City University London

MSc in Computer Games Technology

Project Report

Summer 2013

Virtual Reality Racing Game using Reinforcement Learning

Blair Peter Trusler

Supervisor: Dr Christopher Child

27th September 2013

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the coursework instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed:

Blair Trusler

Abstract

The goal of this project was to develop a racing game using the Oculus Rift and reinforcement learning in the Unity engine. The project involved creating, designing and implementing a racing game using core software engineering principles and methodologies (such as Agile development techniques).

The game was designed and implemented for the Windows platform due to the current limitations of the Oculus Rift hardware interface (which may potentially be expanded to other platforms such as the Xbox One amongst others in the future). The game consists of two game modes; Time Trial and Race. The objective of the Time Trial mode is to achieve the fastest time possible around the race track whilst the goal of the Race mode is to finish three laps in the highest position possible.

A large focus of the project was on the artificial intelligence (AI) of the non-player cars (NPCs). Reinforcement learning and steering behaviour techniques were used to implement this AI. This provided an adaptable solution to the AI and reduced the need to hard code it for different circuits.

This report provides a description of the processes followed in order to achieve the final game build.

Videos of the project can be found at the following link:
http://www.youtube.com/watch?v=rHs_JNVnmp8&list=PLyqnh5008wxDA7NhQAhgtCA8wbQOwajHV

Appendix M contains an academic paper which was written for this project with the aim of submitting the produced results to the IEEE Conference on Computational Intelligence and Games.

Key Words

Unity, Oculus Rift, Reinforcement Learning, Steering Behaviours, Finite State Machines, Artificial Intelligence, Game Design, Software Engineering, Agile

Contents

List of Figures 8
List of Tables 9
List of Algorithms 9
List of Equations 9
1. Introduction and Objectives 10
1.1 Artificial Intelligence 10
1.1.1 Reinforcement Learning in Games 10
1.1.2 Steering Behaviours 10
1.2 The Oculus Rift 11
1.3 Project Scope 11
1.4 High-Level Project Goals 12
1.5 Project Report Outline 12
2. Literature Review 13
2.1 Artificial Intelligence in Racing Games 13
2.1.1 Simple Steering Behaviour – Waypoint Seek 13
2.1.2 Further Steering Behaviours 14
2.1.3 Pathfinding and Navigation Meshes 15
2.1.4 Reinforcement Learning 16
2.1.5 Neural Networks 18
2.1.6 Finite State Machines 19
2.1.7 Decision Trees 19
2.1.8 Chosen Methods for Implementation 19
2.2 Virtual Reality 20
2.3 Game Design 21
2.4 Software Design 22
2.5 Survey Writing and Feedback Collection 23
3. Methods 24
3.1 Overall Game 24
3.2 Game Rules 25
3.2.1 General Gameplay 25
3.2.2 Time Trial 25
3.2.3 Race 25
3.3 Design 25
3.3 Software Design 25
3.3.1 Use Case Specification 25

3.3.2 Menu Flow 28
3.3.3 Game Flow 29
3.4 Builds 31
3.4.1 Build 1: Basic Functionality 31
3.4.2 Build 2: Core Game Features 34
3.4.3 Build 3: Further Enhancements 44
4. Results 47
4.1 System Design 47
4.1.1 Build 1 47
4.1.2 Build 2 47
4.1.3 Build 3 48
4.1.4 Core Components 49
4.2 Q-Learning Implementations 52
4.2.1 State-Action Tables 52
4.2.2 Lap Times 53
4.2.3 Episodes 54
4.2.4 Survey Result 55
4.3 Oculus Rift Integration 55
4.4 Additional Survey Results/Comments 56
5. Discussion 57
5.1 Project Management 57
5.2 The Game 59
5.2.1 Unimplemented Features 59
5.3 Optimization 60
5.4 Q-Learning Implementation Comparison 60
5.5 External Feedback 61
5.6 Similar Work/Projects 62
6. Evaluation and Conclusions 64
6.1 Choice of Milestones 64
6.2 Research and Literature Review 64
6.3 Planning and Agile Development 64
6.4 The Unity Game Engine 65
6.7 Reinforcement Learning and Steering Behaviours 65
6.5 Reinforcement Learning in Unity 65
6.6 The Project as a Whole 66
6.7 Future Developments 66

6.8 Conclusion 67
7. Glossary 68
8. References 69
Appendix A: Project Definition Document 75
Appendix B: Pitch Document 87
Appendix C: Game Visuals 89
Appendix D: Use Case Specifications/Diagrams 94
Appendix E: Game Controls 96
Appendix F: Pseudocode for Core Scripts 97
F.1 Car Controller 97
F.2 Player Controller 97
F.3 AI Controller (SteeringBehaviours) 98
F.4 Q-Learning 98
F.5 Obstacle Avoidance 99
F.6 Execute Policy 100
F.7 Racing Line / Spline Creator 100
F.8 Player Position Tracker 101
F.9 Scene Manager (Singleton) 101
F.9 Finite State Machine and States 102
F.9 Time Trial Mode Initialiser 102
F.10 Race Mode Initialiser 103
F.11 Lap Tracker 103
F.12 Render Texture to Plane (Dashboard Information Example) 104
F.13 Car Effect Scripts 104
Appendix G: Q Table Comparison 105
Appendix H: Q Learning Code 111
H.1 Q Learning Version 1 111
H.2 Q Learning Version 2 113
H.3 Reward Function 115
H.4 Data Structures 116
H.4.1 QStore 116
H.4.2 QActions 118
Appendix I: Oculus Rift Integration 119
Appendix J: Survey 121
Appendix K: Survey Results 122
Appendix L: Installation Guide 123

H.1 DVD/USB Contents 123
H.2 Launch Build 123
H.3 Launch in Unity 123
Appendix M: Academic Paper 124

List of Figures

Figure 2.1 Seek Behaviour, 13
Figure 2.2 Intersection Test, 14
Figure 2.3 Wall Avoidance, 14
Figure 2.4 Race Track Grid for A* Path Finding by Tan et al., 15
Figure 2.5 A*Pathfinding Example and a Unity NavMesh, 16
Figure 2.6 Multi-Layer Perceptron, 18
Figure 2.7 Finite State Machine, 19
Figure 2.8 Iterative Agile Development, 22
Figure 2.9 MVC Pattern, 23

Figure 3.1 Game Screenshot Mock-up, 24
Figure 3.2 Final Game Screen, 24
Figure 3.3 Car Controller Use Case Diagram, 27
Figure 3.4 Menu Flow Diagram, 28
Figure 3.5 Time Trial Mode Game Flow, 29
Figure 3.6 Race Mode Game Flow, 30
Figure 3.7 Unity Wheel Collider, 32
Figure 3.8 Race Track with Spline and Colliders, 35
Figure 3.9 Obstacle Avoidance Technique by Buckland, 39
Figure 3.10 Wall Avoidance by Buckland, 40
Figure 3.11 Sequence diagram of AI controller, 41
Figure 3.12 View Frustums of Oculus Rift Cameras, 41
Figure 3.13 Car Dashboard and Gear Shift Lights, 43
Figure 3.14 Diffuse, Specular and Cubemap Reflection Shaders, 45
Figure 3.15 Cube Map Reflection, 45

Figure 4.1 Build 1 Class Diagram, 47
Figure 4.2 Build 2 Class Diagram, 48
Figure 4.3 Build 3 Class Diagram, 48
Figure 4.4 Car Control Class Diagram, 49
Figure 4.5 Singleton Pattern, 50
Figure 4.6 State Pattern, 50
Figure 4.7 Observer Pattern, 51
Figure 4.8 Relevant Section of the Track (States 93 - 94), 53
Figure 4.9 Standard Deviation of Lap Times (V1 vs V2), 54
Figure 4.10 Standard Deviation of Lap Times based on Episodes, 55
Figure 4.11 Survey Results for AI Question, 55
Figure 4.12 Oculus Rift Survey Results, 56
Figure 4.13 Car Control Survey Results, 56

Figure 5.1 Gantt Chart Representing Actual Workflow, 58

List of Tables

Table 2.1 Components of Agile Development, 22

Table 3.1 Use Case Identification, 25
Table 3.2 Main Menu Use Case, 26
Table 3.3 Car Controller Use Case, 26
Table 3.4 Time Trial Use Case, 27
Table 3.5 Race Use Case, 28

Table 4.1 Action Table, 52
Table 4.2 State-Action Table - Version 1 (red/italics = optimum), 52
Table 4.3 State-Action Table - Version 2 (red/italics = optimum), 53/53
Table 4.4 Lap Time Table, 53/54
Table 4.5 Episodes vs Lap Times, 54

Table 5.1 Objective-Completion Table, 59

List of Algorithms

Algorithm 3.1 Q Learning Pseudocode – Version 1, 36
Algorithm 3.2 Q Learning Pseudocode – Version 2, 37
Algorithm 3.3 Obstacle Avoidance Pseudocode (adapted from Buckland's implementation), 38
Algorithm 3.4 Position Tracking Pseudocode, 42

List of Equations

Equation 2.1 Q-Learning Formula, 16

Equation 3.1 Q Learning Formula, 35
Equation 3.2 Q Learning Formula Components, 35

1. Introduction and Objectives

This report explains the details of a project undertaken to fulfil the requirements for the MSc in Computer Games Technology at City University London. The aim of the project was to develop a virtual reality racing game for the PC. This was designed to be achieved using reinforcement learning techniques for the artificial intelligence (AI) and the Oculus Rift [1] for the virtual reality aspect. The allocated time for the project was 15 weeks over the summer of 2013.

The virtual reality element of the project was inspired by the growing popularity of the Oculus Rift head mounted display (HMD), as described in section 1.1. This notoriety, coupled with native Unity integration, made it an obvious and exciting choice on which to focus the project.

It became apparent in the early stages of development, after receiving the development kit, that integrating the Rift into the game would not be as big of a challenge as initially imagined and as such the emphasis of the project was shifted to focus on the artificial intelligence of the non-player characters (NPCs) in the game. Racing AI has always been a popular and exciting area of game development with a wide variety of techniques and methods at use across different racing games.

The project development was carried out on a Windows-based desktop PC and laptop. A four month Unity Pro License [2] was provided with the Rift development kit which allowed for advanced features of the engine to be accessed (such as custom shaders).

The project lifecycle was focused on replicating a professional schedule and timeline. As such, an Agile [3], iterative approach was followed throughout development. This is reflected in section 2.4 when performing the literature review on software development techniques.

1.1 Artificial Intelligence

There are many approaches to implementing AI agents in racing games. As discussed in the literature review (section 2.1), reinforcement learning (Q learning) [8] was chosen as the primary technique to be implemented in this project. Other behaviours such as obstacle avoidance were also implemented to improve the realism of the AI controller.

1.1.1 Reinforcement Learning in Games

The concept of reinforcement learning is to relieve the programmer of hard coding a controller for the AI agent and therefore creating a system that can “learn” to interact with its environment. This is achieved by representing the game world as a series of states and assigning a large reward when the agent moves to the goal state.

Reinforcement learning was chosen upon as the primary method of creating the AI controller after performing the literature review. The literature review identified several feasible options (such as finite state machines, decision trees and neural networks) however it was ultimately decided to use reinforcement learning due to its many benefits. The primary benefit is the reusability it provides as it reduces the need to hard code a controller for various tracks and environments.

1.1.2 Steering Behaviours

The concept of steering behaviours was also discovered during the literature review. The most useful and pertinent type of steering behaviour was obstacle avoidance. Obstacle avoidance is an essential

behaviour that must be incorporated into any vehicle based game. It is a behaviour that steers a vehicle away from obstacles in its path [9]. This behaviour is essential to create a realistic reaction from the AI cars when faced with opposing vehicles within a close proximity to them. This would help to simulate overtaking and crash avoidance. More details on the theory of this behaviour are discussed in the literature review (section 2.12).

1.2 The Oculus Rift

Over the past decade, video games have become more realistic, engaging and immersive than ever before [1]. This is mainly due to the large improvements in graphics and processing technology [2]. The level of immersion in games will never truly be realised, however, using traditional screen technologies such as monitors, TVs and projectors. The Oculus Rift is a brand new head mounted display (HMD) aimed at bringing virtual reality back to the forefront of gaming and technology in general.

HMDs have always been popular amongst the virtual reality community but until recently the technology has not existed to incorporate them into games [4]. The main issue with traditional HMDs is the high latency experienced in rendering and displaying the scene when the user turns or rotates their head. This issue has been resolved with the release of the Oculus Rift [5], a HMD specifically designed for use with video games and real-time rendering and processing. The Oculus Rift allows for a level of immersion that was previously unachievable with past technology. The main reason for this is due to the wide field of view and low latency that the Rift provides [7]. John Carmack of *id Software* was one of the first developers to test and develop for the Rift. Carmack was quoted as saying his early build of Doom 3 for the Rift was "*the best VR demo ever made*" [7].

One aspect of this project will focus on integrating the Oculus Rift in order to create a truly immersive racing experience.

1.3 Project Scope

The choice to make an entire game as opposed to focusing on a particular area of computer games technology (such as graphics or AI) was made for several reasons. The primary reason was to provide the experience of seeing an entire game project through from start to finish. This would provide exposure to all of the key aspects of game development; design, implementation (AI, graphics and gameplay) and testing.

The game design process took a formal approach, using relevant literature and principles as a foundation to construct a robust design. The use of formal processes helped reduce the risk of flaws and mistakes in the design. The design also took shape from playing similar racing games on the market (for example Forza Motorsport [10], Gran Turismo [11] and the F1 series of games [12]).

The implementation process was also planned in a structured way. Traditional and well established software engineering techniques (such as Agile and SCRUM) were used to ensure that the system was implemented to schedule and that testing and refactoring took place at regular points in the project development.

The choice to make a full game was also made feasible due to the power and flexibility of the Unity engine. Unity provides game developers an entry point into game development which allowed for many of the early stages to be skipped (ie implementing an underlying graphics and audio engine architecture) and focus immediately on implementing the game itself.

As the focus of the project is on the design and implementation of the game, the 3D models and art used in the game were taken from free, open sources from various sites on the internet. It would be

unrealistic to attempt to create custom models and artwork for a project of such a short timescale as well as accomplishing all of the other features of the game that were required to be implemented.

1.4 High-Level Project Goals

A high level plan for the game implementation was outlined at the start of the project. This was split into three builds which each involved several key milestones which would result in the desired game at the end of the project timeframe.

Build 1:

- B1.1 Literature review and game design.
- B1.2 Unity project stub with content hierarchy.
- B1.3 Basic car control and physics on simple terrain.
- B1.4 Integrate Oculus Rift into player car.
- B1.5 Import race track and ensure car and camera behave as expected.

Build 2:

- B2.1 Improve car control physics to achieve higher level of realism.
- B2.2 Create AI car control system.
- B2.3 Integrate head tracking features of Oculus Rift to simulate realistic cockpit.
- B2.4 Game modes (race and time trial).
- B2.5 Implement dashboard and heads-up display.

Build 3:

- B3.1 Weather effects to vary car behaviour
- B3.2 Modify AI into difficulty levels
- B3.3 Shaders and visual effects
- B3.4 Leaderboard table

1.5 Project Report Outline

This report begins with a literature review (section 2). This review is divided into two parts; literature related to game and general software development and then a more technical review on specific parts of the implementation (mostly related to virtual reality and AI).

Section 3 discusses the game design, including the various choices and decisions made during the initial stages of the project. This is followed by the methods (section 3). This section goes into detail regarding how the core elements of the game were implemented in the Unity engine. This involves lots of low level detail on how various problems were tackled and provides pseudocode for the core algorithms used in the game.

This is followed by the results (section 4). This section presents the data collected from the final implementation of the project.

Section 5 provides a general discussion of the project and the results, including whether or not the original specifications had been achieved as well as what further work would be required to create a full retail game.

The concluding section (section 7) provides a final evaluation of the project and discusses what has been learnt from the project and what could have been done differently or improved upon given more time and the potential future developments.

2. Literature Review

It was important to perform a thorough literature review before starting design and development due to the large scope and high ambitions for this project. As such, the literature review was broken down to encapsulate the four core elements of the project; the AI, virtual reality, game design and software design.

Whilst the aim of this project is to create a game through the various stages of game development (alpha, beta and release builds), there was a heavy emphasis on the artificial intelligence and the virtual reality aspects in the game design. As such, a large portion of the literature review was focused on investigating these areas of game development.

It was also important to research current game design and software development techniques to ensure the overall project design and implementation was relevant and robust to modern standards.

2.1 Artificial Intelligence in Racing Games

There are a vast number of options when it comes to implementing a racing AI agent. As such, it was important to review the core principles used across a range of implementations. This investigation revealed that the most commonly used methods were; steering behaviours, pathfinding algorithms, reinforcement learning and neural networks. Each of these methods was considered and reviewed in order to make an informed decision to the approach to be taken by this project.

2.1.1 Simple Steering Behaviour – Waypoint Seek

Buckland (2004) [13] discusses the many steering behaviour techniques which were first created by Craig Reynolds in 1999 [14] and then tailored towards their implementation in video games. The most traditional and simplistic of these behaviours is the *waypoint seek* method. This involves placing several waypoints (target positions) along the race track and creating a simple controller that steers and accelerates to each waypoint in turn.

This method works by producing a scaling force that directs the agent to the waypoint position (the *desired velocity*). This velocity vector represents the vector from the agent's position to the waypoints position, scaled by the maximum speed of the agent. The steering force is then calculated by simply subtracting the current velocity from the desired velocity vector.

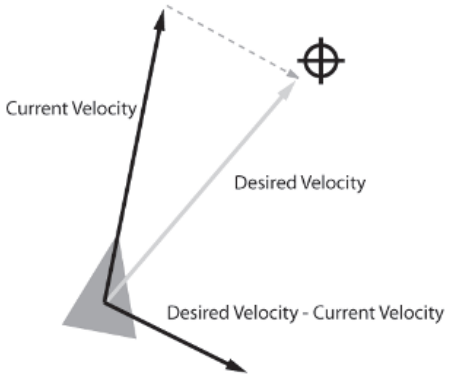


Figure 2.1 Seek Behaviour [13]

This behaviour produces the a very simple AI agent. This method alone however will result in unrealistic actions, as the AI car may hit other objects or cars in the scene because it is not searching for any other game objects apart from the waypoints.

2.1.2 Further Steering Behaviours

Waypoint seek is the simplest form of steering behaviour for autonomous vehicle agents in games. Buckland also presents others which can be combined to create more realistic and fluid behaviour. It was important to review and assess each behaviour to see which could be incorporated into the AI agent.

The first behaviour Buckland discusses is *flee*. This works in the opposite way to seek; it creates a force that steers the agent away from a target position. The vector produced is aimed in the opposite direction to the target position to create a very sharp change of direction and speed.

The *arrive* behaviour is used to slow the agent down when approaching a target position or waypoint. This is often combined with the seek behaviour to bring the agent to a gentle stop when reaching the waypoint. This is accomplished by calculating the time it will take for the agent to reach the target and as a result it is possible to calculate the desired speed of the agent at a given point along the desired velocity vector.

The next behaviour is *pursuit*. This is useful when trying to calculate the position of a moving target at a given time. This requires predicting the target objects trajectory based on its velocity and direction vectors. The *evade* behaviour works in the opposite way to pursuit by fleeing from the estimated future position.

The individual behaviours discussed to this point are not specifically pertinent to a racing AI agent but reviewing them has given further insight about how vehicle behaviour can be manipulated using vectors as forces. This helped when investigating the next two behaviours that Buckland discusses, obstacle avoidance and wall avoidance, which are relevant to a racing AI agent and utilise some of the principles reviewed already.

Obstacle avoidance is a behaviour that steers a vehicle to avoid and move around obstacles in the game world. This is achieved in several steps, firstly by creating a detection box region in front of the vehicle. Tests are then performed to determine if any obstacles are within a radius (of the detection box length) around the vehicle (see figure 2.2). Any obstacles behind the vehicle are ignored. The next stage is to perform line-intersection tests with any obstacles ahead of the vehicle (taking the radius around both the obstacle and the vehicle into account) to ensure that it is within the detection box. If any obstacles are detected, the algorithm sorts them based on their proximity so that nearby obstacles will be avoided before further ones. Once the obstacle is found it is a relatively simple process to calculate the appropriate steering force required to avoid it. This is calculated by subtracting the local x component of each obstacle position. A braking force can also be calculated by creating a vector in the opposite direction to the vehicles velocity vector. This braking vector is scaled based on the vehicle's distance from the obstacle.

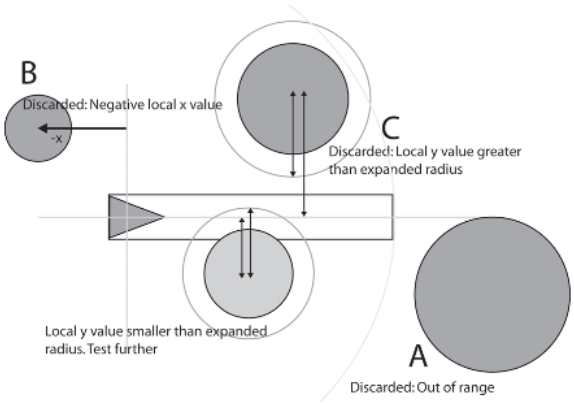


Figure 2.2 Intersection Test [13]

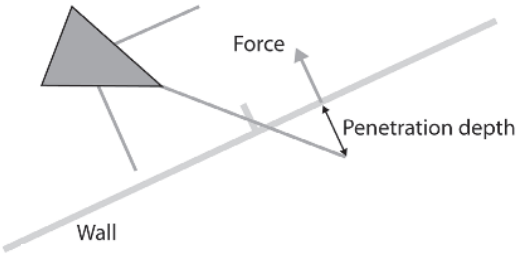


Figure 2.3 Wall Avoidance [13]

Wall avoidance is a behaviour which steers the vehicle away from walls if it gets too close. Walls are detected by

sensing the nearby environment (achieved in Unity by ray casting [15]). Ray casting is the process of shooting a *ray* (a vector) through the game world and tests are performed to determine if any objects or obstacles intersect with it. If a wall is detected by the car, the steering force must be calculated. This force is calculated based on how much the sensor has penetrated the wall and then creating a force of equal magnitude in the direction of the normal of the wall, hence repelling the vehicle.

2.1.3 Pathfinding and Navigation Meshes

Pathfinding algorithms (in particular A* pathfinding [16]) and navigation meshes (*NavMeshes*) [17] are common methods of solving NPC movement across game worlds in games.

A* pathfinding represents the game world as a weighted graph of nodes [16]. The weighting of each node represents how much effort (or *cost*) it would take to traverse along it. For example, a very high brick wall would have a weighting of close to infinity (meaning it was impossible to pass through), whilst a path would have a cost value of 1. The A* algorithm is optimally efficient, meaning it will find the best path in the most cost-effective way (in terms of processing power required). In order to achieve this it makes use of a heuristic function [16].

Even though the algorithm is optimally efficient, the main issue with it in a racing game would be the computational expense of regularly calculating paths. The path would have to be frequently re-calculated based on its updated position around the circuit. If the weighted graph took moving obstacles (such as other cars) into account, the path may need to be recalculated even more often.

This also raises the issue of how to generate the weighted node graph for the track. The car should follow the racing line as accurately as possible, but it would be a challenge representing this in a weighted graph of nodes. It would be much easier and less time consuming to place the racing line in the game visually (such as in the waypoint method) to allow the designer to shape and mould the racing line until it produces the desired effect.

One further issue with using A* is guiding the algorithm to find a path in the correct direction around the circuit. For example, if the car is just past the start/finish line and tries to calculate a path, it will think it is already very close to the destination and return a short path that would turn the car around and drive the wrong way. This could be improved by searching for shorter paths around the circuit or by modifying the heuristic to receive a large cost for going backwards. As with the graph issue, however, this is not the most intuitive way for the designer to modify the agent's behaviour.

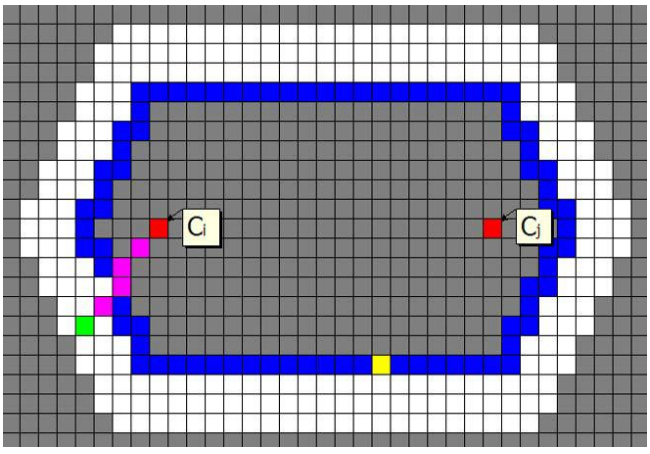


Figure 2.4 Race Track Grid for A* Path Finding by Tan et al. [18]

Tan et al. (2008) [18] focused on creating a tool for a horse racing game using A* to create a route around a track. They also created a system that generated the cost map of each node on the track based on the 3D model that was imported into the system. In addition they used the A* algorithm to

perform real-time obstacle avoidance. One issue they encountered, however, was the performance penalty of using A* for many horses. This meant they had to reduce the resolution of the navigation grid which resulted in less accurate movement around the game world. This problem would be increased if more AI agents were introduced into the game. Figure 2.4 shows how the grid produced for the race track.

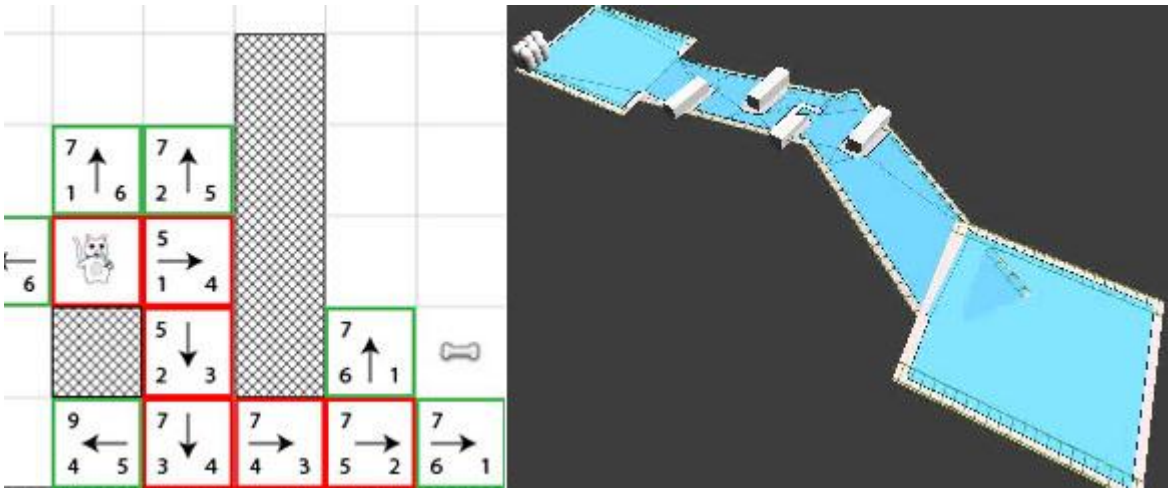


Figure 2.5 A*Pathfinding Example [19] and a Unity NavMesh [20]

NavMeshes provide similar functionality to path finding algorithms but remove the need for a weighted node graph [21]. The NavMesh takes the mesh of the geometry in the game world and creates a simplified mesh layer on top of it to be used for movement around the scene. As a result, complex maps and environments would not be possible. For example a tunnel would create a circular NavMesh meaning that the agents may move along the walls or ceiling if it would result in a more optimal route.

Whilst this would allow for a more rapid development time for the AI, the NavMesh implementation in Unity is still in its early stages of development. It is primarily designed for humanoid AI agents and thus is not optimized for creating realistic vehicle behaviours. It is also not possible to customise a racing line for optimal travel around the track, as the mesh will take the entire width of the track into account. An example of the Unity NavMesh is shown in figure 2.5.

Whilst these approaches to vehicle AI are not optimal for a racing game for the reasons discussed above, they are commonly used in large open-world games such as Grand Theft Auto [22] where many vehicles need to traverse a game world using a shared node graph or NavMesh.

2.1.4 Reinforcement Learning

As briefly discussed in section 1.2, reinforcement learning is a technique which allows the AI agent *learn* how to interact with the game word. This technique is becoming ever more popular for creating NPCs in games.

The most common form of reinforcement learning is Q-Learning [23], which is most useful for finding the optimal action-selection combination (policy) for a finite set of states [23]. Equation 2.1 gives the formula for this algorithm. This formula is discussed in more detail in section 3.2 of this report.

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_a'(Q(s', a')))$$

Equation 2.1 Q-Learning Formula [23]

Patel et al. (2011) [24] used Q-learning to create a bot for the popular first-person shooter game *Counter-Strike* [25]. Their paper discussed traditional game AI techniques, including decision trees and finite state machines (FSMs). These techniques can be very time consuming for the game developers to fine tune the AI agents to achieve the desired behaviour and interactions with the game world. Their solution was to use Q-learning to train a simple AI agent to teach it how to fight and plant a bomb in the correct location. They gave the AI a higher reward value if it accomplished the goal of the game (planting the bomb receives a higher reward than killing an enemy). Their results showed that the Q-learning bots performed competitively against the traditionally programmed bots. However, they did note that this was not tested against players. This would be likely to identify further issues that would need to be resolved in the learning and reward allocation algorithm.

A popular commercial racing game that makes heavy use of reinforcement learning is the Forza series. Forza makes use of "*Drivatar*" technology [26] [27]. In addition to the referenced presentation, I was fortunate enough to have a job interview at Playground Games, creators of the Forza Horizon game and they were able to shed more light on how the AI agents work in the game. The development team created a database of pre-generated racing lines for every corner on a race track (several slightly different lines per corner). For example, some racing lines will be optimal whilst others may go slightly wide and miss the apex of the corner.

The agent uses reinforcement learning (Q learning) to learn the appropriate throttle values to follow each racing line as fast as possible. The cars also learn various overtaking manoeuvres at each part of the track. This learning process is performed offline and stored (before the game is shipped). During a race, the racing lines at each corner are switched to vary the behaviour of the AI car.

This approach relieves the programmers of hard-coding the values for each possible track and corner and creates a reusable and effective tool for creating AI agents. This means that a lot of the processing required for the AI is removed as the learning algorithm executes offline. As a result, the AI only needs to track other car and player positions in order to know when to switch racing lines to avoid collisions. This technique has led to Forza having one of the best AI systems in the racing game market today.

Reinforcement learning provides the option to perform the training process online or offline [28]. Online learning means that the agents train in real-time while the game is running. This gives rise to several key issues. Firstly, it is a highly computationally expensive process and as such could cause an impact on the frame rate and processing of other game logic. Secondly it means that the AI may not behave in a very optimal or "clever" way when the game is first launched as it needs to learn how to behave properly. Furthermore it means that the designers must have complete faith that the agent will learn as expected to produce the desired behaviour. However, in theory it can provide a more adaptive, reactive and unpredictable agent which could potentially improve the quality of the gameplay.

Offline learning counters the issues that arise in online learning. The policy is learnt over thousands of iterations providing a solution that can be tweaked and modified by the designer to achieve the desired behaviour. It also means the computational expense is greatly reduced when running the game as the agent simply has to refer to the policy to determine how to behave. It may also be possible to combine offline and online learning techniques. This could work by training an agent offline and then using a lightweight online learning algorithm to gradually modify and adapt the cars behaviour based on the changes in the game over time. This could create a realistic controller that would learn and react to the counter the player's gameplay tactics.

2.1.5 Neural Networks

Neural networks [29] are systems that attempt to emulate a human brain in order to perform tasks and challenges. AI agents in games are traditionally very rigid and rule-based systems. Neural networks can improve this rigidity by providing the ability to learn and adapt to changing conditions and environments in the game world.

Neural networks consist of interconnecting nodes between several layers. The most common type of neural network is known as a Multi Layered Perceptron (MLP) [30]. MLPs consist of multiple layers with each layer fully connected (with varying weights) to the next layer. Each node in the graph is a neuron (except for the input nodes) with an activation function (defining how to manipulate the input data) [31]. Supervised learning techniques are typically used to train the network [32]. The output layer of the network is determined as a result of the inputs and weights provided. As with reinforcement learning, neural networks can both be trained online and offline and both have the same advantages and disadvantages that arose with reinforcement learning.

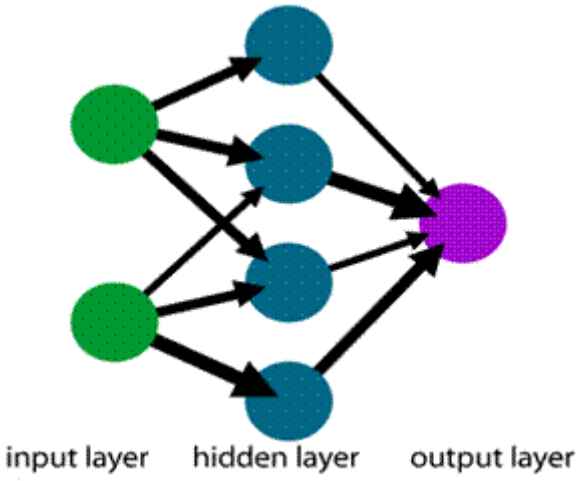


Figure 2.6 Multi-Layer Perceptron [33]

Tan et al (2011) presented the use of a neural network in conjunction with the classic Ms Pac-man game [34]. The training process focused on minimizing the number of hidden neurons whilst maximizing the score in the game using the Pareto Archived Evolution Strategy (PAES) [35] to evolve the network. The controller was then tested using a single-net (choose best evolved network) and multi-net (choose best of five evolved networks from five recorded runs). They found that the best controller was developed using the multi-net approach as it allowed the controller to adapt given the complexity of the game. This paper, however, was focused on using a neural network to tackle the goals of the game from the player's perspective and not from the AI perspective. This would add a great deal more complexity to the implementation as it would have to account for the many different ways the player could play the game.

Neural networks have not been incorporated into many commercial video games. One game that did utilise something similar however was *Black & White* by Lionhead Studios (2001) [36] which made use of a single layer perceptron. This was used in conjunction with decision trees to determine the AI behaviour. This worked well and is a special case as a large portion of the gameplay was designed to teach and interact with a large AI creature to help it evolve and learn [37].

2.1.6 Finite State Machines

Finite state machines (FSMs) are a way of modelling and encapsulating sequential logic and transitions in software systems. This can be extremely useful in game AI to create different behaviours given different states of the agent (for example in a calm state, the agents running speed may be lower whilst in an angry state it will be higher).

FSMs are one of the most traditional methods of implementing an AI behaviour system. The transitions between states are usually based on Boolean logic based on what occurs in the game world. FSMs also provide a good way of encapsulating and hiding unnecessary code between the states, increasing the modularity of the implementation.

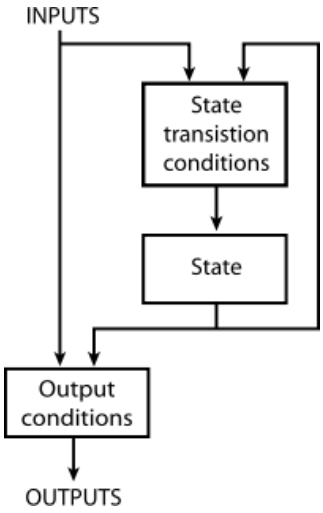


Figure 2.7 Finite State Machine [38]

FSMs can also be hierarchical [39]. This means that a state (a *superstate*) can have states nested within it (*substates*). This can allow for more complex and realistic transitions between states as the FSM will handle an event based on the context within the substate.

2.1.7 Decision Trees

Decision trees are a tool used to create a tree-like model of possible decisions and their relevant consequences. They can also provide the probability of events occurring and the cost of performing an action in the future. Decision trees are useful in helping the AI agent make decisions to reach a goal. Decision trees are similar to hierarchical FSMs but provide more control over the decisions and allow for predictions to be made based on previous decisions.

The main benefit of decision trees is that they are extremely easy to read and understand even with limited knowledge of the game design or programming in general. They can also be easily expanded given new scenarios and decisions.

Pure decision trees alone would not suffice in creating the AI in a racing game where the number of decisions to be made could potentially be quite large. However, it could provide a useful framework to use in conjunction with steering behaviours or with Q learning.

2.1.8 Chosen Methods for Implementation

From the above review and investigation, it became clear that just one method would not be sufficient to create a realistic AI agent. As such, several of the techniques would need to be combined to create the NPC. The combination decided upon was a reinforcement learning agent combined with steering

behaviours (specifically; seek, obstacle avoidance and wall avoidance). This would help to produce a similar version of the AI present in the Forza games whilst utilising real-time steering behaviour techniques as opposed to learning the overtaking and avoidance manoeuvres.

The learning agent would be taught the race track (offline) by varying its throttle values between various sections of the race track. It's steering would be based on a predefined racing line (using the seek steering behaviour on each individual point on the line). Once learnt, the agent could be placed in the game using the saved policy combined with the steering behaviours. This would result in realistic AI agents racing around the track as fast as possible without crashing. This also means that the only AI processing in real-time would be the steering behaviour calculations.

Given a longer timescale for the project, it may have been possible and interesting to integrate a neural network layer above the reinforcement learning algorithm (similar to the way Gerald Tesauro implemented TD-Gammon in 1992 [40]; a backgammon game using a neural network trained by temporal-difference learning). This would allow the AI agent to adapt the policy and racing line during gameplay [41]. An alternative to this could be to use an FSM system to change the driving style of the AI (such as neutral, aggressive, defensive). Various racing lines and policies (using different reward functions) could be used to adapt the cars behaviours accordingly.

In addition to this, a decision tree system would be a welcome addition to improve the readability and reusability of the learning algorithm and steering behaviour techniques.

2.2 Virtual Reality

Virtual reality (VR) was a very popular field within video games and computer science in general until the early 1990s. VR technology and hardware have traditionally been very expensive and cumbersome, with prices commonly exceeding thousands of pounds. A study by TechCast in 2006 [42] found that most people would be willing to pay between $100-500 USD for VR technology for entertainment purposes, with a minimal number of people willing to pay above $1000 USD. This disparity is a clear indicator of one of the many reasons as to why VR fell off the map. A further cause for the drop-off is that VR technology (head mounted displays, HMDs, in particular) typically suffered from high latency problems when moving around a virtual environment. This often causes a break in the immersion from the experience and is another one of the issues that VR has faced in the past.

The Sony HMZ-T1 [43] is a HMD that was released in late 2011 that attempted to tackle the latency issue. Sony managed to succeed in reducing latency issues, but at the cost of image quality (720p screens for each eye). The main stumbling block for the HMZ-T1 was the price tag which was too high (RRP of £700) and as such suffered poorly commercially.

The Oculus Rift has seemingly solved both of these key issues by producing a HMD with 1080p screens for each eye (720p on the development kits as they are working prototypes) and the lowest latency that has been seen on an HMD thus far [44]. Many developers and journalists have been outspoken in how well the hardware operates and many developers are in the process of using it in their projects. Some notable names in the games industry that have endorsed the hardware include John Carmack, Gabe Newell and Cliff Blesinzki. Oculus has also managed to tackle the price barrier with developer kits selling for approximately £200 (the final product with 1080p screens will probably be slightly higher at £250) [45].

The Oculus Rift and upcoming devices such as Google Glass [46] will likely help VR (and Augmented Reality) come back to the forefront of computer science as the consumer market grows and matures. At time of writing, it has been revealed that Sony are working on an HMD of their own

similar to that of the Rift for the upcoming Playstation 4 console [47]. This definitely further points to the fact that VR is once again rising in popularity and hopefully this time it is here to stay.

The goal of the project is to create the most immersive and realistic experience possible (within the constraints of the graphics and art assets available). The paper by Yoon et al. (2010) [48] discusses how HMDs and sound can affect the level of immersion when playing a first-person shooter (FPS) game. They discovered that using the HMD with traditional control methods provided the highest levels of immersion, whilst using the HMD with less traditional inputs (data gloves) proved less successful.

Yoon et al. also found that using headphones with 3D sound increased the levels of immersion. The Unity engine also provides an integrated 3D sound engine. This will allow for realistic 3D audio effects to be implemented into the game with relative ease which should in turn increase the immersion level of the player

The Oculus Rift SDK provides detailed information on how to integrate the device properly into the game using the Unity engine. As the platform is still very young, the developer community forums will help to provide further guidance on virtual reality theory and assist on the implementation of the HMD.

2.3 Game Design

Even though the focus of this project is around the AI and VR fields, a good and solid game design is an essential foundation for any game project. The book *Game Design Complete* by O'Luanaigh (2006) [49] provides a set of design principles that, when followed, should result in a solid game design.

One of the key points O'Luanaigh raises is the *30 Minute Rule*. This rule states that the first 30 minutes of gameplay should be entertaining and showcase enough gameplay mechanics and features to hold the player's attention to keep them engaged and wanting to play more. Aside from gameplay mechanics, additional rewards (such as achievements) and unlockable items can be used as incentives to keep the player interested in playing more of the game. This is a relatively simple addition which can make a big difference in player engagement.

O'Luanaigh also points out that the game should not cause the player to become frustrated. This can be achieved by providing helpful hints and modifying the gameplay difficulty if the player fails at an obstacle several times. This is also closely linked to the game difficulty. Ideally, the game difficulty will adapt to the players level of skill. This is a challenging feat to achieve in many game genres. In racing games, however, this could be relatively easy to accomplish as they provide measurable data (in the form of lap-times) which could be used to adapt the racing style of the NPCs. Many racing games also utilise the *rubber-band technique* [50] which allows the player or NPCs to catch up with the leading car(s) if they are far enough ahead (a good example of this can be seen in the Mario Kart games [51]). This technique has become less frequently used as it can ruin the progression the game from the player's perspective and they can feel like they are not improving or being rewarded for their skill.

The game design will also be formalized using traditional UML techniques to ensure the design is robust and logical (section 3.4). The techniques to be used will include use case diagrams, activity diagrams and state flow diagrams.

2.4 Software Design

The project will follow an Agile development process. Agile is a development technique that consists of several core methodologies, first introduced in the Agile Manifesto in 2001 [52]. The goal of Agile is to focus on the software being developed to ensure that it can adapt easily and quickly to changes and issues that may arise. The table below outlines the core elements of Agile development [53][54].

Process Name	Description
Scrum	An iterative and incremental approach to software development. Goals are divided into small, achievable sub-goals which can be completed within hours or days. Scrum also encourages regular meetings (typically daily or weekly) with the development team to ensure development is on track and any issues that arise can be tackled immediately.
Iterative Development	This is the principle that the software gradually evolves from a state of minimal functionality to eventually reaching the full software as described in the specifications. The software gradually becomes more functional and useful over the course of development.
Sprints	Sprints are the basic unit of Scrum development. Sprints are short time periods of development (typically between one week and one month) which at the end should result in a goal being achieved.
Product Backlog	The product backlog is the complete, ordered list of requirements for the software being developed.
Sprint Backlog	The sprint backlog is the list of requirements and objectives to be implemented in the current sprint.
Extreme Programming (XP)	XP is heavily related to the iterative development process. It focuses on producing frequent releases of the software which can then be reviewed by the management/customer. This allows the management/customer to steer the project back on course early at an early stage if anything falls out of place or is not functioning as intended.
Pair Programming	This is a technique whereby two programmers work together on the same computer and backlog item. One programmer is the *driver* (writing the code) whilst the other is the *observer* (reviews the code as it is written). They swap roles at regular intervals. The observer has a fresh perspective over the implementation and may provide assistance in the event of a problem that may arise. This also helps to reduce the number of bugs and improves the overall quality of the code.

Table 2.1 Components of Agile Development

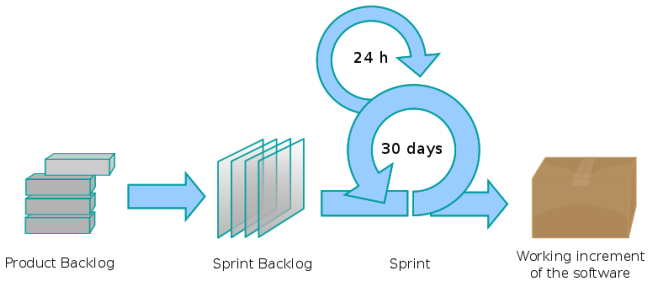


Figure 2.8 Iterative Agile Development [55]

In addition to utilising Agile techniques, one invaluable piece of literature that will be used throughout the project is *Design Patterns* by (Gamma et al, 2004) [56]. This will help to produce a robust and reusable codebase by using industry standard techniques.

The Unity engine has been designed so that it provides a loosely coupled architecture, meaning upgrades and modifications can be done to the Engine through updates without affecting the behaviour of the engine as a whole. This is achieved by using the Model-View-Controller (MVC) pattern [57]. This pattern separates the game logic (the model) from the rendering process (the view) from the input system (the controller).

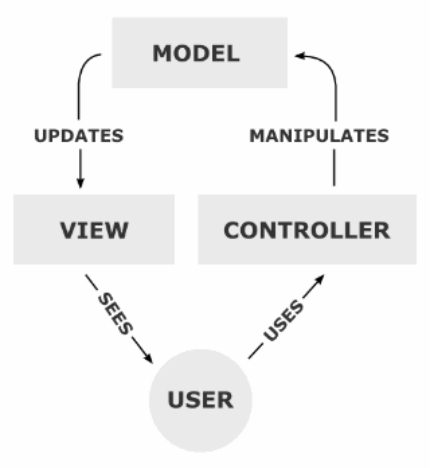


Figure 2.9 MVC Pattern [58]

The other state patterns that will be used in the project include the state pattern, observer pattern and the singleton pattern. Further details on the use of these patterns can be found in section 4.1.4.

2.5 Survey Writing and Feedback Collection

A survey will be used at the end of the project to collect subjective results (for example in relation to the Oculus Rift integration). As a result it is important to understand the fundamentals of survey writing and data collection. The underlying principle of survey writing is to ensure the questions are clear, concise and not misleading in any way [59] as this may result in confused and unrealistic answers and responses. In multiple choice questions it is vital to include enough options to allow the user to feel like they can provide enough feedback that corresponds to their feelings [60]. For non-multiple choice questions it is important to not ask too many vague and open-ended questions.

Finally, not all questions should be required to be completed (such as the text-based response sections). Having all the questions required may be off-putting and cause the user to not fill out the survey at all.

Following these practices should help to ensure a concise and useful set of results is produced from the survey collection process.

3. Methods

3.1 Overall Game

Figure 3.1 shows an early mock-up of the how the game was designed to look. As the game was primarily designed for use with the Oculus Rift, the game was designed so that all of the key details (ie speed, lap-time, lap number and position) could be displayed on the car dashboard to increase levels of immersion with use of the Oculus Rift and hence eliminate the need for a complex HUD.

HUD design for HMDs is a large field in itself and a great deal of research is currently being done by the VR community on how best to design them for such devices. As such, a very simple HUD was implemented to display the necessary additional information.

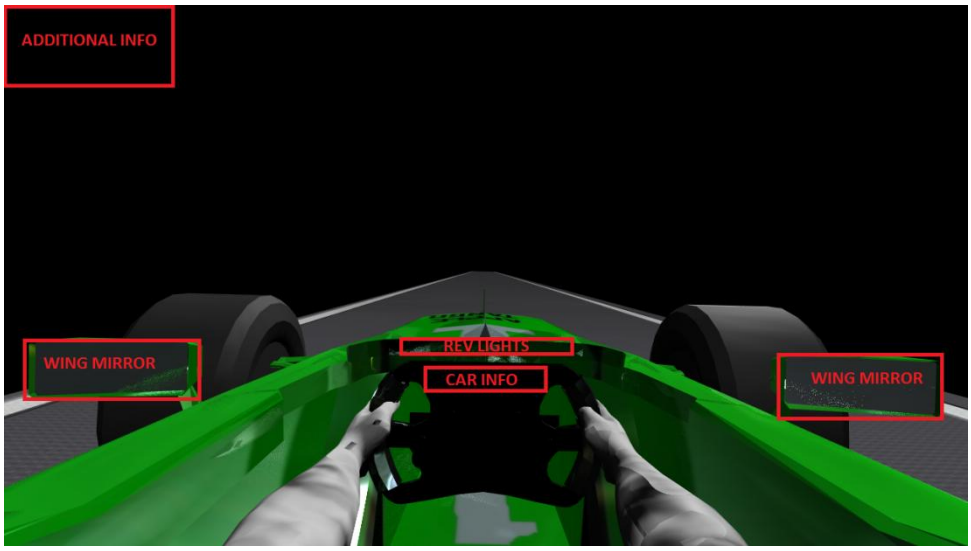


Figure 3.1 Game Screenshot Mock-up [61]

Figure 3.2 Final Game Screen (see Appendix C for more visuals)

3.2 Game Rules

3.2.1 General Gameplay

The game puts the player in control of a Formula 1 racing car. The player drives the car using an Xbox 360 controller while utilising a traditional racing game control scheme (see Appendix E for a full control mapping diagram). The player's objective varies depending on the game mode (see sections 3.2.2 and 3.2.3).

3.2.2 Time Trial

The *Time Trial* game mode challenges the player to achieve the fastest lap time they can around a circuit. There are bronze, silver and gold rated times displayed to the player at the start of the game mode to give them an idea of what is a fast time around that particular circuit. This game mode proceeds indefinitely until the player decides to quit. This is done by accessing the pause menu and returning to the main menu.

3.2.3 Race

The *Race* game mode challenges the player against three AI NPC agents. The player starts from a random position on the grid with the goal being to achieve the highest position possible before the end of three laps. Once the race is over, the final position achieved is displayed and the player is returned to the main menu screen.

3.3 Design

Many racing games have a steep learning curve when it comes to vehicle handling and performance and as such create a barrier to entry for many people. In an effort to counter this, the aim of the game design was to create a racing game that anyone could pick up and play. This would be achieved by creating a car handling system that does not mimic reality completely but still feels fun and responsive to the player.

The game genre is classified as a racing/arcade racing game. It could also be classified as a virtual reality game or experience due to the integration of the Oculus Rift.

3.3 Software Design

The design of the software was planned out using traditional unified modelling language (UML) [62] techniques.

3.3.1 Use Case Specification

Several use case designs were identified and planned before the implementation of the project to ensure the software design was robust and encapsulated all of the specifications. The detail has been abstracted to produce simple, easy to read and understand diagrams of how the system is meant to work. The use case diagrams can be found in Appendix D.

Use Case	Primary Actor
Main Menu	Player
Car Controller	Car (Player/AI)
Time Trial Mode	Player
Race Mode	Player/AI

Table 3.1 Use Case Identification

3.3.1.1 Main Menu

The main menu is where the player can select which game mode to play as well as viewing the leaderboard (displaying quickest lap times and previous race results) and the game controls. If the player selected a game mode, they can then select whether they would like to play in dry or wet conditions.

Use Case Name: Main Menu	**ID:** 01
Primary Actor: Player	
Stakeholders and Interests: Player: wants to play the game.	
Brief Description: This use case describes how the player navigates the main menu.	
Trigger: When the player launches the game.	
Relationships: **Association**: Game Executable **Include**: Play Game, Player Name Details	
Normal Flow of Events: 1. Player launches the game from the executable. 2. Player selects an option. 3. IF: Time Trial Mode/Race Mode a. Select Weather. i. IF: Dry: launch dry version of specified mode. ii. ELSE: launch wet version of specified mode. 4. IF: View Leaderboard a. Display leaderboard. 5. IF: View Controls a. Display controls.	

Table 3.2 Main Menu Use Case

3.3.1.2 Car Controller

The car controller is the system that controls and manages the behaviour of the cars in the game. It is inherited by both the player and AI agents who provide the steering and throttle values. The car controller also initializes and sets the tyres and their respective friction values based on the weather of the game. It also updates the gears, RPM and in addition handles the wheel rotations of the car.

Use Case Name: Car Controller	**ID:** 02
Primary Actor: Player/AI	
Stakeholders and Interests: Player: wants to use controller input to control car. AI: wants to use learnt throttle/steering values to control car.	
Brief Description: This use case describes how the car is controlled.	
Trigger: When a game mode is launched.	
Relationships: **Association**: Q Learner (AI only), Wheel Collider **Include**: MonoBehaviour	
Normal Flow of Events: 1. Game mode is launched. 2. Car is initialized. 3. IF Player: a. Receive input from controller. 4. IF AI: a. Receive input from policy. 5. Set throttle/steering values 6. Calculate and update car information.	

Table 3.3 Car Controller Use Case

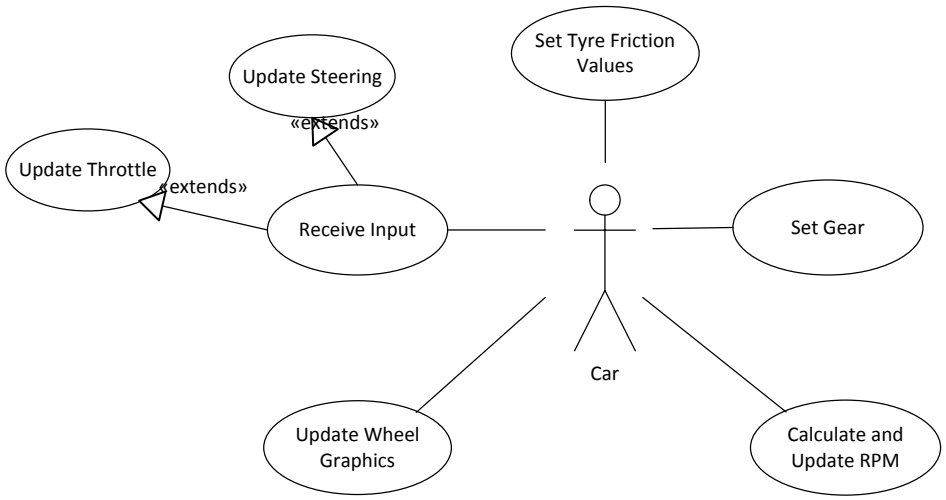


Figure 3.3 Car Controller Use Case Diagram

3.3.1.3 Time Trial Mode

The time trial mode only involves the player. Apart from handling the player's input for car control and the pause menu, it requires a tracker to record the lap times and store the fastest lap achieved.

Use Case Name: Time Trial Mode	**ID:** 03
Primary Actor: Player	
Stakeholders and Interests: Player: wants to play time trial mode.	
Brief Description: This use case describes how the time trial mode operates.	
Trigger: When the player selects the time trial option from the main menu.	
Relationships: **Association**: Car Controller **Include**: Player Name Details	
Normal Flow of Events: 1. Player launches the time trial mode and selects a weather option from the main menu. 2. Mode starts with rolling start, player given control once crossing the start/finish line. 3. IF pause menu open: a. IF "Quit" selected: i. IF recorded lap time better than current top 5: store lap time. ii. Return to main menu. b. IF "Return" selected: i. Continue game. 4. ELSE continue game.	

Table 3.4 Time Trial Use Case

3.3.1.4 Race Mode

Unlike time trial mode, race mode incorporates AI agents. The player implementation is very similar to the time trial mode, but instead of tracking and storing the best lap time it tracks the players position amongst the total number of cars in the race and how many laps are left to be completed.

The AI agent has the ability to perform the reinforcement learning process. This function is disabled in the race mode and is only used in the development environment. Instead, the agent executes the policy that has been stored from the learning algorithm. It also performs the avoidance techniques to ensure it doesn't crash. These algorithms modify the car controller's steering and throttle values accordingly.

Use Case Name: Race Mode	**ID:** 04
Primary Actor: Player	
Stakeholders and Interests: Player: wants to play race mode and finish in the best position possible. AI: wants to drive around circuit as fast as possible whilst avoiding obstacles.	
Brief Description: This use case describes how the race mode operates.	
Trigger: When the player selects the race option from the main menu.	
Relationships: **Association**: Car Controller **Include**: AI Controller	
Normal Flow of Events: 1. Player launches the race mode and selects a weather option from the main menu. 2. Lights countdown to signify start of race and inputs are enabled (for both player and AI) once all lights are out. 3. AI car controllers receive input based on policy and obstacle/wall avoidance values. 4. Track player position amongst all cars. 5. IF player completes 3 laps: a. Race over, display finishing position. b. IF "Quit" selected: return to main menu.	

Table 3.5 Race Use Case

3.3.2 Menu Flow

The menus of the game were designed to use an FSM architecture. As such, the menu states are represented in the state diagram in figure 3.4.

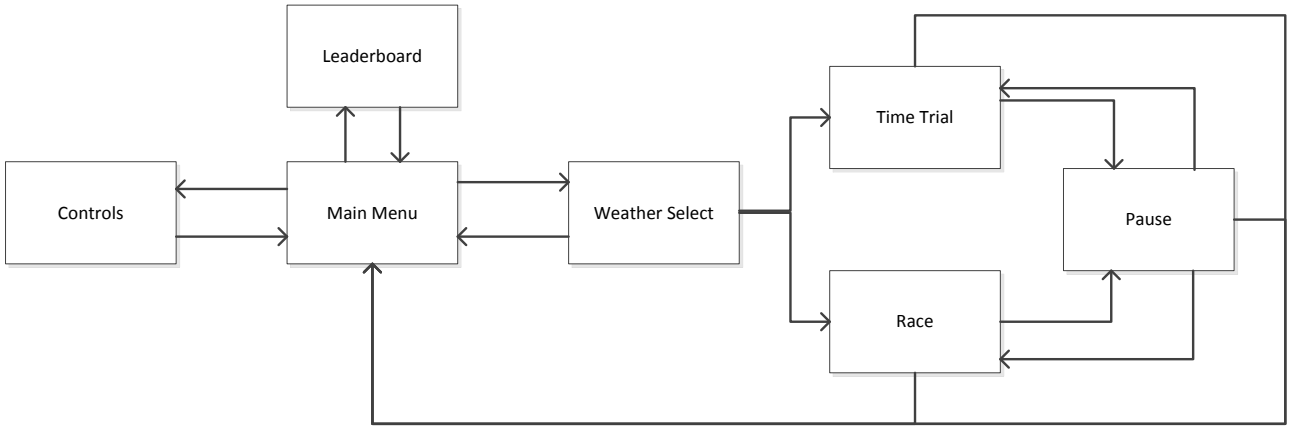


Figure 3.4 Menu Flow Diagram

3.3.3 Game Flow

The activity diagrams in figure 3.5 and 3.6 outline the design of the game flow for each of the game modes.

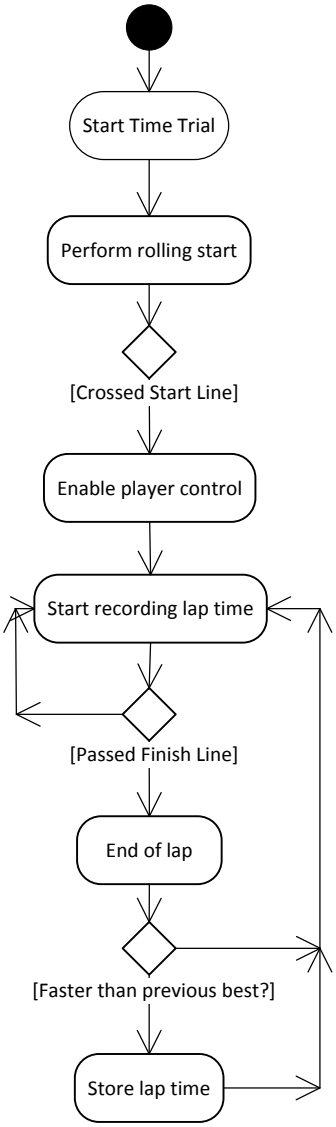


Figure 3.5 Time Trial Mode Game Flow

The time trial game flow has no end node as it is up to the player to decide when they would like to stop playing. This allows the player to play as many laps as they would like to try and achieve the quickest possible lap time. The player can end the game mode through the pause menu of the game.

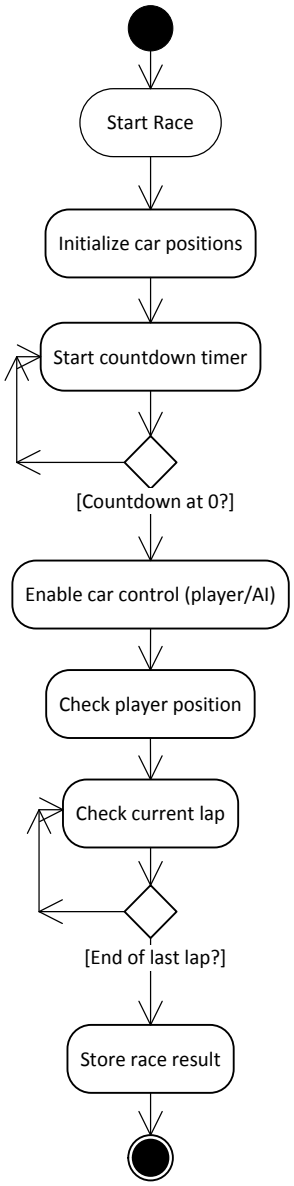


Figure 3.6 Race Mode Game Flow

The race game flow keeps track of the player's position relative to the AI agents and ends the game when the player crosses the finish line at the end of the third and final lap.

3.4 Builds

This section covers the implementation process for each key build component in the game. In addition to the information provided in this section, additional pseudocode for the scripts developed for these builds can be found in Appendix F.

3.4.1 Build 1: Basic Functionality

- B1.1 Literature review and game design.
- B1.2 Unity project stub with content hierarchy.
- B1.3 Basic car control and physics on simple terrain.
- B1.4 Integrate Oculus Rift into player car.
- B1.5 Import race track and ensure car and camera behave as expected.

B1.2 Unity project stub with structured content hierarchy

In order to begin the implementation process it was important to set up a structured Unity project. A project stub was made by creating a folder hierarchy within the *projects* window of Unity. The folders were given short, descriptive names to make it clear where to find specific files within the project (for example the *Scripts* folder contains all scripts used in the game (sub-divided into categories, such as *Controllers* or *Gameplay*).

The 4 month free Unity Pro trial [63] (provided with the Oculus Rift development kit) was also activated to ensure all features of the engine were unlocked and ready to be used.

B1.3 Basic car control and physics on simple terrain

This task comprised the majority of the work in the build 1 phase. Car mechanics are inherently physics-based in their nature. As such it was important to get an understanding of the physics components in Unity before starting the implementation process. Unity natively incorporates the NVIDIA PhysX engine which provides all of the components needed to create physical vehicle behaviour.

The most vital component the engine provides is the *rigidbody* component [64]. Rigidbodies allow objects in the Unity scene to be represented as physically simulated objects. This means that rigidbody objects are susceptible to forces and other objects in the scene. Unity also offers *kinematic* rigidbodies. These are rigidbodies which are not affected by forces and are driven by explicitly specifying positions and rotations of the transform. They can also affect the motion of normal rigidbodies through collisions or joints. As realistic simulation of vehicles is heavily reliant upon forces, a standard rigidbody was applied to the chassis of the car.

The next step was to investigate how to represent the behaviour of the wheels of the car. Unity provides specific colliders for simulating such behaviour through components called *wheel colliders* [65]. Wheel colliders provide many features; collision detection, wheel physics and a tyre friction model. There are many fields within the wheel collider component and it was important to gain an understanding of each individual component in order to achieve the car mechanics required for the game. The first step in setting up a wheel collider was to specify the wheel radius and suspension distance (how far the suspension can extend from the axle).

The next step was to initialize the suspension spring. This consists of three components; the target position, spring and damper values. The target position represents the rest distance along the suspension distance between 0 (full extension) and 1 (full compression). The goal of the suspension spring as a whole is to return the wheel to this target position. The spring value represents the amount

of force exerted by the spring to reach the target position whilst the damper provides damping forces to control the suspension velocity.

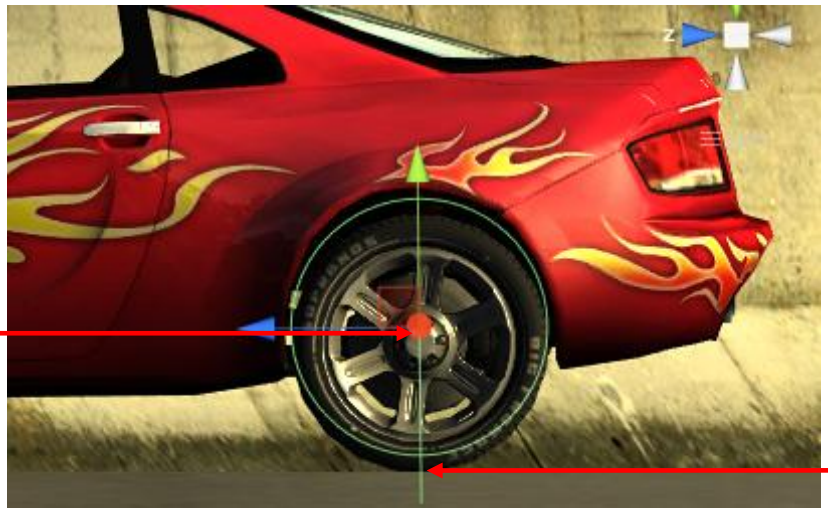


Figure 3.7 Unity Wheel Collider [65]

The final components of the wheel collider are the forward and sideways friction values. These represent the tyre properties when the wheel is rolling forwards and sideways. This is the component that took most time to set up to achieve a realistic feel for the car handling.

After going through the above process of learning about wheel colliders, it became apparent it would be a very time consuming process to manually attach and initialize them for each vehicle in the game (especially when using various car models). As a result, it was decided to initialize them programmatically to make it much easier and quicker to set up cars in a reusable way. The alternative would be to manually create a hierarchy of prefabs for each component which would be time consuming to set up and initialize the collider properties for each wheel.

B1.4 Integrate Oculus Rift into player car

Integrating the Oculus Rift into the Unity project was a relatively painless task. The Oculus Rift software development kit provided a Unity package which was easily imported into the project. This package contained various prefabs [66] (including a camera controller and a character controller) as well as all the scripts and DLLs required to drive the device. A prefab in Unity is a type of reusable game object asset which can be stored in the project. This allows for them to be inserted into scenes multiple times in an efficient and reusable way.

The camera controller prefab was selected. This was because no humanoid character control was required as the player controls the car. The prefab was positioned within the cockpit of the car as a child game object. An option in the Unity inspector window to "*follow Z rotation*" was selected. This enforced the camera to rotate along the Z axis according to the parent game object. This created a realistic effect of turning when the car steered from left to right. Several minor tweaks were made to the near-clipping plane in order not to see through parts of the car and steering wheel.

The game was now Oculus Rift-ready (to a basic extend) and could be played using the device. Further improvements were required (such as restricting complete 360 degrees rotation and accounting for motion blur and sickening effects) and were planned to be tackled at a later stage in the project (see section B2.3).

B1.5 Import race track and ensure car and camera behave as expected

Up until this point the car mechanics had only been tested on a simple, large, flat plane. As such, it was important to create a race track and then ensure the car mechanics were behaving as expected.

The race track was made using the *Race Track Construction Kit* [67], purchased from the Unity Asset Store. This pack provided a set of modular race track components (such as corners and straights) that could be pieced together using the Unity editor to create a race track quickly and efficiently.

Once the track was completed the player car prefab was placed in the scene and tested. This was a successful process apart from one issue with collision detection against the barriers on the side of the track. The geometry of the barriers was relatively complex in nature and was causing the car wheels and front wing to get stuck after a collision. The solution was to scale up the barrier size by four to ensure the wheels and front wing would not get caught between the mesh. Another potential, though more time consuming solution, would have been to replace all of the mesh colliders with simpler box colliders. This would create a simple, flat collision. However this would raise issues when trying to apply a box collider to a curved segment of the track.

3.4.2 Build 2: Core Game Features

- B2.1 Improve car control physics to achieve higher level of realism.
- B2.2 Create AI car control system.
- B2.3 Integrate head tracking features of Oculus Rift to simulate realistic cockpit.
- B2.4 Game modes (race and time trial).
- B2.5 Implement dashboard and heads-up display.

B2.1 Improve car control physics to achieve higher level of realism

The vast majority of the time spend on this requirement was involved in gaining a deeper understanding of the tyre properties and wheel collider implementation in Unity. Many different variations of tyre were tested until the desired handling and car physics were achieved. One further addition to improve the handling was to implement an anti-roll bar to the car. The purpose of an anti roll bar is to provide stability along each of the axles in the vehicle. This is achieved by transferring part of the compression force that may occur on one wheel to the other wheel, limiting the roll of the car's body. This was implemented by retrieving the collision data between the wheel and the terrain and then calculating the suspension distance and compression. This value was then subtracted from the opposing wheels value and multiplied by an anti roll multiplier value. This value was then applied as a force to each wheel on the axle, simulating the effect of an anti roll bar. The implementation was aided and inspired by the Unity community forums [68].

B2.2 Create AI car control system

A great deal of research took place in order to determine an appropriate technique to handle the NPC car control in the game as outlined in the literature review (section 2.1). The ultimate decision was to implement a reinforcement learning algorithm. The primary reason for choosing this method were that it created a reusable AI framework for the car controller that could be learnt for any race track. This meant that the car did not need to be hard coded for each track and additional race tracks could be added easily without the need to re-code the AI system.

B2.2.1 Reinforcement Learning (Q Learning)

As discussed in section 2.1.4, reinforcement learning algorithms typically use an episodic approach (iterative) that gradually constructs a solution (a *policy*) to the problem over time.

The first challenge was deciding how to represent the game world to the algorithm. It was decided at an early stage to use a pre-defined "racing line" [69] to control the NPC car steering as this would provide the most realistic movement of the car around the track whilst reducing the overall state-space for the algorithm. This meant that the only values that needed to be learnt were the appropriate throttle values along the racing line.

The racing line was produced by placing waypoints at regular intervals along the track where the racing line should be. The racing line was positioned to take ensure the line passed the inside apex of each corner and sweep to the outside of the track on straights and exits of corners just like a real racing line. A script was then written to generate a Catmull-Rom Spline [70] to interpolate these points and generate more waypoints along the track thereby creating a racing line.

Box colliders were generated at each point along the racing line to check for collisions with the NPC cars. Initially, collision checking was performed manually through a simple script comparing positions and vectors. However, this proved to be ineffective when the cars were at a higher speed as some collisions were missed. As such, the box colliders were chosen to utilize Unity's in-built continuous collision detection [71] to ensure all collisions were tracked. Figure 3.8 shows a top-down view of the race track with the racing line and box colliders at each state.

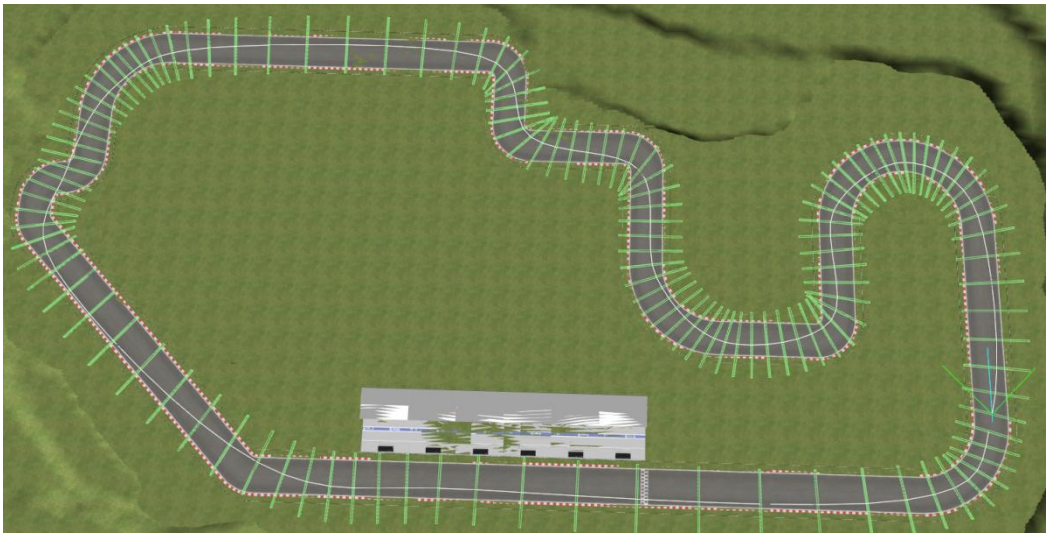


Figure 3.8 Race Track with Spline and Colliders

Before implementing the Q Learning algorithm and class, it was important to establish the data structures that would be required. This was one of the most challenging aspects of the implementation as it was difficult to understand what data was needed to be stored and how to retrieve and use it efficiently and effectively. Two additional classes were implemented; the *QStore* and *QActions* classes. The QStore is responsible for storing the Q value for each state/action combination and the action with the lowest delta time between states. It also has public methods for retrieving various data and information, such as *GetUntriedAction* (returns an action with a Q value of 0) and *GetBestAction* (returns best evaluated action for current state). The QActions class simply stores and initializes an array of possible throttle actions that can be taken by the car.

In addition to these data structures, a simple car percept class was developed. This percept simply performed a simple ray cast test to determine whether it was on or off the track which was necessary in order for the reward function to assign appropriate values.

Once all the above had been established, it was a relatively straight-forward process to implement the algorithm itself.

The first step was to understand the underlying Q learning formula (equation 3.1).

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'}(Q(s', a')))$$

Equation 3.1 Q Learning Formula

Q(s, a) – Q value of the current state-action pair
Q(s', a') – Q value of the next state-action pair
r – reward value associated with next state
α – learning rate parameter
γ – discount value parameter

Equation 3.2 Q Learning Formula Components

At first glance, the formula looks more complicated than it really is. When the parts are broken down as in equation 3.2, it became clearer to understand and in turn to implement the methods required to retrieve the necessary data from the QStore.

The learning rate and discount parameter values required some tweaking (by modifying the value between 0 and 1) and testing to achieve the desired learning effect. The learning rate value (α) determines to what extent the newly acquired information will override the old information [8]. A rate of 0 means that the agent will not learn anything, whilst a rate of 1 would make the agent only consider the most recent data acquired. The discount parameter (γ) determines the importance of future rewards [8]. A factor of 0 makes the agent short-sighted by only considering current rewards, whilst a factor of 1 makes it strive for a higher long-term reward.

Two versions of the learning algorithm were implemented. The first version was initiated by setting the maximum state to the next state from the current state. It would then try every single throttle action between these two states. Once all actions had been attempted, the best of these actions was saved and used for the rest of the learning process. The maximum state was then incremented, and the process repeated until all actions had been evaluated for each state. This approach ensured a finite set of episodes could be used to learn the race track. Algorithm 3.1 outlines a high level view of this implementation. A video of the implementation can be found at: http://www.youtube.com/watch?v=rHs_JNVnmp8&feature=share&list=PLyqnh5008wxDA7NhQAhgtCA8wbQOwajHV.

```
QLearning_V1()
-Episodes = number of states * number of actions
-For each episode:
       -Reset car position/rotation to start
       -State = 0, NextState = State + 1
       -MaxState = highest state that not all actions have been evaluated
                   for

       -For each state below MaxState:
              -If tried all actions for state, use best action
              -Else select an untried action

              -While not at next state
                     -Apply action
                     -If crash, calculate and store negative reward and
                      update Q value for state-action, go to next episode

              -If at next state:
                     -If at MaxState:
                            -Calculate and store positive reward and
                             update Q value for state-action
                     -State++
```

Algorithm 3.1 Q Learning Pseudocode – Version 1

A second version of the algorithm was implemented using a more traditional reinforcement learning approach. Instead of analysing each action for each state individually, the car was simply set to run as far as possible without crashing or deviating too far from the racing line. This meant that the number of episodes required to learn the throttle values was indefinite and as such a much larger number of episodes were executed in order to achieve an appropriate policy. A video of this version can be found at: http://www.youtube.com/watch?v=abXJmdej3EQ&feature=share&list=PLyqnh5008wxDA7NhQAhgtCA8wbQOwajHV.

```
QLearning_V2()
-Episodes = 5000
-For each episode:
       -Reset car position/rotation to start
       -State = 0, NextState = State + 1

       -For each state below StateCount:
              -Get non-negative action for state

              -While not at next state
                     -Apply action
                     -If crash, calculate and store negative reward and
                      update Q value for state-action, go to next episode

              -If at next state:
                     -If at EndState:
                            -Calculate and store positive reward *
                            ReachedEndReward value and update Q value for
                            state-action pair
                     -Else:
                            - Calculate and store Q score
                     -State++
```

Algorithm 3.2 Q Learning Pseudocode – Version 2

In order to perform the learning process effectively, the rendering process in Unity had to be disabled. This was achieved by disabling the cameras on the car and closing any graphical windows (such as the Scene window) in the engine. The process also had to take place in a fixed time step function to ensure the code could be executed at 100 times the normal simulation speed whilst continuing to maintain the desired simulation.

The reward function was relatively straight forward to implement. The value of the reward was primarily based on a simple Boolean check; was the move good or bad (ie did it result in a crash). If the move was bad, a negative value was assigned (scaled by the distance of the vehicle from the racing line). There were a few additional factors to consider when calculating a good move. The proximity to the racing line as well as the time taken to get to the next state were multipliers to the reward value. For example, if the car stuck to the racing line and went from state a to state b in a shorter time than a previous attempt, it would receive a higher multiplier and thus a higher reward value. Additionally, a very large multiplier was rewarded if the evaluated state was the goal state. The reward function for the second version was initially much simpler, only rewarding a negative reward if crashing or going off the track and a positive reward for reaching the goal state. However this resulted in the AI not following the racing line as closely and taking a larger number of episodes to be trained. As a result the reward function was reverted to the original method as described above.

The learned policy was stored in a text file once all of the assigned episodes had been completed. This was necessary to allow the policy to be loaded and used from the game in real-time without having to relearn the track. The text file simply consisted of each of the throttle values for each state along the racing line. A simplified adaptation of the learning function was created in order to execute these stored throttle values in the game (without Q learning and calculating rewards).

B2.2.2 Obstacle Avoidance

Obstacle avoidance was the next major hurdle to overcome after implementing the reinforcement learning algorithms. The method implemented was based on Buckland's version in *Programming Game AI by Example* [16].

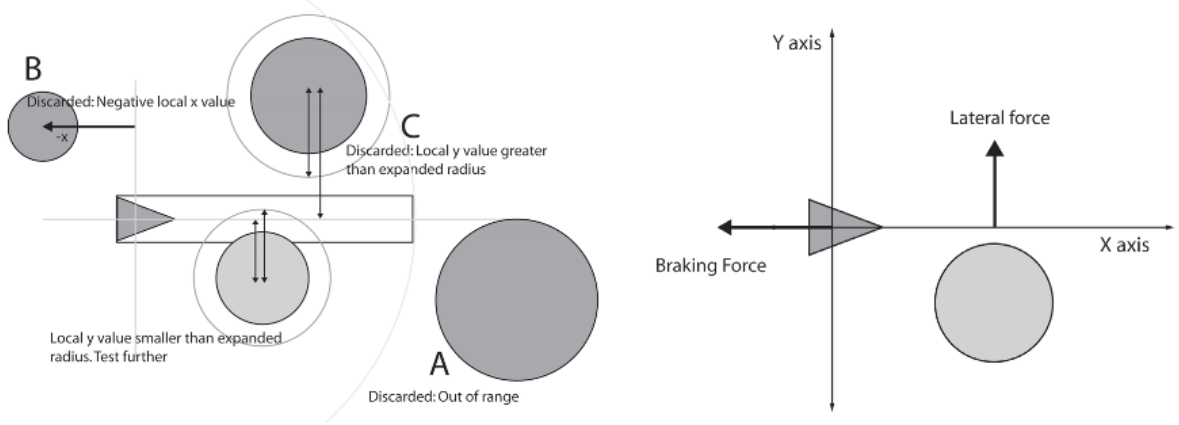


Figure 3.9 Obstacle Avoidance Technique by Buckland [16]

The algorithm described by Buckland is simple yet effective in achieving the desired effect. It works by detecting the nearest obstacle (if any) within a certain range in front of the vehicle. If the obstacle is on target to intersect with the vehicle, an avoidance force is calculated based on the vehicles current velocity and the proximity to the obstacle. This is outlined in further detail by the pseudocode in algorithm 3.3.

```
ObstacleAvoidance()
-ObstacleArray = new array of obstacles
-DetectionBox = MinBoxLength scaled by car velocity

-Detect obstacles within box range:
       -For each car in the scene:
              -If difference in car positions < DetectionBoxLength
                     -Add to ObstacleArray

-For each obstacle in ObstacleArray:
       -If obstacle is behind, discard
       -Calculate expanded radius
       -If obstacle's local X value is within expanded radius
              -Perform line/circle intersection test
       -If intersection point < distance to closest obstacle
              -Distance to closest obstacle = intersection point

-Force multiplier = multiplier based on velocity and proximity to obstacle

-Lateral avoidance force = closest obstacle radius – local x position of
closest obstacle * multiplier

-return avoidance force
```

Algorithm 3.3 Obstacle Avoidance Pseudocode (adapted from Buckland's implementation)

Once the avoidance vector had been calculated and scaled, the value was simply added to the pre-existing steering method (steering based on the spline position). This meant that whilst avoiding obstacles it would still be steered based on the racing line and track layout. This implementation produced a controller which reacted slightly differently in various situations. When driving directly towards an obstacle from a reasonable distance the agent performed a smooth and realistic avoidance behaviour. When the obstacle was detected directly in front of the agent at a very short distance (for example after a sharp corner) the reaction was much more erratic, unrealistic and extreme in nature. It also sometimes created an unrealistic avoidance behaviours when the agent was set to narrowly avoid an obstacle. This was improved upon by creating a tighter bounding radius around the obstacles and down-scaling the avoidance force.

B2.2.3 Wall Avoidance

The gameplay was significantly improved after the obstacle avoidance had been integrated into the game. However, there were still some minor issues. Whilst the AI car would not go off the track while executing the learnt policy, the inclusion of obstacle avoidance meant that this was now a possibility. As such, wall avoidance was implemented.

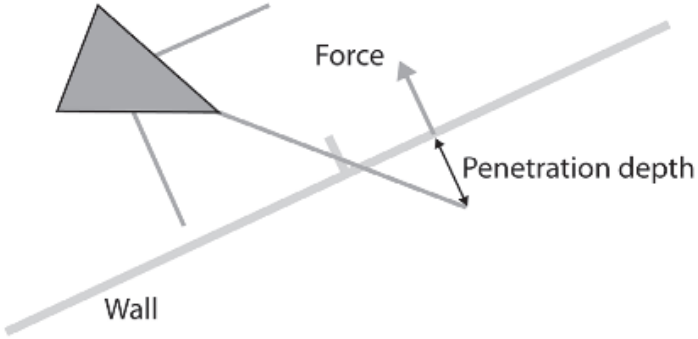


Figure 3.10 Wall Avoidance by Buckland [16]

The first task was to update and improve the car percept class to retrieve the additional necessary visual sensory data for the car. This was achieved by ray casting along a fan within a specified field of view and viewing distance. The tags (identifiers) [72] of the objects were then checked (if any were hit) to check whether the object that was hit was a wall or not.

Now that this sensory data was available, it was possible to tackle the issue of wall avoidance. As with the obstacle avoidance, the implementation was the version provided by Buckland [16] (ported from C++ to a C# script).

This provided a realistic overtaking effect when contending with a single obstacle. However, multiple obstacles caused this method to become less effective. Therefore a second phase was introduced using vector mathematics to content with multiple obstacles.

B2.2.4 AI Summary

The three parts of the AI were combined to form a single controller. The policy provided from the reinforcement learning algorithm essentially acted as a seek function, which was easily combined with the obstacle avoidance and wall avoidance functions. This created a fluid and realistic AI controller for the game. Figure 3.11 shows the logic flow of how the complete AI controller operates. Given more time, the AI could have been improved to create an ever more realistic controller. These potential improvements are discussed in section 6.7.

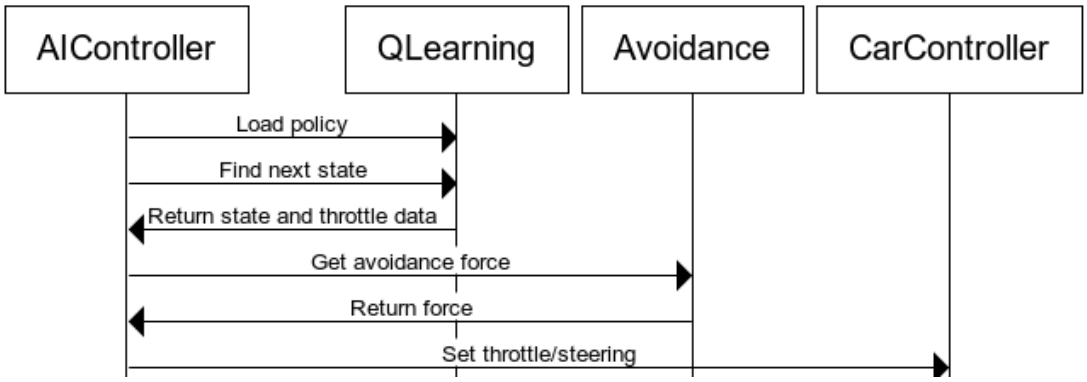


Figure 3.11 Sequence diagram of AI controller

B2.3 Integrate head tracking features of Oculus Rift to simulate realistic cockpit

This requirement took significantly less time than anticipated to implement. As mentioned in section B1.4, the Oculus Rift development kit provided a prefab which handled all of the head tracking and rendering functions. The time spent on this requirement was largely focused on tweaking the near clipping planes to ensure it felt natural in the car (see figure 3.12). Time was also taken to ensure the camera positions were as good as they could be to provide the most realistic feel of sitting in the car, whilst still being able to see the steering wheel dashboard and wing mirrors.

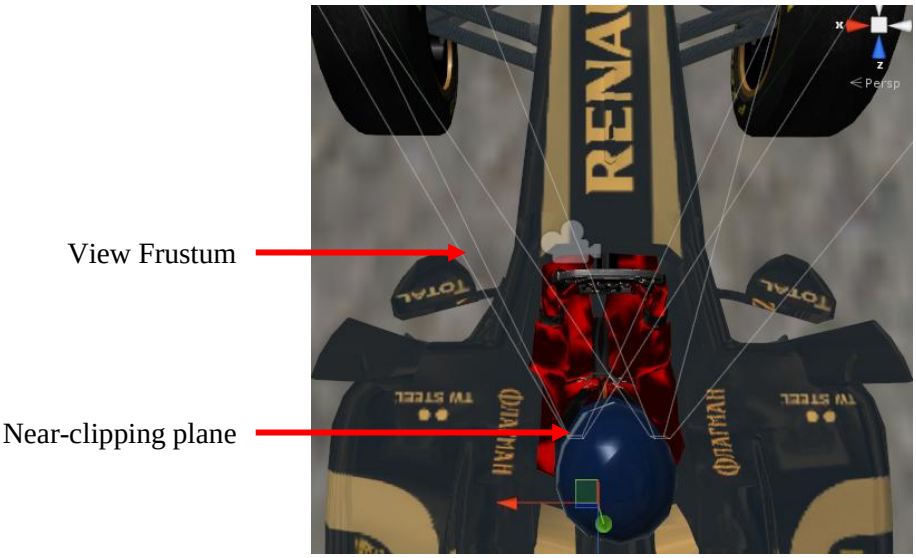


Figure 3.12 View Frustums of Oculus Rift Cameras

Another focal point of this section was to restrict the overall rotation capabilities of the Oculus Rift to simulate the feeling of being strapped in the car (ie so the player could not look directly behind themselves to see through the back of the car). Unfortunately, this was not possible to implement as the low-level rotation code for the Rift was implemented in a secure DLL file which was inaccessible to modify. The Oculus Rift community forums [73] was consulted on this issue and a developer at Oculus communicated that they are working on providing access to these parameters in future versions of the SDK. Unfortunately, this was not available before the end of the project.

B2.4 Game modes; Time Trial and Race

B2.4.1 Time Trial Mode

The time trial mode was a relatively straight forward mode to implement. The main requirement was to capture the time taken for the player to complete a lap from the start to finish. This was accomplished by creating three checkpoints within the race track (using box colliders).

A simple script was attached to the player car to keep track of how many checkpoints the car had passed through. A timer was triggered when the player car passed through the first checkpoint and the overall lap time was stored once the car passed through all three checkpoints and the timer was reset.

Another script was attached to the car to enable a rolling start when first entering the game mode. The player car spawns at the opposite end of the main straight on the track and accelerates towards the finish line. The car is aimed towards the first corner and the player's controls are disabled. Once the car crosses the start/finish line, the controls are activated and the player is then in complete control of the car.

B2.4.2 Race Mode

Unlike the time trial mode, race mode required multiple AI cars to be positioned in the scene with the player. As such, the first step was to position the cars on the racing grid and then have a countdown timer to begin the race. The grid positions were set by positioning simple waypoints on the race track in the Unity editor. A script was then written to spawn the cars randomly on these grid positions. Once spawned, the car handbrakes were enabled to prevent them from driving off before the start of the race. A countdown timer of five seconds was set up to allow for a racing start.

The main objective of the race mode is to achieve the highest possible position before the end of the three laps. As such, it was necessary to create a function to track and return the player position at any given time. This was achieved by creating a series of simple tests to determine whether the player was ahead or behind of another car. The pseudocode for this function is outlined in algorithm 3.4.

```
GetPlayerPosition()
-Position = number of cars
-For each AI car:
       -Does (AI lap number == player lap number)?
       -If no:
              -If (AI lap number > player lap number) player ahead, position--
              -If (AI lap number < player lap number) player behind, position++
       -If yes:
       -Does (AI last checkpoint == player last checkpoint)
              -If no:
                     -If (AI checkpoint > player checkpoint) player ahead,
                            position--
                     -If (AI checkpoint < player checkpoint) player behind
                            position++
              -If yes:
              -Does (AI current waypoint == player current waypoint)
                     -If no:
                            -If (AI checkpoint > player checkpoint) player ahead,
                                   position--
                            -If (AI checkpoint < player checkpoint) player behind,
                                   position++
                     -If yes:
                            -If (AI dist to waypoint > player dist to waypoint)
                                   player ahead, position--
                            -If (AI dist to waypoint < player dist to waypoint)
                                   player behind, position++
-Return Position
```

Algorithm 3.4 Position Tracking Pseudocode

B2.5 Implement Dashboard and Heads-Up-Display (HUD)

As a large focus of the project was based around the Oculus Rift integration, it was important to provide as much information as possible on the dashboard to increase the level of immersion.

The first dashboard element implemented was the gear shift lights. This was implemented using the Lens Flare effects provided by the Unity engine [74]. The aim was to mimic the dashboard of a real Formula 1 car. This meant having three sets of different coloured lights (five lights per set) to depict the proximity to the cars rev limiter. The first set of lights were triggered when reaching 80% of the rev limit in the current gear. The second set were triggered after reaching 90% and the final set were triggered after reaching 95% of the rev limit in the current gear. The lights were positioned behind the

steering wheel on the car chassis (such as in the real Red Bull Racing F1 car). Once this had been implemented, the priority was to display as much information as possible on the car steering wheel. Due to the size restrictions of the wheel, only three pieces of information could be displayed; the current lap number, current gear, and the current speed of the car. This information was retrieved from their relevant classes (ie speed and gear from the car controller class). Several planes were positioned just above the steering wheel (along the z-axis); three planes for the lap time (ie L01), one plane for the gear (ie 6) and three planes for the speed (ie 105). Textures were made for each number (0-9) and other necessary characters to be displayed on the dashboard. A script was then written to render the appropriate texture to each plane based on their current values. Figure 3.13 shows the textures being rendered to the planes on the steering wheel as well as the gear shift lights behind the wheel.

Figure 3.13 Car Dashboard and Gear Shift Lights

Whilst this provided the majority of the information to the player, there was still information that needed to be conveyed (ie the player's position in Race Mode, and fastest lap time in Time Trial Mode). These were displayed in the corner of the screen using the *OnGUI* function provided by Unity.

This worked fine when testing without the Rift, but when testing with the device activated the text did not appear. After consulting the Oculus Rift community forums, it was revealed that the Oculus Rift post-processing effect (of manipulating the image to the lenses in the device) ignores the Unity GUI layer. Consequently it was necessary to render this information to a plane in three dimensional space as had been done with the dashboard information. These planes had to be attached to each Rift camera object to ensure they rotated correctly with the device. It took some testing and adjusting to ensure the text was at an appropriate position and distance from the camera to allow the player to read it easily whilst not being too distracted from the game world. One further issue that was encountered was the planes casting shadows on one another when at certain viewing angles. This was resolved by adding a tag to each plane and only rendering the appropriate plane for its specific camera.

3.4.3 Build 3: Further Enhancements

- B3.1 Weather effects to vary car behaviour
- B3.2 Modify AI into difficulty levels
- B3.3 Shaders and visual effects
- B3.4 Leaderboard table

B3.1 Weather effects to vary car behaviour

Up until this point, the game took place solely on a dry race track on a clear day. A rain weather effect was implemented in the game. This was achieved by attaching a rain particle effect above the player car. A wet surface and puddle shader was then applied to the track surface. Due to the time constraints on the project, the shader and rain particle effects used were taken from a third party source and applied to the game. The particle effect and shader used are referenced in appendix F.13 [75][76]. Creating a realistic wet surface shader would be very time consuming and at this stage in the project it was decided that it was more important to focus on implementing as many specification points as possible.

It was important to alter the behaviour and handling of the car controller once the visual effects were in place. The wheel friction curve of the car controller was then modified to reduce friction, creating the sensation of driving on a slippery surface. This was achieved by lowering the asymptote and extremum values to make it easier for the car to skid at high speeds. These values are responsible for controlling the forwards and sideways friction for the tyres based on the rpm.

B3.2 Modify AI into difficulty levels

Due to the limited timescale of the project, this milestone was not implemented to its fullest. The original plan was to use an FSM architecture to create various levels of aggressiveness in the racing style of the AI. This would involve modifying the rewards function of the reinforcement learning algorithm by setting different levels of rewards for lap times, risk of crashing and proximity to the racing line. However as the base reinforcement learning implementation took up a large part of development it was not possible to implement this system.

However, the two different reinforcement learning versions provided two slightly different policies for the AI cars to use. This gave the AI cars slightly different behaviour. The version of the policy used by the AI cars is randomly determined upon loading the level to create more variety in the gameplay. It would also be possible to learn multiple policies as the algorithms rarely produced the same policy each time. The varying episodes used for the second version of the algorithm could also be used to create a novice and more advanced AI.

B3.3 Shaders and Visual Effects

The first shader effect implemented was a cubemap reflection on the body of the car. The car model used a diffuse shader by default which provided a very matted, unrealistic finish to the car. Applying Unity's specular shader helped to improve the surface finish but it still looked unrealistic. As such, it seemed logical to implement a reflection shader to increase the realism of the cars appearance. Figure 3.14 shows a comparison between the three shaders applied to the car. The shader used was taken from the official Unity documentation [77].

Figure 3.14 Diffuse, Specular and Cubemap Reflection Shaders

This shader was implemented using the Cg shader language [78] wrapped within Shaderlab [79] (as required by Unity). This shader takes the car texture and cubemap of the skybox as inputs. The shader also uses the Lambertian reflectance model [80]. The shader then uses the inputs to set the diffuse (*albedo*) and emission values of the surface. The diffuse term simply uses the RGB values of the car texture at each fragment of the surface. The emission term uses the cubemap and its relative position in world coordinates to create the reflectance effect. This effect is scaled down by 20% to achieve a more realistic reflectance effect on the car body.

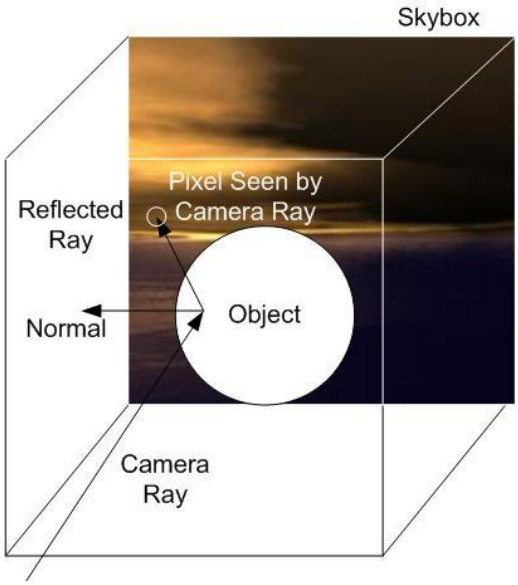


Figure 3.15 Cube Map Reflection [81]

A further shader was utilised to create the wing mirrors of the car. This was implemented using the official Unity wiki page which provided a method of how to achieve the desired effect [82]. This provided an efficient way of creating wing mirrors with no drops in frame rate. Prior to this, the effect was created by using a camera positioned at each wing mirror and rendering the camera's view to a texture which was rendered on a plane in the wing mirror positions. However, this caused the game to have a lower frame rate than before and a stuttering effect when there were moving obstacles in view.

A vignette effect to simulate a helmet was also incorporated in the game. This was relatively straight forward to add into the game, particularly after the lessons learned from creating the HUD (section B2.5). For the regular game camera, this was implemented in the *OnGUI* layer. However, as with the HUD text, the vignette had to be drawn on two planes (one for each camera) when using the Rift cameras. They were attached as child objects to each camera to ensure they rotated with the players head rotation.

It would have been interesting to implement a glass shader to simulate looking through a real visor as opposed to a vignette effect. However this would increase the computational expense drastically and cause frame rate drops if the player moved their head around quickly (especially as it would have to be calculated for each eye).

B3.4 Leaderboard Table

This feature was added in the latter stages of the project to improve the overall quality of the game. The leaderboard table was designed to store and display the top five fastest lap times around the circuit in Time Trial mode and the best position achieved in Race mode.

This was achieved by creating an XML file to easily store and load the data. The file was created upon loading the game (unless it existed already) and loaded upon entering the Leaderboard menu screen. The data for both modes was updated (if necessary) upon quitting or finishing the level.

B1.6, B2.6, B3.5 Testing and Refactoring

A testing and refactoring phase took place after each build of the project. This helped to iron out any underlying bugs that may have existed and improved the readability and reusability of the codebase. Due to the project timescale, these phases were not as long as would have been desired but none the less helped to improve the quality of the project.

4. Results

This section presents the results of the project. This is focused on presenting results from the three core areas of the project; the system design, Q-learning for AI and the Oculus Rift integration.

4.1 System Design

As discussed in the literature review (section 2.4), the system was designed and implemented in the most reusable and modular way possible (following the MVC architecture). This was not fully realised due to the necessity of using *MonoBehaviour* classes (as discussed in section 5.3.1) which is required to use many of the features of the Unity engine (for example accessing the *transform* properties of a game object). Despite this, it was still possible to create modular and reusable code as shown by following class diagrams. Sections 4.1.1, 4.1.2 and 4.1.3 provide high level class diagrams for each major build. Section 4.1.4 provides further detail on the design on some of the core components of the project.

4.1.1 Build 1

Build 1 provided a simple foundation on which the rest of the project could be build upon. This meant using a single test scene consisting of the race track with a car controlled by the player. It also introduced a camera controller which could toggle between the regular camera and the Oculus Rift cameras.

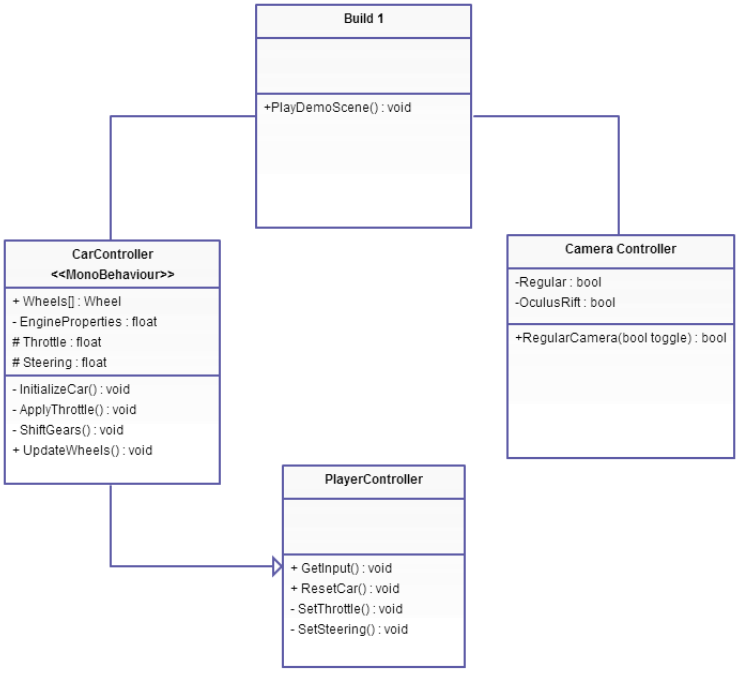


Figure 4.1 Build 1 Class Diagram

4.1.2 Build 2

Build 2 brought a multitude of new and important features to the game, notably the AI, game modes and HUD and dashboard information. These improvements are reflected in figure 4.2.

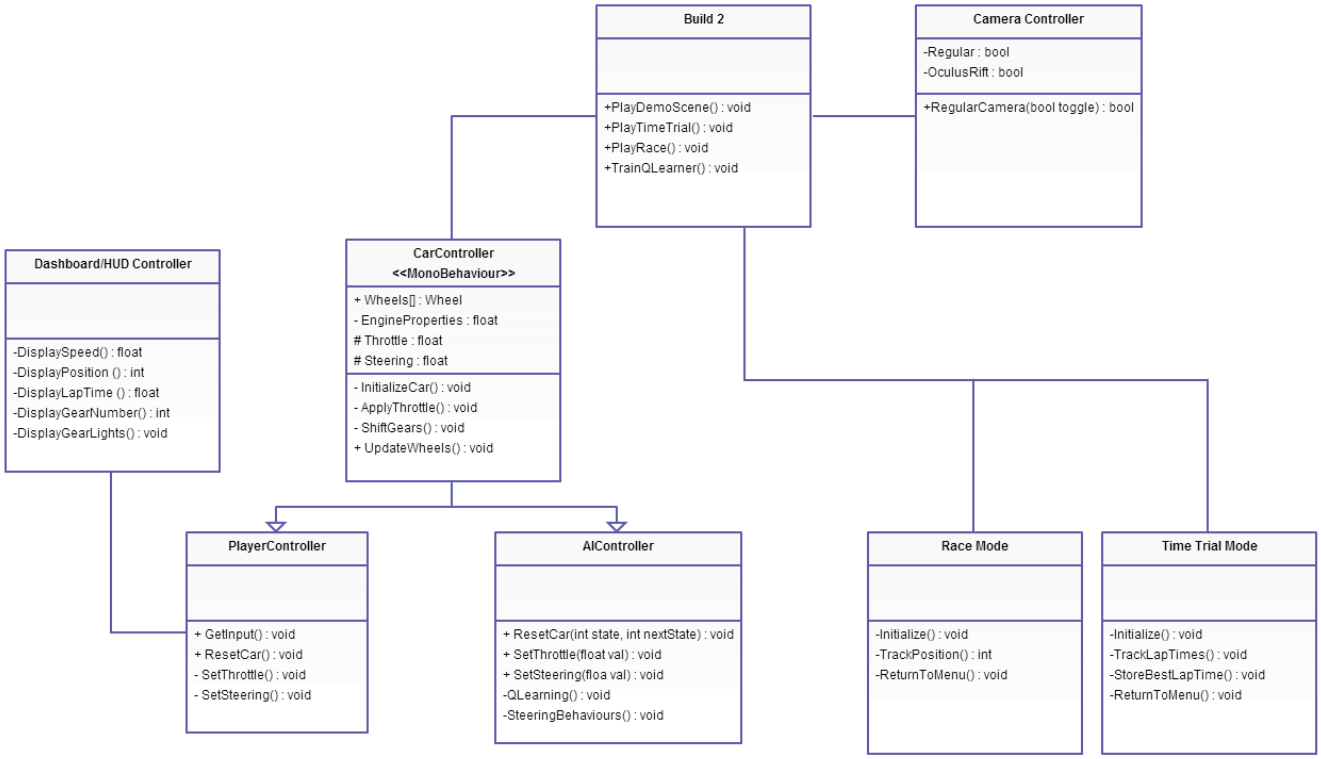


Figure 4.2 Build 2 Class Diagram

4.1.3 Build 3

Build 3 introduced subtle features to the game. The main gameplay related feature implemented as the wet weather mode. This modified the tyre properties in the car controllers as well as adding visual effects. The other features introduced in this build cannot be demonstrated in a class diagram as they are visual effects (such as the car body reflection shader) and variations on the AI policy of which the controller was implemented in build 2.

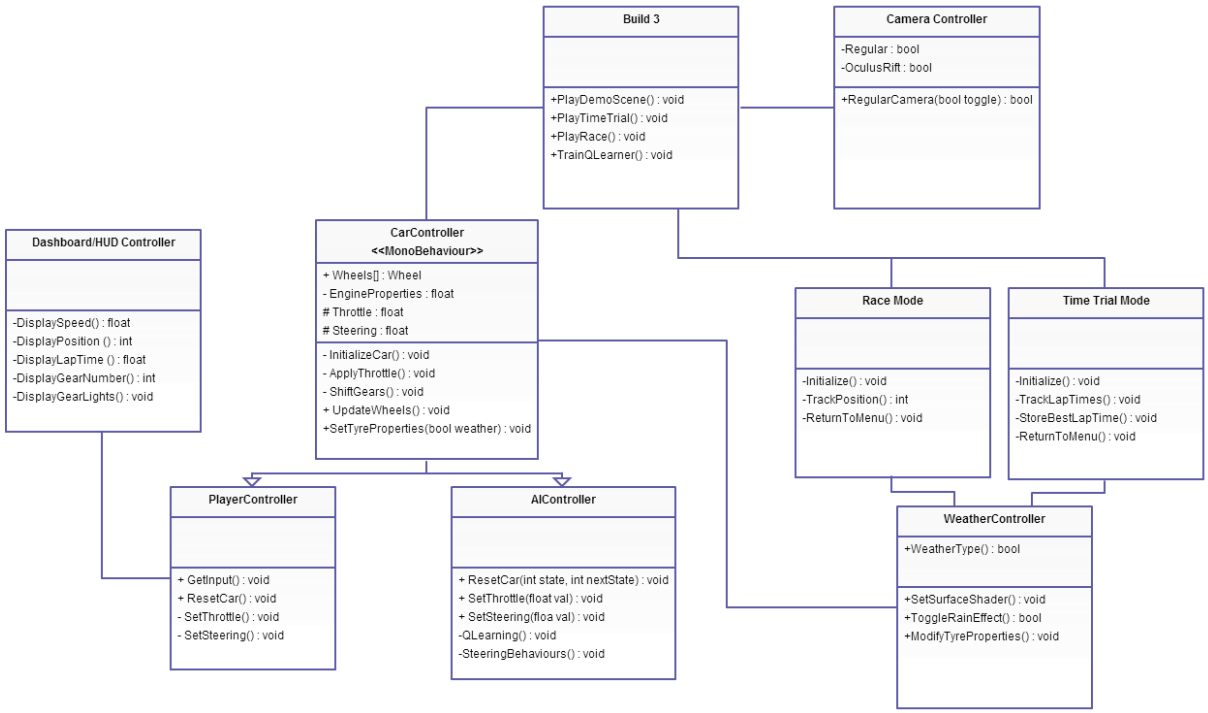


Figure 4.3 Build 3 Class Diagram

4.1.4 Core Components

The *CarController* class acts as the base class for both the player and AI agent cars. This contains the core code required to instantiate and operate a car object. It creates the wheel colliders and has methods to update the throttle and steering values. It also updates the engines RPM and gears accordingly. The *PlayerController* is a simple class that updates the steering and throttle values based on the player's inputs. The *AIController* (*SteeringBehaviours.cs*) is slightly more complicated as it receives inputs from the Q-learning and Avoidance classes, as well as receiving data from the *CarPercept* class.

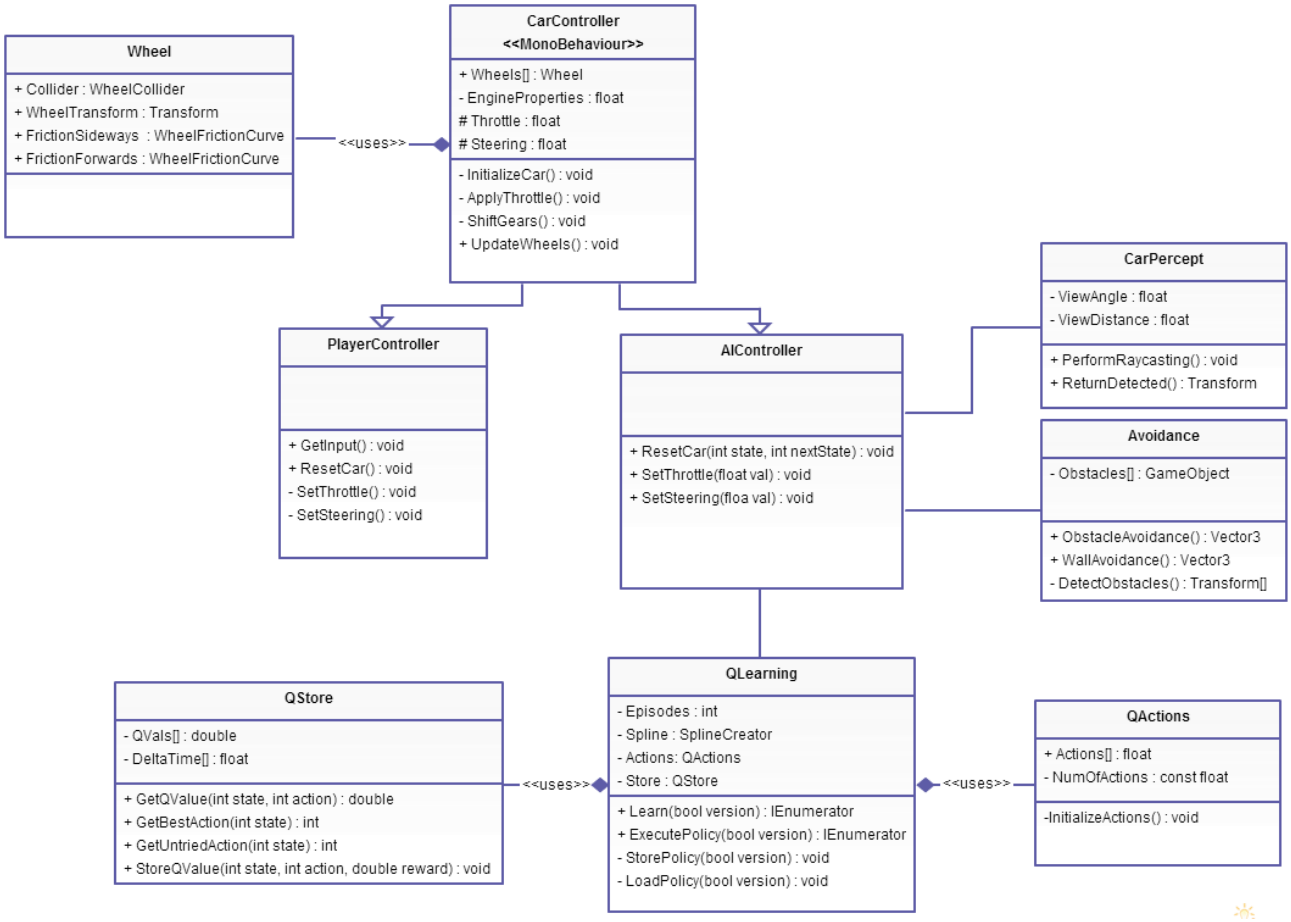


Figure 4.4 Car Control Class Diagram

The *QLearning* class provided the methods required to learn (using each version of the Q-learning algorithm), store, and execute a policy. It also contained instances of the *QActions* and *QStore* classes (the data structures required to perform the learning process). This was also a MonoBehaviour class to make use of the *FixedUpdate* and *IEnumerator* functions (to ensure the learning and execution methods operated at a consistent frame rate).

The *SceneManager* class is a singleton class (based on the *singleton pattern*). The singleton pattern restricts the instantiation of a class to a single object which exists throughout the execution of the game. This meant that it was essential to reduce the number of variables and functions within the class to reduce the constant amount of memory that was allocated. Its primary task is to instantiate and controls the *FiniteStateMachine* class to manage the game states and scene transitions. Use of the singleton pattern has its pros and cons. The main benefit is that it allows for core game data to be obtained across scenes from any class. In addition it means that the memory for the variables and

functions is only allocated once. The main disadvantage of the use of a singleton is that multiple instances cannot be created. This would be an issue of the game was a networked multiplayer game. A solution to this and a general alternative to using a singleton would be to use static classes.

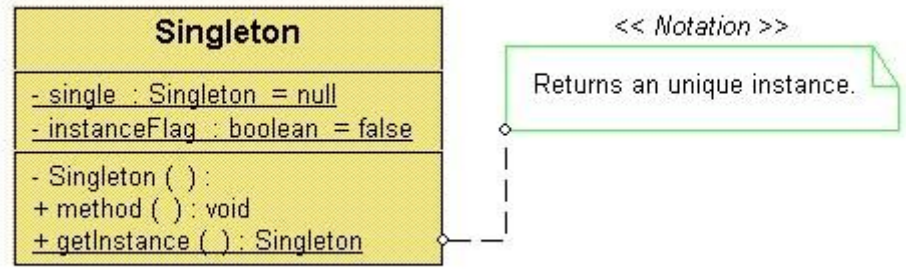


Figure 4.5 Singleton Pattern [83]

The *state pattern* was used in the implementation of the screen flow system in the game. The state pattern encapsulates varying behaviour based on the state of an object. It is a good way to alter behaviour of systems at run-time as opposed to large conditional statements.

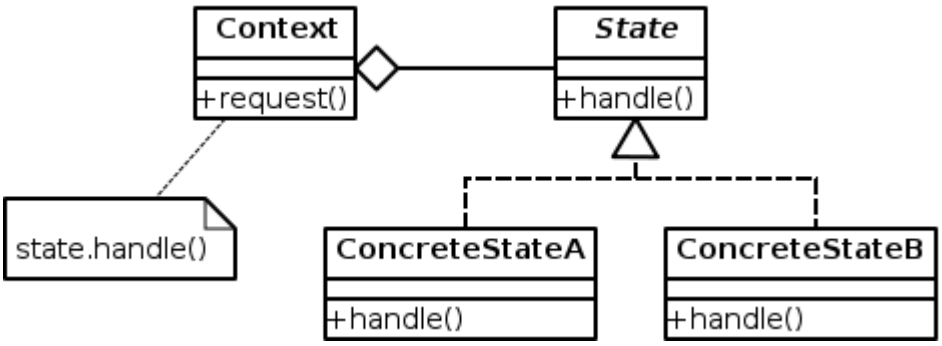


Figure 4.6 State Pattern [84]

The menu flow for the game was implemented using the state pattern through finite state machines (FSMs). Implementing an FSM infrastructure in the game engine helped to provide a reusable and concise method of controlling the game state. This was a logical decision to make as helped to make the code more reusable and also more readable. Figure 4.6 shows the structure of the state pattern. The *Context* class represents the object whose behaviour will be modified by the state change, the *State* defines the behaviour of a particular state and the *Concrete States* implement specific behaviour associated with a state. In terms of the project, the *context* was the game screen flow. The *state* object represents the interface which is in place to transition between states. The *concrete states* were each of the various screens required, for example the main menu, playing and pause screens.

The FSM system was implemented through three core classes: *FiniteStateMachine*, *State* and *Transition*. The state class is an abstract class which defines the properties, data structures and methods that all derived states must implement. The class defines methods for entering, exiting and executing (updating) the state. It also declares a *List* of Transition objects. The Transition class provides the functionality to allow switching between states. This is made up of a state object to transition from one state to another and a delegate which checks whether a condition has been met. The FiniteStateMachine class provides the interface to allow states to be added and updated. The class

contains a *List* of all possible states and a member representing the object that owns the FSM (to allow for multiple FSMs with separate states).

The final design pattern used in the game was the *observer pattern*. This pattern is often useful for implementing features that are based on events (such as player input or collision detection). The Unity engine utilised this pattern for the input detection system and collision detection system and as such it was not necessary to implement this pattern by hand. However, it was important to appreciate how the pattern functioned in order to obtain a full understanding of the engine. The pattern has a subject (for example a collision manager object) and a list of observers (any collisions that have occurred in the game). This provides a modular and reusable way to handle events which can then be easily handled and managed.

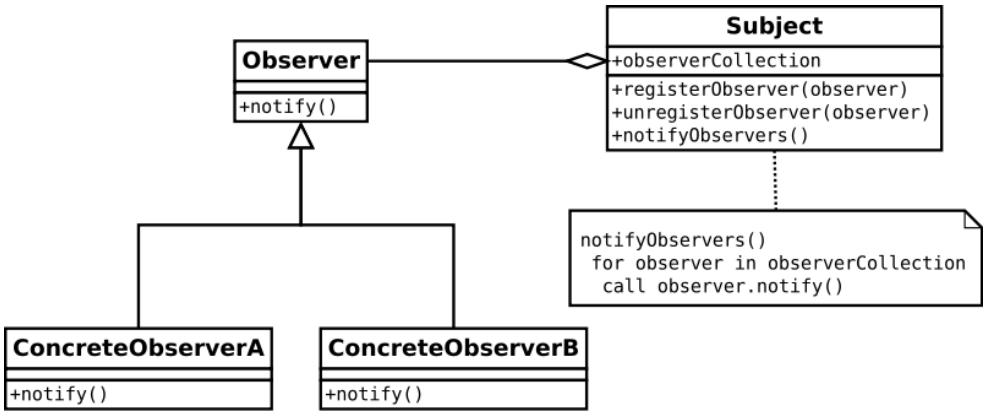


Figure 4.7 Observer Pattern [85]

The rest of the classes implemented were a mixture of MonoBehaviour and Non-MonoBehaviour classes. These were largely focused on gameplay features, animations/effects and menu functions.

4.2 Q-Learning Implementations

Two different versions of the Q-learning algorithm were implemented as mentioned in section 3.4.2. Version 1 of the implementation was the iterative approach of testing each possible action-state pair before moving on to the next state. Version 2 was a more traditional implementation which allowed the car to drive and learn until a crash occurred (or if it reached the goal state). These two contrasting versions allow for a varied data set which can be compared and analysed.

4.2.1 State-Action Tables

Table 4.1 shows the action table used for the learning algorithm (in the *QActions* class). These actions were based on modifying the cars throttle values while the steering was based on the racing line. Tables 4.2 and 4.3 show the State-Action tables for states 93 and 94. These states represent section of the race track shown in figure 4.8 (one of the tightest corners on the track). The tables show the Q values calculated by each algorithm for each state-action combination. See appendix G for further comparisons between tables.

Action	Throttle Value
0	1.0
1	0.75
2	0.5
3	0.25
4	0.0
5	-0.25
6	-0.5
7	-0.75
8	-1.0

Table 4.1 Action Table

State	Action	Q Value
93	0	255963350.08605
93	1	255963350.08605
93	2	255963350.08605
93	3	255963350.08605
93	4	255963350.08605
93	5	255963350.08605
93	***6***	***2805597255.12183***
93	7	2573942438.35473
93	8	2657868839.60533
94	***0***	***2920734984.09786***
94	1	2920734972.65661
94	2	2920734845.53158
94	3	2920733433.0312
94	4	2920717738.5824
94	5	2920543355.81592
94	6	2918605769.49921
94	7	2897077032.39744
94	8	2657868839.60533

Table 4.2 State-Action Table - Version 1 (red/italics = optimum)

State	Action	Q Value
93	***0***	***730021813***
93	1	-2147483648
93	2	-2147483648

93	3	-2147483648
93	4	-2147483648
93	5	-2147483648
93	6	-2147483648
93	7	-2147483648
93	8	-2147483648
94	***0***	***531860033***
94	1	-2147483648
94	2	-2147483648
94	3	-2147483648
94	4	-2147483648
94	5	-2147483648
94	6	-2147483648
94	7	-2147483648
94	8	-2147483648

Table 4.3 State-Action Table - Version 2 (red/italics = optimum)

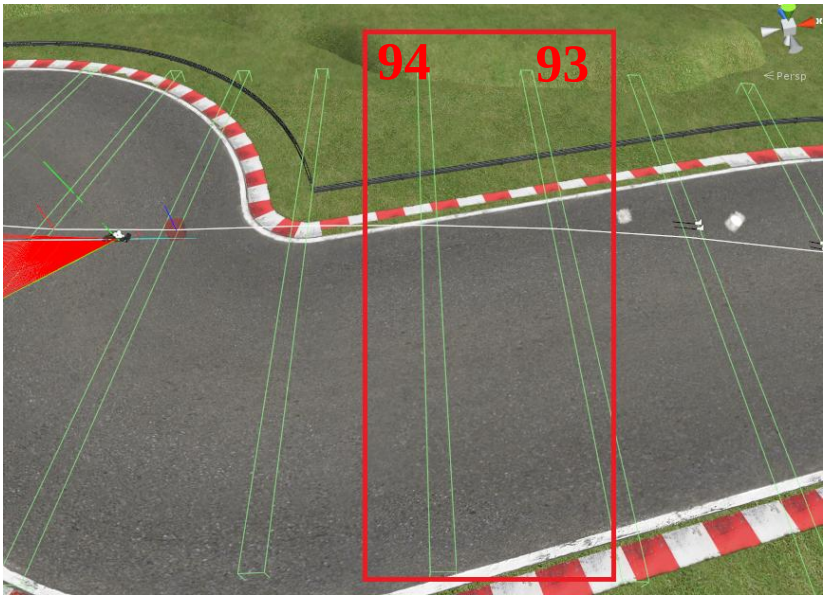


Figure 4.8 Relevant Section of the Track (States 93 - 94)

It is interesting to note the difference in Q Values produced between each version of the algorithm. The first version produced very erratic values whilst the second version produced a more consistent set of values. The tables also show the different state-action combinations for state 93. The first version selected action 6 (heavy braking) whilst the second version selected action 0 (full throttle).

4.2.2 Lap Times

Each policy was executed for a set of 10 laps each. By comparison, a lap time completed by myself and a normal player typically took between 39-42 seconds.

Lap Number	Version 1	Version 2
1	42.71785	41.96450
2	42.80776	41.30627
3	42.76348	45.01987
4	43.11338	44.86371
5	42.57750	41.39682
6	43.61481	41.64060

7	43.36933	44.75314
8	41.99917	42.81601
9	42.21012	41.57589
10	42.18604	41.24636
Average	**42.73594**	**42.65832**
Standard Deviation	**0.52378007**	**1.597068**

Table 4.4 Lap Time Table

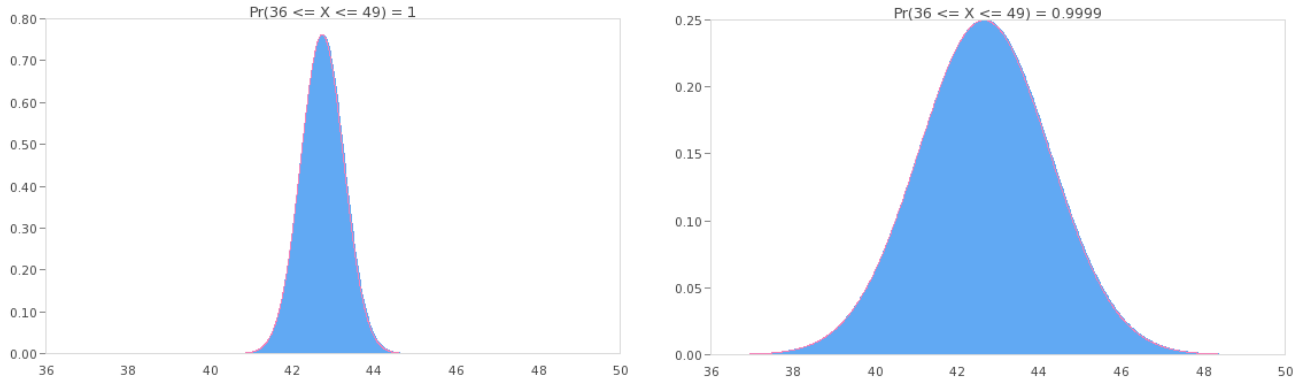


Figure 4.9 Standard Deviation of Lap Times (V1 vs V2)

The results produced in Table 4.4 show that the average lap time between versions was very similar whilst the standard deviation is vastly different. The second version of the algorithm produced less consistent lap-times as compared to the first version.

4.2.3 Episodes

The first version of the algorithm required a finite number of episodes in order to learn the track (number of episodes * number of possible actions). The second version, however, could be learnt in an indefinite number of episodes. The version used for results up until this point was taught using the same number of iterations as the first (approximately 1000). The algorithm was taught using a varying number of episodes to see the effect on lap-time. The lap-times on display in Table 4.5 are the result of an average of 10 lap-times. The policies which caused the car to crash still managed to complete the lap as the car was built with a reset function to reset the car after 2.5 seconds to a point slightly further long the racing line.

Episodes	**Lap Time / Result**
10	44.33456 (crashed into wall)
100	44.96534 (crashed into wall)
1000	42.65832
1500	41.74825
2500	40.95938
5000	41.46755
Average	**42.6889**
Standard Deviation	**1.62844**

Table 4.5 Episodes vs Lap Times

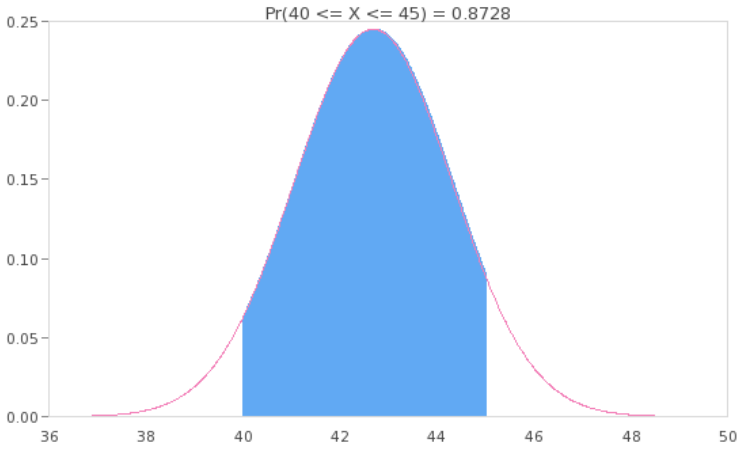


Figure 4.10 Standard Deviation of Lap Times based on Episodes

The results produced in Table 4.5 show that controllers taught with 10 and 100 episodes resulted in policies which caused the car to crash into a wall. The policies with 1000 episodes or more produced reasonable lap-times whilst not crashing. It is interesting to note the fastest lap time was produced from a 2500 episode policy and not from the 5000 episode policy.

4.2.4 Survey Result

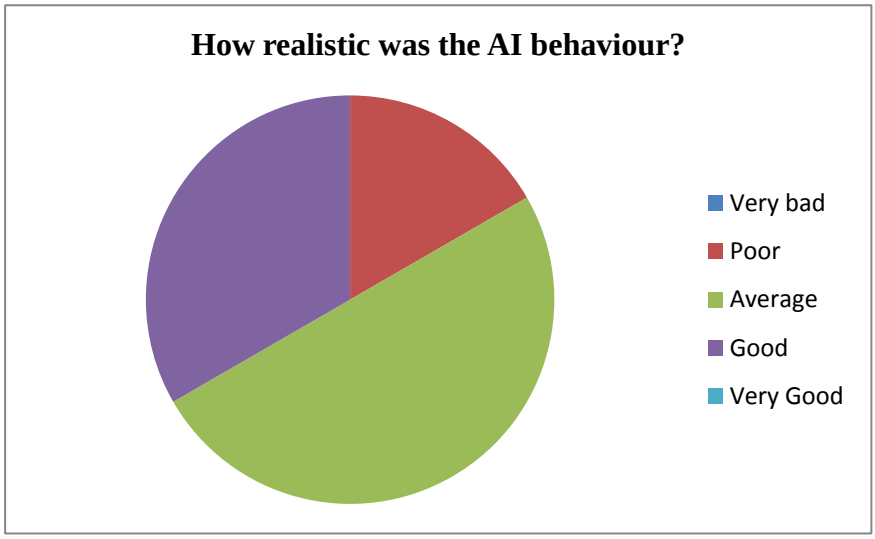


Figure 4.11 Survey Results for AI Question

4.3 Oculus Rift Integration

Unlike the Q-learning implementations, collecting results for the Oculus Rift integration was more challenging and subjective. From an implementation perspective, the device was fully integrated and fully functional with the project. It was simple for the user to toggle between the normal game mode and the Rift optimized mode (dual rendering for each eye).

From a gameplay perspective, it was important to receive feedback on the integration from a focus group. This information was collected from a survey (see appendix J). The results from this survey are shown in figure 4.12 and appendix K.

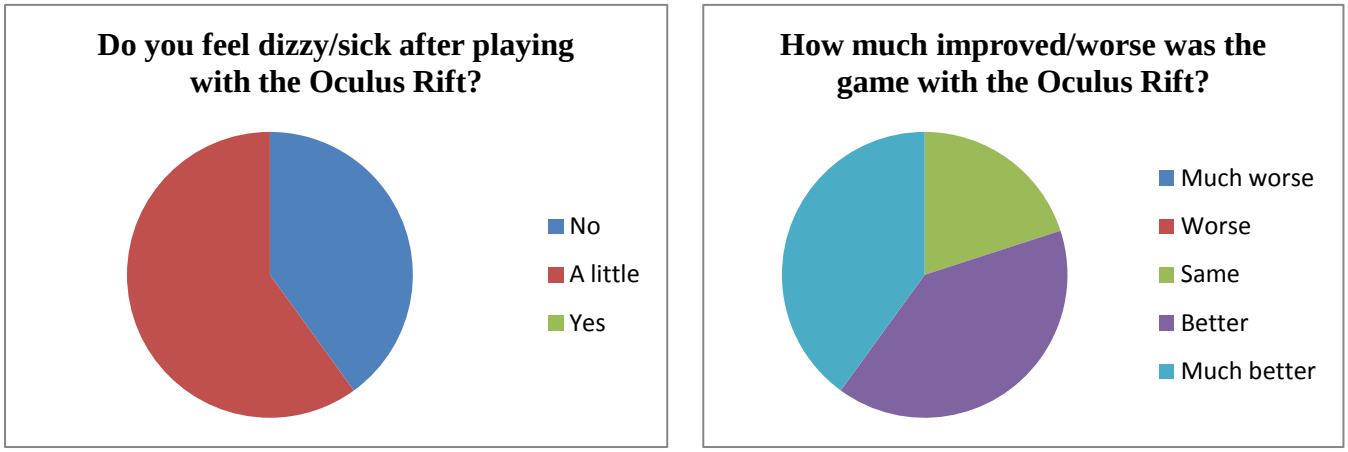


Figure 4.12 Oculus Rift Survey Results

4.4 Additional Survey Results/Comments

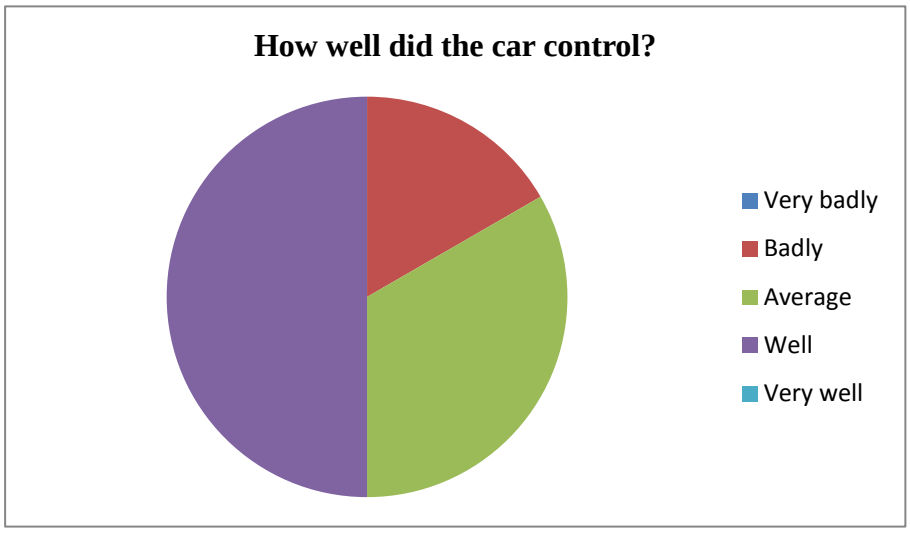


Figure 4.13 Car Control Survey Results

Suggestions to improve the game:

- More game modes and tracks.
- More tracks and weather effects.
- Gears sometimes feel slow and clunky after a crash.
- AI could be more realistic, crashed a few times.
- Shift camera view higher to make it easier to see the track.

5. Discussion

5.1 Project Management

The project was initially planned out in the form of a Gantt chart (see Appendix A). Each milestone was set based on the best possible estimates to complete each major task as described in the specifications.

As the timescales set in the original plan were estimates, various tasks took more or less time than expected. For example, implementing the AI took much longer than expected as the project focus shifted in the early stages. This became evident whilst performing the literature review on the AI for racing games and as such the Gantt chart was modified to accommodate this. The Gantt chart and plan were constantly modified throughout development to ensure the core tasks were on track.

A further issue throughout development that had not been considered was the effect of using Unity for the game. The abstraction of the Unity engine is a blessing and a curse. It allows for some very rapid developments, it can also be a hindrance in some situations. The use of non-MonoBehaviour [86] classes, for example, can be tricky and confusing to integrate into the game scene (this is discussed further in section 5.3). Figure 5.1 shows the final Gantt chard produced showing the actual time spent on each task in the project.

Figure 5.1 Gantt Chart Representing Actual Workflow

5.2 The Game

The produced game has reached and exceeded initial expectations, particularly on the AI side of the project. The majority of features specified at the beginning of the project were achieved to the expected standard or higher (see table 5.1). Even so there are still some unimplemented features and interesting areas for future developments. There is also a great deal of optimization that could be done to the project to make it run more efficiently and to be more reusable. Given a larger timescale, these optimizations would have been a core area of focus in the latter stages of development.

Objective	Completion Status	Notes/Improvements
B1.2 Unity project stub	Completed	N/A
B1.3 Basic car control	Completed	N/A
B1.4 Basic Oculus Rift integration	Completed	N/A
B1.5 Race track	Completed	N/A
B2.1 Improve car control	Completed	Further tweaking and improvements still possible
B2.2 AI car control	Completed	Further improvements can be made (ie use learning for overtaking, multiple racing lines etc)
B2.3 Full Oculus Rift integration	Completed	Could be improved by restricting overall rotation
B2.4 Game modes	Completed	Could be improved with more tracks
B2.5 Dashboard/HUD	Completed	N/A
B3.1 Weather effects	Completed	Could add dynamic weather, more weather modes
B3.2 AI difficulties	Partial Completion	Use FSM and varied learning to create difficulty hierarchies
B3.3 Shaders and Effects	Partial Completion	Could improve (on car reflection and visor) and add more shaders
B3.4 Leaderboard	Completed	Add multiple pages of lap times for multiple tracks

Table 5.1 Objective-Completion Table

5.2.1 Unimplemented Features

As discussed in section 4.3, the AI difficulty specification was not implemented to its completion. This was largely due to the time constraints of the project. Providing varying levels of AI difficulty would have been a good addition to the gameplay. This could have been achieved using an FSM architecture to vary the aggressiveness of the AI.

The Oculus Rift rotation lock was also not implemented as mentioned earlier in the report. This was due to the restrictions on the DLL files which meant that they could not be accessed at time of writing.

The final section was build 3.3; shaders and visual effects. Given more time it would have been interesting to delve deeper into Unity shader programming using ShaderLab and Cg. The shaders implemented were largely developed using tutorials and Unity wiki pages. It would be useful to develop more shaders by hand to create an interesting and unique visual style for the game.

5.3 Optimization

5.3.1 Use of MonoBehaviour Classes

MonoBehaviour is a base class that is used by scripts attached to game objects in the Unity engine. They classes provide many features and functions (such as *Start* and *Update*) which are very useful for producing quick prototypes and samples. However this can result in scattered and messy code in places, particularly as the codebase gets larger and larger.

This was an issue that was encountered towards the latter portion of the project development. Many of the classes and features implemented in the first and second builds made use of MonoBehaviour. This created many issues when trying to implement the FSM system for menu flow and also when transitioning between scenes in an organized manner. Given more time or a fresh start on the project, it would be a good idea to only use these classes when absolutely necessary. This would provide more control and a cleaner and more manageable codebase.

5.3.2 Custom Unity Plugins

Creating Unity plugins would be an excellent way of improving the reusability of the code. For example, it would have been helpful to create a plugin for the spline creator class. The current implementation requires a game object to be manually positioned in the scene and added to an array of waypoints in the inspector window. A plugin that tracked mouse position and clicks on the screen would have been useful to create and store these waypoints in a more user friendly fashion. It would also reduce the amount of time needed to create a spline.

Another useful plugin would have been for the reinforcement learning process. A proper interface for setting parameters and values for the learning algorithm would have been useful and more intuitive than the simple options provided in the inspector window.

5.3.3 Use of C++ Dynamic Link Libraries

The entire codebase of the project was developed using C# as a high level scripting language within the Unity engine. Using a lower-level language such as C++ in the form of Dynamic Link Libraries (DLLs) [87] when appropriate would have helped to increase the performance and efficiency of the game. DLL files can be integrated using Unity "*plugins*" [88].

DLLs are libraries that contain code and data that can be used by multiple programs at the same time. As such, this saves memory and reduces swapping between processes. This could have been a useful way of implementing the car AI techniques as multiple agents will be executing the same AI logic at the same time. Another reason for using DLLs is that they allow for easy upgrades and modifications to be made; when a change to a DLL is made, the whole game does not require re-compiling or re-linking as long as the function names and arguments remain the same. Because of these factors, the use of DLL files would have helped to increase the game performance and overall code reusability.

5.4 Q-Learning Implementation Comparison

The focal part of the project was the reinforcement learning implementation for the AI controller. Section 4.2 provided results for each implementation of the algorithm which allow for some comparisons and reflections to be made.

The state-action table comparison was the first set of results to be collected. The states chosen were located before the tightest corner on the track. The table shows the differing Q-tables for each version of the algorithm. The initial observation from comparing the tables is that the Q values for the first version produced larger and erratic numbers. This is because of the reward calculation process taking

place more often for each state and action pair. It is interesting to note the different actions selected for state 93. The first version selected a braking action whilst the second version selected the full throttle action. This was because the first version was focused on individual states at a time, meaning it often braked at the latest possible state as it didn't keep track of the reward based on the final end goal state. The second version had a more long-term view and as a result performed the braking action earlier (states 89, 90 and 92, after consulting the full Q table) in order to achieve a better speed and line through the corner.

The second set of results compared the lap times between the two versions of the algorithm (which were both taught using the same number of episodes). The average lap time between the two algorithms was extremely close. The standard deviation, however, was very different (0.5 for the first version and 1.5 for the second version). The first version appeared to produce very consistent lap times and results, whilst the second produced a wider range of very fast and relatively slow lap times. The slow lap times were often a result of going off track or hitting a wall. This would indicate that the number of episodes used to teach the second version were too low.

The second test results were the inspiration for the third test. The question was at what point the number of episodes used cease to have an effect on the second version of the algorithm. This test was not done for the first version because it learns in a finite number of episodes (number of actions * number of states). The results show that for 100 episodes or less, the car crashed or had an incident causing the lap-time to be increased. This was to be expected given the number of possible actions for the number of states in the game world. Interestingly, it also shows that the fastest lap time was produced from a policy created from 2500 episodes, whilst the policy produced by 1000 and 5000 episodes produced relatively similar lap times. The time produced by the 1000 episodes was to be expected, but for 5000 episodes one would expect the lap time to be at least as fast as the one produced for 2500. This result is possibly due to the algorithm performing further learning and discovering that a policy for this type of lap-time would result in a crash in the tighter parts of the racetrack. This is evident in the Q tables as highlighted in appendix G. Therefore it made safer choices whilst still maintaining a good overall speed.

Whilst the lap-times produced are relatively competitive compared to player lap-times, the overall performance of the algorithm in terms of lap-time is restricted by the optimality of the racing line. The line was implemented by hand and based on what appeared to be the best line around each corner. Better lap times would possibly have been achieved if this line was produced algorithmically to create a minimum-curvature line around the race track.

5.5 External Feedback

As mentioned in section 4.3, it was important to receive some meaningful feedback about the game in order to achieve a broad perspective on the areas of the game that worked well and the areas that could be improved. A survey was created to collect this data and a small group of testers (consisting of friends and family) tested the game (see appendix J for the survey and appendix K for the results). The focus of the survey was on the game AI and the Oculus Rift integration.

The outcome from the AI behaviour question provided some mixed results. The general consensus was that the AI performed "Average". Some testers found it to be of "Good" quality whilst others stated "Bad" quality. Those who rated it good found the AI fun and competitive to play against whilst those who rated it bad were regular racing game players and as such found the AI to be too easy. This disparity may also be due to the use of real-time obstacle avoidance and avoidance techniques that were implemented in the AI cars. As this was performing in real-time, it is difficult to predict exactly

how the car will behave when given a moving obstacle to avoid. A solution to this would be to perform Q-learning for overtaking manoeuvres and for multiple racing lines. These could be combined to create fluid and realistic avoidance behaviours. It could also be possible to create a hierarchy of RL policies for each racing line. For example, the top policy would be the optimum policy and then the derived policies would provide avoidance and overtaking manoeuvres in addition to less-optimum policies.

Many of the testers found the Oculus Rift enhanced the game experience. This is unsurprising as the Oculus Rift provides a very new and unique experience to video games and not many had tried the device beforehand. It was positive to note that none of the testers answered "Yes" to the question regarding sickness and dizziness. A great deal of effort was put into to development to ensure the cars controller was smooth enough to avoid a sickening effect. In early testing phases, the cars were faster and more sensitive to player inputs which caused more motion blur and rapid movement and this induced some undesired effects.

One final note that was received from the feedback was that some of the players did not find the car controls intuitive or easy to use. They did, however, note that they do not normally play or enjoy racing games. A concerted effort in the development of the project was to make the car control easy enough to pick up and play however there is room for improvement. This would require further tweaking and modifying the tyre slip and friction values to achieve a simpler car controller. A useful future development may be to provide varying levels of realism for the tyre properties allowing the player to make a selection when entering a game mode.

5.6 Similar Work/Projects

As discussed in section 6.5, no games using reinforcement learning had been implemented in Unity to date. As a result, all similar works are using different game engines.

The most relevant and similar work to this project can be seen in several examples using the TORCS (The Open Racing Car Simulator) game [89]. TORCS is an open-source game which encourages developers to create unique and novel AI agents. These agents can then be entered into competitions and championships to determine the best AI.

The most similar AI agent to that implemented in this project was developed by a student at Albert-Ludwigs University in Germany [90]. This project utilised reinforcement learning to learn not only throttle values but also steering values around a race track. TORCS provides a very useful car percept which can detect the track ahead and the direction to travel around the track. The agent produced was very impressive and raced around the track fairly realistically. However, there is noticeable twitching of the car on long straights and just before entering a corner. The reward function used was solely based on lap time and not about following a racing line around the track.

Another similar project using TORCS was to use reinforcement learning to learn how to overtake an opponent [91]. This work provided very realistic overtaking behaviours as compared to the obstacle avoidance steering behaviour produced in this project. Learning overtakes would require a great deal more data collection as there are many possible combinations of speeds and positions around the track. Furthermore it is hard to predict opponent speeds and steering behaviours, especially if it is a player controlling the opposing vehicle. As a result, it would be best to use a combination of the learned overtakes as well as the obstacle avoidance steering behaviour.

Lucas et al. performed a study comparing the use of a neural network (trained using evolution) and Q-learning to create a controller in a simple car simulation [92]. Their work used a simple waypoint-

based map and tested each type of algorithm to see which car passed through each of the waypoints in the shortest amount of time. Their work found that the neural network produced a more reliable controller than the Q-learning controller however the Q-learning controller was able to be trained in a much shorter period of time. At the end of the paper Lucas et al. suggest that the use of both the neural network and Q-learning algorithms may help to provide an even better controller.

6. Evaluation and Conclusions

This section of the report is focused on evaluating the project in hindsight now that the implementation is complete and results have been collected.

6.1 Choice of Milestones

The milestones and goals set at the start of the project were set based on planning and research, which resulted in the initial Gantt chart (appendix A). Producing the Gantt chart required a great deal of thought to be put into each milestone and as a result the project managed to stay on schedule and to achieve a good implementation for every section.

The milestones were varied and of differing levels of difficulty as reflected by the time allocated in the Gantt chart. This variation was a good idea as it helped to keep the project fresh and exciting after completing each milestone. If the project was solely focused on the AI, it may have become more difficult to find motivation in the later stages of the project. Variation was also a necessary factor as game development encompasses many disciplines (such as graphics, AI, gameplay and UI) and it was interesting to receive some exposure to each of these areas.

6.2 Research and Literature Review

In hindsight, the research and literature review process was essential to ensuring the project accomplished its goals. This was particularly true for the AI research. This research provided a broad view on the techniques available as well as a review on other projects. This information showed how other projects have attempted to use each technique and their varying levels of success. This allowed for a fully informed decision to be made before beginning the development and implementation process.

The software development research was also essential to ensure that the relevant methodologies were fully understood and this allowed the project to stay on track throughout development.

6.3 Planning and Agile Development

As mentioned in 6.1, the planning stage was essential in order to complete the project within the allocated timeframe. This was focused on defining and allocating milestones which were heavily informed and influenced by the literature review process.

The main principles of agile development were utilized throughout the development of the project. The main methodology used was the iterative approach to development in the SCRUM method. This was a useful tool to implement each milestone. Each milestone was broken up into sub-parts in order to create a step-by-step way of achieving each goal and treated as short sprints. Each sub-task was broken down into hours or days at the start of each milestone. Iterative development also allowed for rapid changes to be made to the specifications upon encountering issues or new ideas. This was very useful when implementing the Oculus Rift as it became apparent it would take much less time than anticipated. As a result, more time was allocated to the AI section which, as it turns out, was greatly appreciated due to the complexity of the reinforcement learning algorithm.

As this was an individual project, the SCRUM meetings were not heavily used. Tutor meetings, however, were treated as SCRUM meetings during which project milestones and requirements were discussed relative to the project timeframe. From this modifications were made to the overall project plan.

6.4 The Unity Game Engine

The Unity engine provided an excellent platform to launch into the game development process. The learning curve was somewhat less steep than it would be for most people due to previous experience and familiarity from using the engine to make simple game prototypes.

Unity provided the ability to program in C#, JavaScript and Boo. C# was the language of choice due to prior knowledge and experience with the language and its close proximity to C++ (which many games are written in).

As mentioned in section 5.3.1, the main downside of using the Unity engine was the dependency to inherit from MonoBehaviour classes. This caused the codebase to become more fragmented and less modular than was initially intended. The main cause of this was the Start and Update methods in the MonoBehaviour classes. These were handled internally by Unity and as a result it was not possible to see the execution order which caused some issues (especially when initialising a scene). A reduced dependency on these classes would allow more control in the ordering of these functions.

One of the most difficult decisions at the start of the project was deciding which game engine to use (between Unity, Unreal Engine 3, and Ogre3D). In hindsight, Unity was the best choice for several reasons. Primarily, the learning curve to use Unreal and Ogre would have been much steeper due to having no experience in either platform. The main factor in choosing Unity over Unreal was the fact that Unreal Engine 4 will be released in the not too distant future. This will deprecate much of the current functionality (such as UnrealScript in favour of C++) whilst Unity shows no signs of changing from its current format. The Ogre engine would also be more challenging as it is a code-heavy engine and does not provide a user-friendly GUI. This would have added an extra layer of difficulty in developing the game which was not feasible in the short timescale of the project.

Another benefit of Unity was the ease of deployment on multiple platforms. Currently, the project is restricted to the Windows PC platform due to the Oculus Rift integration. Should the Oculus Rift become supported by other platforms in the future however the game could be easily exported to those platforms thanks to the engines capabilities.

6.7 Reinforcement Learning and Steering Behaviours

There were two main challenges in the reinforcement learning process; understanding the data structures required and how to represent the game world to the algorithm. The data structures were developed over a course of research and experimentation. Creating the representation of the world, however, was more challenging as this required converting the 3D world into something that the algorithm could understand and process in simple terms of states and actions. Once these had been implemented, the rest of the algorithm was fairly intuitive and simple to write.

The first version of the algorithm was implemented intuitively due to unfamiliarity with traditional reinforcement learning techniques and theory. The second version of the algorithm arose after performing further research into reinforcement learning in general. This unveiled the more traditional approach of only assigning negative rewards until the goal was reached. It was very interesting to have two different approaches to the algorithm and consequently to be able to compare and contrast the results.

6.5 Reinforcement Learning in Unity

A major hurdle in the project was understanding how to perform the reinforcement learning process in Unity. From research and searching through the Unity forums it became apparent that no one had

attempted this before (or at least made it known to the public). This made finding answers to seemingly simple problems more challenging.

One of the biggest challenges in performing the Q-learning process in Unity was being able to run simulations fast enough without causing the engine to crash and at the same time ensure that the code executed as expected. The goal was simple enough; disable the graphics rendering process and increase the simulation speed. The simulation speed was simple to tackle with a few lines of code (while ensuring all relevant learning code was operating at a fixed rate to ensure it behaved as expected).

The graphics issue was initially tackled by using Unity's *batch mode* feature [93]. This, however, is primarily designed for server-side code and as a result was not fully successful in running the reinforcement learner. It was then discovered, after discussing the issue with some other developers and Unity employees, that the graphics could be disabled by simply disabling all cameras in the scene and closing all windows with graphics being displayed (such as the scene window). This allowed for the learning to take place in-engine with full access to the console output which was very useful in the early stages of development.

6.6 The Project as a Whole

This project has been a fascinating learning experience and has linked together all aspects of game development as intended. It is very rewarding to look back and see all of the features that have been implemented into the game. The planning process that took place before the project began (with the initial Gantt chart) and following an Agile development process throughout were instrumental in achieving the goals of the project. This ensured that milestones were hit on time and allowed the project to develop at a swift rate which was essential in a project with such a short timescale.

6.7 Future Developments

There are several potential developments that could be made for this project in the future. The first and most useful improvement would be to make the AI system more reusable by integrating it as a Unity plugin. This would allow for faster and easier modifications to the learning agents whilst also allowing the system to be shared with others through the Unity community forums or through the Unity asset store.

Another development that would improve the AI would be to use a similar technique to that of Forza's Drivatars. This would involve having the AI agent learn multiple variations of the racing line around the circuit. It would also be interesting to use different reward functions to create further variance in the behaviour of the AI to create varying skill levels or difficulties. A further improvement could also be to increase the state-space in the game which would increase the number of state-action combinations possible around the track. This could result in improved agent behaviour, especially around tight and twisting corners.

As mentioned in the literature review, it would be possible to implement a neural network layer above the reinforcement learner to provide an even more advanced AI agent. The neural network could be used to help learn which combination of racing lines and overtaking manoeuvres is best at each part of the track given different scenarios. It would be informative to compare an online and offline version of the neural network agent and to see how well both versions fair in a race-like scenario. This is a similar concept to that proposed by Lucas et al. [92] in their comparison of Q-learning and neural networks for a car controller which could be used to evolve the controller as the simulation progresses.

It would also be interesting to see if this approach could be adapted to work in the TORCS game (discussed in section 5.6). It would be insightful to see how well the agent would fare against the other AI agents produced for the TORCS competitions and see whether it was competitive or not in a race.

There are many more ways to improve the game itself. These improvements to the gameplay would include creating and integrating more race tracks, more weather effects, an achievements system to reward the player for playing and an online multiplayer mode amongst others.

6.8 Conclusion

This has been an educational and informative project in learning how to develop a video game from start to finish whilst at the same time maintaining an iterative, Agile development approach. It was a true test of dedication and passion to complete the project as there were no distractions from other projects. Consequently maintaining motivation and concentration demanded a great deal of discipline. Finally there was great deal of pressure to ensure the project accomplished its goals.

The project provided the opportunity to advance personal knowledge and skills in the AI field. This was twofold from implementing both the reinforcement learning algorithm and steering behaviours and also through the research that took place in the early stages of the project.

The project has accomplished a great deal, but has also left options open for further developments in the future. This is an exciting prospect. Hopefully this project will help to inspire more Unity developers to delve into the realms of advanced AI techniques such as reinforcement learning and potentially neural networks.

7. Glossary

Term	Definition
AI	Artificial Intelligence
C#	The programming language used to script in Unity
Catmull-Rom Spline	Represents a curve by intersecting all specified points
FSM	Finite State Machine
Gantt Chart	Tool used to plan and manage project timescales
HMD	Head Mounted Display
IDE	Integrated Development Environment
NPC	Non-Player Character
Oculus Rift	The low-latency HMD used for the game
Q Learning	A specific form of reinforcement learning
Reinforcement Learning	Algorithm used for creating AI agents without hard-coding
Spline	Mathematical method of representing a curve
UML	Unified Modelling Language
Unity	The 3D game engine used to develop the project
Visual Studio	The IDE used for programming
VR	Virtual Reality

8. References

[1] Oculus Rift. (2013). Oculus Rift. Available: http://www.oculusvr.com/. Last accessed 30th August 2013.

[2] Oculus VR. (2013). 4-Month Unity Pro Trial for Oculus Devs. Available: http://www.oculusvr.com/blog/4-month-unity-pro-trial-for-oculus-devs/. Last accessed 30th August 2013.

[3] Agile Alliance. (2013). The Agile Manifesto. Available: http://www.agilealliance.org/the-alliance/the-agile-manifesto/. Last accessed 30th August 2013.

[4] Buckley, Sean. (2013). Oculus' Palmer Luckey and Nate Mitchell on the past, present and future of the Rift. Available: http://www.engadget.com/2013/03/19/oculus-rift-luckey-mitchell-interview/. Last accessed 30th August.

[5] Oculus Rift. (2013). Oculus Rift. Available: http://www.oculusvr.com/. Last accessed 30th August 2013.

[6] Edge (June 2013). Edge Magazine. London: Future Publishing. p74-81.

[7] G4. (2012). John Carmack Interview At E3 2012: Oculus Rift Virtual Reality Headset. Available: http://www.youtube.com/watch?v=UyuMVazQPos. Last accessed 30th August.

[8] Sutton, R and Barto, A (1998). Reinforcement Learning:An Introduction. United States: MIT Press. p330-332.

[9] Buckland, Mat (2004). Programming Game AI by Example. United States: Wordware Publishing. p86-110.

[10] Turn 10 Studios. (2013). Forza Motorsport. Available: http://forzamotorsport.net/en-US/en-GB. Last accessed 30th August 2013.

[11] Polyphony Digital. (2013). Gran Turismo. Available: http://www.gran-turismo.com/gb/. Last accessed 30th August 2013.

[12] Codemasters. (2012). F1 2012. Available: http://www.codemasters.com/uk/f12012/pc/. Last accessed 30th August 2013.

[13] Buckland, M (2004). Programming Game AI by Example. United States: Wordware Publishing Inc.. p85-125.

[14] Reynolds, C. (1999). Steering Behaviors For Autonomous Characters. Game Developers Conference. 1 (1), p763-782.

[15] Permadi, F. (2010). What is Ray-Casting?. Available: http://www.permadi.com/tutorial/raycast/rayc1.html#WHAT IS RAY-CASTING. Last accessed 16th September 2013.

[16] Buckland, M (2004). Programming Game AI by Example. United States: Wordware Publishing Inc.. p241-247.

[17] Pratt, S. (2010). Study: Navigation Mesh Generation. Available: http://www.critterai.org/book/export/html/2. Last accessed 16th September 2013.

[18] Tan, C, Chen, C, Tai, W, Yen, S . (2008). An AI Tool: Generating Paths for Racing Game. International Conference on Machine Learning and Cybernetics. 6 (1), p3132-3137.

[19] Wenderlich, R. (2011). Introduction to A* Pathfinding. Available: http://www.raywenderlich.com/4946/introduction-to-a-pathfinding. Last accessed 16th September 2013.

[20] Kumar, K. (2011). Unity3d 3.5 nav mesh pathfinding. Available: http://www.youtube.com/watch?v=8fMPjl7QJDw. Last accessed 16th September 2013.

[21] Unity. (2012). Navmesh and Pathfinding (Pro only). Available: http://docs.unity3d.com/Documentation/Manual/NavmeshandPathfinding.html. Last accessed 16th September 2013.

[22] Rockstar Games. (2013). Grand Theft Auto. Available: http://www.rockstargames.com/grandtheftauto/. Last accessed 16th September 2013.

[23] Sutton, R, Barto, A (1998). Reinforcement Learning: An Introduction. United States: MIT Press. p330-334.

[24] Patel, P, Carver, N, Rahimi, S. (2011). Tuning Computer Gaming Agents using Q-Learning. Proceedings of the Federated Conference on Computer Science and Information Systems. 1 (1), p581-588.

[25] Steam. (2013). Counter-Strike. Available: http://store.steampowered.com/app/10/. Last accessed 16th September 2013.

[26] Microsoft. (2004). Drivatar. Available: http://research.microsoft.com/en-us/projects/drivatar/. Last accessed 16th September 2013.

[27] Candela, J, Herbrich, R, Graepel, T. (2011). Machine Learning in Games. Available: http://research.microsoft.com/en-us/events/2011summerschool/jqcandela2011.pdf. Last accessed 16th September 2013.

[28] David, S, Kushilevitz, E, Mansour, Y. (1997). Online Learning versus Offline Learning. Machine Learning. 29 (1), p45-63.

[29] Smith, L. (1996). An Introduction to Neural Networks. Available: http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html. Last accessed 16th September 2013.

[30] Honkela, A. (2001). Multilayer perceptrons. Available: http://users.ics.aalto.fi/ahonkela/dippa/node41.html. Last accessed 16th September 2013.

[31] Yu, Z. (2010). Fundamentals of Neural Networks. Available: http://www.yukool.com/nn/fundamental.htm. Last accessed 16th September 2013.

[32] Reingold, E. (1999). Artificial Neural Networks Technology. Available: http://www.psych.utoronto.ca/users/reingold/courses/ai/cache/neural3.html. Last accessed 16th September 2013.

[33] N/A. (2007). Multilayer Perceptron. Available: http://faculty.ksu.edu.sa/zitouni/Pictures%20Library/Forms/DispForm.aspx?ID=12. Last accessed 16th September 2013.

[34] Tan, T, Jason, T, Anthony, P, Ong, Jia. (2011). Neural Network Ensembles for Video Game AI Using Evolutionary Multi-Objective Optimization. International Conference on Hybrid Intelligent Systems (HIS). 11 (1), p605-610.

[35] Knowles, J. (2004). The Pareto Archived Evolution Strategy (PAES). Available: http://www.cs.man.ac.uk/~jknowles/multi/. Last accessed 16th September 2013.

[36] Lionhead Studios. (2001). Black & White. Available: http://www.lionhead.com/games/black-white/. Last accessed 16th September 2013.

[37] Champandard, A. (2007). Top 10 Most Influential AI Games. Available: http://aigamedev.com/open/highlights/top-ai-games/. Last accessed 16th September 2013.

[38] Wikipedia. (2006). Finite State Machine Logic. Available: http://en.wikipedia.org/wiki/File:Finite_State_Machine_Logic.svg. Last accessed 16th September 2013.

[39] Champandard, A. (2007). The Gist of Hierarchical FSM. Available: http://aigamedev.com/open/article/hfsm-gist/. Last accessed 16th September 2013.

[40] Tesauro, G. (1995). Temporal difference learning and TD-Gammon. Communications of the ACM. 38 (3), p58-68.

[41] Mohamed, A. (2010). Artificial Intelligence in Racing Games. Available: http://www.cs.bham.ac.uk/~ddp/AIP/RacingGames.pdf. Last accessed 16th September 2013.

[42] Druck, Aaron. (2006). When Will Virtual Reality Become a Reality?. Available: http://www.techcast.org/Upload/PDFs/061026231112tc%20%20aaron.pdf. Last accessed 24th May 2013.

[43] Bierton, David. (2011). Sony HMZ-T1 Personal 3D Viewer Review. Available: http://www.eurogamer.net/articles/digitalfoundry-sony-hmz-t1-personal-3d-viewer-review. Last accessed 24th May 2013.

[44] Connors, Devin. (2013). Oculus Rift at CES: Bigger Screen, Lower Latency. Available: http://www.gamefront.com/oculus-rift-at-ces-bigger-screen-lower-latency/. Last accessed 24th May 2013.

[45] Oculus. (2013). Oculus Rift Development Kit. Available: https://www.oculusvr.com/pre-order/. Last accessed 24th May 2013.

[46] Google. (2013). Google Glass. Available: http://www.google.com/glass/start/. Last accessed 24th May 2013.

[47] Poole, W. (2013). Sony set to go big with virtual reality on PlayStation 4. Available: http://www.eurogamer.net/articles/2013-09-03-sony-set-to-go-big-with-virtual-reality-on-playstation-4. Last accessed 16th September 2013.

[48] Yoon, Jong-Won et al.. (2010). Enhanced User Immersive Experience with a Virtual Reality based FPS Game Interface. IEEE Conference on Computational Intelligence and Games. 1 (1), p69-74.

[49] O'Luanaigh (2005). Game Design Complete. United States: Paraglyph Press. p19-26.

[50] Giant Bomb. (2010). Rubber Band AI. Available: http://www.giantbomb.com/rubber-band-ai/3015-35/. Last accessed 16th September 2013.

[51] Nintendo. (2013). Mario Kart 7. Available: http://mariokart7.nintendo.com/. Last accessed 16th September 2013.

[52] Agile Manifesto. (2001). Manifesto for Agile Software Development. Available: http://agilemanifesto.org/. Last accessed 16th September 2013.

[53] Schwaber, K (2004). Agile Project Management with Scrum. United States: Microsoft Press. p1-138.

[54] Schwaber, K. (2011). The Scrum Guide. Available: https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guide.pdf. Last accessed 16th September 2013.

[55] Effective AGILE Development. (2013). Effective AGILE Development. Available: http:// http://effectiveagiledev.com/portals/0/scrum-process-basic.png. Last accessed 16th September 2013.

[56] Gamma, E, Helm, R, Johnson, R, Vlissides, J (1998). Design Patterns: Elements of Reusable Object-Oriented Software. United States: Addison Wesley. p10-390.

[57] Microsoft. (2013). Model-View-Controller. Available: http://msdn.microsoft.com/en-us/library/ff649643.aspx. Last accessed 16th September 2013.

[58] Wikipedia. (2010). MVC-Process. Available: http://en.wikipedia.org/wiki/File:MVC-Process.png. Last accessed 16th September 2013.

[59] SurveyMonkey. (2013). Writing Survey Questions: Tips for effective and relevant questions.. Available: http://help.surveymonkey.com/articles/en_US/kb/Writing-Survey-Questions-Tips-for-effective-and-relevant-questions. Last accessed 16th September 2013.

[60] Halteman, E. (2011). 10 Common Mistakes Made When Writing Surveys. Available: https://www.surveygizmo.com/survey-blog/10-common-survey-mistakes-part-1/. Last accessed 16th September 2013.

[61] SXPanda. (2010). F1 car cockpit view. Available: http://sxpanda.webs.com/apps/photos/photo?photoid=101303694. Last accessed 16th September 2013.

[62] Unified Modelling Language. (2013). UML Resource Page. Available: http://www.uml.org/. Last accessed 16th September 2013.

[63] Oculus VR. (2013). 4-Month Unity Pro Trial for Oculus Devs. Available: http://www.oculusvr.com/blog/4-month-unity-pro-trial-for-oculus-devs/. Last accessed 24th May 2013.

[64] Unity. (2013). Rigidbody. Available: http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody.html. Last accessed 16th September 2013.

[65] Unity. (2013). Wheel Collider. Available: http://docs.unity3d.com/Documentation/Components/class-WheelCollider.html. Last accessed 16th September 2013.

[66] Unity. (2013). Prefabs. Available: http://docs.unity3d.com/Documentation/Manual/Prefabs.html. Last accessed 16th September 2013.

[67] Rivermill Studios. (2013). Race Track Construction Kit. Available: https://www.assetstore.unity3d.com/#/content/6843. Last accessed 16th September 2013.

[68] Edy. (2010). How to make a physically real, stable car using WheelColliders. Available: http://forum.unity3d.com/threads/50643-How-to-make-a-physically-real-stable-car-with-WheelColliders. Last accessed 16th September 2013.

[69] DrivingFast.net. (2013). The Racing Line. Available: http://www.drivingfast.net/techniques/racing-line.htm#.UjcmA8akoYE. Last accessed 16th September 2013.

[70] Dunlop, R. (2005). Introduction to Catmull-Rom Splines. Available: http://www.mvps.org/directx/articles/catmull/. Last accessed 16th September 2013.

[71] Digital Rune. (2006). Discrete Collision Detection vs. Continuous Collision Detection. Available: http://www.digitalrune.com/Documentation/html/cd2fc090-d3fd-4a0f-a7d3-1759241c8545.htm. Last accessed 16th September 2013.

[72] Unity. (2013). GameObject.tag. Available: http://docs.unity3d.com/Documentation/ScriptReference/GameObject-tag.html. Last accessed 16th September 2013.

[73] Oculus. (2013). Oculus Developer Forums. Available: https://developer.oculusvr.com/forums/. Last accessed 16th September 2013.

[74] Unity. (2013). Lens Flare. Available: http://docs.unity3d.com/Documentation/Components/class-LensFlare.html. Last accessed 16th September 2013.

[75] Grespon. (2013). Wet Shaders. Available: https://www.assetstore.unity3d.com/#/content/8849. Last accessed 16th September 2013.

[76] Ashnfara. (2013). Rainscape. Available: http://u3d.as/content/ashnfara/rainscape/1Aj. Last accessed 16th September 2013.

[77] Unity. (2013). Surface Shader Examples. Available: http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaderExamples.html. Last accessed 16th September 2013.

[78] NVidia. (2003). The Cg Tutorial. Available: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html. Last accessed 16th September 2013.

[79] Unity. (2013). ShaderLab syntax: Shader. Available: http://docs.unity3d.com/Documentation/Components/SL-Shader.html. Last accessed 16th September 2013.

[80] University of Waterloo. (2000). *Lambertian Reflection.* Available: http://medialab.di.unipi.it/web/IUM/Waterloo/node78.html. Last accessed 16th September 2013.

[81] Wikipedia. (2005). Cube mapped reflection example. Available: http://en.wikipedia.org/wiki/File:Cube_mapped_reflection_example.jpg. Last accessed 16th September 2013.

[82] Pranckevicius, A. (2007). Mirror Reflection. Available: http://wiki.unity3d.com/index.php?title=MirrorReflection2. Last accessed 16th September 2013.

[83] Code Project. (2002). Singleton Pattern & its implementation with C++. Available: http://www.codeproject.com/Articles/1921/Singleton-Pattern-its-implementation-with-C. Last accessed 17th April 2013.

[84] Code Project. (2009). State Design Pattern. Available: http://www.codeproject.com/Articles/38962/State-Design-Pattern. Last accessed 17th April 2013.

[85] Wikipedia. (2013). Observer. Available: http://en.wikipedia.org/wiki/File:Observer.svg. Last accessed 17th April 2013.

[86] Unity. (2013). MonoBehaviour. Available: http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html. Last accessed 16th September 2013.

[87] MSDN. (2013). What is a DLL?. Available: http://support.microsoft.com/kb/815065. Last accessed 24th May 2013.

[88] Unity. (2013). Plugins (Pro/Mobile-Only Feature). Available: http://docs.unity3d.com/Documentation/Manual/Plugins.html. Last accessed 24th May 2013.

[89] TORCS. (2013). Welcome to TORCS. Available: http://torcs.sourceforge.net/. Last accessed 16th September 2013.

[90] Engesser, T. (2011). Learning to drive fast in TORCS using Batch Mode Reinforcement Learning. Available: http://www.youtube.com/watch?v=bawh8YAnXTg. Last accessed 16th September 2013.

[91] Loiacono, D. (2008). RL applied to TORCS: Learning to overtake. Available: http://www.youtube.com/watch?v=jFr9YagbgIo. Last accessed 16th September 2013.

[92] Lucas, S, Togelius, J. (2007). Point-to-Point Car Racing: an Initial Study of Evolution Versus Temporal Difference Learning. Symposium on Computational Intelligence and Games. 1 (1), p260-267.

[93] Unity. (2013). Command line arguments. Available: http://docs.unity3d.com/Documentation/Manual/CommandLineArguments.html. Last accessed 16th September 2013.

Appendix A: Project Definition Document

1. Introduction

Over the past decade, video games have become more realistic, engaging and immersive than ever before [1]. This is mainly due to the large improvements in graphics and processing technology [2]. The level of immersion in games will never truly be realised, however, using traditional screen technologies such as monitors, TVs and projectors.

Head-mounted displays (HMDs) have always been popular amongst the virtual reality community but until recently the technology has not existed to incorporate this into games [3]. The main issue with traditional HMDs is the high latency experienced in rendering and displaying the scene when the user turns or rotates their head. This issue has been resolved with the release of the Oculus Rift [4], a HMD specifically designed for use with video games. The Oculus Rift may allow for a level of immersion to be created that was previously unachievable with past technology. The main reason for this is due to the wide field of view and low latency that the Rift provides [5]. John Carmack of iD Software was one of the first developers to test and develop for the Rift. Carmack was quoted as saying his early build of Doom 3 for the Rift was "*the best VR demo ever made*" [6].

This project is aimed at creating a racing game utilising the Oculus Rift technology to create an immersive racing experience. This document will highlight the key objectives and challenges that are to be undertaken. It will also outline the estimated timescale for development (in Gantt chart form) as well as discuss the various risks associated with the project and how these risks are to be mitigated.

2. Project Description

The project will focus on integrating the Oculus Rift into a racing game using the Unity game engine [7]. Unity engine was selected as the development platform for several reasons. Primarily, Unity is natively integrated with the NVIDIA PhysX engine [8]. This will help to create realistic behaviour and mechanics for the car control.

Unity also boasts integration with the Oculus Rift hardware [9]. This will help in reducing the development time required to integrate the hardware with the game and allow for more time configuring and optimizing the integration.

An alternative to using an engine would be to develop a C++ interface between the Oculus Rift hardware and the graphics code directly (using OpenGL or DirectX). Whilst Oculus have provided detailed documentation on how to integrate the hardware in this way, there is a great deal of complexity in doing so and this could possibly take up the majority of the allocated time for the project. One issue, for example, is the *distortion scale* [10]. This needs to be accounted for in order to create a realistic looking image as the lenses in the device create a pincushion distortion effect [11]. This is counteracted by warping the image using a barrel distortion effect [12], resulting in a normal looking image. This is just one of the many factors that would need to be considered and implemented (other issues include head tracking, rendering images to each screen and latency).

The game will be an open-wheeler racing game. Open-wheeler cars are cars with wheels that are exposed on the outside of the car's main body (such as a Formula 1 car). Such cars are designed specifically for racing and are designed to function at high speeds. The racing game genre was selected for several reasons. Firstly, a vehicular-based game seems to be the optimal genre for the hardware available. This is because the player will be expecting to interact with an additional peripheral (such as a steering wheel or a controller) in order to control the vehicle. Secondly, genres

such as first-person shooters (FPS) could feel slightly unnatural as the player will expect to be walking or running in order to move their character throughout the world.

While playing the game, in order to maximize the level of immersion using the Rift, the player will be locked into first-person mode within the cockpit of the car. Controlling the game view using another style of camera (such as a third-person camera) could reduce the level of immersion and realism. It could also feel unnatural and may cause undesired, nauseating effects for the player [13].

The game will have two game modes; time trial and race. The time trial mode will challenge the player to complete a lap of the selected race track as quickly as possible. The best times achieved for each track will be stored in a high score table. The race mode will incorporate some artificial intelligence (AI) into the game. The AI could be developed using the Finite State Machine (FSM) architecture to control its movement around the track. Other options will be researched in the early stages of the project. The AI will be required to complete laps of the tracks without crashing into the environment and also take action in order to avoid colliding with the player.

The race tracks in the game will be imported using 3D models with collision hulls (likely in *.FBX* format). This will demonstrate a loose coupling of the game system from the art assets and allow for additional tracks to be added easily without affecting the game functionality.

The project will be developed using a combination of C# scripting (within the Unity environment) and C++ *Dynamic Link Library* (DLL) files [14]. DLL files can be integrated using Unity "*plugins*" [15]. DLLs are libraries that contain code and data that can be used by multiple programs at the same time. As such, this saves memory and reduces swapping between processes. This will be a useful way of implementing the car AI techniques as multiple agents will be executing the same AI logic at the same time. Another reason for using DLLs is that they also allow for easy upgrades and modifications to be made; when a change to a DLL is made, the whole game does not require recompiling or relinking as long as the function names and arguments remain the same. Because of these factors, DLL files will be developed in order to increase the game performance and code reusability.

3. High Level Objectives

The development has been divided into three builds to ensure that the main aspects of the game are completed before the project deadline. The goal of Build 1 (B1) will be to achieve the basic infrastructure that is required to build the foundation for later goals. Build 2 (B2) will integrate the core game functionality. Build 3 (B3) is the final game build. This build will add the advanced features and functionality required to create a complete, polished game.

3.1 B1 – Build 1: Basic Functionality

- B1.1 – Review relevant literature and perform game design
- B1.2 – Unity project stub with structured content hierarchy
- B1.3 – Basic car control and physics on simple terrain.
- B1.4 – Integrate Oculus Rift as camera positioned in car.
- B1.5 – Import race track model and ensure car and camera behave as expected.
- B1.6 – Testing and refactoring

3.2 B2 – Build 2: Core Game Features

- B2.1 – Improve car control physics to achieve higher level of realism.
- B2.2 – Create AI car control system (using FSM) and integrate three AI cars.

- B2.3 – Integrate head tracking features of the Oculus Rift to allow player to move head realistically in cockpit.
- B2.4 – Incorporate separate game modes; *Time Trial* and *Race*.
- B2.5 – Implement Heads-Up-Display (HUD).
- B2.6 – Testing and refactoring

3.3 B3 – Build 3: Further Enhancements

- B3.1 – Incorporate weather effects to vary car behaviour.
- B3.2 – Modify AI to create two difficulties; easy and hard.
- B3.3 – Create shaders to implement a realistic helmet visor across players view (i.e. lens flare, rain drops).
- B3.4 – Testing and refactoring

4. Beneficiaries

There are several beneficiaries from the development of this project. I am the primary beneficiary as the development will provide a positive and in-depth learning experience in several core areas of games development, primarily game physics and artificial intelligence. It will also provide exposure to the new technology of the Rift and the bleeding edge of the field of virtual reality. In addition, the project will also give me experience with using a commercial engine (Unity) to a level of depth that I have not previously achieved. Should the completed game be of a high standard, it may also provide possible commercialisation opportunities through popular indie game outlets such as the Steam Greenlight program [16].

The project may also prove beneficial for Unity developers looking to integrate the Oculus Rift into their games as the hardware is still at an early stage in its lifecycle.

In the event that this project demonstrates a satisfying level of immersion using the Oculus Rift in a racing game, then this study will provide a positive endorsement of the product for Oculus.

5. Methodology

The project will be developed using an Agile methodology [17] with frequent iterations and builds (as outlined in section 3) to ensure that the specifications are met. This will allow for modifications and improvements to be made to the system as the development process unfolds. Agile also ensures that regular testing will take place throughout the development of the game. As such, errors will be spotted early on in development allowing for easier debugging.

There will also be a focus on implementing the various features of the game in a modular, loosely coupled manor. This will allow for modifications to be made to specific parts of the game without affecting the system as a whole.

The project consists of the following elements:

1. Literature review on car mechanics/physics, racing AI, virtual reality (specifically using HMDs) in games, Oculus Rift and the Unity engine.
2. Design and develop game environment.
3. Design and develop car physics using Unity PhysX integration.
4. Design and develop AI cars using FSMs.
5. Implement Oculus Rift head tracking and ensure realistic behaviour.
6. Iterate over implemented features and improve functionality.
7. Implement advanced features.

8. Evaluate completed system and document.

The timescale for these sections is outlined in the project work plan in section 8.

6. Areas of Theory

There are three core areas of literature to study throughout the early stages of the project; the car mechanics and physics, racing AI and virtual reality in video games (particularly using HMDs).

6.1 Car Mechanics and Physics

The development of the racing game mechanics and physics is a focal area of study. This will be assisted by Jim Parker's book "*Start Your Engines*" (2005) [18] which is focused on developing the mechanics of a car in a video game. The concepts will also prove useful when implementing weather effects to alter the vehicles performance. The Unity engine also provides PhysX integration. This will allow for realistic modelling of forces on the vehicles in the game.

6.2 Racing AI

There are several main methods in creating a racing AI. In the early days of racing game development, the traditional method was to implement a *racing line* by generating waypoints along the race track and creating a spline by interpolating between these points [19]. This can, however, result in unrealistic behaviour as the AI car may hit other objects or cars in the scene.

An improvement upon this is to use an A* pathfinding algorithm between these waypoints. This could potentially allow the car to check for obstacles and react accordingly whilst trying to follow the racing line. This concept is outlined in the paper by Wang and Lin (2012) [20]. They discuss adding a visibility factor to the car to improve the efficiency of the A* algorithm. This method, however, would result in a very deterministic AI.

One other method would be to use neural networks or fuzzy logic systems (as discussed by Fujii et al, 2008 [21]) to create a potentially more realistic behaving AI. However, due to the timeframe and scope of the project implementing these methods would not be feasible.

Mat Buckland discusses moving game agents in great detail in the book Programming Game AI by Example [22]. Buckland outlines that the movement of an agent or vehicle can be divided into three layers; action selection (select goals and decide on plan to follow), steering (calculate trajectories to achieve the goals) and locomotion (apply mechanical aspects to the agent to create motion). Buckland also proposes useful techniques such as seeking, fleeing and obstacle avoidance amongst others. These will be researched further during the literature review and design stage (B1.1).

The AI to be developed in the project will borrow from several of the ideas discussed (primarily from the techniques discussed by Buckland), wrapped within an FSM architecture. The A* or waypoint method could be used for a general racing state when no surrounding vehicles are present. An overtaking or avoiding state would then be required in order to avoid collisions if surrounding vehicles are present. The development of the AI system will also be assisted by ideas from the book by Kyaw and Swe (2013) [23] which is specifically focused on game AI development and implementing an FSM architecture in Unity.

6.3 Virtual Reality

Virtual reality (VR) was a very popular field within video games and computer science in general until the early 1990s. VR technology and hardware have traditionally been very expensive and cumbersome, with prices commonly exceeding thousands of pounds. A study by TechCast in 2006 [24] found that most people would be willing to pay between $100-500 USD for VR technology for entertainment purposes, with a minimal number of people willing to pay above $1000 USD. This disparity is a clear indicator of one of the many reasons as to why VR fell off the map. A further cause

for the drop-off is that VR technology (HMDs in particular) typically suffered from high latency problems when moving around a virtual environment. This often causes a break in the immersion from the experience and is another one of the issues that VR has faced in the past.

The Sony HMZ-T1 [25] is a HMD that was released in late 2011 that attempted to tackle the latency issue. Sony managed to succeed in reducing latency issues, but at the sacrifice of image quality (720p screens for each eye). The main stumbling block for the HMZ-T1 was the price tag which was too high (RRP of £700) and as such suffered poorly commercially.

The Oculus Rift has seemingly solved both of these key issues by producing a HMD with 1080p screens for each eye (720p on the development kits as they are working prototypes) and the lowest latency that has been seen on an HMD thus far [26]. Many developers and journalists have been outspoken in how well the hardware operates and many developers are in the process of using it in their projects. Some notable names in the games industry that have endorsed the hardware include John Carmack, Gabe Newell and Cliff Blesinzki. Oculus has also managed to tackle the price barrier with developer kits selling for approximately £200 (the final product with 1080p screens will probably be slightly higher at £250) [27].

The Oculus Rift and upcoming devices such as Google Glass [28] will likely help VR come back to the forefront of computer science as the consumer market grows and matures.

The goal of the project is to create the most immersive and realistic experience possible (within the constraints of the graphics and art assets available). The paper by Yoon et al. (2010) [29] discusses how HMDs and sound can affect the level of immersion when playing a first-person shooter (FPS) game. They discovered that using the HMD with traditional control methods provided the highest levels of immersion, whilst using the HMD with less traditional inputs (data gloves) proved less successful.

The goal of this project is to use HMD with traditional control methods (a controller or race wheel) and as such this should provide an acceptable level of immersion.

Yoon et al. found that using headphones with 3D sound increased the levels of immersion. The Unity engine also provides an integrated 3D sound engine. This will allow for realistic 3D audio effects to be implemented into the game with relative ease which should in turn increase the immersion level of the player

The Oculus Rift SDK and developer community forums will provide further guidance on virtual reality theory and assist on the implementation of the HMD.

7. Project Feasibility

There are many aspects to the project and as such many factors to consider in determining the feasibility of the project. Firstly, the feasibility of developing the game using the Unity engine is positive as I have had previous exposure to the engine and I am confident in using the basics of the engine. I do not anticipate it to be an overly steep learning curve to understand and incorporate the more advanced features in the Pro version of the engine.

Achieving the goals set in B1 will be a relatively straight-forward process as my previous knowledge attained about the Unity engine will be utilised. This should be a good stepping stone into the project.

Developing the car AI will be a new area of exposure, fortunately there are many techniques that are documented on how to achieve a racing AI. Using the FSM architecture will bring a level of familiarity to the AI control and as such the AI should be feasible to complete in the project timeframe.

Integrating the Oculus Rift may prove to be challenging due to the immaturity of the platform. However, the Oculus Rift software development kit contains detailed documentation on incorporating the hardware into Unity (as well as various other platforms). Oculus also provides developer forums which has a very active and supportive community. This interaction will prove useful throughout the development of the project.

The features highlighted in B3 are for the most part new areas of study for me. However, they have been defined in terms that are not overly ambitious in their nature and as such it should be feasible to achieve these goals.

8. Project Evaluation

The overall success of the project will be evaluated in several ways. Primarily, it will be measured by comparing whether the initial goals of the project as outlined in this document correspond with the final features in the produced game. If the majority of the goals set for builds 1, 2 and 3 have been achieved then the project will have been a success. The game performance will be analysed by checking that all algorithms are performing optimally using profiling tools. This will help to ensure that the game does not suffer from frame rate drops and other performance related issues.

The immersion of the game will also be evaluated using user acceptance tests. These tests will consist of users playing the game with and without the Oculus Rift and then completing a survey. This survey will gather data on how the user's immersion was affected when using the HMD. It may also be useful to capture lap time and race result data for each of the versions played (with and without the HMD) to see whether a higher level of immersion has an impact on player performance.

9. Project Work Plan

Task Name	Duration
1. Review Literature and Design	19 days
2. Build 1: Basic Functionality	15 days
B1.1 Review literature and perform game design	15 days
B1.2 Unity project stub	1 day
B1.3 Basic car control and physics	6 days
B1.4 Simple Oculus Rift integration	2 days
B1.5 Import and test with race track model	4 days
B1.6 Testing and Refactoring	15 days
3. Build 2: Core Game Features	50 days
B2.1 Improve car control to behave realistically	10 days
B2.2 Car AI control system using FSM	14 days
B2.3 Integrate Oculus Rift head tracking	12 days
B2.4 Game modes - Time Trial/Race	6 days
B2.5 Heads-Up-Display	4 days
B2.6 Testing and Refactoring	50 days
4. Build 3: Further Enhancements	22 days
B3.1 Weather effects	6 days
B3.2 Varying AI difficulties	6 days
B3.3 Shaders to simulate realistic helmet effects	6 days
B3.4 Testing and Refactoring	22 days
5. Final bug fixing and improvement	6 days
6. Testing and evaluation	8 days
7. Project Report and Documentation	91 days
Project Deadline	1 day

10. Risk Assessment

The following table assigns a level of risk (between 1 and 5) to each objective of the project.

Objective	Severity of Failure	Risk of Failure	Impact (severity * risk)	Mitigation Strategy
B1.1 Review relevant literature and perform game design	5	1	5	Relevant books have been purchased or are available in university library. Large part of game design already done in preparation for Game Development Process exam.
B1.2 Unity project stub with structured content hierarchy	5	1	5	Already familiar with Unity interface, free 4 month Pro license with Oculus Rift.
B1.3 Basic car control and physics on simple terrain	5	1	5	Many examples of basic car mechanics. Test thoroughly to ensure fully functional and reacting as expected.
B1.4 Integrate Oculus Rift as camera positioned in car	5	1	5	Achievable without hardware. Detailed documentation and developer community available. Develop early to ensure integration.
B1.5 Import race track model and ensure car and camera behave as expected	4	1	4	Use race track model from third party source as not focus of project. Test thoroughly to ensure collisions are detected as expected.
B2.1 Improve car control physics to achieve higher level of realism	5	3	15	Refactor car control code to modularise it. Detailed documentation available on Unity PhysX integration. Thorough testing to ensure the controls feel accurate and realistic.
B2.2 Create AI car control system (using FSM) and integrate three AI cars	4	3	12	Techniques for implementing car AI available in literature.
B2.3 Integrate head tracking features of the Oculus Rift to allow player to move head realistically in cockpit	5	3	15	Scheduled mid-way through project to ensure hardware will arrive in time. Detailed documentation and developer community available.

B2.4 Incorporate separate game modes; *Time Trial* and *Race*	2	1	2	Simple switch to enable AI or enable timed mode. Not crucial to project as a whole.
B2.5 Implement Heads-Up-Display (HUD)	1	1	1	All data required to output to HUD produced in physics calculations in B2.1.
B3.1 Incorporate weather effects to vary car behaviour	1	4	4	Car controller will be developed in a modular, reusable fashion to allow for variances in car performance to be added.
B3.2 Modify AI to create two difficulties; easy and hard	1	3	3	AI developed in modular, reusable fashion. As such, adding variances should be feasible.
B3.3 Create shaders to implement a realistic helmet visor across players view (i.e. lens flare, rain drops).	1	5	5	Detailed documentation available on Unity shaders. Implementation similar to OpenGL shaders (used in previous projects).

10.1 High Risk Impact Factors

B2.1 Improve car control physics to achieve higher level of realism

This is a high impact factor as the gameplay relies on a high quality, realistic car controller. This risk has been mitigated by reviewing relevant literature in B1.1. In addition, there is a great deal of documentation and examples available online on how to develop a car controller (in Unity and in general). I have also been working in my spare time on learning how to implement various car mechanics in Unity.

B2.2 Create AI car control system and integrate three AI cars

The AI controller is another integral part of the game design and gameplay and as such would have a large impact on the game if it was not completed. This is a high risk area due to the vast complexity of such an AI. As with B2.1, the literature review process will help mitigate this risk by analysing the various methods thoroughly and adapting them accordingly. This risk also been alleviated by the fact that the controller will be developed using an FSM architecture which I am familiar with and provides an intuitive way of creating NPC behaviours. The controller will also be developed using C++ in DLL files. I am very familiar with C++ and as such this should help reduce the risk further.

B2.3 Integrate head tracking features of the Oculus Rift

This is a high risk factor and the potential impact regarding the quality of the Oculus Rift integration is dependent on testing with the hardware. It is possible to set up scene for the Oculus Rift without the hardware (as in B1.4) but thorough testing will be required in order to achieve a realistic camera simulation.

The main risk involved with this objective is receiving the Oculus Rift hardware in time. The estimated delivery date for the hardware is mid-May. This risk has been mitigated by pushing the task back to a later stage of the project (mid-July) to ensure that the hardware will be available.

Build 3 Objectives

All objectives in B3 are advanced developments and as such can be removed or modified if required due to time constraints.

11. Research Ethics Checklist

Number	Question	Yes/No
If the answer to any of the following questions is NO, then the project plan needs to be modified, because the project should not continue as currently planned. Seek advice very early about it		
1	Does the planned project pose only minimal and predictable risk to the student?	Yes
2	Does it pose only minimal and predictable risk to other people affected by the project?	Yes
3	Are arrangements for the supervision of the project appropriate?	Yes
4	Is the project carried out or supervised by competent researchers?	Yes
5	Do the foreseeable benefits of the project outweigh the foreseeable risks?	Yes
If the answer to any of the following questions is YES, then authorisation from the Senate's Ethics Committee is required. Seek advice very early about it		
6	Does the project involve interaction with, or collecting personal information about, people who are vulnerable because of their social, psychological or medical circumstances?	No
7	Does the project involve animals?	No
8	Does the project involve research on pregnant women or women in labour?	No
9	Does the project involve research on persons under the age of 18?	No
10	Does the project involve research on human tissue?	No
11	Does the project involve research on vulnerable categories of people who may include minority groups?	No
Will the student ensure that any people subject to observation or data collection are:		
12	Fully informed about the procedures affecting them and affecting any information collected about them (how the data will be used, to whom they will be disclosed, how long they will be kept)?	Yes
13	Fully informed about the purpose of the research?	Yes
14	Will the consent of these people be obtained?	Yes
15	When these people can be classified as research subjects, will it be clear to them that they may withdraw at any time?	Yes
16	Will the student make arrangements to ensure that material or private information obtained from or about these people remains confidential?	Yes
17	Not sufferers of motion sickness (and fully notified that they may experience some disorientation from using the system).	Yes
18	Over the age of 18 years of age (due to the potential unexpected effects of the virtual reality experience)	Yes

12. References

[1] Doyle, Patrick. (2011). A Study of the Immersion Enhancement Techniques Utilised by Video Games. Available: http://teapowered.co.uk/documents/A-Study-of-the-Immersion-Enhancement-Techniques-Utilised-by-Video-Games_Patrick-Doyle_Teapowered.pdf. Last accessed 24th May 2013.

[2] Madigan, Jamie. (2010). The Psychology of Immersion in Video Games. Available: http://www.psychologyofgames.com/2010/07/the-psychology-of-immersion-in-video-games/. Last accessed 24th May 2013.

[3] Buckley, Sean. (2013). Oculus' Palmer Luckey and Nate Mitchell on the past, present and future of the Rift. Available: http://www.engadget.com/2013/03/19/oculus-rift-luckey-mitchell-interview/. Last accessed 24th May 2013.

[4] Oculus Rift. (2013). Oculus Rift. Available: http://www.oculusvr.com/. Last accessed 24th May 2013.

[5] Edge (2013). Edge Magazine. London: Future Publishing. p74-81.

[6] G4. (2012). John Carmack Interview At E3 2012: Oculus Rift Virtual Reality Headset. Available: http://www.youtube.com/watch?v=UyuMVazQPos. Last accessed 24th May 2013.

[7] Unity. (2013). Unity Game Engine. Available: http://unity3d.com/. Last accessed 24th May 2013.

[8] NVidia. (2013). PhysX. Available: http://www.geforce.co.uk/hardware/technology/physx. Last accessed 24th May 2013.

[9] Oculus VR. (2013). 4-Month Unity Pro Trial for Oculus Devs. Available: http://www.oculusvr.com/blog/4-month-unity-pro-trial-for-oculus-devs/. Last accessed 24th May 2013.

[10] Oculus (2013). Oculus Rift SDK Overview. United States: Oculus. p28-33.

[11] Bockaert, Vincent. (2013). Pincushion Distortion. Available: http://www.dpreview.com/glossary/optical/pincushion-distortion. Last accessed 24th May 2013.

[12] Bockaert, Vincent. (2013). Barrel Distortion. Available: http://www.dpreview.com/glossary/optical/barrel-distortion. Last accessed 24th May 2013.

[13] Häkkinen, Jukka et al.. (2001). Postural Stability and Sickness Symptoms after HMD Use. Available: http://s3.amazonaws.com/publicationslist.org/data/jukka.hakkinen/ref-22/Hakkinen%202002.pdf. Last accessed 24th May 2013.

[14] MSDN. (2013). What is a DLL?. Available: http://support.microsoft.com/kb/815065. Last accessed 24th May 2013.

[15] Unity. (2013). Plugins (Pro/Mobile-Only Feature). Available: http://docs.unity3d.com/Documentation/Manual/Plugins.html. Last accessed 24th May 2013.

[16] Valve. (2013). Steam Greenlight. Available: http://steamcommunity.com/greenlight. Last accessed 24th May 2013.

[17] Agile Alliance. (2013). The Agile Manifesto. Available: http://www.agilealliance.org/the-alliance/the-agile-manifesto/. Last accessed 24th May 2013.

[18] Jim Parker (2005). Start Your Engines: Developing Driving & Racing GamesStart Your Engines: Developing Driving & Racing Games. United States: Paraglyph Press.

[19] Mohamed, Abdal. (2012). Artificial Intelligence in Racing Games. Available: http://www.cs.bham.ac.uk/~ddp/AIP/RacingGames.pdf. Last accessed 24th May 2013.

[20] Wang, Jung-Ying and Lin, Yong-Bin. (2012). Game AI: Simulating Car Racing Game by Applying Pathfinding Algorithms. International Journal of Machine Learning and Computing. 2 (1), p13-18.

[21] Fujii, Seiya. (2008). A Study on Constructing Fuzzy Systems for High-Level Decision Making in a Car Racing Game. IEEE International Conference on Fuzzy Systems. 1 (1), p2299-2306.

[22] Buckland, Mat (2004). Programming Game AI by Example. United States: Wordware Publishing. p86.

[23] Kyaw, Aung and Swe, Thet (2013). Unity 4.x Game AI Programming. United States: Packt Publishing.

[24] Druck, Aaron. (2006). When Will Virtual Reality Become a Reality?. Available: http://www.techcast.org/Upload/PDFs/061026231112tc%20%20aaron.pdf. Last accessed 24th May 2013.

[25] Bierton, David. (2011). Sony HMZ-T1 Personal 3D Viewer Review. Available: http://www.eurogamer.net/articles/digitalfoundry-sony-hmz-t1-personal-3d-viewer-review. Last accessed 24th May 2013.

[26] Connors, Devin. (2013). Oculus Rift at CES: Bigger Screen, Lower Latency. Available: http://www.gamefront.com/oculus-rift-at-ces-bigger-screen-lower-latency/. Last accessed 24th May 2013.

[27] Oculus. (2013). Oculus Rift Development Kit. Available: https://www.oculusvr.com/pre-order/. Last accessed 24th May 2013.

[28] Google. (2013). Google Glass. Available: http://www.google.com/glass/start/. Last accessed 24th May 2013.

[29] Yoon, Jong-Won et al.. (2010). Enhanced User Immersive Experience with a Virtual Reality based FPS Game Interface. IEEE Conference on Computational Intelligence and Games. 1 (1), p69-74.

Appendix B: Pitch Document

Hook

Traditional arcade-style racing game with realistic graphics in open-wheeler cars with full native integration with the Oculus Rift. This will be one of the first fully Rift supported racing games available when it hits the market in 2014.

The game will be one of the most immersive racing experiences ever produced without the need for expensive simulators or multiple monitors.

Pitch Presentation Key Points

Slide 1: Introduction

- A first-person arcade-style racing game.
- Realistic, eye popping 3D visuals utilising the full potential of the Oculus Rift with a wide field of view.
- Similar gameplay style to Forza and Need for Speed series of games.

Slide 2: Game Modes

- Time Trial:
    - Solo race against the clock to achieve the fastest lap time.
    - Bronze, Silver and Gold medal times.
    - Online leaderboards add competitive element.
- Race:
    - Race against 3 opponents of specified difficulty.
    - Optional: qualifying session to earn grid slot (otherwise randomly determined).
- *Type of car available will be dependent on the game mode or race track chosen.*

Slide 3: Game Controls 1

- View Control (using Rift):
    - The overall field of view will be restricted.
    - Facing forwards, rotation 90 degrees left and right, up and down.
    - i.e. they cannot look directly behind themselves at the back of the race seat (unrealistic as race drivers are typically strapped into the seat).
- Race Tracks:
    - Popular, real racing circuits will be used from around the world to provide variety and differing challenges.

Slide 4: Game Controls 2

- Game Input: Traditional racing style controls using a Controller and/or Race Wheel (Keyboard also supported).
- Focus on controller/wheel not a barrier to entry:
    - Most PC gamers own a console or have a controller.
    - If purchasing an Oculus Rift they will likely have one!
    - Hardcore racing fans will likely have race wheel set ups (most of these are compatible cross-platform from console to PC).

Slide 5: Game Interface and HUD

- Mock-ups of game menus, interface and heads-up display.

Slide 6: Future Plans and DLC

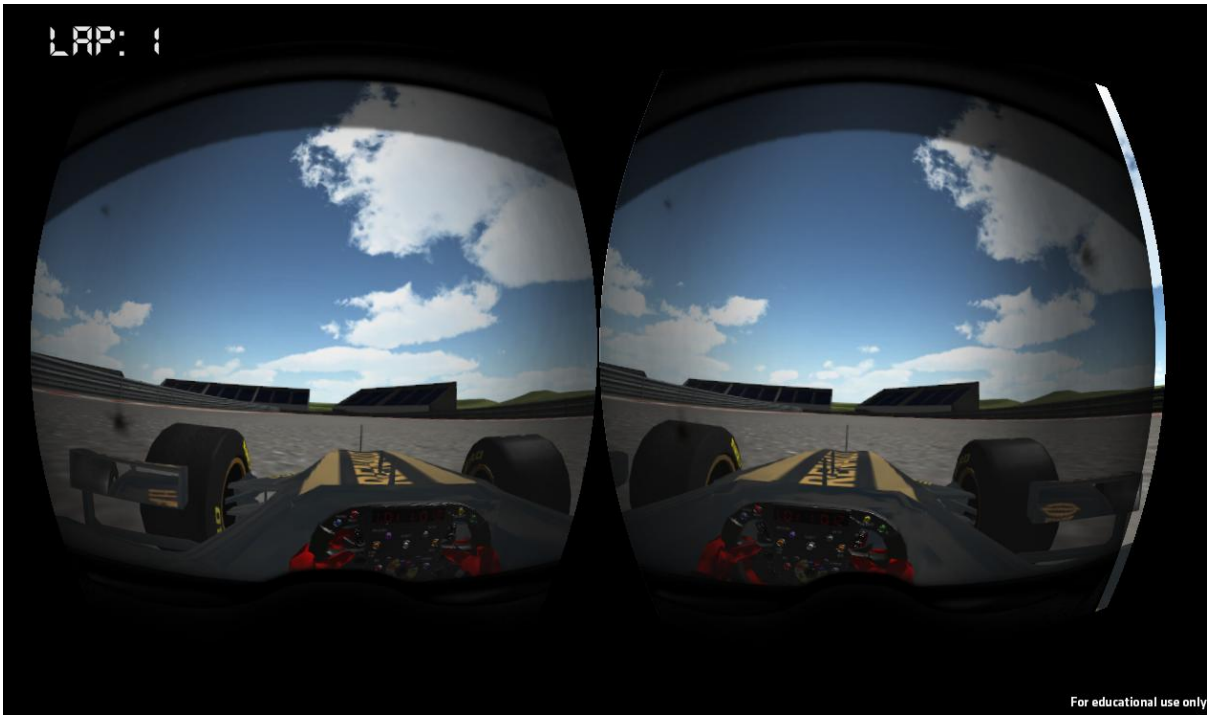
- Multiplayer Mode:
    - Race against up to 8 people online (grid slot determined by skill level - based on previous online race results - best player starts at the back of the pack).
- Additional Cars, Tracks, Liveries.

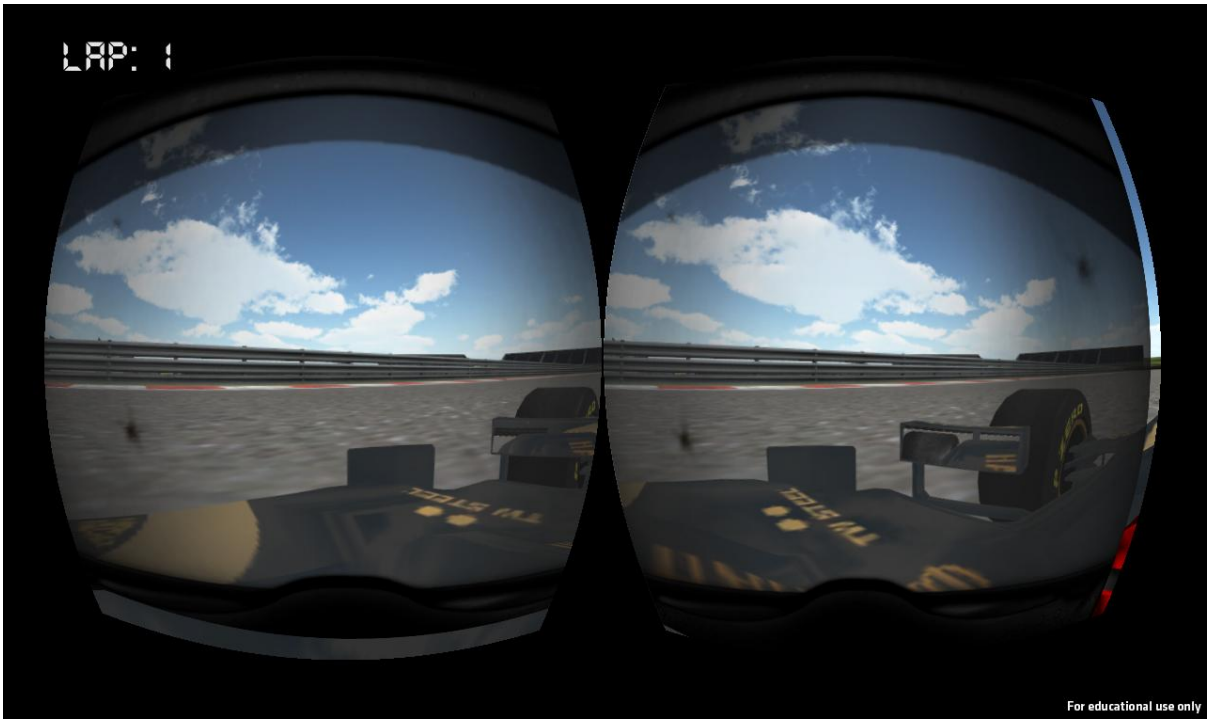
Appendix C: Game Visuals

Main Menu

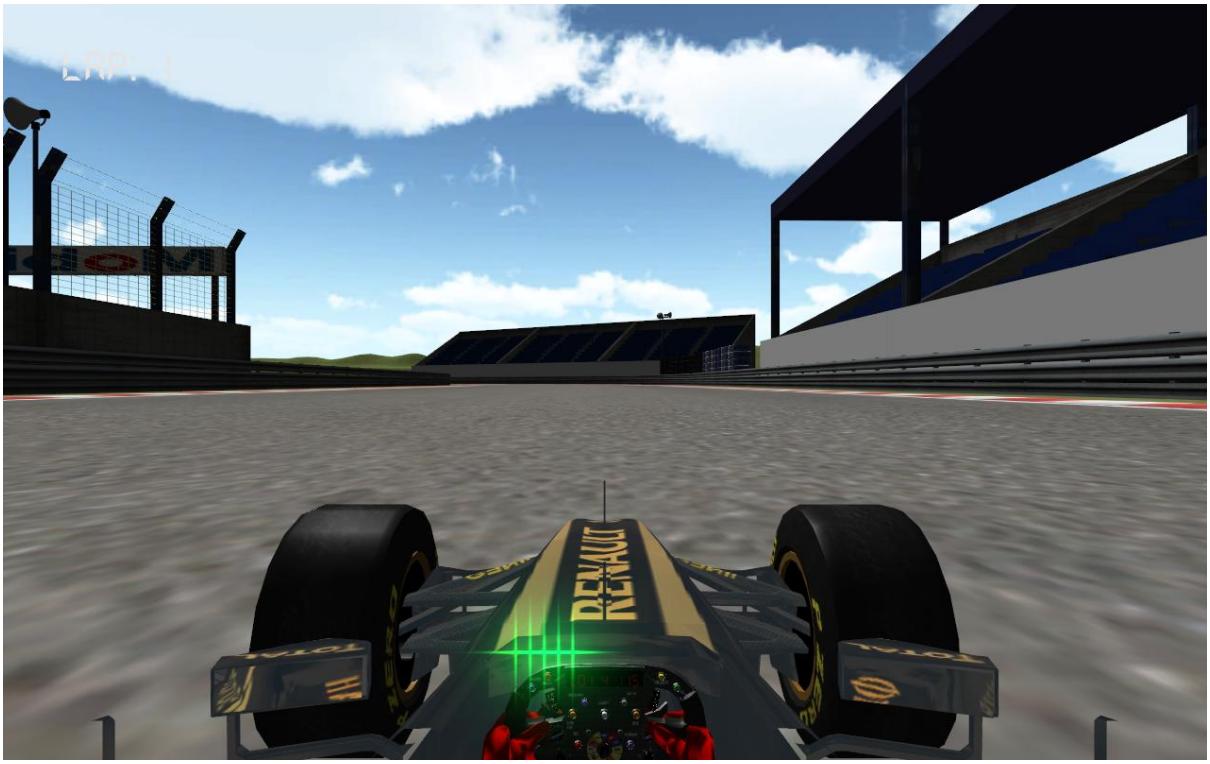
First Person View (without Oculus Rift)

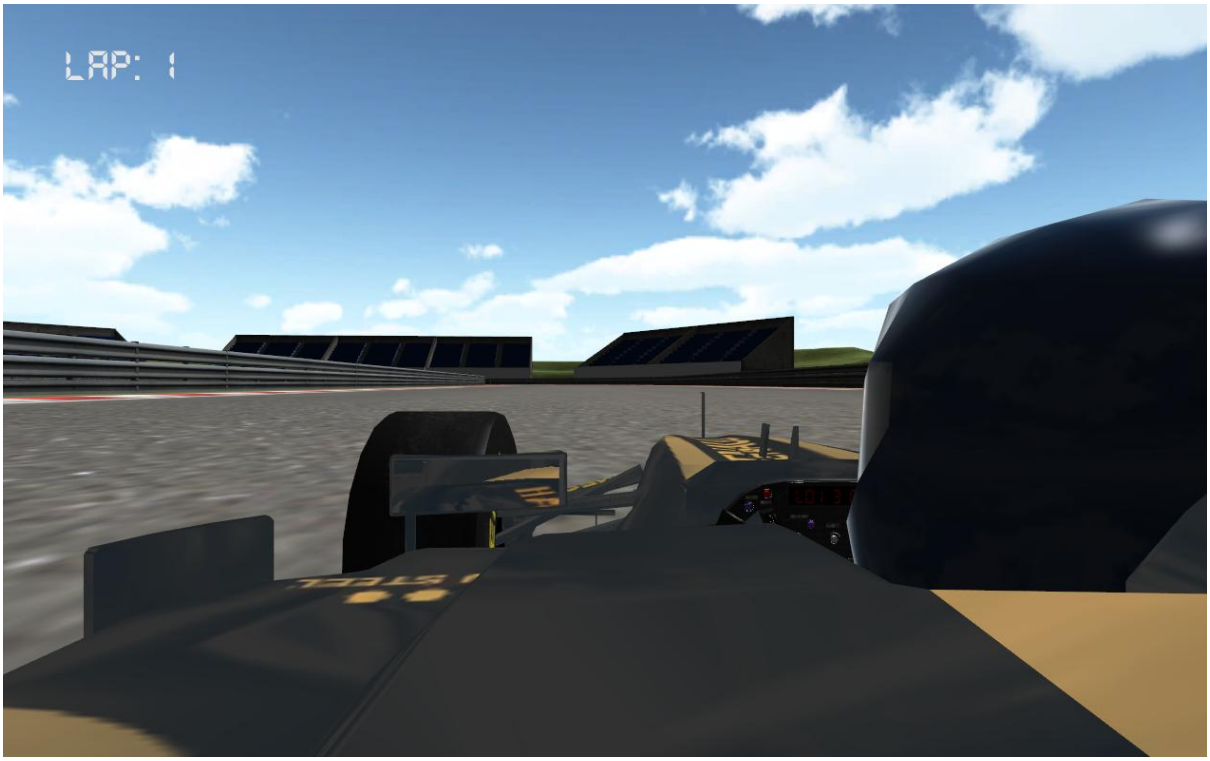
First Person View (with Oculus Rift)

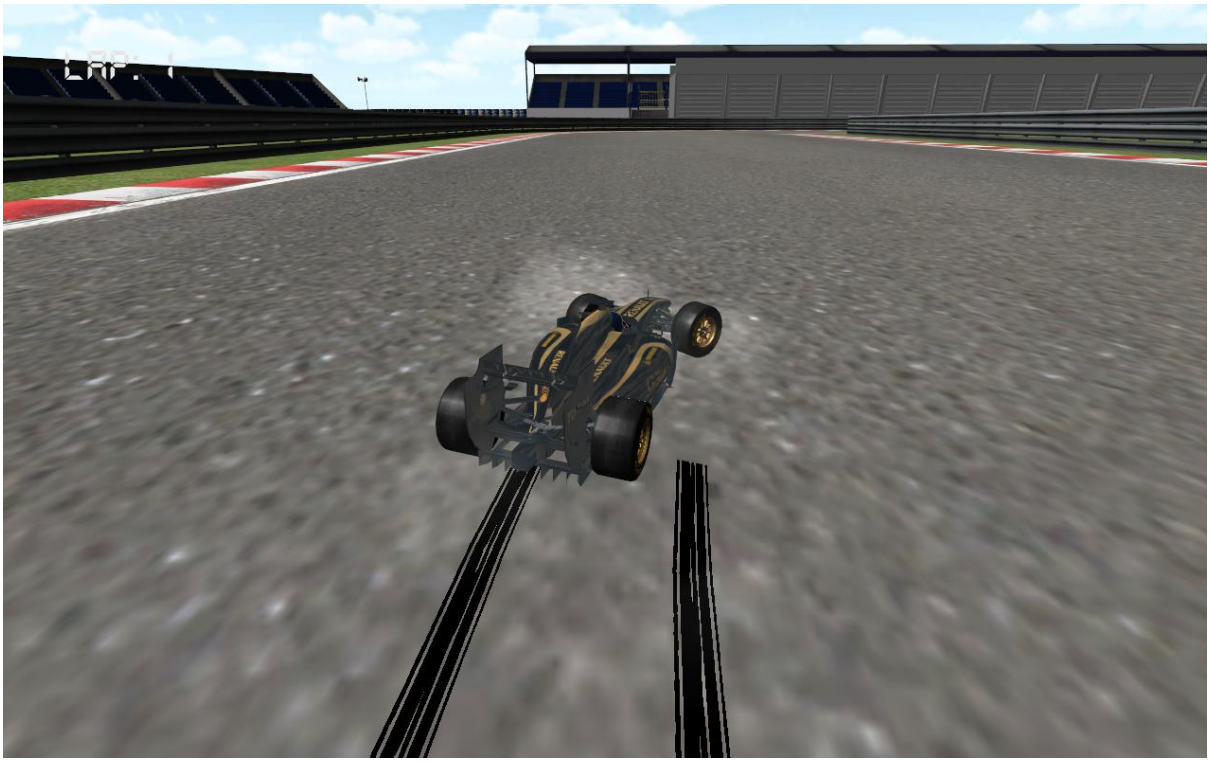




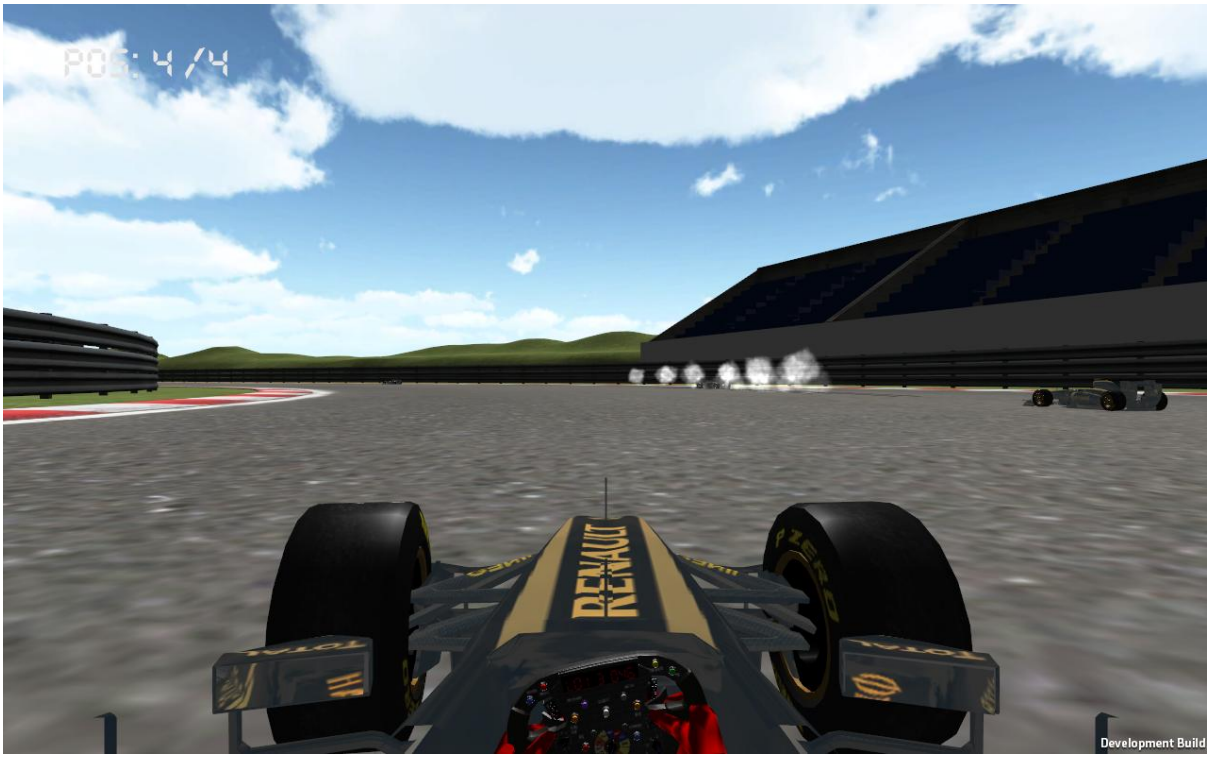
Other Camera Views







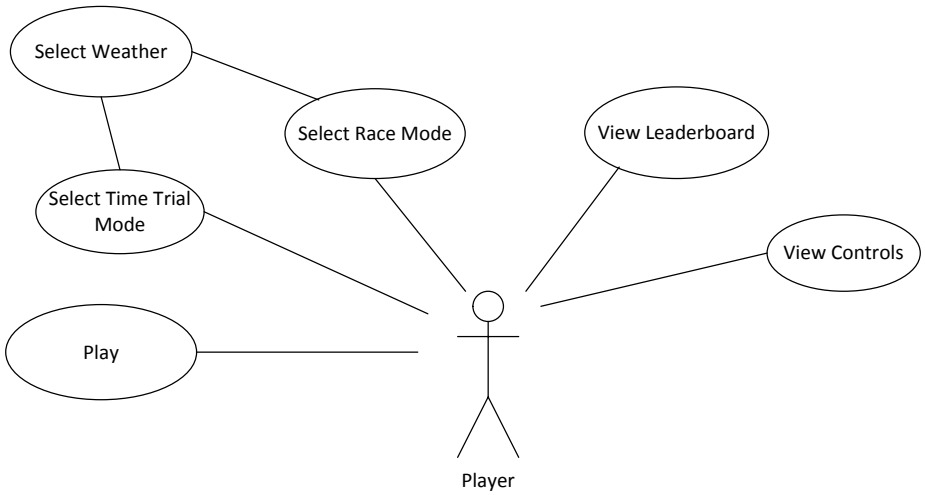
Race Mode



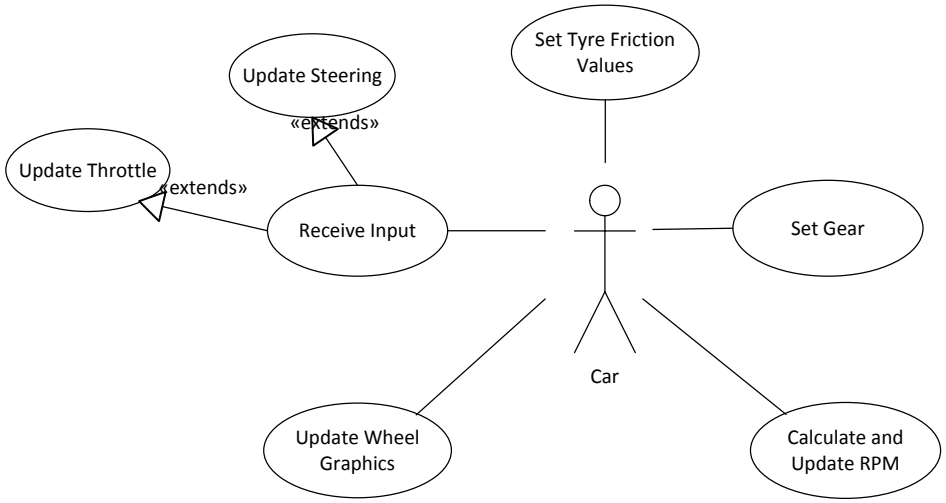
Wet Weather

Appendix D: Use Case Specifications/Diagrams

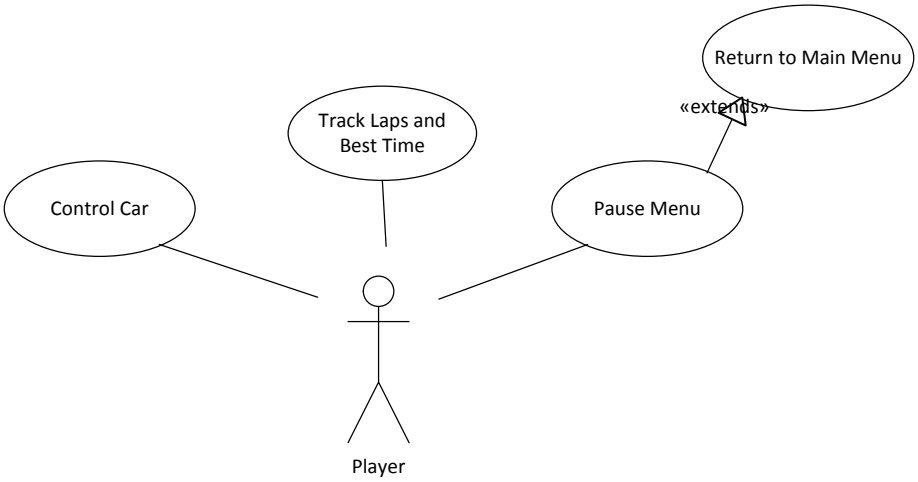
Main Menu



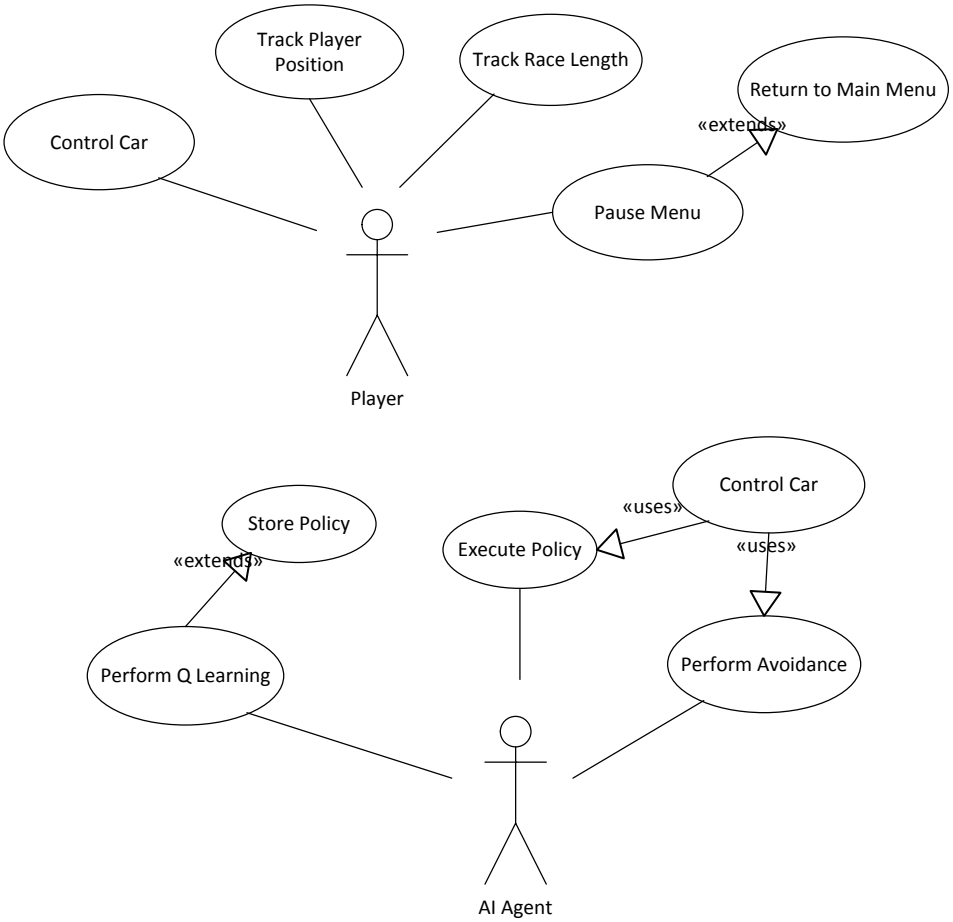
3.3.1.2 Car Controller



3.3.1.3 Time Trial Mode



3.3.1.4 Race Mode



Appendix E: Game Controls

The figure below shows the in-game controls for the game. In addition to these, the D-Pad and A button are used to navigate the games menus. The right analogue stick can also be used to look behind the car (when not using the Oculus Rift).

Appendix F: Pseudocode for Core Scripts

F.1 Car Controller

Developed using Unity official car tutorial: http://unity3d.com/gallery/demos/demo-projects

```
CarController()
-Initialize():
	-Generate wheel colliders
		-Assign friction curves
	-Set centre of mass

-Fixed Update():
	-Apply throttle/steering
	-Apply anti roll bars
	-Update wheel rotation graphics and suspension

-Update():
	-Update car/dash information
	-Check if handbrake on
```

```
Apply Throttle()
-Set dynamic wheel slip based on velocity
-Update centre of mass based on velocity
-Apply drag to limit overall top speed

-Set engine RPM
	-Using current wheel RPM divided by current gear ratio

-Calculate gears
	-Shift up/down if RPM is on threshold

-Apply motor torque to wheels
	-Engine torque * gear ratio * throttle value

-Apply steering based on steering value
```

F.2 Player Controller

```
PlayerController() : Car Controller
-Initialize():
	-Throttle/Steering = 0.0f
	-Initialize base

-Update():
	-DetectInput()

-DetectInput():
	-Throttle = LT/RT trigger value
	-Steering = Left analogue stick value
	-Handbrake Toggle = B button down/up
	-Reset car = back button
		-Resets car to nearest point on racing line, angled towards
		 next point along line
```

F.3 AI Controller (SteeringBehaviours)

```
SteeringBehaviours() : Car Controller
-Initialize():
       -Throttle/Steering = 0.0f
       -Initialize base
       -Detect racing line
       -Initialize percept and avoidance classes

-SteerToWaypoint():
       -waypointSteer = relative waypoint position.x / relative waypoint
       position magnitude
       -obstacleAvoidance = obstacleAvoidance.x /
       obstacleAvoidance.magnitude
       -wallAvoidance = wallAvoidance.x/ wallAvoidance.magnitude

       -Steering = waypointSteer + obstacleAvoidance + wallAvoidance;

-SetThrottle(float value) : Throttle = value
```

F.4 Q-Learning

```
QLearning_V1()
-Episodes = number of states * number of actions
-For each episode:
       -Reset car position/rotation to start
       -State = 0, NextState = State + 1
       -MaxState = highest state that not all actions have been evaluated
                   for

       -For each state below MaxState:
              -If tried all actions for state, use best action
              -Else select an untried action

              -While not at next state
                     -Apply action
                     -If crash, calculate and store negative reward and
                      update Q value for state-action, go to next episode

              -If at next state:
                     -If at MaxState:
                            -Calculate and store positive reward and
                              update Q value for state-action
                     -State++
```

```
QLearning_V2()
-Episodes = 5000
-For each episode:
	-Reset car position/rotation to start
	-State = 0, NextState = State + 1

	-For each state below StateCount:
		-Get non-negative action for state

		-While not at next state
			-Apply action
			-If crash, calculate and store negative reward and
			 update Q value for state-action, go to next episode

		-If at next state:
			-If at EndState:
				-Calculate and store positive reward *
				ReachedEndReward value and update Q value for
				state-action pair
			-Else:
				- Calculate and store Q score
			-State++
```

F.5 Obstacle Avoidance

```
ObstacleAvoidance()
-ObstacleArray = new array of obstacles
-DetectionBox = MinBoxLength scaled by car velocity

-Detect obstacles within box range:
	-For each car in the scene:
		-If difference in car positions < DetectionBoxLength
			-Add to ObstacleArray

-For each obstacle in ObstacleArray:
	-If obstacle is behind, discard
	-Calculate expanded radius
	-If obstacle's local X value is within expanded radius
		-Perform line/circle intersection test
	-If intersection point < distance to closest obstacle
		-Distance to closest obstacle = intersection point

-Force multiplier = multiplier based on velocity and proximity to obstacle

-Lateral avoidance force = closest obstacle radius – local x position of
closest obstacle * multiplier

-return avoidance force
```

F.6 Execute Policy

```
ExecutePolicy()
-Load policy file from project/build location
-State = Find closest waypoint
-While (State != MaxState)
	-Steer to next state
		-SteeringBehaviours.SteerToNextWaypoint()
	-Throttle to next state
		-SteeringBehaviours.SetThrottle(storedPolicy[state])
	-If (trigger next state)
		-State++
	-If (crashed and stationary for 2.5 seconds)
		-Reset car

	-If (State == MaxState) State = 0
```

F.7 Racing Line / Spline Creator

```
SplineCreator()
-Initialize:
	-Load defined waypoints
	-GenerateSpline(waypoints)
	- GenerateColliders()

-GenerateSpline(waypoints):
	-Interpolate waypoints using Catmull-Rom technique to generate
	spline points
	-Store points in container

-GenerateColliders():
	-Iterate through spline container
		-Create box collider at point,
			-Rotated along direction of line
```

F.8 Player Position Tracker

```
GetPlayerPosition()
-Position = number of cars
-For each AI car:
	-Does (AI lap number == player lap number)?
	-If no:
		-If (AI lap number > player lap number) player ahead, position--
		-If (AI lap number < player lap number) player behind, position++
	-If yes:
	-Does (AI last checkpoint == player last checkpoint)
		-If no:
			-If (AI checkpoint > player checkpoint) player ahead,
				position--
			-If (AI checkpoint < player checkpoint) player behind
				position++
		-If yes:
		-Does (AI current waypoint == player current waypoint)
			-If no:
				-If (AI checkpoint > player checkpoint) player ahead,
					position--
				-If (AI checkpoint < player checkpoint) player behind,
					position++
			-If yes:
				-If (AI dist to waypoint > player dist to waypoint)
					player ahead, position--
				-If (AI dist to waypoint < player dist to waypoint)
					player behind, position++
-Return Position
```

F.9 Scene Manager (Singleton)

```
SceneManager()
-Static instance

-Initialize:
	-Initialize FSM and States
	-Initialize score saving system

-Update:
	-Update FSM

-LoadSceneAsynchronously(string sceneName):
	-Loads specified scene asynchronously (used for loading screen
	transitions)
```

F.9 Finite State Machine and States

```
FiniteStateMachine()
-Initialize:
	-Initialize state container
	-Load starting state

-Update:
	-Check for state transitions
		-If so, call destructor of current state and initialiser of
		next state

-AddState(State state)
	-Adds state to container
```

```
State() : Abstract
-Abstract Enter, Exit, Update
-Container of transitions between states
-AddState()
	-Adds state to container
```

```
Transition()
-Contains next state and relevant transition
-Initialize:
	-Initializes next state/condition
```

F.9 Time Trial Mode Initialiser

```
TimeTrialInitializer()
-Initialize:
	-Get waypoints
	-Get car lap tracker component

-Update:
	-If current lap == 0:
		-Find first waypoint on track
		-Steer and accelerate at full speed towards waypoint
```

F.10 Race Mode Initialiser

```
RaceInitializer()
-Initialize:
	-Countdown timer = 5
	-Spawn cars on grid
	-Start countdown

-Update:
-If counting down:
	-Apply car handbrakes
-Else:
	-Disable car handbrakes

-Countdown():
	-Counts down from 5 seconds to 0
```

F.11 Lap Tracker

```
LapTracker()
-Initialize:
	-Initialize lap time container
	-Load checkpoints and waypoints

-Update:
	-If completed a lap (crossed start/finish line):
		-Add time to lap time container
		-Save to file

-OnTriggerEnter(Collision object):
	-If object == checkpoint:
		-if object == checkpoint 0
			-Lap complete
			-Current lap++, currentCheckpoint = 0
		-currentCheckpoint++

	-If object == waypoint:
		-Set current waypoint to object

-RecordLapTime():
	-lapTime += Time.deltaTime

-GetBestLap():
	-Iterate through lap time container
	-Return lowest time
```

F.12 Render Texture to Plane (Dashboard Information Example)

```
GearNumberIndicator() (Attached to plane on dashboard)
-Initialize:
	-Get player car controller
	-Initialize container of textures

-Update:
	-playerController.GetGear()
	-Render relevant texture to plane
		-By setting renderer material main texture
```

F.13 Car Effect Scripts

The scripts written to create tyre marks, tyre smoke and collision effects were implemented by following a set of tutorials found here: http://www.youtube.com/user/FlatTutorials.

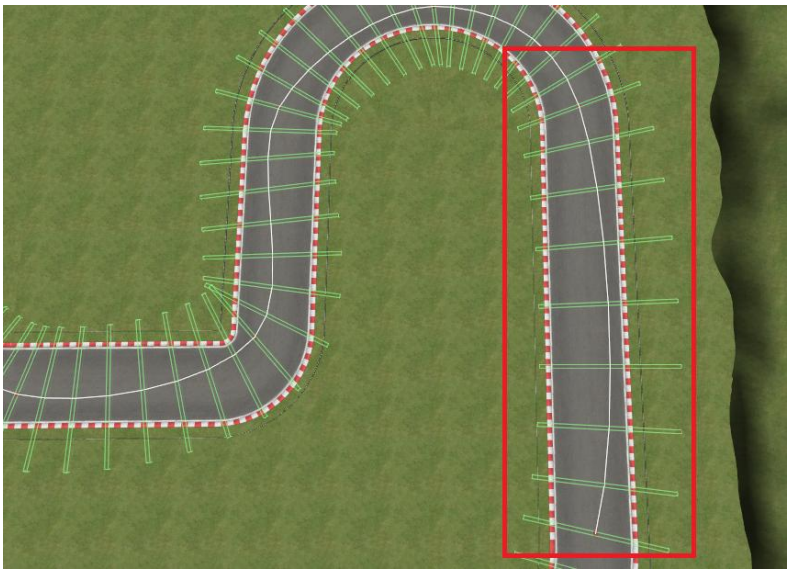
The shader scripts used for further visual effects can be found at the following:

- Wing Mirrors: http://wiki.unity3d.com/index.php?title=MirrorReflection2
- Cubemap Reflection: http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaderExamples.html
- Wet Surface: https://www.assetstore.unity3d.com/#/content/8849
- Rain: http://forum.unity3d.com/threads/78751-Rainscapes-Coming-Soon!

Appendix G: Q Table Comparison

This section highlights a larger portion of the Q Tables produced by each learning algorithm (at approximately 1000 iterations).The full Q Tables can be found in the "*ORGameData*" folder in the Build and Project folders labelled "*QDataV1*" and "*QDataV2*". The tables show that the second version of the algorithm produces a clearer results table and for the most part very similar outcomes as that of the first version despite their differing implementations.

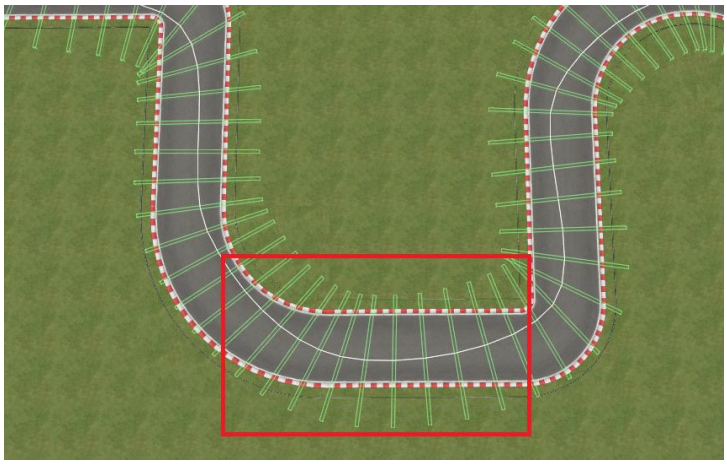
States 0-10:



Version 1	Version 2
State: 0, Action: 0, Value: 0	***State: 0, Action: 0, Value: 319130***
State: 0, Action: 1, Value: 0	State: 0, Action: 1, Value: 0
State: 0, Action: 2, Value: 0	State: 0, Action: 2, Value: 0
State: 0, Action: 3, Value: 0	State: 0, Action: 3, Value: 0
State: 0, Action: 4, Value: 0	State: 0, Action: 4, Value: 0
State: 0, Action: 5, Value: 0	State: 0, Action: 5, Value: 0
State: 0, Action: 6, Value: 0	State: 0, Action: 6, Value: 0
State: 0, Action: 7, Value: 0	State: 0, Action: 7, Value: 0
State: 0, Action: 8, Value: 0	State: 0, Action: 8, Value: 0
State: 1, Action: 0, Value: 3516.48169604905	***State: 1, Action: 0, Value: 319180***
State: 1, Action: 1, Value: 3516.46280036986	State: 1, Action: 1, Value: 0
State: 1, Action: 2, Value: 3516.25284837647	State: 1, Action: 2, Value: 0
State: 1, Action: 3, Value: 3513.92004842281	State: 1, Action: 3, Value: 0
State: 1, Action: 4, Value: 3488.00004863739	State: 1, Action: 4, Value: 0
State: 1, Action: 5, Value: 3200.00004768372	State: 1, Action: 5, Value: 0
State: 1, Action: 6, Value: -2000	State: 1, Action: 6, Value: 0
State: 1, Action: 7, Value: -2001	State: 1, Action: 7, Value: 0
State: 1, Action: 8, Value: -2002	State: 1, Action: 8, Value: 0
State: 2, Action: 0, Value: 28131.8683977212	***State: 2, Action: 0, Value: 2553794***
State: 2, Action: 1, Value: 28131.8671732812	State: 2, Action: 1, Value: 0
State: 2, Action: 2, Value: 28131.8535683924	State: 2, Action: 2, Value: 0
State: 2, Action: 3, Value: 28131.7024029589	State: 2, Action: 3, Value: 0
State: 2, Action: 4, Value: 28130.0227870117	State: 2, Action: 4, Value: 0
State: 2, Action: 5, Value: 28111.3603873825	State: 2, Action: 5, Value: 0
State: 2, Action: 6, Value: 27904.0003890991	State: 2, Action: 6, Value: 0
State: 2, Action: 7, Value: 25600.0003814697	State: 2, Action: 7, Value: 0
State: 2, Action: 8, Value: -2001	State: 2, Action: 8, Value: 0
State: 3, Action: 0, Value: 94945.0562142327	***State: 3, Action: 0, Value: 8619167***
State: 3, Action: 1, Value: 94945.0558423091	State: 3, Action: 1, Value: 0
State: 3, Action: 2, Value: 94945.0517098242	State: 3, Action: 2, Value: 0
State: 3, Action: 3, Value: 94945.0057933243	State: 3, Action: 3, Value: 0
State: 3, Action: 4, Value: 94944.4956099862	State: 3, Action: 4, Value: 0
State: 3, Action: 5, Value: 94938.8269061645	State: 3, Action: 5, Value: 0

State: 3, Action: 6, Value: 94875.8413074159 State: 3, Action: 7, Value: 94176.0013132095 State: 3, Action: 8, Value: 86400.0012874603	State: 3, Action: 6, Value: 0 State: 3, Action: 7, Value: 0 State: 3, Action: 8, Value: 0
State: 4, Action: 0, Value: 225054.948063366 State: 4, Action: 1, Value: 225054.94718177 State: 4, Action: 2, Value: 225054.93738625 State: 4, Action: 3, Value: 225054.828547139 State: 4, Action: 4, Value: 225053.619223671 State: 4, Action: 5, Value: 225040.182296094 State: 4, Action: 6, Value: 224890.88309906 State: 4, Action: 7, Value: 223232.003112793 State: 4, Action: 8, Value: 204800.003051758	***State: 4, Action: 0, Value: 20430691*** State: 4, Action: 1, Value: 0 State: 4, Action: 2, Value: 0 State: 4, Action: 3, Value: 0 State: 4, Action: 4, Value: 0 State: 4, Action: 5, Value: 0 State: 4, Action: 6, Value: 0 State: 4, Action: 7, Value: 0 State: 4, Action: 8, Value: 0
State: 5, Action: 0, Value: 439560.445436263 State: 5, Action: 1, Value: 439560.443714394 State: 5, Action: 2, Value: 439560.424582519 State: 5, Action: 3, Value: 439560.212006131 State: 5, Action: 4, Value: 439557.850046233 State: 5, Action: 5, Value: 439531.606047058 State: 5, Action: 6, Value: 439240.006052852 State: 5, Action: 7, Value: 436000.006079674 State: 5, Action: 8, Value: 400000.005960464	***State: 5, Action: 0, Value: 39903740*** State: 5, Action: 1, Value: 0 State: 5, Action: 2, Value: 0 State: 5, Action: 3, Value: 0 State: 5, Action: 4, Value: 0 State: 5, Action: 5, Value: 0 State: 5, Action: 6, Value: 0 State: 5, Action: 7, Value: 0 State: 5, Action: 8, Value: 0
State: 6, Action: 0, Value: 759560.449713862 State: 6, Action: 1, Value: 759560.446738473 State: 6, Action: 2, Value: 759560.413678593 State: 6, Action: 3, Value: 759560.046346594 State: 6, Action: 4, Value: 759555.96487989 State: 6, Action: 5, Value: 759510.615249316 State: 6, Action: 6, Value: 759006.730459328 State: 6, Action: 7, Value: 753408.010505676 State: 6, Action: 8, Value: 691200.010299683	***State: 6, Action: 0, Value: 68953700*** State: 6, Action: 1, Value: 0 State: 6, Action: 2, Value: 0 State: 6, Action: 3, Value: 0 State: 6, Action: 4, Value: 0 State: 6, Action: 5, Value: 0 State: 6, Action: 6, Value: 0 State: 6, Action: 7, Value: 0 State: 6, Action: 8, Value: 0
State: 7, Action: 0, Value: 1206153.8622771 State: 7, Action: 1, Value: 1206153.8575523 State: 7, Action: 2, Value: 1206153.80505443 State: 7, Action: 3, Value: 1206153.22174482 State: 7, Action: 4, Value: 1206146.74052686 State: 7, Action: 5, Value: 1206074.72699313 State: 7, Action: 6, Value: 1205274.57660903 State: 7, Action: 7, Value: 1196384.01668262 State: 7, Action: 8, Value: 1097600.01635551	***State: 7, Action: 0, Value: 109495949*** State: 7, Action: 1, Value: 0 State: 7, Action: 2, Value: 0 State: 7, Action: 3, Value: 0 State: 7, Action: 4, Value: 0 State: 7, Action: 5, Value: 0 State: 7, Action: 6, Value: 0 State: 7, Action: 7, Value: 0 State: 7, Action: 8, Value: 0
State: 8, Action: 0, Value: 28131.8685079208 State: 8, Action: 1, Value: 28131.8683977212 State: 8, Action: 2, Value: 28131.8671732812 State: 8, Action: 3, Value: 28131.8535683924 State: 8, Action: 4, Value: 28131.7024029589 State: 8, Action: 5, Value: 28130.0227870117 State: 8, Action: 6, Value: 28111.3603873825 State: 8, Action: 7, Value: 27904.0003890991 State: 8, Action: 8, Value: 25600.0003814697	***State: 8, Action: 0, Value: 2553794*** State: 8, Action: 1, Value: 0 State: 8, Action: 2, Value: 0 State: 8, Action: 3, Value: 0 State: 8, Action: 4, Value: 0 State: 8, Action: 5, Value: 0 State: 8, Action: 6, Value: 0 State: 8, Action: 7, Value: 0 State: 8, Action: 8, Value: 0
State: 9, Action: 0, Value: 35604.3960803373 State: 9, Action: 1, Value: 35604.3959408659 State: 9, Action: 2, Value: 35604.3943911841 State: 9, Action: 3, Value: 35604.3771724966 State: 9, Action: 4, Value: 35604.1858537448 State: 9, Action: 5, Value: 35602.0600898117 State: 9, Action: 6, Value: 35578.440490281 State: 9, Action: 7, Value: 35316.0004924536 State: 9, Action: 8, Value: 32400.0004827976	***State: 9, Action: 0, Value: 1766379*** State: 9, Action: 1, Value: 0 State: 9, Action: 2, Value: 0 State: 9, Action: 3, Value: 0 State: 9, Action: 4, Value: 0 State: 9, Action: 5, Value: 0 State: 9, Action: 6, Value: 0 State: 9, Action: 7, Value: 0 State: 9, Action: 8, Value: 0
State: 10, Action: 0, Value: 1098.90111359066 State: 10, Action: 1, Value: 1098.90110928598 State: 10, Action: 2, Value: 1098.9010614563 State: 10, Action: 3, Value: 1098.90053001533 State: 10, Action: 4, Value: 1098.89462511558 State: 10, Action: 5, Value: 1098.82901511765 State: 10, Action: 6, Value: 1098.10001513213 State: 10, Action: 7, Value: 1090.00001519918 State: 10, Action: 8, Value: 1000.00001490116	***State: 10, Action: 0, Value: 99709*** State: 10, Action: 1, Value: 0 State: 10, Action: 2, Value: 0 State: 10, Action: 3, Value: 0 State: 10, Action: 4, Value: 0 State: 10, Action: 5, Value: 0 State: 10, Action: 6, Value: 0 State: 10, Action: 7, Value: 0 State: 10, Action: 8, Value: 0

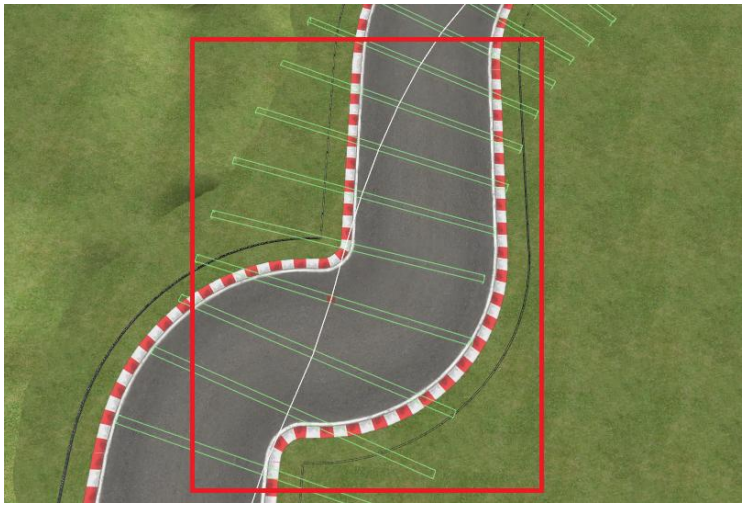
States 35-45:



Version 1	Version 2
State: 35, Action: 0, Value: 150769232.784638	***State: 35, Action: 0, Value: 1515044504***
State: 35, Action: 1, Value: 150769232.194037	State: 35, Action: 1, Value: -2147483648
State: 35, Action: 2, Value: 150769225.631804	State: 35, Action: 2, Value: -2147483648
State: 35, Action: 3, Value: 150769152.718103	State: 35, Action: 3, Value: -2147483648
State: 35, Action: 4, Value: 150768342.565858	State: 35, Action: 4, Value: -2147483648
State: 35, Action: 5, Value: 150759340.874141	State: 35, Action: 5, Value: -2147483648
State: 35, Action: 6, Value: 150659322.076128	State: 35, Action: 6, Value: -2147483648
State: 35, Action: 7, Value: 149548002.085328	State: 35, Action: 7, Value: -2147483648
State: 35, Action: 8, Value: 137200002.044439	State: 35, Action: 8, Value: -2147483648
State: 36, Action: 0, Value: 569670.337285396	***State: 36, Action: 0, Value: 51715263***
State: 36, Action: 1, Value: 569670.335053855	State: 36, Action: 1, Value: 0
State: 36, Action: 2, Value: 569670.310258945	State: 36, Action: 2, Value: 0
State: 36, Action: 3, Value: 569670.034759946	State: 36, Action: 3, Value: 0
State: 36, Action: 4, Value: 569666.973659917	State: 36, Action: 4, Value: 0
State: 36, Action: 5, Value: 569632.961436987	State: 36, Action: 5, Value: 0
State: 36, Action: 6, Value: 569255.047844496	State: 36, Action: 6, Value: 0
State: 36, Action: 7, Value: 565056.007879257	State: 36, Action: 7, Value: 0
State: 36, Action: 8, Value: 518400.007724762	State: 36, Action: 8, Value: 0
State: 37, Action: 0, Value: 178120441.941464	***State: 37, Action: 0, Value: 735520840***
State: 37, Action: 1, Value: 178120441.243722	State: 37, Action: 1, Value: -2147483648
State: 37, Action: 2, Value: 178120433.491027	State: 37, Action: 2, Value: -2147483648
State: 37, Action: 3, Value: 178120347.349972	State: 37, Action: 3, Value: -2147483648
State: 37, Action: 4, Value: 178119390.227135	State: 37, Action: 4, Value: -2147483648
State: 37, Action: 5, Value: 178108755.528813	State: 37, Action: 5, Value: -2147483648
State: 37, Action: 6, Value: 177990592.212761	State: 37, Action: 6, Value: -2147483648
State: 37, Action: 7, Value: 176677666.46363	State: 37, Action: 7, Value: -2147483648
State: 37, Action: 8, Value: 162089602.415323	State: 37, Action: 8, Value: -2147483648
State: 38, Action: 0, Value: 634725.283209963	***State: 38, Action: 0, Value: 57621021***
State: 38, Action: 1, Value: 634725.280723585	State: 38, Action: 1, Value: 0
State: 38, Action: 2, Value: 634725.253097158	State: 38, Action: 2, Value: 0
State: 38, Action: 3, Value: 634724.946136853	State: 38, Action: 3, Value: 0
State: 38, Action: 4, Value: 634721.53546676	State: 38, Action: 4, Value: 0
State: 38, Action: 5, Value: 634683.639131952	State: 38, Action: 5, Value: 0
State: 38, Action: 6, Value: 634262.568740318	State: 38, Action: 6, Value: 0
State: 38, Action: 7, Value: 629584.008779049	State: 38, Action: 7, Value: 0
State: 38, Action: 8, Value: 577600.008606911	State: 38, Action: 8, Value: 0
State: 39, Action: 0, Value: 4285.71434300356	***State: 39, Action: 0, Value: 813643***
State: 39, Action: 1, Value: 4285.71432621534	State: 39, Action: 1, Value: 0
State: 39, Action: 2, Value: 4285.71413967956	State: 39, Action: 2, Value: 0
State: 39, Action: 3, Value: 4285.71206705978	State: 39, Action: 3, Value: 0
State: 39, Action: 4, Value: 4285.68903795077	State: 39, Action: 4, Value: 0
State: 39, Action: 5, Value: 4285.43315895882	State: 39, Action: 5, Value: 0
State: 39, Action: 6, Value: 4282.5900590153	State: 39, Action: 6, Value: 0
State: 39, Action: 7, Value: 4251.00005927682	State: 39, Action: 7, Value: 0
State: 39, Action: 8, Value: 3900.00005811453	State: 39, Action: 8, Value: 0
State: 40, Action: 0, Value: 4395.60445436263	***State: 40, Action: 0, Value: 692704***

State: 40, Action: 1, Value: 4395.60443714394 State: 40, Action: 2, Value: 4395.60424582519 State: 40, Action: 3, Value: 4395.60212006131 State: 40, Action: 4, Value: 4395.57850046232 State: 40, Action: 5, Value: 4395.31606047058 State: 40, Action: 6, Value: 4392.40006052852 State: 40, Action: 7, Value: 4360.00006079674 State: 40, Action: 8, Value: 4000.00005960464	State: 40, Action: 1, Value: 0 State: 40, Action: 2, Value: 0 State: 40, Action: 3, Value: 0 State: 40, Action: 4, Value: 0 State: 40, Action: 5, Value: 0 State: 40, Action: 6, Value: 0 State: 40, Action: 7, Value: 0 State: 40, Action: 8, Value: 0
State: 41, Action: 0, Value: 4505.49456572169 State: 41, Action: 1, Value: 4505.49454807254 State: 41, Action: 2, Value: 4505.49435197082 State: 41, Action: 3, Value: 4505.49217306284 State: 41, Action: 4, Value: 4505.46796297388 State: 41, Action: 5, Value: 4505.19896198235 State: 41, Action: 6, Value: 4502.21006204173 State: 41, Action: 7, Value: 4469.00006231666 State: 41, Action: 8, Value: 4100.00006109476	***State: 41, Action: 0, Value: 776392*** State: 41, Action: 1, Value: 0 State: 41, Action: 2, Value: 0 State: 41, Action: 3, Value: 0 State: 41, Action: 4, Value: 0 State: 41, Action: 5, Value: 0 State: 41, Action: 6, Value: 0 State: 41, Action: 7, Value: 0 State: 41, Action: 8, Value: 0
State: 42, Action: 0, Value: 706015.395128755 State: 42, Action: 1, Value: 4615.38465900114 State: 42, Action: 2, Value: 4615.38445811645 State: 42, Action: 3, Value: 4615.38222606437 State: 42, Action: 4, Value: 4615.35742548544 State: 42, Action: 5, Value: 4615.08186349411 State: 42, Action: 6, Value: 4612.02006355494 State: 42, Action: 7, Value: 4578.00006383657 State: 42, Action: 8, Value: 4200.00006258488	***State: 42, Action: 0, Value: 2915413*** State: 42, Action: 1, Value: 0 State: 42, Action: 2, Value: 0 State: 42, Action: 3, Value: 0 State: 42, Action: 4, Value: 0 State: 42, Action: 5, Value: 0 State: 42, Action: 6, Value: 0 State: 42, Action: 7, Value: 0 State: 42, Action: 8, Value: 0
State: 43, Action: 0, Value: 812747.26361165 State: 43, Action: 1, Value: 812747.260427914 State: 43, Action: 2, Value: 812747.225053078 State: 43, Action: 3, Value: 812746.831999336 State: 43, Action: 4, Value: 812742.464735484 State: 43, Action: 5, Value: 812693.93958101 State: 43, Action: 6, Value: 812154.771191723 State: 43, Action: 7, Value: 806164.011241317 State: 43, Action: 8, Value: 739600.011020899	State: 43, Action: 0, Value: -2147483648 State: 43, Action: 1, Value: -2147483648 State: 43, Action: 2, Value: -2147483648 State: 43, Action: 3, Value: -2147483648 ***State: 43, Action: 4, Value: 104322946*** State: 43, Action: 5, Value: 0 State: 43, Action: 6, Value: 0 State: 43, Action: 7, Value: 0 State: 43, Action: 8, Value: 0
State: 44, Action: 0, Value: 299548135.872341 State: 44, Action: 1, Value: 299548134.698935 State: 44, Action: 2, Value: 299548121.661098 State: 44, Action: 3, Value: 299547976.796242 State: 44, Action: 4, Value: 299546367.186706 State: 44, Action: 5, Value: 299528482.636101 State: 44, Action: 6, Value: 299329765.404849 State: 44, Action: 7, Value: 297121796.143127 State: 44, Action: 8, Value: 272588804.06189	***State: 44, Action: 0, Value: 1712224774*** State: 44, Action: 1, Value: -2147483648 State: 44, Action: 2, Value: -2147483648 State: 44, Action: 3, Value: -2147483648 State: 44, Action: 4, Value: -2147483648 State: 44, Action: 5, Value: -2147483648 State: 44, Action: 6, Value: -2147483648 State: 44, Action: 7, Value: -2147483648 State: 44, Action: 8, Value: -2147483648
State: 45, Action: 0, Value: 320439564.723035 State: 45, Action: 1, Value: 320439563.467793 State: 45, Action: 2, Value: 320439549.520656 State: 45, Action: 3, Value: 320439394.552469 State: 45, Action: 4, Value: 320437672.683703 State: 45, Action: 5, Value: 320418540.808305 State: 45, Action: 6, Value: 320205964.412529 State: 45, Action: 7, Value: 317844004.432082 State: 45, Action: 8, Value: 291600004.345179	***State: 45, Action: 0, Value: 563919791*** State: 45, Action: 1, Value: -2147483648 State: 45, Action: 2, Value: -2147483648 State: 45, Action: 3, Value: -2147483648 State: 45, Action: 4, Value: -2147483648 State: 45, Action: 5, Value: -2147483648 State: 45, Action: 6, Value: -2147483648 State: 45, Action: 7, Value: -2147483648 State: 45, Action: 8, Value: -2147483648

States 92-98:



Version 1	Version 2
State: 92, Action: 0, Value: 3720439.61017253	State: 92, Action: 0, Value: -2147483648
State: 92, Action: 1, Value: 3720439.59559863	State: 92, Action: 1, Value: -2147483648
State: 92, Action: 2, Value: 3720439.43366644	State: 92, Action: 2, Value: -2147483648
State: 92, Action: 3, Value: 3720437.63441989	State: 92, Action: 3, Value: -2147483648
State: 92, Action: 4, Value: 3720417.64279131	State: 92, Action: 4, Value: -2147483648
State: 92, Action: 5, Value: 3720195.5135823	***State: 92, Action: 5, Value: 336690554***
State: 92, Action: 6, Value: 3717727.41123134	State: 92, Action: 6, Value: 0
State: 92, Action: 7, Value: 3690304.05145836	State: 92, Action: 7, Value: 0
State: 92, Action: 8, Value: 3385600.05044937	State: 92, Action: 8, Value: 0
State: 93, Action: 0, Value: 255963350.08605	***State: 93, Action: 0, Value: 730021813***
State: 93, Action: 1, Value: 255963350.08605	State: 93, Action: 1, Value: -2147483648
State: 93, Action: 2, Value: 255963350.08605	State: 93, Action: 2, Value: -2147483648
State: 93, Action: 3, Value: 255963350.08605	State: 93, Action: 3, Value: -2147483648
State: 93, Action: 4, Value: 255963350.08605	State: 93, Action: 4, Value: -2147483648
State: 93, Action: 5, Value: 255963350.08605	State: 93, Action: 5, Value: -2147483648
State: 93, Action: 6, Value: 2805597255.12183	State: 93, Action: 6, Value: -2147483648
State: 93, Action: 7, Value: 235114416.818647	State: 93, Action: 7, Value: -2147483648
State: 93, Action: 8, Value: 2573942438.35473	State: 93, Action: 8, Value: -2147483648
State: 94, Action: 0, Value: 2920734984.09786	***State: 94, Action: 0, Value: 531860033***
State: 94, Action: 1, Value: 2920734972.65661	State: 94, Action: 1, Value: -2147483648
State: 94, Action: 2, Value: 2920734845.53158	State: 94, Action: 2, Value: -2147483648
State: 94, Action: 3, Value: 2920733433.0312	State: 94, Action: 3, Value: -2147483648
State: 94, Action: 4, Value: 2920717738.5824	State: 94, Action: 4, Value: -2147483648
State: 94, Action: 5, Value: 2920543355.81592	State: 94, Action: 5, Value: -2147483648
State: 94, Action: 6, Value: 2918605769.49921	State: 94, Action: 6, Value: -2147483648
State: 94, Action: 7, Value: 2897077032.39744	State: 94, Action: 7, Value: -2147483648
State: 94, Action: 8, Value: 2657868839.60533	State: 94, Action: 8, Value: -2147483648
State: 95, Action: 0, Value: 3014945083.43703	***State: 95, Action: 0, Value: 617591234***
State: 95, Action: 1, Value: 3014944952.2115	State: 95, Action: 1, Value: -2147483648
State: 95, Action: 2, Value: 3014943494.15005	State: 95, Action: 2, Value: -2147483648
State: 95, Action: 3, Value: 3014927293.46711	State: 95, Action: 3, Value: -2147483648
State: 95, Action: 4, Value: 3014747285.87677	State: 95, Action: 4, Value: -2147483648
State: 95, Action: 5, Value: 3012747201.51651	State: 95, Action: 5, Value: -2147483648
State: 95, Action: 6, Value: 2990524041.70048	State: 95, Action: 6, Value: -2147483648
State: 95, Action: 7, Value: 250534000.87145	State: 95, Action: 7, Value: -2147483648
State: 95, Action: 8, Value: 2743600040.88283	State: 95, Action: 8, Value: -2147483648
State: 96, Action: 0, Value: 258523157.988677	***State: 96, Action: 0, Value: 850917217***
State: 96, Action: 1, Value: 258523157.988677	State: 96, Action: 1, Value: -2147483648
State: 96, Action: 2, Value: 258523157.988677	State: 96, Action: 2, Value: -2147483648
State: 96, Action: 3, Value: 258523157.988677	State: 96, Action: 3, Value: -2147483648
State: 96, Action: 4, Value: 258523157.988677	State: 96, Action: 4, Value: -2147483648
State: 96, Action: 5, Value: 2831519565.4139	State: 96, Action: 5, Value: -2147483648
State: 96, Action: 6, Value: 4048035.89578308	State: 96, Action: 6, Value: -2147483648
State: 96, Action: 7, Value: 4018176.05603027	State: 96, Action: 7, Value: -2147483648

State: 96, Action: 8, Value: 3686400.05493164	State: 96, Action: 8, Value: -2147483648
State: 97, Action: 0, Value: 4135824.2311098	State: 97, Action: 0, Value: -2147483648
State: 97, Action: 1, Value: 4135824.21490873	State: 97, Action: 1, Value: -2147483648
State: 97, Action: 2, Value: 4135824.03489692	State: 97, Action: 2, Value: -2147483648
State: 97, Action: 3, Value: 4135822.03476569	State: 97, Action: 3, Value: -2147483648
State: 97, Action: 4, Value: 4135799.811085	State: 97, Action: 4, Value: -2147483648
State: 97, Action: 5, Value: 4135552.88129677	State: 97, Action: 5, Value: -2147483648
State: 97, Action: 6, Value: 4132809.21695128	State: 97, Action: 6, Value: -2147483648
State: 97, Action: 7, Value: 4102324.05720365	***State: 97, Action: 7, Value: 374530681***
State: 97, Action: 8, Value: 3763600.05608201	State: 97, Action: 8, Value: 0
State: 98, Action: 0, Value: 3309686041.06936	***State: 98, Action: 0, Value: 885805638***
State: 98, Action: 1, Value: 3309684440.46779	State: 98, Action: 1, Value: -2147483648
State: 98, Action: 2, Value: 3309666656.00571	State: 98, Action: 2, Value: -2147483648
State: 98, Action: 3, Value: 3309469050.86914	State: 98, Action: 3, Value: -2147483648
State: 98, Action: 4, Value: 3307273438.21516	State: 98, Action: 4, Value: -2147483648
State: 98, Action: 5, Value: 299300593.392877	State: 98, Action: 5, Value: -2147483648
State: 98, Action: 6, Value: 3282877741.77712	State: 98, Action: 6, Value: -2147483648
State: 98, Action: 7, Value: 274904896.954835	State: 98, Action: 7, Value: -2147483648
State: 98, Action: 8, Value: 3011814444.87953	State: 98, Action: 8, Value: -2147483648

Appendix H: Q Learning Code

H.1 Q Learning Version 1

```
private IEnumerator TrainingV1()
    {
        Debug.Log("Training Begun");
        bool nextEpisode;
        int iteration = 0;
        bool initialNode = true;

        for (int i = 0; i < EPISODES; i++)
        {
            System.Random rand = new System.Random();
            nextEpisode = false;

            if (iteration >= splinePoints.Count - 1)
            {
                iteration = 0;
            }

            int state;

            if (iteration < splinePoints.Count - 1)
            {
                state = iteration;
            }
            else state = 0;
            state = 0;

            theCar.ResetCar(0, 1);
            theCar.ResetSteering(1);

            nextState = state + 1;
            int maxState = qStore.GetMaxState();
            int evalState = maxState - 1;

            //do max state check
            Debug.Log("MAX STATE " + maxState);

            for (int s = 0; s < maxState; s++)
            {
                //Start timer upon entering new state:
                float dt;
                float startTime = Time.time;

                nextState = state + 1;
                if (nextState > splinePoints.Count - 1)
                {
                    nextState = 0;
                }
                //select action for current state
                float[] throttleAction = actions.Actions;

                int actionIndex;
                float action;

                double this_Q;
                double max_Q;
                double new_Q;

                if (!qStore.DeterminedAllQValuesForState(state))
                {
                    Debug.Log("Find action for " + state);
                    actionIndex = qStore.GetUntriedAction(state);
                }
                else
                {
                    Debug.Log("Evaluated all actions for state: " + state);
                    actionIndex = qStore.GetBestAction(state);
                }

                if (state == 0) actionIndex = 0;

                action = throttleAction[actionIndex];

                Vector3 RelativePositionToState = transform.InverseTransformPoint(splinePoints[nextState].x,
transform.position.y, splinePoints[nextState].z);

                while (!Triggered)
                {
                    float timer = Time.time - startTime;
                    theCar.SetThrottle(action);
```

```
                    theCar.SteerToSplinePointNew(nextState);

                    Vector3 distanceVector = splinePoints[nextState] - splinePoints[state];

                    if (state > iteration)
                    {
                        if ((!theCarPercept.IsOnTrack) || (RelativePositionToState.magnitude >
distanceVector.magnitude * 3.0f) || (theCar.transform.InverseTransformDirection(theCar.rigidbody.velocity).z < 0.0f))
//|| (theCar.transform.rigidbody.velocity.magnitude < 0.5f))
                        {
                            theCar.SetThrottle(0.0f);
                            //GIVE REWARD
                            dt = Time.time - startTime;
                            reward = CalcReward(false, dt, state);
                            if (state == maxState - 1)  //only learn current evaluation state values
                            {
                                this_Q = qStore.GetQValue(state, actionIndex);
                                max_Q = qStore.GetMaxQValue(state);
                                new_Q = this_Q + ALPHA * (reward + GAMMA * max_Q - this_Q);
                                qStore.StoreQValue(state, actionIndex, (int)new_Q);
                            }
                            //next episode
                            nextEpisode = true;
                            break;
                        }

                    }
                    if (state == iteration)
                    {
                        if (timer >= 3.0f)
                        {
                            Debug.Log("TIMER");
                            if ((!theCarPercept.IsOnTrack) || (RelativePositionToState.magnitude >
distanceVector.magnitude * 3.0f) || (theCar.transform.rigidbody.velocity.magnitude < 0.5f))
                            {
                                theCar.SetThrottle(0.0f);
                                //GIVE REWARD
                                dt = Time.time - startTime;
                                reward = CalcReward(false, dt, state);
                                if (state == maxState - 1)  //only learn current evaluation state values
                                {
                                    this_Q = qStore.GetQValue(state, actionIndex);
                                    max_Q = qStore.GetMaxQValue(state);
                                    new_Q = this_Q + ALPHA * (reward + GAMMA * max_Q - this_Q);
                                    qStore.StoreQValue(state, actionIndex, (int)new_Q);
                                }
                                //next episode
                                nextEpisode = true;
                                break;
                            }
                        }
                    }
                    //break;
                    yield return new WaitForFixedUpdate();
                }
                if ((Triggered) && (!nextEpisode))
                {
                    if (state == 0)
                    {
                        state++;    //dont store q value that started on (always the same value)
                    }
                    else
                    {
                        if (state == evalState)  //only learn current evaluation state values
                        {
                            dt = Time.time - startTime;
                            if (qStore.deltaTimeStore[state] == 0)
                            {
                                qStore.AddDeltaTime(state, dt);
                            }
                            if (qStore.deltaTimeStore[state] > dt)
                            {
                                qStore.AddDeltaTime(state, dt);
                            }
                            reward = CalcReward(true, dt, state);
                            //calc Q value
                            this_Q = qStore.GetQValue(state, actionIndex);
                            max_Q = qStore.GetMaxQValue(state);
                            new_Q = this_Q + ALPHA * (reward + GAMMA * max_Q - this_Q);
                            qStore.StoreQValue(state, actionIndex, new_Q);
                            nextEpisode = true;
                        }
                        state++;    //Go to next state
                    }
                    Triggered = false;
```

```
                }
                if (nextEpisode)    //Break out of episode to reset car to start (only when crashed)
                {
                    Triggered = false;
                    nextEpisode = false;
                    break;
                }
            }

        }   //end of for loop
        Debug.Log("End of Training V1");
        isTraining = false;
        yield return new WaitForFixedUpdate();
    }
```

H.2 Q Learning Version 2

```
private IEnumerator TrainingV2()
    {
        bool nextEpisode;
        Debug.Log("Start Training V2");
        for (int i = 0; i < NUM_OF_EPISODES; i++)
        {
            nextEpisode = false;
            Debug.Log("Episode: " + i);
            //Reset car to start for beginning of episode
            theCar.ResetCar(0, 1);
            theCar.ResetSteering(1);

            int state = 0;
            nextState = state + 1;

            for (int st = 0; st < splinePoints.Count - 1; st++)//while haven't reached goal
            {
                //Timing variables
                float startTime = Time.time;
                float dt;

                nextState = state + 1;

                //Action variables
                float[] throttleAction = actions.Actions;
                int actionIndex = qStore.GetBestAction(state);
                float action = throttleAction[actionIndex];

                //Q Learning variables
                double this_Q;
                double max_Q;
                double new_Q;

                Vector3 RelativePositionToState = transform.InverseTransformPoint(splinePoints[nextState].x,
transform.position.y, splinePoints[nextState].z);
                //while not at next state
                while (!Triggered)
                {
                    float timer = Time.time - startTime;
                    theCar.SetThrottle(action);
                    theCar.SteerToSplinePointNew(nextState);
                    Vector3 distanceVector = splinePoints[nextState] - splinePoints[state];
                    if (state == 0)
                    {
                        if (timer >= 3.0f)
                        {

                            if ((!theCarPercept.IsOnTrack) || (RelativePositionToState.magnitude >
distanceVector.magnitude * 3.0f) || (theCar.transform.rigidbody.velocity.magnitude < 0.5f) ||
(theCar.transform.InverseTransformDirection(theCar.rigidbody.velocity).z < 0.0f))
                            {
                                theCar.SetThrottle(0.0f);
                                //GIVE REWARD
                                dt = Time.time - startTime;
                                reward = CalcReward(false, dt, state);
                                this_Q = qStore.GetQValue(state, actionIndex);
                                max_Q = qStore.GetMaxQValue(state);
                                new_Q = this_Q + ALPHA * (reward + GAMMA * max_Q - this_Q);
                                qStore.StoreQValue(state, actionIndex, (int)new_Q);
                                //next episode
                                nextEpisode = true;
                            }
                        }
                    }
                    else
```

```
                    {
                        //If off track, going backwards, or 3* further away from next state than at the start, assign
negative reward for action:
                        if ((!theCarPercept.IsOnTrack) || (RelativePositionToState.magnitude >
distanceVector.magnitude * 3.0f) || (theCar.transform.InverseTransformDirection(theCar.rigidbody.velocity).z < 0.0f)
)
                        {
                            //Negative reward
                            theCar.SetThrottle(0.0f);
                            //GIVE REWARD
                            dt = Time.time - startTime;
                            reward = CalcReward(false, dt, state);
                            this_Q = qStore.GetQValue(state, actionIndex);
                            max_Q = qStore.GetMaxQValue(state);
                            new_Q = this_Q + ALPHA * (reward + GAMMA * max_Q - this_Q);
                            qStore.StoreQValue(state, actionIndex, (int)new_Q);
                            //next episode
                            nextEpisode = true;
                        }
                    }
                    if (nextEpisode)
                    {
                        Triggered = false;
                        nextEpisode = false;
                        break;
                    }
                    yield return new WaitForFixedUpdate();
                }

                //If trigger next state,
                if ((Triggered) && (!nextEpisode))
                {
                    dt = Time.time - startTime;
                    //If at end state, assign huge reward
                    if (state == splinePoints.Count - 2)
                    {
                        //REWARD!
                        reward = CalcReward(true, dt, state) * 100000;
                        this_Q = qStore.GetQValue(state, actionIndex);
                        max_Q = qStore.GetMaxQValue(state);
                        new_Q = this_Q + ALPHA * (reward + GAMMA * max_Q - this_Q);
                        //Change the way store:
                        qStore.StoreQValue(state, actionIndex, (int)new_Q);
                    }
                    else
                    {
                        reward = CalcReward(true, dt, state);
                        this_Q = qStore.GetQValue(state, actionIndex);
                        max_Q = qStore.GetMaxQValue(state);
                        new_Q = this_Q + ALPHA * (reward + GAMMA * max_Q - this_Q);
                        //Change the way store:
                        qStore.StoreQValue(state, actionIndex, (int)new_Q);
                    }
                    nextEpisode = false;
                    state++;
                    Triggered = false;
                }
                if (nextEpisode)
                {
                    Triggered = false;
                    //isTriggered = false;
                    nextEpisode = false;
                    break;
                }
            }
        }
        Debug.Log("End of Training V2");
        isTraining = false;
        yield return new WaitForFixedUpdate();
    }
```

H.3 Reward Function

```
private double CalcReward(bool goodMove, float dt, int state)
{
    double newReward = 0;
    if (!goodMove)    //have gone off track,
    {
       // Debug.Log("NEG REWARD OFF TRACK");
        if (theCar.transform.InverseTransformDirection(theCar.rigidbody.velocity).z < 0.0f)
        {
            newReward -= penaltyValue;
        }
        newReward -=  penaltyValue;
        Vector3 posDifference = theCar.transform.position - splinePoints[state + 1];
        newReward -= (posDifference.magnitude * 2.0f);
    }
    else  //haven't gone off track
    {
        double timeBonus = 1; // = 1 / dt;
        bool storeTime = false;
        if (dt < qStore.GetDeltaTime(state))
        {
            storeTime = true;
           // timeBonus = 2;
        }

        newReward += rewardValue;// +timeBonus;
        float posDifference = Vector3.Distance(theCar.transform.position, splinePoints[state + 1]);
        float dist = Vector3.Distance(splinePoints[state], splinePoints[state + 1]);
        if (posDifference < dist * 0.5f)
        {
            newReward *= 2.0f;
            if (state != 0) newReward *= state;
            if (dt < qStore.GetDeltaTime(state))
            {
                storeTime = true;
            }
        }

        if (posDifference <= dist * 0.25f)
        {
            newReward *= 4.0f;
            if (state != 0) newReward *= state;

            if (dt <= qStore.GetDeltaTime(state))
            {
                storeTime = true;
            }
        }

        if (posDifference <= dist * 0.1f)
        {
            newReward *= 8.0f;
            if (state != 0) newReward *= state;

            if (dt <= qStore.GetDeltaTime(state))
            {
                storeTime = true;
            }
        }
        if (storeTime)
        {
            qStore.AddDeltaTime(state, dt);
        }

        newReward *= timeBonus;
    }
    return newReward;
}
```

H.4 Data Structures

H.4.1 QStore

```
public class QStore
{
    public double[,] q; //state, action = reward
    public Dictionary<int, int>[] qStore;  //State<action, reward>
    public bool[] HaveBestAction;

    public float[] deltaTimeStore;

    public void AddDeltaTime(int state, float dt)
    {
        deltaTimeStore[state] = dt;
    }

    public float GetDeltaTime(int state)
    {
        return deltaTimeStore[state];
    }

    private int stateCount;
    private int actionCount = 9;
    public QStore(int stateCount)
    {
        this.stateCount = stateCount;
        Initialize();
    }

    private void Initialize()
    {
        q = new double[stateCount, actionCount];
        deltaTimeStore = new float[stateCount];
        HaveBestAction = new bool[stateCount];

        for (int i = 0; i < stateCount; i++)
        {
            for (int j = 0; j < actionCount; j++)
            {
                q[i, j] = 0;
                deltaTimeStore[i] = 0.0f;
                HaveBestAction[i] = false;

            }
        }
    }

    public double GetQValue(int state, int actionIndex)
    {
        return q[state, actionIndex];
    }

    public double GetMaxQValue(int state)
    {
        return GetQValue(state, GetBestAction(state));
    }

    public bool DeterminedAllQValuesForState(int state)
    {
        if (state == 0) return true;
        for (int i = 0; i < actionCount; i++)
        {
            if (q[state, i] == 0)   //hasnt been assigned a value (other than the 0 at initialization)
            {
                return false;
            }
        }
        return true;
    }

    //Will always try and get fastest non negative action (as 0 is full throttle!)
    public int GetNonNegativeAction(int state)
    {
        int action = 0;
        for (int i = 0; i < actionCount; i++)
        {
            if (q[state, i] >= 0)
            {
                action = i;
                break;
            }
        }
```

```
        return action;
    }

    //not returning properly
    public int GetUntriedAction(int state)
    {
        int action = 0;
        for (int i = 0; i < actionCount; i++)
        {
            if (q[state, i] == 0)   //Q hasnt been assigned a value (other than the 0 at initialization)
            {
                action = i;
            }
        }
        return action;
    }

    public int GetBestAction(int state)
    {
        double highestQ = 0;
        int action = 0;
        for (int i = 0; i < actionCount; i++)
        {
            if (q[state, i] > highestQ)
            {
                highestQ = q[state, i];
            }
        }
        for (int j = 0; j < actionCount; j++)
        {
            if (q[state, j] == highestQ)
            {
                action = j;
                break;
            }
        }
        return action;
    }

    public int GetMaxState()
    {
        int maxState = 0;
        for (int i = 0; i < stateCount; i++)
        {
            if (!DeterminedAllQValuesForState(i))
            {
                return i + 1;
            }
        }
        return maxState;
    }

    public int GetBestActionForPolicy(int state)
    {
        int action = 0;
        double highestQ = 0;
        for (int i = 0; i < actionCount; i++)
        {
            if (q[state, i] > highestQ)
            {
                highestQ = q[state, i];
                action = i;
            }
        }
        return action;
    }

    public void StoreQValue(int state, int action, double reward)
    {
        q[state, action] = reward;
    }
}
```

H.4.2 QActions

```
public class QActions
{
    public float[] Actions { get; private set; }
    private int NUM_OF_ACTIONS = 9;

    public QActions()
    {
        InitializeActions();
    }

    private void InitializeActions()
    {
        Actions = new float[NUM_OF_ACTIONS];
        Actions[0] = 1.0f;
        Actions[1] = 0.75f;
        Actions[2] = 0.5f;
        Actions[3] = 0.25f;
        Actions[4] = 0.0f;
        Actions[5] = -0.25f;
        Actions[6] = -0.5f;
        Actions[7] = -0.75f;
        Actions[8] = -1.0f;
    }
}
```

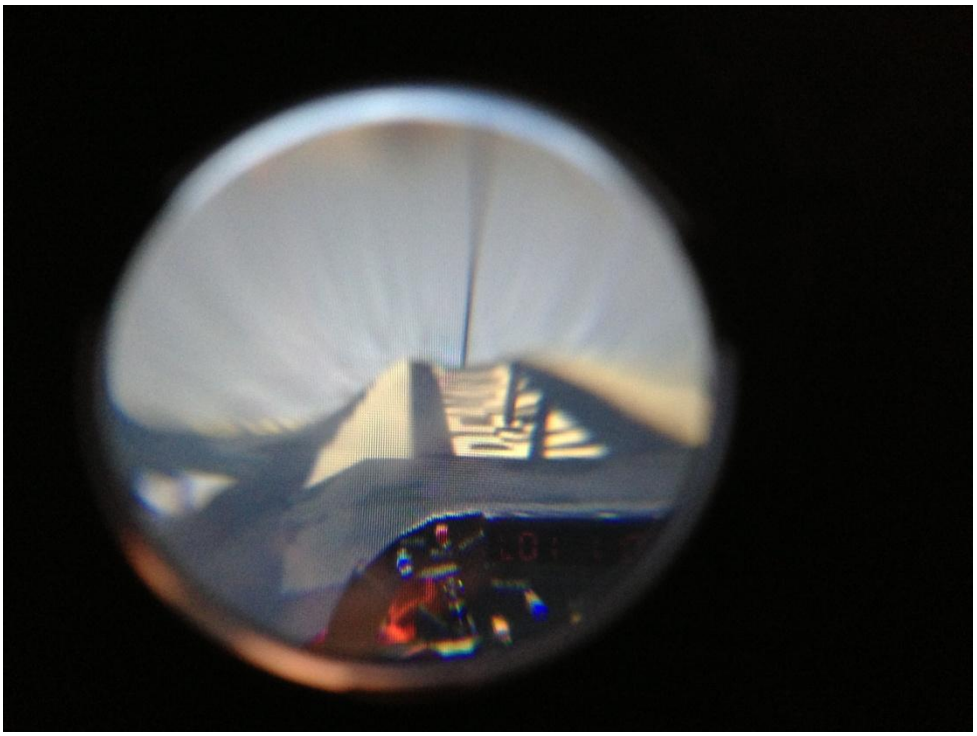
Appendix I: Oculus Rift Integration

The figures below demonstrate how the Oculus Rift works in the game.

Please visit http://www.youtube.com/watch?v=UzAuJ5bIHj8 for a short video demonstrating the integration.

The game in use with the Oculus Rift

View of the Oculus Rift with the lens off/on.



View through the lens of the Oculus Rift.

Appendix J: Survey

Rift Racer Survey

1. How well did the car control?

Very badly	Badly	Average	Well	Very well
○	○	○	○	○

2. How much improved/worse was the game with the Oculus Rift?

Much worse	Worse	Same	Better	Much better
○	○	○	○	○

3. Did you feel dizzy/sick after playing with the Oculus Rift?

No	A bit	Yes
○	○	○

4. How realistic was the AI behaviour?

Very bad	Poor	Average	Good	Very
○	○	○	○	○

5. Any suggestions to improve the game?

6. Additional comments

URL: http://www.surveymonkey.com/s/KTYV2ZL

Appendix K: Survey Results

1. How well did the car control?

Option	Response
Very badly	0.0%
Badly	16.7%
Average	33.3%
Well	50.0%
Very well	0.0%

2. How much improved/worse was the game with the Oculus Rift?

Option	Response
Much worse	0.0%
Worse	0.0%
Same	16.7%
Better	50.0%
Much better	33.3%

3. Did you feel dizzy/sick after playing with the Oculus Rift?

Option	Response
No	50%
A bit	50%
Yes	0.0%

4. How realistic was the AI behaviour?

Option	Response
Very bad	0.0%
Poor	16.7%
Average	50.0%
Good	33.3%
Very good	0.0%

5. Suggestions to improve the game:

- Gears sometimes feel clunky
- More tracks and dynamic weather effects
- More game modes and race tracks
- Improve AI, improve handling, add multiplayer mode
- An option to shift the camera angle higher up to assist learning
- Steering wheel controls would be good

6. Additional comments:

- Bad car control probably down to the user, not the game.
- Great sense of immersion with the Rift.

Appendix L: Installation Guide

H.1 DVD/USB Contents

The DVD/USB contains several folders:

Project Build
Contains the executable files needed to play the game. Select the "Build.exe" file to start the game. Ensure the entire folder hierarchy is copied across if copying the files from the DVD to a local destination.

Project Documentation
Contains a pdf and docx of the project report.

Source Code
Contains the C# scripts and shaders used in the project. They are organized in sub-folders based on their function.

Unity Project
This folder contains the full Unity 4.X project. This contains the files and assets used to create the project as used throughout development. Ensure the entire folder hierarchy is copied across if copying the files from the DVD to a local destination.

H.2 Launch Build

Perform the following steps to launch the final project build:

1. Open the DVD contents
2. Select the "Project Build" folder
3. Select the "*Build .exe*" file
4. Select the resolution (recommended resolution is 1280x800 to work properly with the Oculus Rift) and quality
5. Press *Play* and the game will launch

H.3 Launch in Unity

Perform the following steps to open the project in Unity:

1. Open Unity 4.X Pro (Pro is required to function correctly due to use of Oculus Rift and shader effects)
2. Navigate to *File -> Open Project* and navigate to the project location
3. Select the folder named "Unity Project/OR_Racing_Game" then press *OK*
4. Unity will then load the project (this may take a few minutes if it is the first time opening the project as Unity needs to compile the files and assets)

The Q-Learning and AI scenes are located in: *Scenes->TestScenes* in the *Project* window and are named:

- TestScene_AI_QLearn
- TestScene_AI_QLearnNewSpline
- TestScene_AI_ObstacleAvoidance
- TestScene_AI_LapTimer (scene used to generate results used in this project)

IMPORTANT: The game requires an XBOX 360 controller to play/navigate the menus properly! (Ensure it is switched ON before launching the build!)

Appendix M: Academic Paper

An academic paper was written for this project with the aim of submitting the results of the project to the IEEE Conference on Computational Intelligence and Games. See the following pages for the academic paper.

Implementing Racing AI using Q-Learning and Steering Behaviours

Blair Trusler
School of Informatics
City University London
Northampton Square, London, UK
Blair.Trusler.1@city.ac.uk

Dr. Chris Child
School of Informatics
City University London
Northampton Square, London, UK
C.Child@city.ac.uk

***Abstract* – Artificial intelligence has become a fundamental component of modern computer games as developers are producing ever more realistic experiences. This is particularly true of the racing game genre in which AI plays a fundamental role. Reinforcement learning techniques, notably Q-Learning (QL), have been growing as feasible methods for implementing AI in racing games in recent years. The focus of this research is on implementing QL to create a policy which the AI agents to utilise in a racing game using the Unity 3D game engine. QL is used (offline) to teach the agent appropriate throttle values around each part of the circuit whilst the steering is handled using a predefined racing line. Two variations of the QL algorithm were implemented to examine their effectiveness. The agents also make use of Steering Behaviours (including obstacle avoidance) to ensure that they can adapt their movements in real-time against other agents and players. Initial experiments showed that both types performed well and produced competitive lap times when compared to a player.**

Keywords – Q-Learning; reinforcement learning; steering behaviours; artificial intelligence; computer games; racing game; Unity

1. INTRODUCTION

Reinforcement learning (RL) techniques such as Q-Learning (QL, Watkins 1989) [1] have grown in popularity in games in recent years. The drive for more realistic artificial intelligence (AI) has increased commensurably alongside the high fidelity of experience which is now possible with modern hardware. An example of a modern commercial game making heavy use of RL techniques is the *Forza Motorsport* series [2] with its *Drivatar* [3][4] system. RL has also been used in games such as *Halo 3* [5] for matchmaking purposes [4]. RL can produce an effective AI controller whilst removing the need for a programmer to hard-code the behaviour of the agent.

The racing game used for performing the QL experiments was built using the Unity game engine [5]. The game was built as a side-project in conjunction with this research. The cars in the game were created so that the throttle and steering values could be easily manipulated to control the car.

The first challenge when considering implementing RL is to determine how to represent and simplify the agent's state representation of the game world in an effective way to use as input for the algorithm. The information needs to be abstracted to a high level in order to ensure that only necessary details are provided. This allows the agent to learn efficiently and to ensure the resulting policy is not tainted or biased in any way. Two versions of the QL algorithm were implemented; an iterative approach and a traditional RL approach.

The results from the experiments demonstrate that when combined with steering behaviours both QL implementations produced an effective AI controller that could complete competitive lap times.

2. BACKGROUND

2.1 *Steering Behaviours*

The concept of steering behaviours (SBs) was first introduced by Craig Reynolds (1999) [6]. SBs provide a mechanism of control for autonomous game agents. Reynolds proposed myriad behaviours which could be used independently of one another or holistically to achieve different behaviours.

There are three relevant SBs for this project; seek, obstacle avoidance and wall avoidance. Seek simply aims and propels the agent towards a specific point in the world. This is achieved by creating a force which is calculated by subtracting the desired velocity vector from the agent to the desired position by the current velocity vector.

Obstacle and wall avoidance are not a focus of this paper. However they were used to perform real-time avoidance techniques during the game when multiple agents

were in the scene. The seek behaviour however is an important behaviour as it allowed for the cars to steer towards the points along the racing line when performing the learning process and executing the learnt policy.

2.2 *Reinforcement Learning*

Reinforcement learning is the method for teaching an AI agent to take actions in a given scenario. The goal is to maximise the cumulative reward, known as the *utility* (Sutton and Barto, 1988) [7]. The result of the RL process is a policy which provides the agent a roadmap of how to perform optimally. The RL process can be performed *online* or *offline*.

Online learning is the process of teaching the AI agent in real-time. Offline learning involves teaching the agent before releasing the game or software. Both methods have their merits and issues. For several reasons the offline version is most commonly used when RL is applied to games. Primarily, it ensures that the agent will behave as expected when the game is finished. It also means there is less computational expense in real-time as the AI is behaving based on a saved policy and does not need to perform as many calculations in real-time.

The offline RL process works by performing a large number of iterations (*episodes*) of a simulation in order to build up a data store of learned Q values relative to their state-action combination.

2.3 *Q-Learning*

Q-Learning is one of the most commonly used forms of RL and is a form of temporal difference learning (Sutton and Barto, 1988) [7]. QL is used to find the best action-selection policy for a finite number of states. It works by assigning utility values to state-action pairs based on previous actions which have led to a goal state. As the number of episodes increases, the utility estimates and predictions improve and become more reliable.

A *state* can comprise of any piece of information from the agent's environment (for example a position on a grid). An *action* is the operation that the agent can perform at each state (for example move left or move right). The action selection policy is a key component to the learning process. The two common types of policies are *greedy* and *ε-greedy* (Sutton and Barto, 1988) [7]. Greedy policies always choose the optimal available action according to the current utility estimates. ε-greedy policies in contrast have a small probability of selecting a random action to explore instead of choosing the greedy option. A third type of policy for action selection is known as *softmax* selection [8]. Softmax is very similar to ε-greedy but its goal is to try "better" actions more often.

The QL formula is performed upon reaching a state. The QL formula is defined as follows [7]:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(r + \gamma \max_{a'}(Q(s', a')))$$

Where:

- Q(s, a) – Q value of the current state-action pair
- Q(s', a') – Q value of the next state-action pair
- r – reward value associated with next state
- α – learning rate parameter
- γ – discount value parameter

The learning rate and discount value parameters are crucial in defining the learning process. The learning rate determines to what extent newly acquired information will override the previously stored information. A learning rate value of 0 will mean that the agent will not learn anything whilst a rate of 1 means that the agent will only consider the most recently acquired data. The discount parameter defines the importance of future rewards to the agent. A factor of 0 creates a short-sighted agent which only considers current rewards, whilst a factor of 1 ensures the agent will aim for the highest possible long-term reward.

2.4 *Q-Learning in Games*

Patel et al (2011) [9] used QL to create an AI agent for the popular first-person shooter game *Counter-Strike* [10]. Their paper discussed traditional AI techniques, including decision trees and finite state machines (FSMs). These techniques can be very time consuming for game developers when fine tuning the AI agents to achieve the desired behaviour and interactions with the game world. The solution by Patel et al was to use Q-learning to train a simple AI agent in order to teach it how to fight and plant a bomb in the correct location. A higher reward value was assigned to the AI if it accomplished the goal of the game. For example planting the bomb produced a higher reward than killing an enemy. Their results showed that the Q-learning bots performed competitively against the traditionally programmed bots. However, they did note that this was not tested against players. This could possibly identify further issues that would need to be resolved in the learning and reward allocation algorithm.

A popular commercial racing game that makes heavy use of reinforcement learning is the Forza series. Forza makes use of "Drivatar" technology [3]. The

development team created a database of pre-generated racing lines for every corner on a race track (several slightly different lines per corner). For example, some racing lines will be optimal whilst others may go wide and miss the apex of the corner. The agent uses QL to learn the appropriate throttle values to follow each racing line as fast as possible. The cars also learn various overtaking manoeuvres at each part of the track. This learning process is performed offline and stored (before the game is shipped). During a race, the racing lines at each corner are switched to vary the behaviour of the AI car.

This approach meant that the programmers were not required to hard-code the values for each track and corner and produced a reusable and effective tool for creating AI agents for each type of vehicle. This technique has resulted in the Forza series having one of the most advanced AI systems in the racing game market today [11].

3. IMPLEMENTING Q-LEARNING

This section will explain how the QL concepts were applied to the racing game to produce a policy that could be use by the AI agents. The QL process was designed to take place offline in order to achieve a high quality controller and to ensure the AI would not impact the overall performance of the game itself.

3.1 *Game World Representation*

The first challenge was converting the three dimensional game world into a series of states for the algorithm to interpret. This was achieved in two steps. Firstly, a racing line was generated by positioning waypoints along the race track and creating a Catmull-Rom spline [12] by interpolating between these points.

The second step after generating the racing line was to create the states. A state was defined as a track segment (each point along the racing line). The region was implemented by placing a simple box collider at each point as shown in Figure 1. The collider width was equal to that of the race track width and rotated based on the direction of the spline. The quality of the state is evaluated based on the agent's proximity to the centre of the racing line and time taken to reach the state.

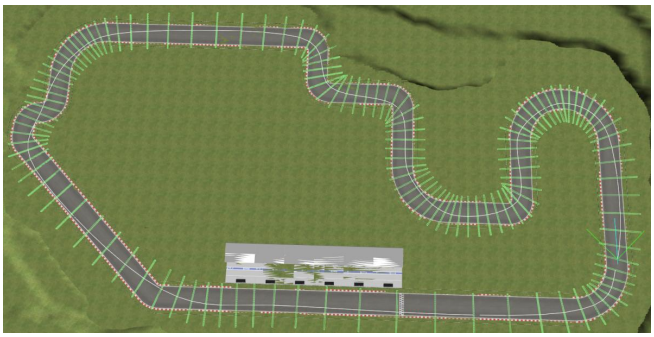


Figure 1 Q States in Game World

3.2 *Discrete Action Space*

It was decided to focus the QL on learning the cars throttle values whilst using the racing line to generate the appropriate steering values. This helped to reduce the action space to an appropriate size in order to minimise the number of iterations required to perform the learning process.

The action space was set to nine evenly spaced throttle values ranging from +1.0 to -1.0 (where +1.0 represents full throttle and -1.0 represents full braking or reversing). The full action table is highlighted in Table 1.

Action	Throttle Value
0	1.00
1	0.75
2	0.50
3	0.25
4	0.00
5	-0.25
6	-0.50
7	-0.75
8	-1.00

Table 1: Action Table

3.3 *QStore Data Structure*

A data structure (named the *QStore*) was implemented to store all of the data required by the learning algorithm. The QStore maintained a two-dimensional array of doubles. The first dimension in the array represented the state values whilst the second dimension represented the action values. This allowed for the Q value for each state-action pair to be easily stored and accessed as shown below:

```
double q[state, action] = QValue
```

The QStore also implemented several useful methods for obtaining relevant information from the array. It provided methods for getting and setting the Q values, obtaining an untried action and checking whether Q values had been generated for each state. The key method however was to get the best action available for a given state. This

was required for the learning algorithm to select an action. It was particularly useful for creating the policy file at the end of the learning process. Algorithm 1 outlines the process which was implemented in order to retrieve the best action available from the data structure.

```
GetBestAction(state)
  -HighestQ = 0, Action = 0
  -for each action, i:
       -if q[state, i] > HighestQ
              -HighestQ = q[state, i]
  -for each action, i:
       -if q[state, i] == HighestQ
              -Action = i;
  -return Action
```

Algorithm 1: GetBestAction returns the best action available for a given state

The *GetBestAction* function was essential for utilising a greedy action-selection policy as it provided the highest valued action upon each selection stage.

3.4 *Q-Learning Algorithm*

As previously mentioned two versions of the QL algorithm were implemented. Both versions are very similar in nature but with some key differences as highlighted in the following sections. The algorithm works by applying each action (throttle values) at each state on the track. A reward was calculated if the car reached or did not reach the next state and the QL formula was calculated and stored. Both versions used the greedy action selection policy.

The action policy generated from each version of the algorithm was stored in a text file in the project folder hierarchy. This allowed the policy to be retrieved and utilised without having to re-perform the learning process each time.

3.4.1 *First* (*Iterative) Version*

The first version of the algorithm was based on an iterative approach. The learning agent was designed to evaluate each possible action for a state before moving on to the next state. The agent would continually reset to the starting state after each evaluation.

This meant that the agent would gradually make its way along the racing line and during the process the agent would ultimately evaluate the actions between the penultimate state and the goal state. This iterative approach meant that the number of episodes could be predetermined (number of states * number of actions). Algorithm 2 provides the detailed pseudocode for the implementation.

```
QLearning_V1()
  -Episodes = number of states * number of actions
  -For each episode:
    -Reset car position/rotation to start
      -State = 0, NextState = State + 1
      -MaxState = highest state that not all actions
                  have been evaluated

      -For each state below MaxState:
        -If tried all actions for state, use best
         Action
        -Else select an untried action

        -While not at next state
          -Apply action

          -If crash:
            -Calculate and store negative reward
            -Update Q value using QL formula
            -Go to next episode

          -If at next state:
            -If at MaxState:
              -Calculate and store positive reward
              -Update Q value using QL formula
            -State++
```

Algorithm 2: Version 1 of the QL implementation

3.4.2 *Second (Traditional) Version*

The second version of the algorithm was based on a more traditional RL approach. Unlike the first version the learning process did not continually reset in an iterative manner. This version of the QL algorithm is outlined in Algorithm 3.

```
QLearning_V2()
  -Episodes = 10 || 100 || 1000 || 2500 || 5000
  -For each episode:
    -Reset car position/rotation to start
    -State = 0, NextState = State + 1

    -For each state below StateCount:
      -Get non-negative action for state

      -While not at next state
        -Apply action
        -If crash:
          -Calculate and store negative reward
          -Update Q value using QL formula
          -Go to next episode

          -If at next state:
            -If at EndState:
            -Calculate and store positive reward *
             ReachedEndReward value
            -Update and store Q value using QL
             formula

            -Else:
              - Update and store Q value using QL
                formula
              -State++
```

Algorithm 3: Version 2 of the QL implementation

3.4.3 *Reward Function*

The reward function used for the agents produced a reward value based on the quality of the action performed at the current state. The value returned by the function was based on whether the action performed was good or bad.

A bad move (such as crashing into a wall or going off the race track) would result in the function returning a large negative reward value. A good move would return a positive scaling reward value based on two key factors. Firstly, the value would be scaled according to the proximity of the car to the racing line. The second factor was the time taken to get between two states. If the time taken was less than previously attempted actions, the reward value would be increased further. A final large multiplier would be added to the reward value if the car reached the goal state (the final point on the racing line).

Reward = -PenaltyValue - PositionDifference

Equation 1: Reward Calculation for a Negative Move

*Reward = RewardValue * ProximityScaler * TimeBonus*

Where *ProximityScaler* increases based on proximity to centre of racing line

Equation 2: Reward Calculation for a Positive Move

3.4.4 *Execute Policy*

The final piece of the puzzle was to implement a function which could execute the policy that had been stored following the learning process. The policy file was a text file that consisted of a single value (representing the action number) per line. This was loaded into the game and executed using a similar algorithm to that used in the second version of the QL algorithm. The agent would identify its current state and apply the corresponding action as specified in the file until reaching the next state.

4. TESTING AND RESULTS

This initial aim of this research was to investigate whether QL could be used to create a high quality controller for a racing game. Subsequent to this goal, the two versions of the QL algorithm suggested a further area of research in order to determine how they differed and which performed to a higher level. Each version of the agent was taught using the same racing line, race track and car properties. The two agents were taught using the same number of episodes (approximately 1,000) for the first two experiments. The third experiment involved varying the number of episodes for the second version of the algorithm.

4.1 *State-Action Tables (Q Tables)*

The first area of comparison was between the Q Tables produced by each version of the algorithm. These tables were produced after the learning process was completed by retrieving the data from the QStore.

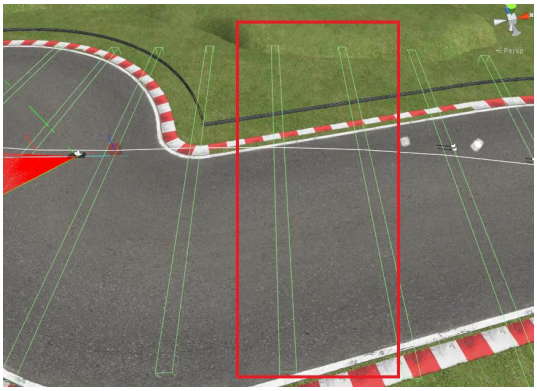


Figure 5: States 93 (right) and 94 (left) on the track

State	Action	Q Value
93	0	255963350.08605
93	1	255963350.08605
93	2	255963350.08605
93	3	255963350.08605
93	4	255963350.08605
93	5	255963350.08605
93	***6***	***2805597255.12183***
93	7	2573942438.35473
93	8	2657868839.60533
94	***0***	***2920734984.09786***
94	1	2920734972.65661
94	2	2920734845.53158
94	3	2920733433.0312
94	4	2920717738.5824
94	5	2920543355.81592
94	6	2918605769.49921
94	7	2897077032.39744
94	8	2657868839.60533

Table 2 State-Action Table (Version 1)

State	Action	Q Value
93	***0***	***730021813***
93	1	-2147483648
93	2	-2147483648
93	3	-2147483648
93	4	-2147483648
93	5	-2147483648
93	6	-2147483648
93	7	-2147483648
93	8	-2147483648
94	***0***	***531860033***
94	1	-2147483648
94	2	-2147483648
94	3	-2147483648
94	4	-2147483648
94	5	-2147483648
94	6	-2147483648
94	7	-2147483648
94	8	-2147483648

Table 3: State-Action Table (Version 2)

Tables 2 and 3 show that there was a difference in action selection at state 93 whilst the same action was

picked at state 94. The negative reward values produced in table 3 are identical. This is because the algorithm was designed to pick the highest throttle value (lowest action value) action unless it resulted in a crash. It would be interesting to see the difference in Q values using a different action selection policy as discussed in section 5.1.

4.2 *Lap Times*

The overall goal of this research was to produce a high quality AI controller for a racing game using the two variations of the QL algorithm. As a result the most tangible measurement of performance provided by the project was in terms of lap-times.

The same race track and racing line was used for each version and they both started from the same position at the beginning of each lap. Ten laps times were recorded for each version and recorded as shown in Table 4. The lap times were performed with the obstacle avoidance and wall avoidance behaviours disabled as there were no obstacles present in the scene to check for in real-time.

Lap Number	Version 1	Version 2
1	42.71785	41.96450
2	42.80776	41.30627
3	42.76348	45.01987
4	43.11338	44.86371
5	42.57750	41.39682
6	43.61481	41.64060
7	43.36933	44.75314
8	41.99917	42.81601
9	42.21012	41.57589
10	42.18604	41.24636

Table 4: Lap Time Comparison

The lap times produced are very similar however the first version appears to produce more consistent times whereas the second version has more erratic results (see table 6 for the average and standard deviation).

4.3 *Episode Variation*

As mentioned earlier in this report, the first version of the algorithm required a finite number of episodes in order to perform the learning process. The second version however could be taught using an indefinite number of episodes. This raised the question of what effect would varying numbers of episodes have on the lap-time produced by the agent. Up to this point, the results produced for the second version was taught using the same number of episodes as the first version of the algorithm (approximately 1,000).

Episodes	Lap Time / Result
10	44.33456 (crashed into wall)
100	44.96534 (crashed into wall)
1000	42.65832
1500	41.74825
2500	***40.95938***
5000	41.46755

Table 5: Episode Variation Table

The policies which caused the car to crash (10 and 100 episodes) still managed to complete their laps as the car was built with a reset function to reset the car after 2.5 seconds to a point slightly further long the racing line. Table 5 shows that the fastest lap time was produced by the 2500 iteration version whilst similar lap times were produced by the 1000, 1500 and 5000 versions. See table 7 for the average and standard deviation.

5. EVALUATION

5.1 *State-Action Tables (Q Tables)*

The state-action tables (Tables 1 and 2) showed that the learning agents took a different approach entering the corner. The states chosen (93 and 94) were located before the tightest corner on the track. The initial observation from comparing the tables is that the Q values for the first version (table 2) produced larger and more erratic numbers than the second version (table 3). This is because the reward calculation process took place more often as each possible state-action pair was evaluated.

It is interesting to note the different actions selected for state 93. The first version selected a braking action whilst the second version selected the full throttle action. This was because the first version was focused on one individual state at a time. This meant it often braked at the last possible state as it didn't keep track of the reward based on the final end goal state. The second version had a more long-term view and as a result performed the braking action earlier (states 89, 90 and 92, after consulting the full Q table) in order to achieve a better speed and line through the corner. This is because the QL function is aimed at achieving the highest possible long-term reward which is provided upon reaching the goal state. It would have been interesting to see the effect of action-selection policy on the Q values produced.

5.2 *Lap Times*

The lap time comparison produced an interesting set of results. Table 6 shows the average and standard deviation between lap times for each version. The average lap time between the two algorithms was extremely close.

The standard deviation, however, was very different (0.5 for the first version and 1.5 for the second version), as shown in Table 5.

Lap Number	Version 1	Version 2
Average	42.73594	42.65832
Standard Deviation	0.52378007	1.597068

Table 6: Average and Standard Deviation of Lap Times

The first version appeared to produce very consistent lap times and results, whilst the second produced a wider range of very fast and relatively slow lap times. The slow lap times were often a result of going off track or hitting a wall. This would indicate that the number of episodes used to teach the second version was too low.

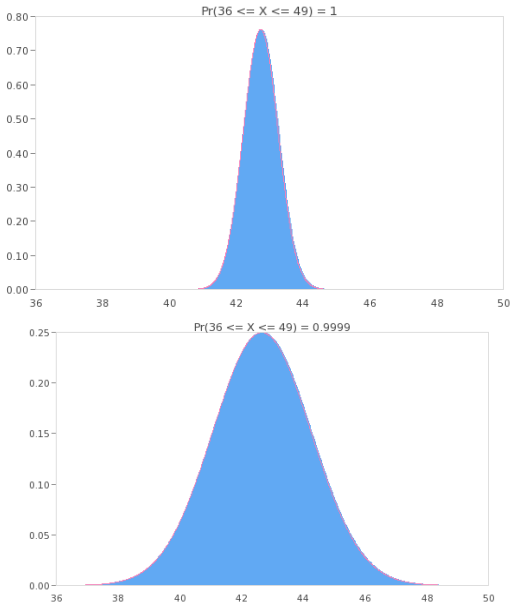


Figure 6: Standard Deviation of Lap Times (Top: V1, Bottom: V2)

5.3 *Episode Variation*

This experiment was inspired by the standard deviation result in the lap-time test. The question raised was at what point was it that the number of episodes used cease to have an effect on the second version of the algorithm. A test was worked by writing a lap-time tracker script which kept a log of the lap-times produced by the car for 10 laps. This test was not performed for the first version because it was taught in a finite number of episodes as discussed earlier. Table 7 highlights the average lap times produced and the standard deviation between them.

Episodes	Lap Time / Result
Average	42.6889
Standard Deviation	1.62844

Table 7: Average and Standard Deviation for Episode Variation of Lap Times (Version 2 only)

The results show that for 100 episodes or less, the car crashed or had an incident causing the lap-time to be increased. This was to be expected given the number of possible actions for the number of states in the game world. Interestingly, it also shows that the fastest lap time was produced from a policy created by 2500 episodes. In contrast the policies produced by 1500 and 5000 episodes produced relatively similar lap times.

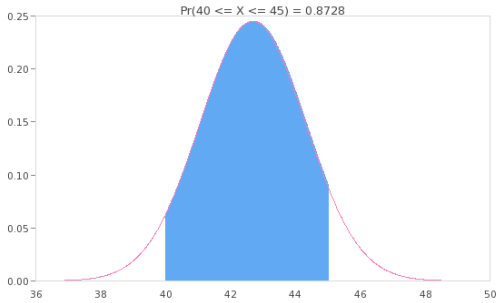


Figure 7: Standard Deviation of Episode-Based Lap Times

The time produced by 1000 episodes version was to be expected. However, one would have imagined that the lap time for 5000 episodes would have been at least as quick if not faster than the controller produced from 2500 episodes. This result is possibly due to the algorithm performing further learning and discovering that a policy for this type of lap-time would result in a crash in the tighter parts of the racetrack. Therefore it made safer choices whilst still maintaining a good overall speed.

5.4 *Results Discussion*

The lap-times produced by both versions are relatively competitive compared to player lap-times (with times ranging between 39 and 42 seconds on average depending on the type of player). The overall performance of the algorithm in terms of lap-time is restricted by the optimality of the racing line. The line was generated from waypoints that were implemented by hand and based on what appeared to be the best line around each corner. Better lap times would possibly have been achieved if this line was produced algorithmically to create a minimum-curvature line around the race track [13]. It was also surprising to note that both versions produced relatively similar lap times despite the differing approach to the QL process.

One drawback with using QL to create the controller and generate the results was the time taken to execute the large number of episodes. This was largely due to the use of the Unity game engine whereby the maximum simulation speed increase is only 100 times [14]. The process also had to take place within the engine itself and could not be run as a batch mode process. This meant that

teaching the controllers took a significant amount of time, particularly for controllers being taught with over 1000 episodes. Given another engine or framework this could have been improved upon.

6. CONCLUSIONS AND FUTURE WORK

This paper has presented the use of QL to produce an AI controller in a racing game. The results have shown that the controller produces reasonable lap-times and performance compared to a player (as discussed in section 5.4).

The QL formula used in this project was the standard QL approach. There are several alternative methods that could have been used (such as SARSA [7]) which may have produced differing or even improved policies for the AI controller.

There are several other areas that are open to investigation in the future. The most pertinent of these would be to utilise alternative reward functions. This could be used to create different types of AI controllers (for example varying skill level, difficulties or driver traits). A further development could have been to use multiple racing lines with differing lines into and out of corners. These lines could have been learnt and switched in real-time to produce more realistic and seemingly human behaviour.

One simple modification that could be made would be to increase the state-space of the game world. This would increase the size of the QStore but in turn increase the number of possible actions that can be taken around the race track. This could potentially result in enhanced behaviour, in particular through tight or twisting corners. The state space could be expanded further by taking other factors into account such as the car velocity. This would further improve the quality of the agent behaviour.

It may also be possible to implement a neural network layer above the reinforcement learner to provide a more advanced AI agent. The neural network could be used to help learn which combination of racing lines and overtaking manoeuvres is best at each part of the track given different scenarios. It would be informative to compare an online and offline version of the neural network agent and to see how well both versions fair in a race-like scenario. This is a similar concept to that proposed by Lucas et al. [15] in their comparison of Q-learning and neural networks for a car controller which could be used to train the controller as the simulation progresses. The neural network could also be used to learn an approximate utility function as discussed in the work by Tesauro in TD-Gammon (1995) [16].

This project has shown that QL produces a reasonable controller, and also that it can reduce the need for developers to focus their time and effort on hard-coding a car controller for specific race tracks and scenarios. The racing line is the principle requirement to be implemented into the game world. In the future QL could be used to teach the agent how to steer based on its current position on the track and what lies ahead. This would then allow AI developers to focus their efforts on improving the agent's steering behaviours to create more realistic real-time interactions.

7. REFERENCES

[1] Watkins, C (1989). Learning from Delayed Rewards. London: King's College.

[2] Turn 10 Studios. (2013). Forza Motorsport. Available: http://forzamotorsport.net/en-US/en-GB. Last accessed 30th August 2013.

[3] Microsoft. (2004). Drivatar. Available: http://research.microsoft.com/en-us/projects/drivatar/. Last accessed 16th September 2013.

[4] Candela, J, Herbrich, R, Graepel, T. (2011). Machine Learning in Games. Available: http://research.microsoft.com/en-us/events/2011summerschool/jqcandela2011.pdf. Last accessed 16th September 2013.

[5] Unity. (2013). Unity Game Engine. Available: http://unity3d.com/. Last accessed 24th May 2013.

[6] Reynolds, C. (1999). Steering Behaviors For Autonomous Characters. Game Developers Conference. 1 (1), p763-782.

[7] Sutton, R and Barto, A (1998). Reinforcement Learning:An Introduction. United States: MIT Press. p324-332.

[8] FIAS. (2010). Reinforcement Learning. Available: http://www.cs.utexas.edu/~dana/RL08.pdf. Last accessed 20th September 2013.

[9] Patel, P, Carver, N, Rahimi, S. (2011). Tuning Computer Gaming Agents using Q-Learning. Proceedings of the Federated Conference on Computer Science and Information Systems. 1 (1), p581-588.

[10] Steam. (2013). Counter-Strike. Available: http://store.steampowered.com/app/10/. Last accessed 16th September 2013.

[11] Thirwell, E. (2013). Forza 5's AI is "much more engaging than anything you'll see in another racing game". Available: http://www.oxm.co.uk/62293/forza-5s-ai-is-much-more-engaging-than-anything-youll-see-in-another-racing-game/. Last accessed 20th September 2013.

[12] Dunlop, R. (2005). Introduction to Catmull-Rom Splines. Available: http://www.mvps.org/directx/articles/catmull/. Last accessed 16th September 2013.

[13] Moreton, H (1983). Minimum Curvature Variation Curves, Networks, and Surfaces for Fair Free-Form Shape Design. United States: Berkeley. p1-213.

[14] Unity Documentation. (2013). Time Manager. Available: http://docs.unity3d.com/Documentation/Components/class-TimeManager.html. Last accessed 20th September 2013.

[15] Lucas, S, Togelius, J. (2007). Point-to-Point Car Racing: an Initial Study of Evolution Versus Temporal Difference Learning. Symposium on Computational Intelligence and Games. 1 (1), p260-267.

[16] Tesauro, G. (1995). Temporal difference learning and TD-Gammon. Communications of the ACM. 38 (3), p58-68.