Technical Specification

CA400

Name:	Kyrylo Khaletskyy
Student Number:	15363521
Supervisor:	Martin Crane
Date of Completion:	17/05/2019

Table of Contents

Abstract

AIRace demonstrates how machine learning can be used to drive cars autonomously. I developed two neural network algorithms, backpropagation and NEAT (NeuroEvolution of Augmented Topologies) a genetic algorithm to train the cars using inputs from its sensors. A vehicle pathfinding algorithm provided by Unity is also implemented in AIRace in order to contrast the best case scenario without using machine learning. This project determines which one performs best under particular scenarios. To showcase these algorithms I created a 3D racing game, where a user can either play by themselves or race against other cars controlled by said algorithms. More advanced users can observe the training process of either algorithm and experiment with various attributes pertaining to them. In this document, you will find the full development process of AIRace. You will find details about the motivation behind this project and how the research was carried out. You will see how and why I made certain choices pertaining to the technical aspects of the system such as system architecture, design, implementation and validation. An in-depth analysis of the machine learning algorithms, how they perform and why they perform as they do. Finally, a review of the results gathered, directions for future work and conclusions.

1. Introduction

1.1 Overview

AIRace is a 3D racing game targeted at computer gamers and racing game enthusiasts, the game aims to enable users to drive around the track, either on their own or race against enemy cars. Enemy cars are driven by three separate algorithms, backpropagation, NEAT (NeuroEvolution of Augmented Topologies) a genetic algorithm and a pathfinding AI. AIRace was created using the Unity Game Engine, it implements two main features, Racing and Learning mode.

Racing mode focuses on the user interaction with the car and track, enabling a smooth and realistic gaming experience. AIRace uses custom car physics, including acceleration curves, steering, drag and camera movements which provides an authentic and unique driving experience. The user is able to select between a set of different cars, each of these cars will have different speed, handling and drivetrain characteristics. Within Racing mode, the game contains two separate features, timetrial and competition. In a timetrial, a user can race around a track on their own trying to beat their best time or practice for a future competition. In a competition, they will face against enemy cars which use the aforementioned algorithms to drive around the track.

Learning mode allows users to visualise how the computer learns to go around a track. They can choose between genetic or backpropagation algorithms and experiment with various attributes pertaining to them such as fitness functions, mutation probabilities, learning rates, sensors lengths and population sizes. Experimenting with these attributes will show which one performs best under particular scenarios, allowing the user to compare both algorithms and their attributes.

1.2 Glossary

- **Neural Network:** is a series of algorithms that endeavours to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria.
- **Torque:** is the measure of the force that can cause an object to rotate about an axis. Just as force is what causes an object to accelerate in linear kinematics, torque is what causes an object to acquire angular acceleration.
- **Drag:** is the resistance force caused by the motion of a body through a fluid, such as water or air.
- **Drivewheel:** is the transforming torque into tractive force from the tires (front, rear or all wheels), causing the vehicle to move.
- **Reinforcement Learning:** is about taking suitable action to maximize reward in a particular situation, in this case, with the use of a fitness function.
- **Machine Learning:** is an application of artificial intelligence that provides systems with the ability to automatically learn and improve from experience without being explicitly programmed.
- **Steam:** is a digital distribution platform developed by Valve Corporation for purchasing and playing video games.
- **NPC:** is a non player character found in video games.
- **Scene:** contains the objects of a game. They can be used to create a main menu, individual levels, and anything else visible in a 3D or 2D environment.
- **Colliders:** are physical objects that "collide" with other objects in the 3D environment.
- **Meshes:** non-collidable objects that make up the 3D environment.
- **Doppler effect:** is an increase (or decrease) in the frequency of sound, light, or other waves as the source and observer move towards (or away from) each other.
- **Sigmoid:** is an activation function which sets a number between 0 and 1.
- **TanH:** is an activation function which sets a number between -1 and 1.
- **Postprocessing:** is used in real-time 3D rendering (such as in video games) to add additional effects that enhance the quality and visual aesthetic.

2. General Description

2.1 Business Context & Motivation

I have always enjoyed playing video games and ever since starting the Computer Applications course, I realised that I would like to centre my career in game development. Video games have been at the forefront of entertainment for decades. The video game industry has been an enormous source of income for companies such as Ubisoft, EA and Rockstar. This project is an entry into the game development industry and its advantages as a potentially profitable business idea. The user demographic will be computer gaming enthusiasts who enjoy various types of racing games. Currently, on Steam, the top 10 racing games alone sold 20 million copies, this doesn't include games sold on other platforms and consoles. This project also extends into car automation in the real world.

The use of various machine learning algorithms to autonomously drive around the track ties in with the recent gain in popularity of the car automation industry. Companies such as Tesla, Google and Nvidia have invested millions into the development of car automation making this project a very valuable learning experience in both fields of machine learning and car automation. Researching this topic thoroughly and applying it to my project will make me gain invaluable skills that I can apply in my future career.

2.2 Research

Unity Game Engine Environment

I have never worked with Unity, C# or 3D Modelling before, therefore needed to focus my time on learning these skills. It was important to get a good understanding of the Unity Game Engine Environment as the entire project has been developed within the Unity editor and Visual Studio. To teach myself these skills I looked at:

1. **Official Unity Documentation**
   1.1. https://docs.unity3d.com/Manual/index.html
2. **Youtube Tutorials (channels)**
   2.1. https://www.youtube.com/user/Brackeys
   2.2. https://www.youtube.com/channel/UCX_b3NNQN5bzExm-22-NVVg

Car Physics

Car physics was a huge aspect of AIRace, it was important to create a consistent, realistic and challenging racing experience. I wanted each car to drive and handle slightly different in order to mimic real cars. While this project doesn't focus on car driving physics simulation it was still important to make it as realistic as possible. I derived my own torque algorithm shown in section 4.1 and other car physics from the following sources:

1. http://lancet.mit.edu/motors/motors3.html
2. http://www.asawicki.info/Mirror/Car%20Physics%20for%20Games/Car%20Physics%20for%20Games.html
3. https://nccastaff.bournemouth.ac.uk/jmacey/MastersProjects/MSc12/Srisuchat/Thesis.pdf

Genetic Algorithm

Generally speaking, autonomous and self-driving cars implement Genetic algorithms. This unsupervised reinforcement learning approach was perfect for my implementation. The plan for this project was to have cars learn how to go around a track by themselves, this eliminates the need to set each car up on each track individually, which is an extremely long and cumbersome process, as mentioned in section 4.4. I decided to combine Neural Networks and Genetic Algorithms, this will give me the best of both worlds when it comes to machine learning centred around cars. For my own implementation of this algorithm I have followed a similar process provided by the following sources:

1. https://www.aitrends.com/ai-insider/genetic-algorithms-self-driving-cars-darwinism-optimization/
2. https://www.researchgate.net/publication/220702292_Driving_Cars_by_Means_of_Genetic_Algorithms
3. https://towardsdatascience.com/reinforcement-learning-towards-general-ai-1bd68256c72d
4. https://blog.coast.ai/lets-evolve-a-neural-network-with-a-genetic-algorithm-code-included-8809bece164

Backpropagation

For the second machine learning AI, I decided to implement backpropagation. The reason I chose this algorithm is that this is the exact opposite of the genetic algorithm which I have already implemented. As I have already had experience with Neural Networks this gave me an even greater insight into the topic. Additionally, backpropagation takes user input and works in a supervised learning approach, which provided an excellent comparison between the two algorithms. For my own implementation of this algorithm I have followed a similar process provided by the following sources:

1. https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e
2. http://neuralnetworksanddeeplearning.com/chap2.html

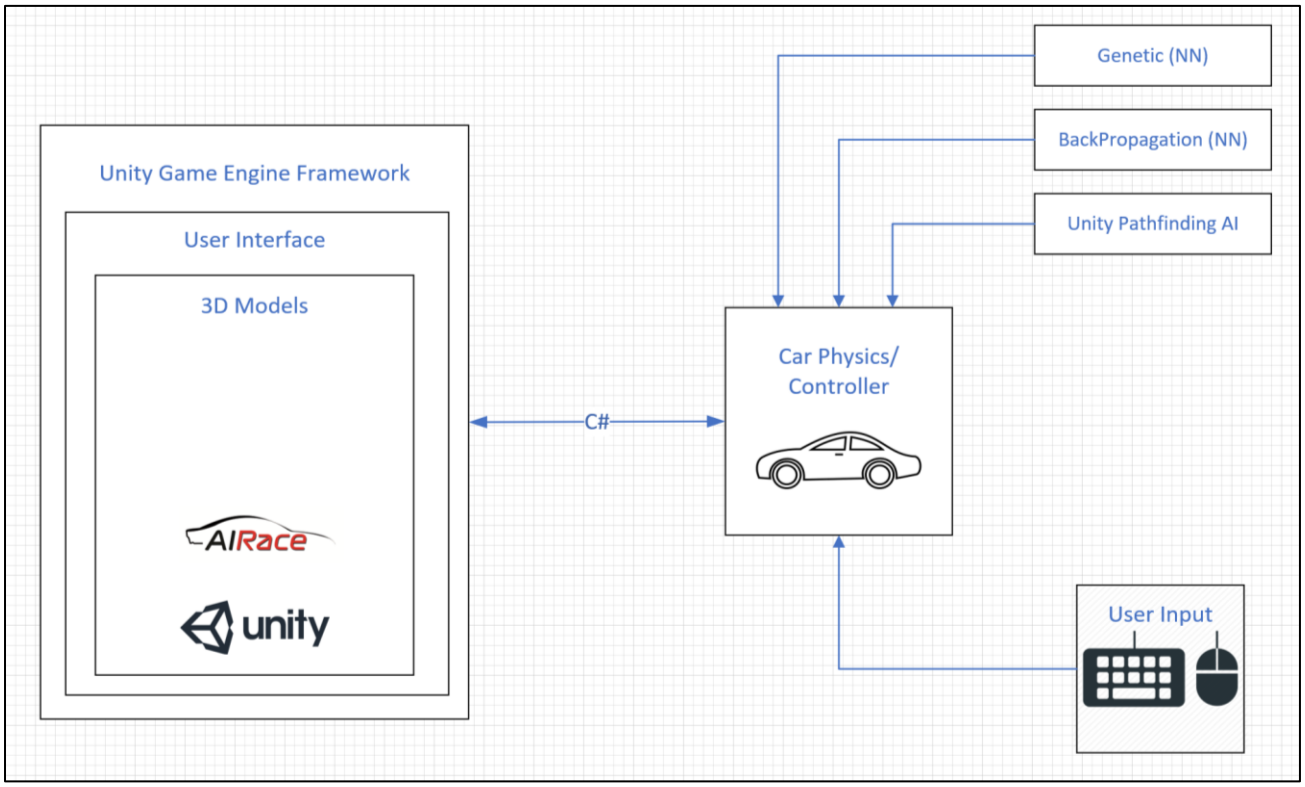
2.3 User Characteristics & Objectives

The game is aimed at anyone with a personal computer and an interest in video games, while the target audience will be young people it is expected that all age groups will have an interest in the game. As the user demographic is very broad it was essential to make the game and user interface usable in all scenarios for all age groups. In competition mode, the objective of the player is to traverse through a race track and attempt to beat the enemy cars. In time trial mode the objective of the player is to practice for a race and attempt to set the best time possible. The main goal of this game is to create a challenging and interesting gaming experience that will leave the player with nothing but enjoyment while they are playing it.

Some more technically advanced players may wish to visit the machine learning mode where they can observe how a car learns to go around a track, but this is not mandatory. Nevertheless, regardless of their technical expertise they will able to operate this mode with ease. The objective of this section is to show the user how machine learning algorithms work. A user can experiment with different algorithms, attributes and cars, which will give them insight into machine learning and AI. Experimenting with the various attributes will yield different results during each session which will grant the user the knowledge of how each attribute affects the result.

3. Design

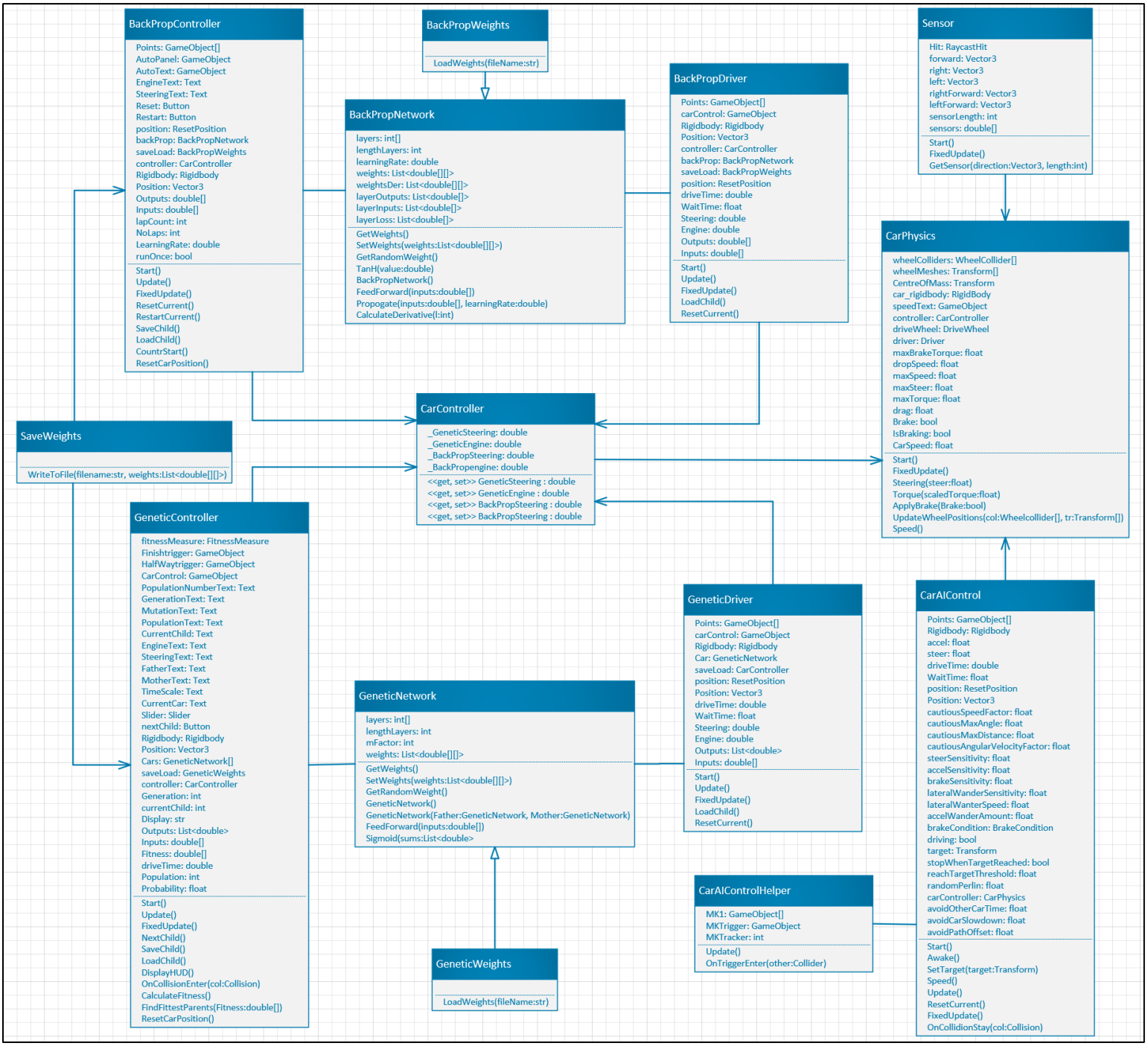
3.1 System Architecture



The above system architecture diagram shows the interaction between the Unity game engine and its components. The user input and various AI (Genetic, BackPropagation and Unity Pathfinding AI) interact with the car physics/controller. In turn, the car physics/controller interacts with the Unity game engine to control the 3D models and give visual and auditory feedback to the user through the User Interface.

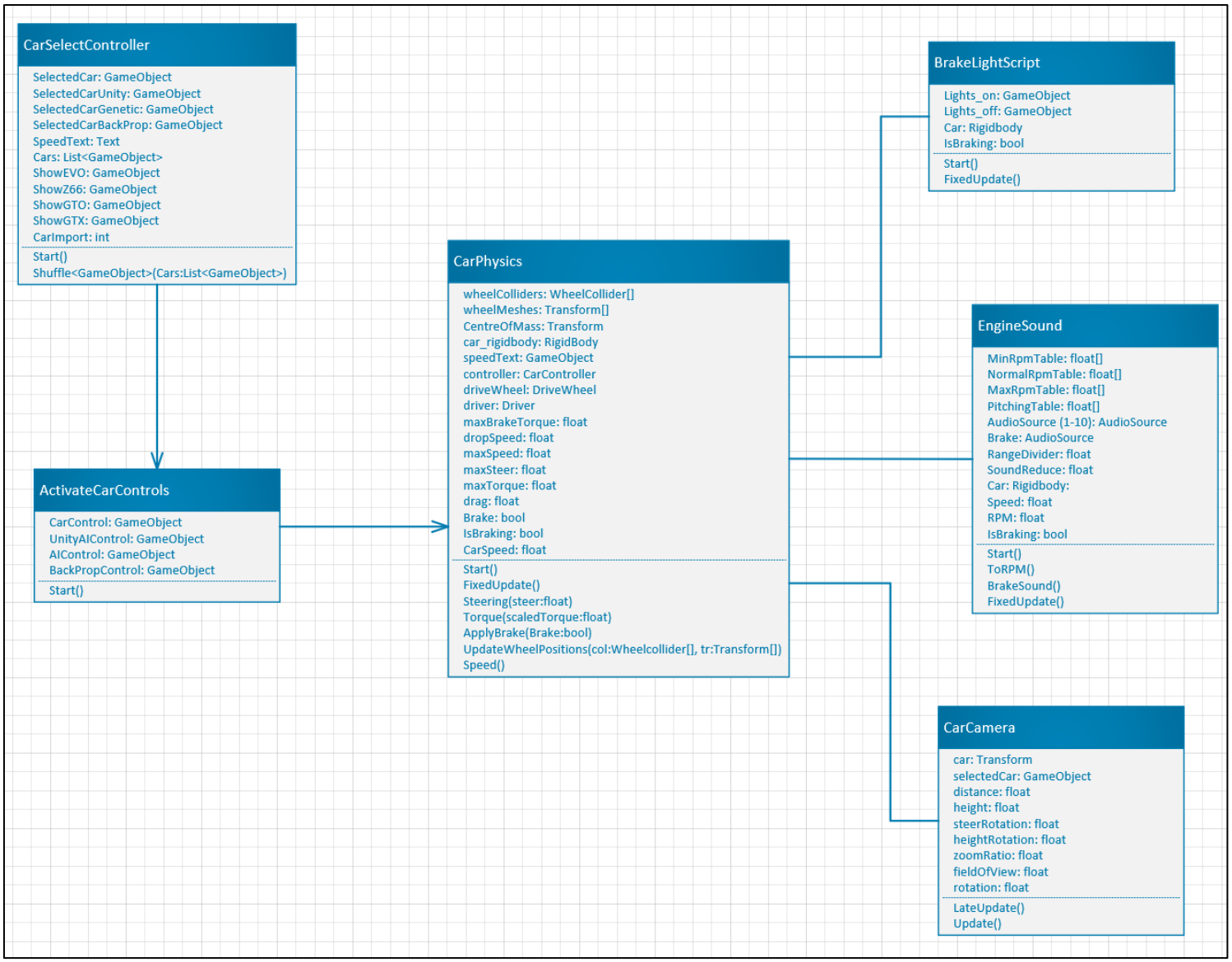
3.2 Class Diagrams

3.2.1 AI Controller Class Diagram



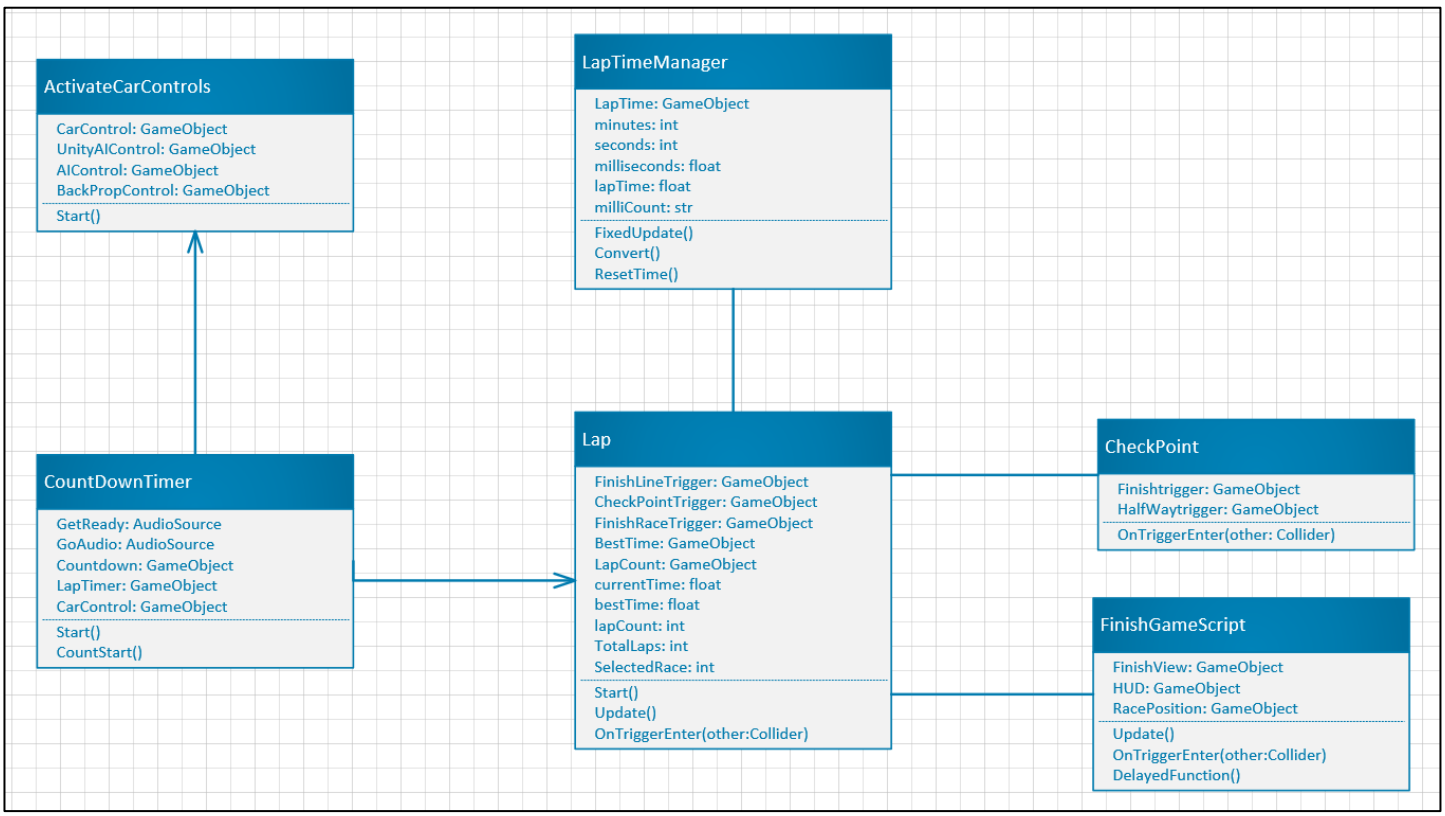
Above is a class diagram of the various AI that interacts with the CarPhysics class. The GeneticNetwork class is used by the GeneticDriver and GeneticController which set their values to the CarController depending on which one is currently active. Similarly, the BackPropNetwork is used by the BackPropDriver and BackPropController which set their values to the CarController depending on which one is currently active. The CarPhysics class then gets the values from the CarController and applies them to individual cars. The CarAIControl uses the CarAIControlHelper to find the next point on the lap to follow and applies the necessary engine and steering value to the CarPhysics class.

3.2.2 Car Physics Class Diagram



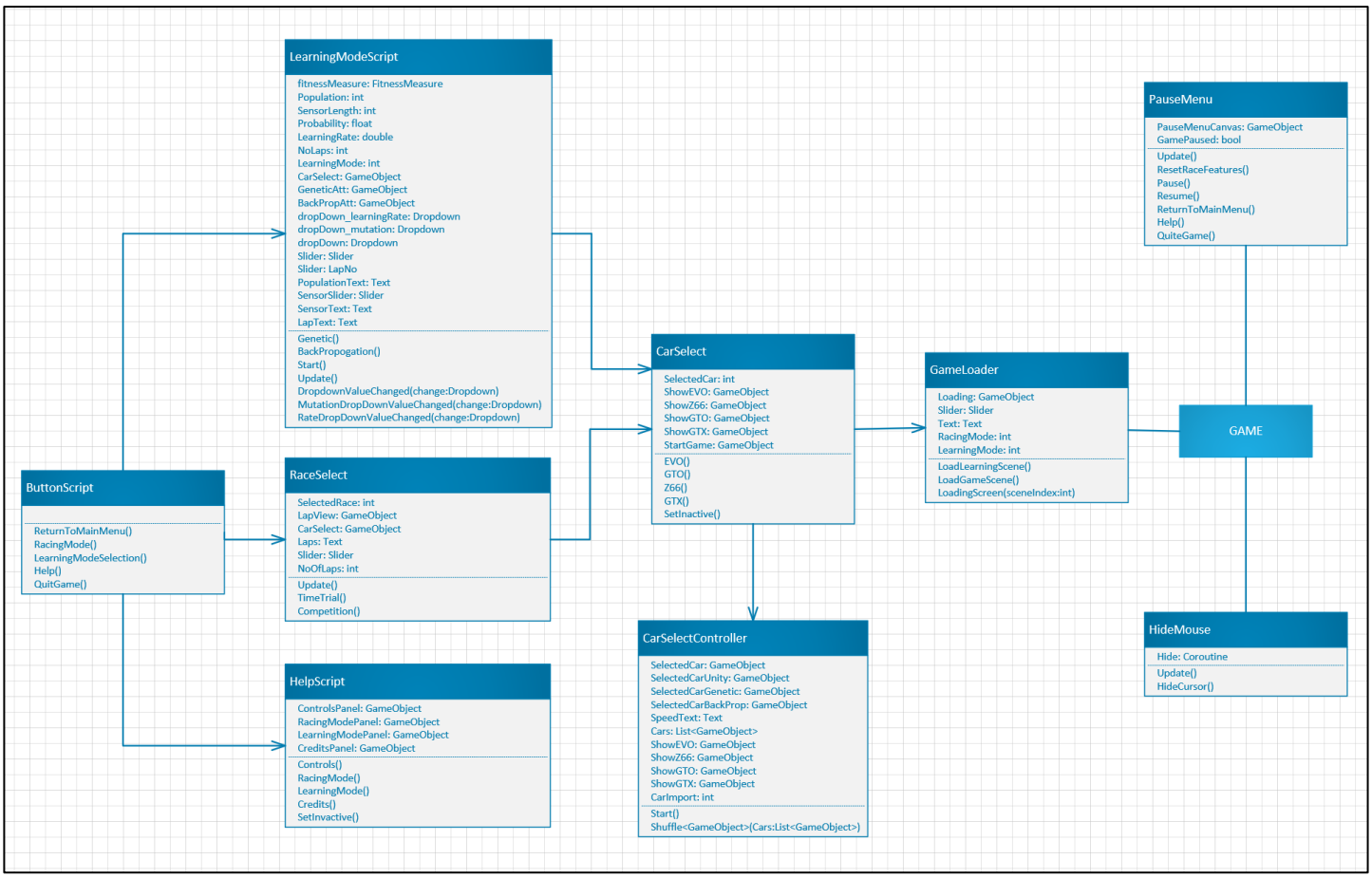
Above is a class diagram pertaining to all the car physics, sounds, models and attributes. After the CarSelectController class receives which car the user has chosen it assigns the various AI randomly to each other car. After the countdown timer returns 0 ActivateCarControls class activates the controls to all the cars that were spawned into a scene. The CarPhysics utilises the BrakeLightScript, EngineSound and CarCamera classes, each one loading in separate attributes depending on which driver is assigned to a car.

3.2.3 Lap Timer & Race Physics Class Diagram



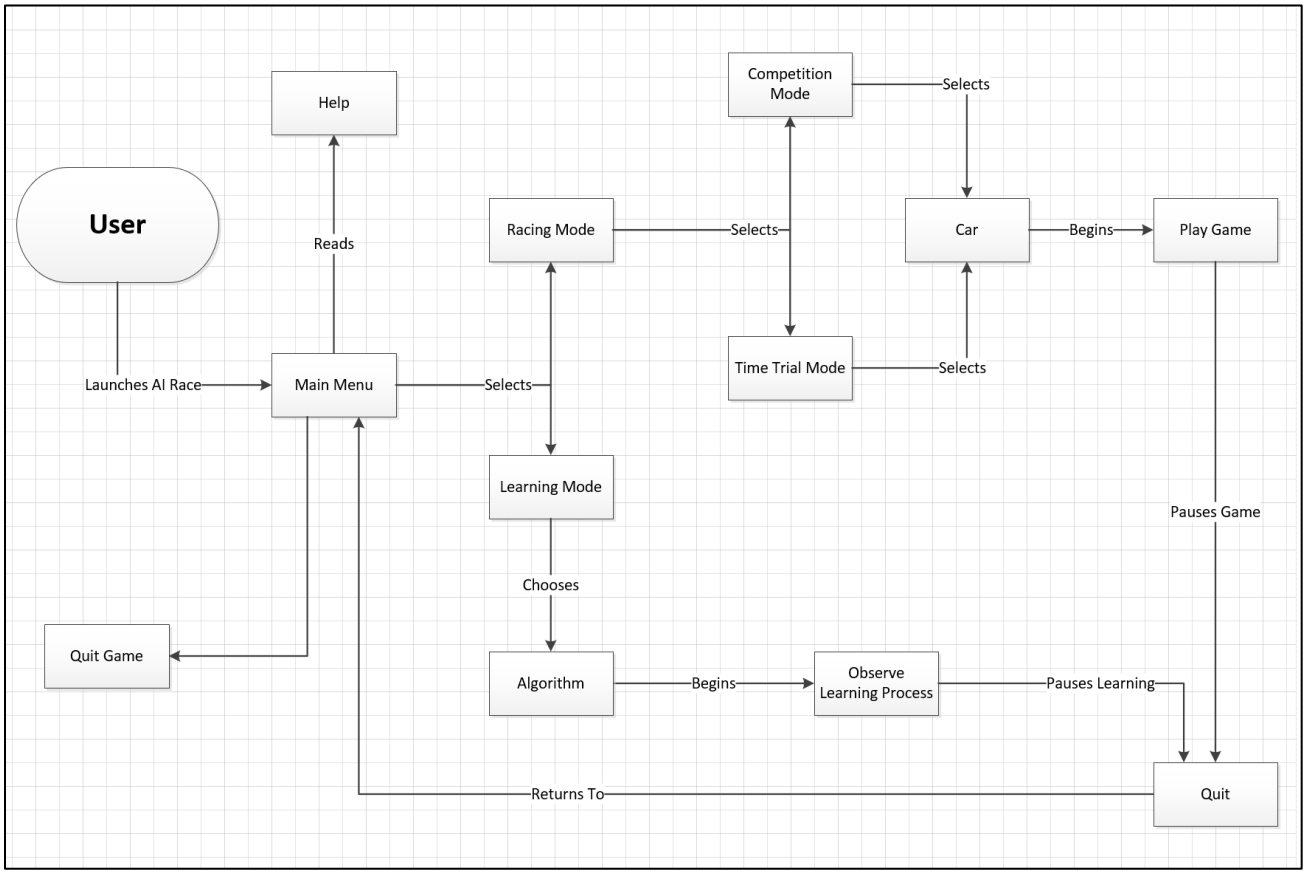
Above is a class diagram pertaining to all the racing features such as lap timers, lap counters, finish line triggers and checkpoint triggers. The Lap class utilises the LapTimeManager, Checkpoint and FinishGame classes to set up various racing scenarios. When the CountDownTimer class returns 0 the controls of the car are activated and the lap timer starts running. Depending on which type of race it is the Lap class will activate different triggers around the track.

3.2.4 UI Class Diagram



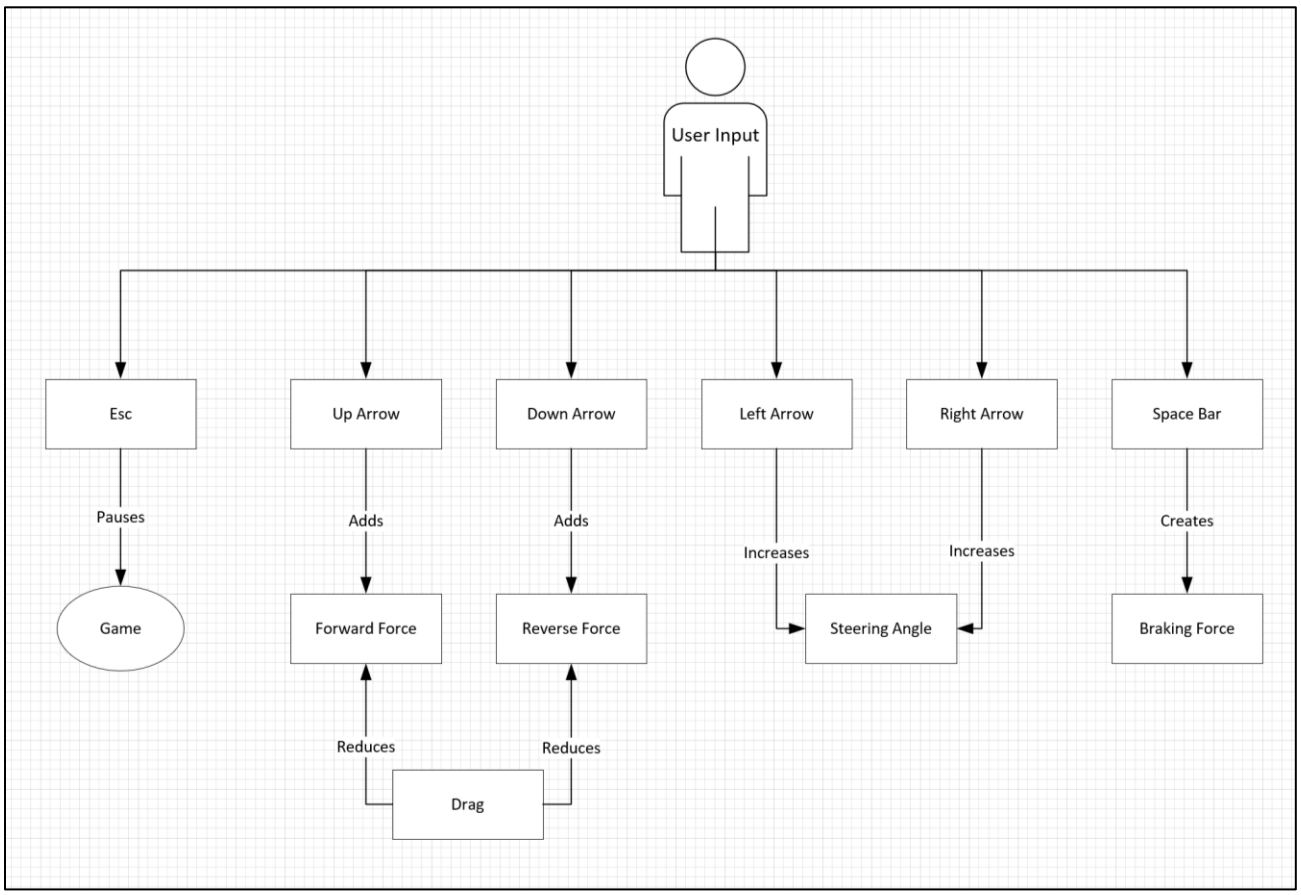
Above is a class diagram for all the UI related features such as screens, buttons, game loaders and selectors. When on the main menu a user has three options, LearningMode, RacingMode and Help Screen. Selecting each option will activate a separate class. After a user selects either RacingMode or LearningMode, they will also activate the CarSelect class which interacts with the CarSelectController class mentioned in section 3.2.2. When in game, AIRace activates the PauseMenu and HideMouse classes.

3.3 Process Diagram



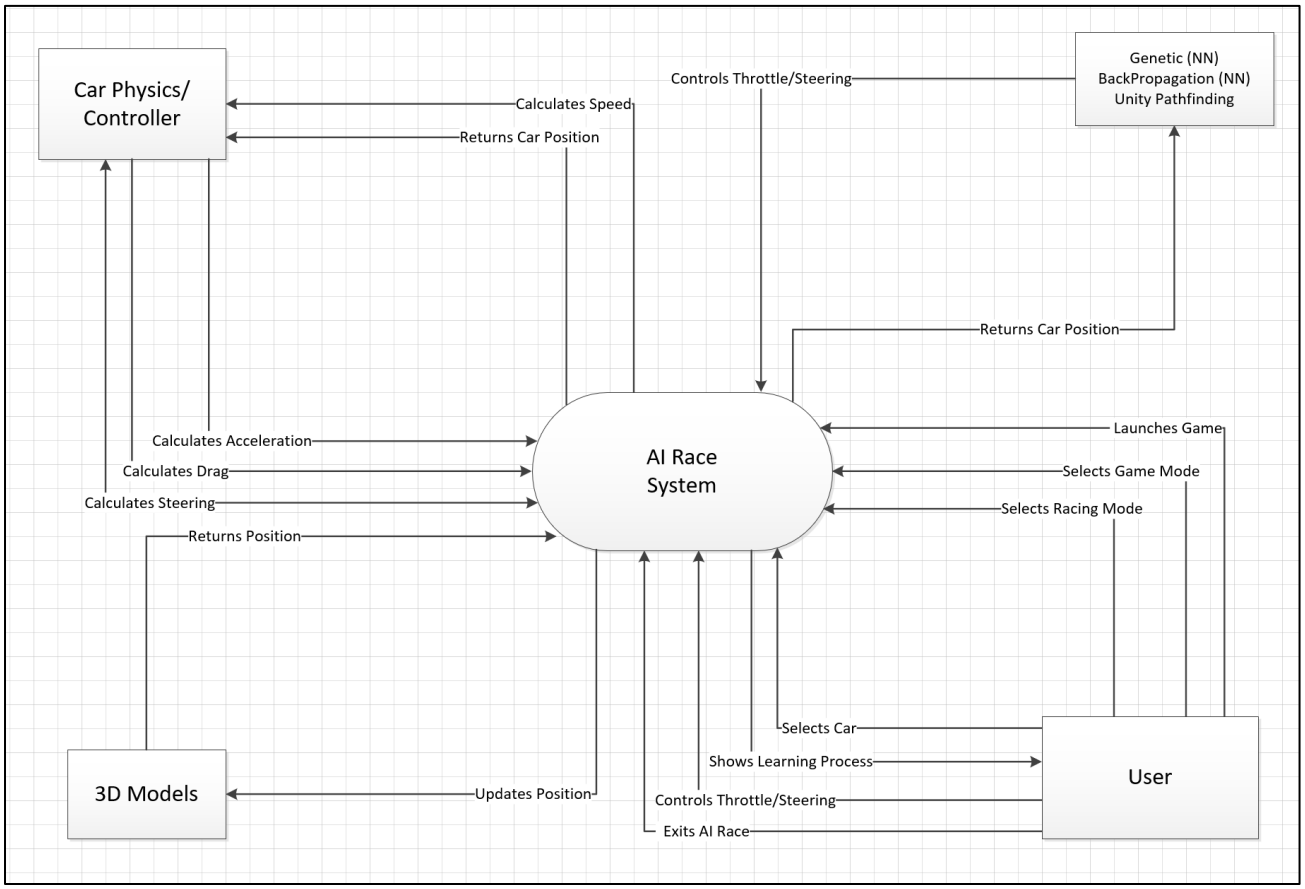
The process diagram above is a graphical representation of the movement of processes through the system. It shows how each process works at step by step intervals, and which order a user must interact with the various features of AIRace. For example, when a user launches AI Race they will be met with the main menu, from where they can select the help screen, racing mode or learning mode.

3.4 User Input Diagram



The user input process diagram above shows a more in-depth version of the “Play Game” process from the previous diagram (Process Diagram shown in section 3.3). The user input and the process that follows that input is shown.

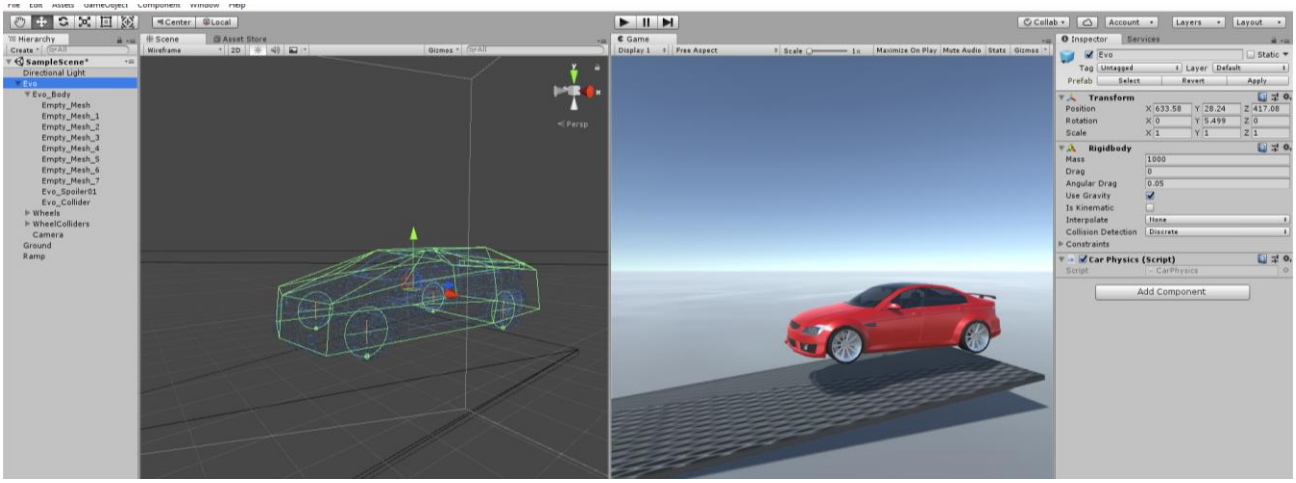
3.5 Context Diagram



The context diagram above shows how each feature or person interacts with the system, each feature shows a set of functions it controls within the system and a set of functions that the system controls within the feature.

4. Implementation

4.1 Car Physics



```
//Makes the wheel model rotate along with its wheel colliders
    void UpdateWheelPositions(WheelCollider[] col, Transform[] tr){
        //For each wheel apply a quaternion to create rotation on the wheel transform
        for (int i = 0; i < 4; i++){
            Vector3 pos = tr[i].position;
            Quaternion rot = tr[i].rotation;
            col[i].GetWorldPose(out pos, out rot);
            //If its the FL or RL wheel rotate the wheels 180 degrees on the Z-axis (refer to docs)
            if (i == 0 || i == 2){
                rot = rot * Quaternion.Euler(new Vector3(0, 0, 180));
            } else{
                rot = rot * Quaternion.Euler(new Vector3(0, 0, 0));
            }
            tr[i].position = pos;
            tr[i].rotation = rot;
        }
    }
```

To begin, I needed a 3D modelled car, I found some assets on the Unity Asset store. I didn't have any previous experience 3D modelling so I decided to get one there and focus my time on other aspects of the game. When you get a car model on the store all you get is a 3D model, after which you have to add your own colliders. The 3D model can be seen on the right and the colliders can be seen on the left (outlined as green lines). After I added all the colliders I put in a simple ramp which would test the car rolling down the ramp. The issue now is that the wheel colliders need to be mapped to the actual wheel models in order to rotate as the wheels are rotating. The UpdateWheelPositions() method shown above fixes this issue and allows the wheels to rotate with the wheel colliders.

Next, I worked on the inputs that control the steering and acceleration. In order to get the input from the keyboard, you use Input.GetButton() which finds a button you are pressing on the keyboard and Input.GetAxis() which returns a value for either left-right or up-down. Using Input.GetAxis() we can find the desired steering angle. This function returns a float, by multiplying this value by 25f we get a max steer angle of 25 degrees, and a smooth transition in between 0 and 25 degrees on both

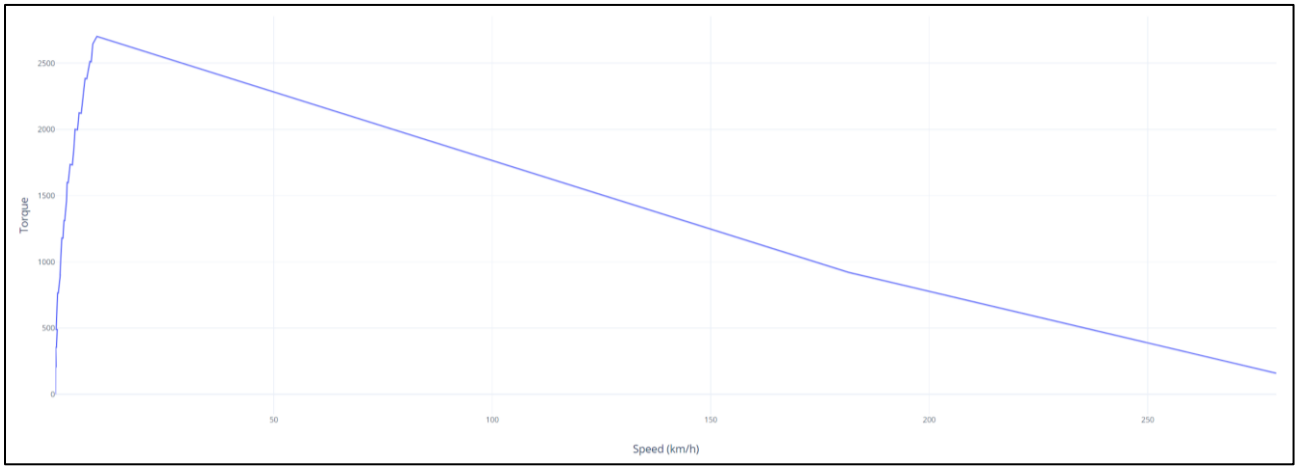
wheels. Now both front wheel colliders (and subsequently wheel meshes) are turned to a steering angle desired by the user. Changing the max steer will change the force at which the car turns, this value can later be adjusted and even tweaked depending on the car selected, allowing for different characteristics between each car.

```
public class CarController {

    public static double _GeneticSteering;
    public static double _GeneticEngine;

    public static double _BackPropSteering;
    public static double _BackPropEngine;

    public double GeneticSteering {
        get {return _GeneticSteering; }
        set { _GeneticSteering = value; }
    }

    public double GeneticEngine {
        get { return _GeneticEngine; }
        set { _GeneticEngine = value; }
    }

    public double BackPropSteering {
        get { return _BackPropSteering; }
        set { _BackPropSteering = value; }
    }

    public double BackPropEngine {
        get { return _BackPropEngine; }
        set { _BackPropEngine = value; }
    }
}
```

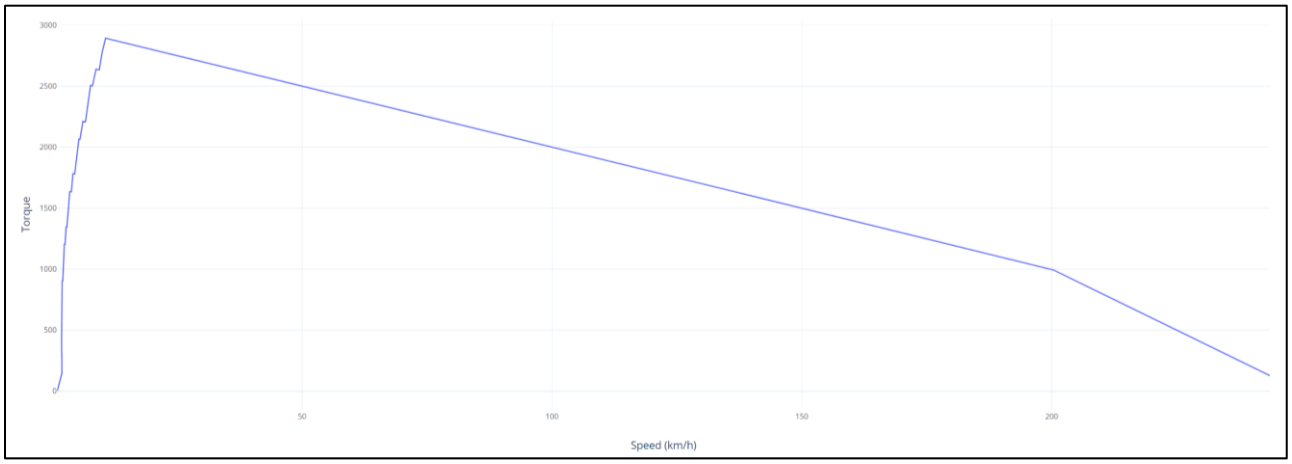
Unlike user input which is generated from Input.GetAxis(), the engine and steering values which are returned from the BackPropDriver, BackPropController, GeneticController and GeneticDriver are returned from the CarController class, which gets and sets the outputs of their respective AI (depending which class is currently active).

As the wheelcolliders act as turnable objects there is no way to control their speed, only torque. My first acceleration implementation consisted of a torque that I applied to the wheels, the problem here is that the car will infinitely speed up (and stay at that speed). Then I applied drag which slows the velocity of the whole car slowly, but this didn't stop the car from speeding up past realism. I have researched online that a typical acceleration curve looks like an "S", where at first there is a high acceleration, then it stays constant, then it starts slowing down until it reaches max speed. Based off that I decided to implement my own acceleration curve. As each car has different characteristics, their respective torque curves are also different. Here are the torque curves for each car:

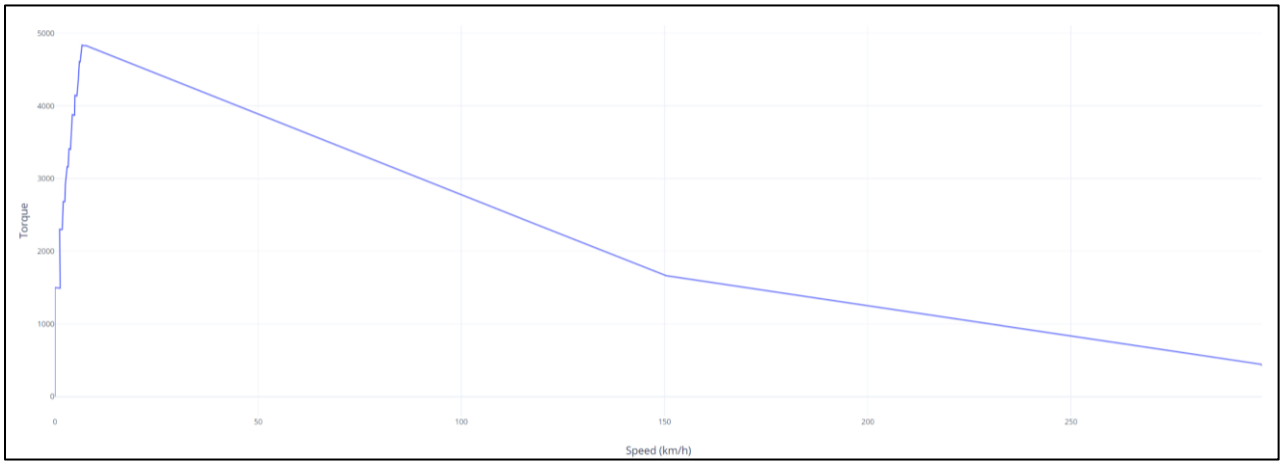
EVO:


Torque
2500
2000
1500
1000
500
0
50
100
150
200
250
Speed (km/h)

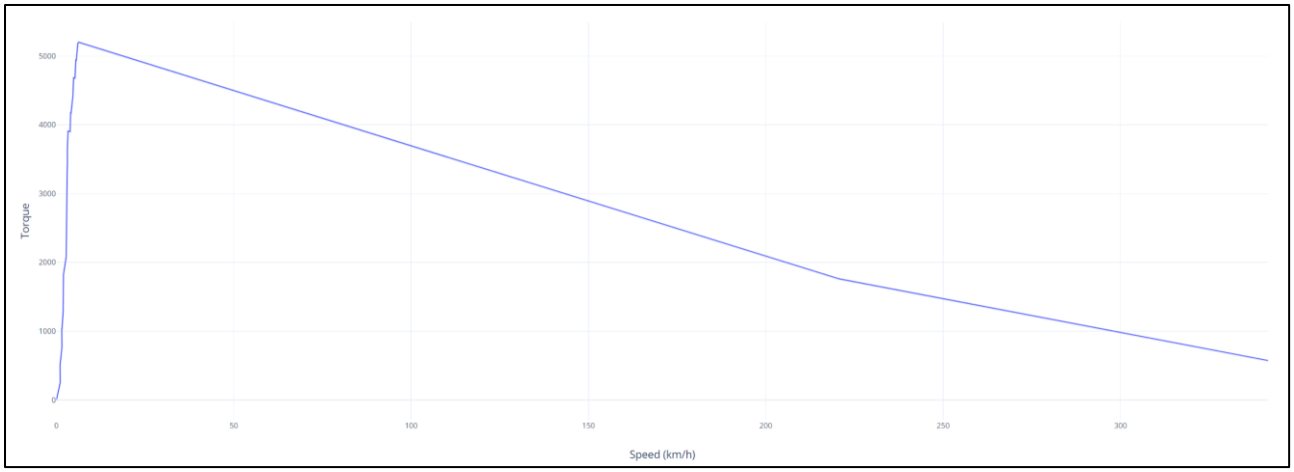
Z66:


Torque
3000
2500
2000
1500
1000
500
0
50
100
150
200
Speed (km/h)

GTO:


Torque
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
Speed (km/h)

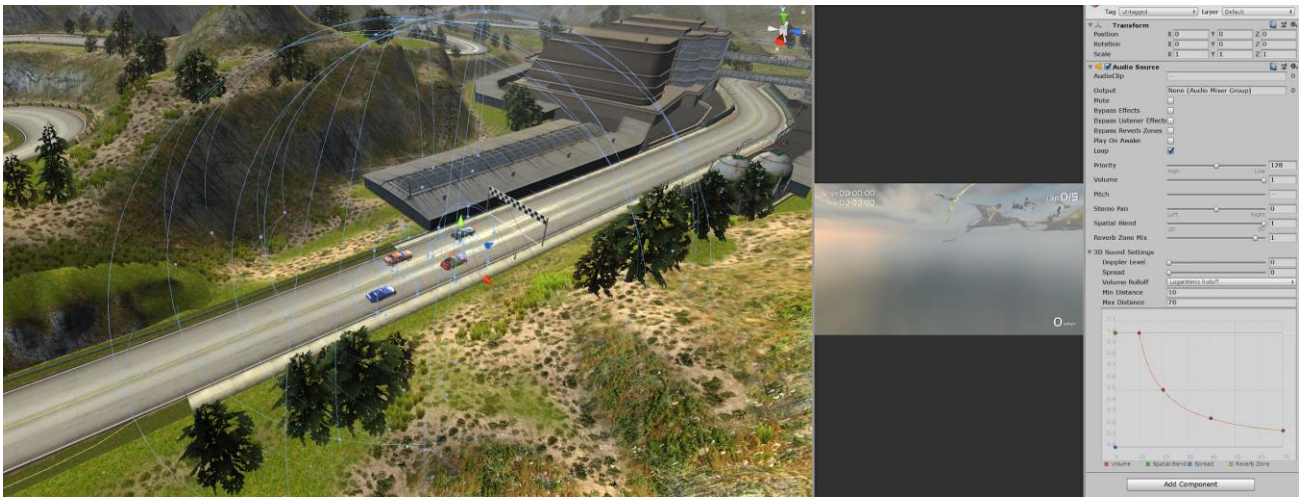
GTX:




```
public void Torque(float scaledTorque) {
        //If speed less than dropSpeed, increase acceleration (refer to docs)
        if (Speed() < dropSpeed) {
            scaledTorque = Mathf.Lerp(scaledTorque, scaledTorque / 3f, Speed() / dropSpeed);
        //If speed greater than dropSpeed, decrease acceleration (refer to docs)
        } else {
            scaledTorque = Mathf.Lerp(scaledTorque / 3f, 0, (Speed() - dropSpeed) / (maxSpeed - dropSpeed));
        }

        //Apply torque to specific wheels depending on drivetrain
        if (driveWheel == DriveWheel.AWD) {
            for (int i = 0; i < 4; i++) {
                wheelColliders[i].motorTorque = scaledTorque;
            }
        } else if (driveWheel == DriveWheel.FWD) {
            wheelColliders[0].motorTorque = scaledTorque;
            wheelColliders[1].motorTorque = scaledTorque;
        } else if (driveWheel == DriveWheel.RWD) {
            wheelColliders[2].motorTorque = scaledTorque;
            wheelColliders[3].motorTorque = scaledTorque;
        }
    }
```

As shown in the graphs above you can see the amount of torque (Y-Axis) applied to the wheels and the speed at which the car is going (X-Axis). Dropspeed is the speed at which the acceleration begins to level off. Maxspeed is the maximum speed a car can reach. If the speed is less than the dropspeed the interpolation value t grows between 0 and 1 depending on the speed. If the speed is greater than the dropspeed the interpolation value drops between 1 and 0 depending on the speed. The algorithm shown in the code above creates an acceleration curve shown in the graphs above and applies the scaledTorque value to the wheels by linearly interpolating the first, second and third values in their respective Mathf.Lerp() method. All these values are variables this allows me to have one car controller script and input different values for maxSpeed, dropSpeed and maxTorque. Allowing for different car characteristics, this can also be seen in the driveWheel enum which I implemented, this checks if the car is either rear/front/4 wheel drive and will apply torque to the wheels depending on the car.

Car Sounds:



Engine sounds were extremely finicky and hard to implement. After a lot of research, I have finally found a selection of engine sounds which I could use for my cars. I wanted to implement a different sound for each car, to add to the realism of this game. Engine sounds also add a sense of atmosphere to the game. I found a selection of engine sounds found here:
https://www.mediafire.com/folder/7d3mwfrbjytd4/FM4_soundbanks_%40_VStanced.com

After numerous failed attempts at implementing my own engine sound script, I found that it is near impossible to get pitch and amplitude to sound correct, therefore I decided to import a script used by other many car games found here:
https://pastebin.com/ikUeaA70

Initially, I had a few problems with this script, please see section 5.4 for further details on how I resolved this issue. Next, I configured 3D sounds within the Unity Editor, The larger sphere (shown in the screenshot above) corresponds to the maximum distance a sound can be heard. A drop off curve is then configured with the relative amplitude and Doppler effect.

Sensors:

```
        left = new Vector3(-0.6f, 0, 0.7f);
        leftForward = new Vector3(-0.3f, 0, 1f);
        forward = Vector3.forward;
        rightForward = new Vector3(0.3f, 0, 1f);
        right = new Vector3(0.6f, 0, 0.7f);
```

The sensors (left, leftForward, forward, rightForward, right) are drawn from the centre point of the car, when this vector comes in contact with an item on the track it will turn red. The length of these values can be tweaked by the user in LearningMode, I have decided to set up 5 sensors, these 5 sensors are inputs for the neural network. All 5 sensors are stored in an array, and get processed through the GetSensor() method within the Sensor class, which returns a value between 0 and 1 if the sensor is hitting anything, otherwise the value returned will be 0.

4.2 Genetic (NN)

My implementation of the genetic algorithm works as follows:

1. Create an initial population of randomly generated children (size given by the user).
2. Score each child based on a fitness function selected by the user.
3. At the end of a generation, select the 2 most fit members of the population.
4. Mutate new children randomly based on the mother and father.
5. Repeat step 2 with the new children mutated in step 4.
6. Continue until the car has achieved the desired performance.

```
public GeneticNetwork() {
    this.weights = new List<double[][]>();
    //for each layer create new matrix
    for (int i = 0; i < lengthLayers - 1; i++) {
        double[][] layerWeights = new double[layers[i]][];
        //for each row
        for (int j = 0; j < layers[i]; j++) {
            layerWeights[j] = new double[layers[i + 1]];
            //and for each column
            for (int k = 0; k < layers[i + 1]; k++) {
                //set the current layer to a random weight
                layerWeights[j][k] = GetRandomWeight();
            }
        }
        //add that weight to the list of weights
        weights.Add(layerWeights);
    }
}
```

The init method for the GeneticNetwork class shown above creates a random List<double[][]> weights, which act as the first set of children in generation 0. The method is written in such a way that the developer can change the architecture of the network just by changing the layers array. The random values are in range of -1 and 1, and retrieved using the GetRandomWeight() method.

```
public List<double> FeedForward(double[] inputs) {
    //Create list of outputs and populate it with inputs
    List<double> outputs = new List<double>();
    for (int i = 0; i < inputs.Length; i++) {
        outputs.Add(inputs[i]);
    }
    for (int i = 0; i < lengthLayers - 1; i++) {
        List<double> sums = new List<double>(new double[layers[i + 1]]);
        //for each output and for each hidden and output layer
        for (int j = 0; j < outputs.Count; j++) {
            for (int k = 0; k < layers[i + 1]; k++) {
                //multiply inputs by the weights of the current car
                sums[k] += outputs[j] * weights[i][j][k];
            }
        }
        //Apply sigmoid function on each sum
        outputs = Sigmoid(sums);
    }
    return outputs;
}
```

For inputs, the FeedForward method in the neural network takes in the distances returned by the sensors mentioned in section 4.1. The inputs are multiplied by the weights and activated using a

Sigmoid function. The outputs of the first layer are used as inputs for the second layer. The final result is then returned as outputs. There are two outputs, one for steering and one for the engine, the cars brake automatically if the motor torque drops below a certain number.

```
public void OnCollisionEnter (Collision col) {
        CalculateFitness();
        ResetCarPosition();
        //display the current child fitness
        Display += "Child "+ (currentChild + 1) +":  "+ Fitness[currentChild].ToString("f2") + "\n";
        PopulationText.GetComponent<Text>().text = Display;
        //if all children have died
        if (currentChild == Fitness.Length - 1) {
            int[] FitestParents = new int[2];
            //find 2 of the fittest parents
            FitestParents = FindFittestParents(Fitness);
            GeneticNetwork Father = Cars[FitestParents[0]];
            GeneticNetwork Mother = Cars[FitestParents[1]];
            //create a new generation of children based on 2 fittest parents
            for (int i = 0; i < Population; i++){
                Fitness[i] = 0;
                Cars[i] = new GeneticNetwork(Father, Mother, Probability);
            }
            //increment generation, reset current child and fitness display
            Generation++;
            currentChild = -1;
            Display = "";
        }
        currentChild++;
}
```

The cars are fitted with an OnCollisionEnter (shown above) when a car collides with something the next child takes control of the car, but before that, the position of the car and the lap times are reset.

```
    public int[] FindFittestParents(double[] Fitness) {
        int[] FittestParents = new int[2];
        double MaxFather = 0; //Biggest Value
        double MaxMother = 0; //2nd Biggest Value
        //For each fitness value
        for (int i = 0; i < Fitness.Length; i++) {
            double temp = Fitness[i];
            //and check if temp is larger than highest
            if (temp > MaxFather) {
                MaxMother = MaxFather;
                MaxFather = temp;
                FittestParents[0] = i;
            //and find second largest
            } else if (temp > MaxMother && temp <= MaxFather) {
                MaxMother = temp;
            }
        }
        //assign -1 to father
        Fitness[FittestParents[0]] = -1;
        //find index of mother
        FittestParents[1] = Array.IndexOf(Fitness, MaxMother);
        //display fitness of mother and father on screen
        FatherText.GetComponent<Text>().text = "Father: Child " + (FittestParents[0] + 1) + " (" +
MaxFather.ToString("f1") + ")";
        MotherText.GetComponent<Text>().text = "Mother: Child " + (FittestParents[1] + 1) + " (" +
MaxMother.ToString("f1") + ")";

        return FittestParents; }
```

When all cars in the generation have died, then new children are created based on who has the highest fitness, the method is shown above (FindFittestParents) finds the two highest parents while retaining their index in order to keep the same weights when they are being mutated. I have created 3 different fitness functions. First, one is simply the distance travelled. Second is the distance travelled divided by the amount of time that has passed, this forces cars to drive faster. The third one begins as distance travelled, but as soon as a car completes a full lap it begins to compare the different lap times that the cars complete and taking them as their respective fitness values, this way when a new child performs a worse lap than the previous will be disregarded.

```
public GeneticNetwork(GeneticNetwork Father, GeneticNetwork Mother, float probability) {
        System.Random randomBool = new System.Random();
        this.weights = new List<double[][]>();
        //For each layer create new matrix
        for (int i = 0; i < lengthLayers - 1; i++) {
            double[][] layerWeights = new double[layers[i]][];
            //for each row
            for (int j = 0; j < layers[i]; j++) {
                layerWeights[j] = new double[layers[i + 1]];
                //and for each column
                for (int k = 0; k < layers[i + 1]; k++) {
                    //randomly crossover either mother or father
                    if (randomBool.Next(2) == 0) {
                        layerWeights[j][k] = Father.weights[i][j][k];
                    } else {
                        layerWeights[j][k] = Mother.weights[i][j][k];
                    }
                    //mutate that layer to a random weight if it falls under the probability
                    if (Random.Range(0f, 1f) < probability) {
                        layerWeights[j][k] = GetRandomWeight();
                    }
                }
            }
            //Add that new weight to the new list of weights (children)
            weights.Add(layerWeights);
        }
}
```

After both parents have been chosen a new population is created based randomly on both parents, also a random weight to a random layer is assigned in order to avoid being stuck in local maximum (shown in the code above). In learning mode, the user also has the option to set the mutation probability for new children. Eventually, when the car has reached its peak performance in its respective fitness function and there is no room (or very little room) for improvement the weights are frozen in order to later be assigned when playing against a user. When saving/loading the weights used in a trained network I used a similar method as to the Neural Network which goes through the entire List<double[][]> and writes the values into a text file, similarly the load method goes through the file and puts the values into a List of matrices which is assigned to the weights of a child, therefore there is no need to train that child and it can be used to race against a player.

4.3 Backpropagation (NN)

My implementation of the genetic algorithm works as follows:

1. Create an initial set of random weights.
2. Feedforward these weights with the inputs provided by the car sensors.
3. Compare the user input (controls) to the output of the feedforward of the weights.
4. Adjust the weights to reduce the error (difference between user controls and feedforward outputs) of the output.
5. Repeat steps 2-4 until the error becomes as small as possible.
6. After a given amount of time, the user input will be turned off and the algorithm will pick up on its own.

```
public double TanH(double value) {
    return 1 - (value * value);
}
```

To begin, I used a similar init and feedforward method as the one from my genetic algorithm (see Section 4.2), I only did a few small changes to accommodate for the new propagation of data throughout the algorithm, I added more lists to be able to store outputs, inputs, loss and weights derivative. While the user is controlling the car, the feedforward and backpropagation methods are running in tandem. As the feedforward method is running it saves the current layer inputs and outputs, this happens for every layer in the network. Meanwhile, the backpropagation method is working in reverse, by inputting the outputs (user input) and propagating it through the network in reverse. For every layer, it compares the input of that layer, to the output of the previous layer, and by doing it so it calculates the error. For genetic I used Sigmoid, therefore for this algorithm I decided to use TanH (shown above).

```
    public void Propogate(double[] inputs, double learningRate) {
        this.learningRate = learningRate;
        //For all layers (run in reverse)
        for (int l = lengthLayers - 2; l >= 0; l--) {
            //if its the last layer
            if (l == lengthLayers - 2) {
                for (int i = 0; i < l; i++) {
                    //calculate error (user vs actual) and multiply by activated output to get loss
                    double error = layerOutputs[l][i] - inputs[i];
                    this.layerLoss[l][i] = TanH(layerOutputs[l][i]) * error;
                }
                CalculateDerivative(l);
            //if its the hidden layers
            } else {
                for (int i = 0; i < layers[l + 1]; i++) {
                    //calculate current layer loss based on weights and loss of next layer
                    layerLoss[l][i] = 0;
                    for (int j = 0; j < layerLoss[l + 1].Length; j++) {
                        layerLoss[l][i] += weights[l + 1][j][i] * layerLoss[l + 1][j];
                    }
                    //activate loss using the tanh function
                    this.layerLoss[l][i] *= TanH(layerOutputs[l][i]);
                }
                CalculateDerivative(l);
            }
```

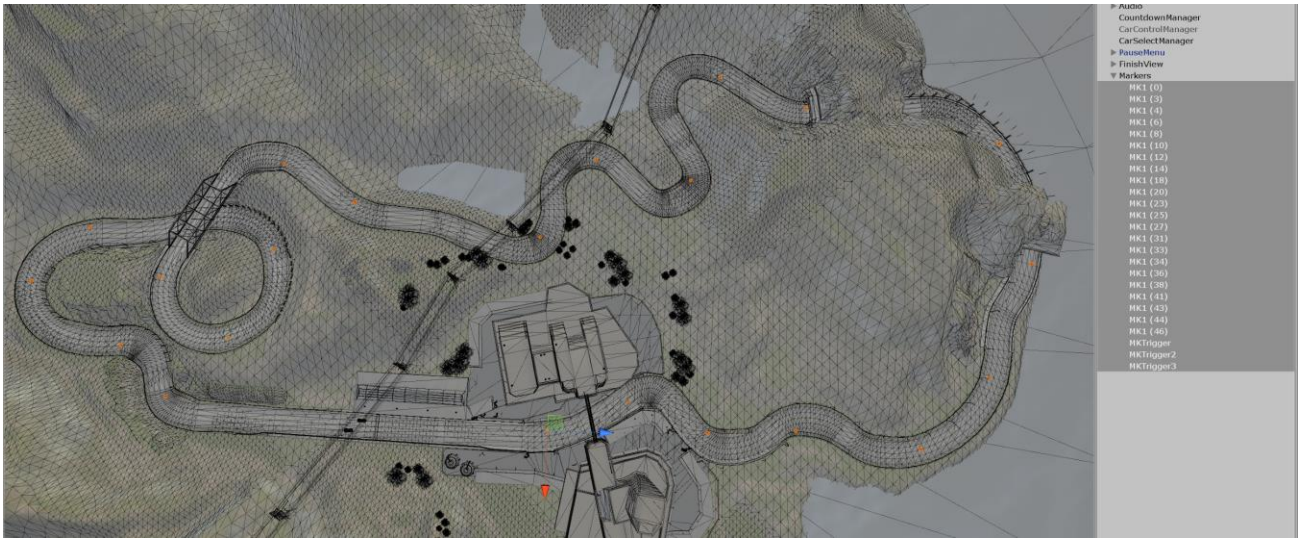
```
            //for each output layer
            for (int i = 0; i < layers[l + 1]; i++) {
                //and each input layer
                for (int j = 0; j < layers[l]; j++) {
                    //update the weights and smooth them by the learning rate
                    this.weights[l][i][j] -= weightsDer[l][i][j] * learningRate;
                }
            }
        }
    }
```

The weights are subsequently adjusted depending on the difference in outputs vs expected outputs. This error is activated using TanH and the loss is then multiplied by the inputs of the current layer in order to get the weights derivative. Lastly, the weights derivative are multiplied by the learning rate (this number changes depending on user input) to smooth it, and this number is taken away from the actual weights. This process repeats itself until the loss is brought down to as little as possible. The code shown above shows the new method I implemented to achieve this new propagation of data different from the genetic algorithm. This algorithm uses gradient descent to reduce the error, a weight starts off being too high or too low, but as the backpropagation method is recording the expected outputs of the weights in comparison to the outputs given by the feedforward method we can change the weights in accordance with reducing the error, as shown in the graph above. This algorithm is also be used in Competition mode as one of the drivers for a car.

```
        //if still training
        if (lapCount <= NoLaps) {
            //feed inputs, backpropogate expected values and hide training panel
            backProp.FeedForward(Inputs);
            backProp.Propogate(new double[] { steer, torque }, LearningRate);
            AutoPanel.SetActive(false);

        //when finished training
        } else {
            //set the backprop alg as driver, feed inputs to create outputs
            Rigidbody.GetComponent<CarPhysics>().driver = Driver.BackProp;
            Outputs = backProp.FeedForward(Inputs);

            //set outputs to CarController class
            controller.BackPropSteering = Outputs[0];
            controller.BackPropEngine = Outputs[1];

            //show training panel and tell user to release all controls
            AutoPanel.SetActive(true);
            if (runOnce) {
                StartCoroutine(CountStart());
                runOnce = false;
            }
        }
```

When in BackPropagation Learning Mode, the user selects how many laps he/she wants to train the car for. The BackPropController then gets the current lapCount and while the lapCount is less than or equal to the number of laps selected the FeedForward and Propagate methods mentioned earlier will be running. As this method is contained in the Update() method, it runs once every 0.01 seconds. In a scenario where the user has selected 2 laps, and each lap is 1.15min on average the BackPropNetwork algorithm will be running approximately 15,000 times.

4.4 Unity Pathfinding AI



The Unity Pathfinding AI works by following a set of nodes laid out by the developer. These nodes usually correspond to the turns on a track. These nodes can be represented as any GameObject on the track, for my instance, I chose them to be cubes. After I laid out the cubes I turned off their "Mesh Renderer" which in turn made them invisible when playing in the game. I also made their collider be a "trigger" which would stop the cars crashing with the cube, and instead, let them act as triggers for the waypoint system. The nodes can be seen as orange points in the screenshot above.

```
void Update () {
    for (int i = 0; i < MK1.Length; i++) {
        if (MKTracker == i) {
            MKTrigger.transform.position = MK1[i].transform.position;
        }
    }
}

IEnumerator OnTriggerEnter(Collider other) {
    if (other.gameObject.tag == "UnityAI") {
        this.GetComponent<BoxCollider>().enabled = false;
        MKTracker += 1;
        if (MKTracker == MK1.Length) {
            MKTracker = 0;
        }
        yield return new WaitForSeconds(1);
        this.GetComponent<BoxCollider>().enabled = true;
    }
}
```

Although Unity provides both the waypoint system and the AI, I decided to create my own waypoint system as the built-in one was quite outdated and gave me bad results when the Unity Car AI was implemented. The code above shows my own waypoint system. It uses a list of GameObjects to iterate through them and put the MKTrigger to the location of the next node. This only happens when the current trigger has collided with the car which attached with the Unity Car AI controller. Similarly, with the Unity Car AI controller, I had to do some alterations to make it perform faster around the track. In the beginning, the car I was testing was getting approximately 1.45min/lap, this was hugely unacceptable as personally, I was getting about 1.15min/lap (although I had a lot of practice during

testing). My goal was to have it perform at least 1.25min/lap which would make it more competitive in relation to other users/AI. I made it better by tweaking the position of the nodes and the Unity Car AI algorithm. This was an extremely long and cumbersome process since it was a car physically going around a track I couldn't automate the process in the console, and I had to watch it drive around each time getting better and better. Spending so long implementing this feature is the biggest reason I'm making my own AI for this project since my AI can learn by itself on any track without any supervision. This creates a much better and faster implementation of opponent cars. For example, If I had another track I would have had to repeat the process mentioned above again and spend hours shaving time off lap times. Instead, my Neural Network can do it on its own and achieve much better lap times.

4.5 Lap Time/Race Physics

First I did the time tracking and lap counter. Unity provides developers with a function called Time.deltaTime, but this only returns a timer, in order for me to time the cars I had to multiply this value by 10, which would return a more accurate representation of the time. Then for every 1000 milliseconds, I added 1 second, and for every 60 seconds I added 1 minute, I decided to leave hours out. I parsed milliseconds to only show 2 decimal points.

```
void FixedUpdate () {
        //Turn seconds into milliseconds and remove decimal
        milliseconds += Time.deltaTime * 10;
        milliCount = milliseconds.ToString("f1").Replace(".","");
        //For every 10 milliseonds add 1 second
        if(milliseconds >= 10) {
            milliseconds = 0;
            seconds += 1;
            lapTime += 1;
        }
        //For every 60 seconds add 1 minute
        if (seconds >= 60) {
            seconds = 0;
            minutes += 1;
        }
        //Format the laptime as seen in game
        LapTime.GetComponent<Text>().text = ""
            + Convert(minutes) + ":"
            + Convert(seconds) + "."
            + milliCount.Substring(0, 2);
}
```

Next, I had a problem, and that was the timer was showing the time 1:5.45, and I needed to show it in the format 01:05.45, which would stop the element on the screen constantly changing its size. To fix this I wrote a simple method called Convert() which added a 0 in front of numbers less than 9.

```
public static string Convert(int x) {
    if (x <= 9) {
        return "0" + x;
    } else {
        return "" + x;
    }
}
```

Lap counters and lap times are counted using colliders set on the finish line, the finish line will only activate if the half point marker was activated, this will stop people cheating on the track. The first time the car goes around the track the time is saved as "best time" and if the user beats that time a

new best will be recorded. The lap counter is done the same way by increasing the value every time the user crosses the finish line (given that they also crossed the halfway line).

Timetrial:

Before starting the time trial mode I needed to create a new Scene. When starting a new scene you can attach it with all the previous scripts you wrote. This saves time as you can make one "CarPhysics" script which can interact with all cars, this is good practice in both programming terms and unity game design. Previously, I was testing my car physics and neural network, therefore, I only needed one scene, but now that features diverge I must create more scenes, where each scene will work in different ways, for example, the time trial and competition will be using the CarPhysics script but each will have different scripts in order to define what they are. Luckily, for the timetrial mode, I have most of the features I need, the only thing that is left to load it with a selected car. So far I have the all the necessary time tracking scripts and car physics/camera scripts. In a timetrial, a user should be able to drive around a track indefinitely, therefore, there won't be a scenario where the race finishes, except for when the user wants to exit the game. After setting up the new scene called "TimeTrial" I loaded all necessary scripts, models, variables and visual details (these include lighting, display settings, UI elements). Now when testing it when I load the scene on its own there shouldn't be any cars present as a user must first load a car, and depending which car he/she chooses will be the car the will load in.

Competition:

Unlike the Timetrial mode, Competition has a set number of laps that a user must complete in order to finish the race. First in my lap script I added logic to check if the selected game mode is "Competition", if it is then set the max laps to a variable amount and on the last lap when the user finishes the race display the position and a finish screen (from where the user can return to the main menu). Next, I created a screen called "Finish Screen" which shows up as soon as the player crossed the finish line, it disables all controls, HUD and sounds and allows the user to return to the main menu.

```
public static void WriteToFile(string filename, List<double[][]> weights) {
     using (var file = new StreamWriter(filename)) {
         //for each weight in weights
         foreach (var weight in weights) {
             //for each row in weight
             foreach (var i in weight) {
                 //and for each column in row write it to file
                 foreach (var j in i) {
                     file.WriteLine(j.ToString());
                 }
             }
         }
     }
}
```

This mode implements AI previously discussed to drive the car around the track autonomously (opponent cars). But, instead of the cars progressively learning the cars are using pre-trained models. The learning data was frozen at the fittest point it could achieve in a given amount of time and then

utilised in the Competition Mode. The weights are then saved using the SaveWeights class, used by both GeneticNetwork, and BackPropNetwork.

```
public class GeneticWeights : GeneticNetwork {
    //Loades weights from text file
    public List<double[][]> LoadWeights(string fileName) {
        List<double[][]> weights = new List<double[][]>();
        using (var file = new StreamReader(fileName)) {
            //for each layer
            for (int i = 0; i < lengthLayers - 1; i++) {
                double[][] layerWeights = new double[layers[i]][];
                //for each row
                for (int j = 0; j < layers[i]; j++) {
                    layerWeights[j] = new double[layers[i + 1]];
                    //and for each column read a line and add it to layerweights
                    for (int k = 0; k < layers[i + 1]; k++) {
                        layerWeights[j][k] = double.Parse(file.ReadLine());
                    }
                }
                //add that layerweight to weights list
                weights.Add(layerWeights);
            }
        }
        return weights;
    }
}
```

When in Competition mode, the weights that were saved need to be loaded back into GeneticDriver and BackPropDriver. In order to do this, I created the GeneticWeights class (shown above) which inherits from the GeneticNetwork, it goes through a text file which contains a set of weights and adds them back into a List<double[][]>. Similarly, the BackPropWeights class (not shown in this document) inherits from the BackPropNetwork class and works in a similar manner.

```
    //shuffles list of cars in a random order, based on the Fisher-Yates shuffle
    public static List<GameObject> Shuffle<GameObject>(List<GameObject> Cars) {
        int n = Cars.Count;
        System.Random rnd = new System.Random();
        while (n > 1) {
            int k = (rnd.Next(0, n) % n);
            n--;
            GameObject value = Cars[k];
            Cars[k] = Cars[n];
            Cars[n] = value;
        }
        return Cars;
    }
```

The 3 other opponent cars use:

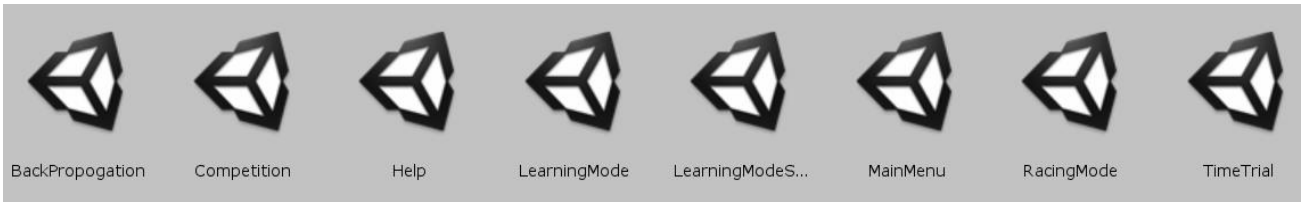
- Genetic Algorithm (Neural Network)
- BackPropagation (Neural Network)
- Pathfinding (Unity)

All cars are added to a list called Cars. When a user selects a car, that car is removed from the list. The remaining three cars are assigned to a random AI algorithm from the list shown above. Before being assigned, the list of Cars is shuffled using the Fisher-Yates shuffle shown in the algorithm above.

4.6 User Interface

```
private void LateUpdate() {
        //Set the camera to follow the selected car
        selectedCar = CarSelectController.SelectedCar;
        if (selectedCar != null) {
            car = selectedCar.transform;
        } else {
            Debug.Log("Camera object not found");
        }
        //Set variables (depending on car)
        float wantedAngle = rotation;
        float wantedHeight = car.position.y + height;
        float carAngle = transform.eulerAngles.y;
        float carHeight = transform.position.y;

        //Find angle and height of camera depending on set variables
        carAngle = Mathf.LerpAngle(carAngle, wantedAngle, steerRotation * Time.deltaTime);
        carHeight = Mathf.LerpAngle(carHeight, wantedHeight, heightRotation * Time.deltaTime);

        //Set new camera angle based on car position and follow the car
        Quaternion currentRotation = Quaternion.Euler(0, carAngle, 0);
        transform.position = car.position;
        transform.position -= currentRotation * Vector3.forward * distance;
        Vector3 temp = transform.position;
        temp.y = carHeight;
        transform.position = temp;
        transform.LookAt(car);
    }

    void Update() {
        //Get velocity, if its less than 10km/h (in reverse) turn the camera around (for reversing)
        Vector3 velocity = car.InverseTransformDirection(car.GetComponent<Rigidbody>().velocity);
        if (velocity.z < -3f) {
            rotation = car.eulerAngles.y + 160;
        } else {
            rotation = car.eulerAngles.y;
        }
    }
```

When attaching a camera as an object of the car the camera is stuck to the car. Instead, I wanted the camera to follow the car and when the car turns the camera would stay (mostly) stationary and the user would be able to see the side of the car when it is turning. Another problem that lies with the camera being stuck to the car is that if it flips the screen would be upside down aswell. To eliminate this I implemented a camera script shown above. Instead of the camera being a child object of the car like it was before, the new camera script follows the car. As the car velocity increases/decreases so does the velocity if the camera. The height and distance away from the car are variables, allowing fine-tuning the position of the camera depending on the car. When a car turns a damping force acts on the camera which counteracts the rotation of the car to the camera, allowing the camera to remain straight while the car turns. Now when the car rotates you can see the side of the car, this gives it a nicer look and feel while driving the car. Also, now when a user flips a car, or encounters a bumpy part of the track the camera won't be jittery and flip with the car, this creates a smoother and more aesthetically pleasing driving experience.



First I started by creating the various Scenes in the unity editor, as shown above. While I am testing I did not focus on how well things look, rather just make sure everything works together. I set out the various buttons needed on each screen and added it to the "Scenes in Build", scenes in the game so far are shown in the screenshot above. After I set up all the different navigation screens I started to add scripts to each one in order to define their functionality. Throughout the development of the project, I implemented Shneiderman's Eight Golden Rules. All buttons, text, layouts and features were developed with consistency in mind. I tried to implement good game development practices, for example, when configuring the pause menu I created a darker overlay which would enable when the "Esc" key is pressed, this takes the attention away from the race and focuses it on the buttons.

```
public void ReturnToMainMenu() {
    SceneManager.LoadScene(0);
}

public void QuitGame() {
    Debug.Log("QuitGame!");
    Application.Quit();
}
```

In the example shown above the first method is mapped to all the "Main Menu" buttons which returns a user to the main menu, it runs Scene(0), which I have defined as my main menu scene in "Scenes in Build".

```
public void TimeTrial() {
    SelectedRace = 1;
    CarSelect.SetActive(true);
}
```

```
    public void EVO () {
        SelectedCar = 1;
        SetInactive();
        ShowEVO.SetActive(true);
        StartGame.SetActive(true);
    }
```

The two methods above correspond to a user clicking either the "Time Trial" button or the “EVO” button found in RacingMode. The static variable SelectedRace code corresponds to a time trial, and when a user clicks that button the option for a Car Selection will be given where they can select a car. The static variable SelectedCar will also be used later by the CarSelectManager to decide which car was chosen by the user. Clicking on a car also activates the “Start Race” button. This way has two benefits, it forces a user to select an option (reducing errors), and saves me time by creating multiple scenes.

HUD:

```
BestTime.GetComponent<Text>().text = ""
                    + LapTimeManager.Convert(LapTimeManager.minutes)
                    + ":" + LapTimeManager.Convert(LapTimeManager.seconds)
                    + "." + LapTimeManager.milliCount.Substring(0, 2);
```

The HUD, which is comprised of things such as lap counters, current speed and lap timers are all made of on-screen objects of type Text, Button, Slider etc. When playing in a race or observing learning mode these objects are displayed in the User Interface. In the case of the BestTime, which can be seen in TimeTrial and Learningmode, I made a Text object and attached it to the on-screen element. Within my script, I set the BestTime (Text) object if the time is less than the current lap time. The time is retrieved from the LapTimeManager and transformed using the Convert() method mentioned in section 4.5.

```
void Start () {
    //Set dropdown as available fitness values
    string[] fitnessNames = Enum.GetNames(typeof(FitnessMeasure));
    List<string> names = new List<string>(fitnessNames);
    dropDown.AddOptions(names);
}

//Update 2 Dropdown and Population slider values
void Update() {
    dropDown.onValueChanged.AddListener(delegate {
        DropdownValueChanged(dropDown);
    });
    //Set population text to the slider value
    Population = (int)Slider.value;
    PopulationText.GetComponent<Text>().text = "Population: " + (Population);
}

//When dropdown value is changed, also change the fitnessMeasure static var
void DropdownValueChanged(Dropdown change) {
    fitnessMeasure = (FitnessMeasure)change.value;
}
```

When in LearningMode selection, there are a number of on-screen elements which can be controlled by the user. These are set up as shown in the code above. The values from the fitnessNames enum are added to the DropDown menu and the one which is selected is assigned to the fitnessMeasure static variable. Similarly, the Slider.value (which is attached to a specific object in Unity Editor) changes the static int Population, which is used later in the GeneticController.

Button Sounds:

To conform with "Informative feedback" in Shneiderman's Eight Golden Rules I needed to add a tactile sound to all the buttons throughout the system. The goal is for users to know when/if they pressed a button. Adding sounds to buttons was an easy and mostly mechanical process within Unity, the problem was finding an appropriate sound which didn't stand out too much. Next, I had to tweak the volume of a button click, if it was too loud it would become annoying to press buttons for the user.

5. Problems & Resolution

5.1 Wheel Models:

After mapping the wheel colliders to the wheel meshes I found that the front right and rear right wheels are both rotated 180 degrees on the wrong axis. This is because when you add a wheel collider it only faces one direction, and in order to rotate it you need to write a script to rotate the wheels 180 degrees as it moving.

```
if (i == 0 || i == 2){
    rot = rot * Quaternion.Euler(new Vector3(0, 0, 180));
}
else{
    rot = rot * Quaternion.Euler(new Vector3(0, 0, 0));
}
```

As the wheel meshes are stored in a list then i[1] and i[3] are the front-right and rear-right wheels, I rotated the wheel by 180 degrees in the Y-axis and now all 4 wheels face the correct direction. Using the same principle I made the wheel meshes (rims) rotate in the same fashion as the car drives on the terrain. I also implemented a "centre of gravity" to prevent the car from flipping over while driving through intense corners, I've had some trouble with the 3D model axis at first, to fix this I created an empty object and made it faces the axis I need and moved all the objects of the car into the new temporary object in order to correct the axis of the car.

5.2 Reset Race Positions:

```
    public GameObject ResetCurrentCar(Rigidbody Body, Vector3 Position, GameObject[] Points) {

        //set closes distance to infinity
        float closestDistance = Mathf.Infinity;
        GameObject closest = null;

        //check each point and find one that's closest to the car
        foreach (GameObject point in Points) {
            float currentDistance = (point.transform.position - Position).sqrMagnitude;
            if (currentDistance < closestDistance) {
                closestDistance = currentDistance;
                closest = point;
            }
        }
        return closest;
    }
```

After I got all the three algorithms to work in Competition Mode, I found that cars often crashed with each other. This was especially apparent with the UnityAI which had no collision avoidance like the two machine learning algorithms created by me. Due to this, the UnityAI would often crash into a wall or another car and subsequently make both cars stuck in one position. The main problem is that when a car is stuck it will remain in that position for the entire duration of the race. To fix this issue I developed a way for the cars to reset their position on the track if they have been stationary for a period of time. I created an abundance of spawn points around the track and fitted each car (except the user car) with a function that resets its position to one of these spawn points (whichever is the closest one). To find the closest point I created a ResetPosition class which can be used across all AI

algorithms. The ResetCurrentCar() method then takes in a list of spawn points around the track, the car position, and the car Rigidbody, and finds the closest spawn point to the car. When the point is returned the position and rotation of the car are set at the closest spawn point on the track.

5.3 Loading Screen:

When transitioning from one scene to another I have found that there is a slight delay between pressing the button and the scene appearing. As I was working on a powerful enough computer it was my suspicion that when this game will be built and running on other computers that this delay will increase a lot. During the user testing phase, I found that when loading into a race this delay increased dramatically. To solve this issue I decided to add a loading/transition screen between scenes that require a lot of time to load. I implemented a simple progress bar with a percentage which will be a great indicator of how much has loaded. It also gives a user the impression that the game isn't "stuck". To do this I created a canvas which activates the one you select to load a game, but now instead of loading it instantly, the GameLoader script runs an AsyncOperation which halts the loading of a new level until it has already loaded. Within this method, the operation.progress also returns a value on how much of the level has been loaded. Overall this created a much smoother transition between levels.

5.4 Gear Change Sounds:

```
public void ToRPM() {
    Speed = Car.velocity.magnitude * 3.6f;

    if (Speed <= firstGear) {
        RPM = Speed * 55;
    } else if (Speed <= secondGear) {
        RPM = Speed * 27;
    } else if (Speed <= thirdGear) {
        RPM = Speed * 20;
    } else if (Speed <= fourthGear) {
        RPM = Speed * 18;
    } else if (Speed <= fifthGear) {
        RPM = Speed * 15;
    }
}
```

After implementing the engine sound script I found a massive incompatibility between how they work together. The Torque() method which I created works more like a single gear electric motor than an actual engine. Whereas the engine sound script breaks the engine sound into RPM ranges which depending on the RPM of the engine will increase/decrease the amplitude and frequency of the selected sound. Therefore when I ran the script I found that the engine sound would accelerate indefinitely since the RPM wouldn't iterate from 0-X depending on the gear. To fix this issue I got the speed of the car and broke it up into a series of gears. I created a ToRPM() method (shown above) which converted the speed into its relative RPM. Now each engine sound (from 1-10) gets maxed out depending on the gear the car is in. This created a good transition when the acceleration was increasing, but when it was decreasing quickly then it created an unpleasant sound, especially when the player activates the brakes on the car. To fix this I added skid sounds to mimic tire squeal when

a car brakes at high speeds. To create skid sounds I looked for free looping skid sounds. When sounds are looping it means that when it is continuously being played it doesn't sound interrupted. The brake sound is played when the speed of the car is greater than 1, this means that there won't be a sound when the car isn't moving. When I added the skid sound it removed the problem with decreasing acceleration by masking the sound. I also tweaked the algorithm to continuously play a low-frequency engine sound when the car is idle, this removed the silence when you are in-game and adds a sense of atmosphere.

5.5 Postprocessing:

When completing the project I found that it lacked a visual aesthetic. While this is a small part of the project that doesn't require any programming I decided that it is important that my game looks good. For this I used a number of post processing techniques such as motion blur, vignette, anti-aliasing, bloom and ambient occlusion. When all these are applied it makes the game more cinematic and better looking. This part is completely subjective to each person therefore I tried my best not to overdo each one as it would make the game look messy. I used the tutorial linked above to help me with this section.

6. Testing & Validation

During the development of AIRace, I implemented the agile software development methodology with a focus on continuous testing and integration. This allowed me to easily make alterations to my design while still delivering frequent working and tested code. Because this project relies on physical movement throughout the environment (for example driving a car), a lot of testing was done by interacting with the physical object in a scene where unit testing was not possible. This meant that I had to implement Unit Testing as much as possible so that I could easily check and modify code without performing that action in the game (which can take a lot of time depending on the action). During the lifetime of this project the following tests were performed:

1. Unit Testing
2. Integration Testing
3. Physics Testing
4. GUI Testing
5. System Testing
6. Acceptance (User) Testing
7. Shneiderman's Eight Golden Rules
8. Interface Testing

When performing User Testing, both qualitative and quantitative tests were done in order to remove any inherent bias of the user. AI Architecture Testing involved changing the attributes and architecture of each algorithm in order to yield the best possible lap times. For further details on each section please refer to the testing document located in the following location in my Gitlab repository: https://gitlab.computing.dcu.ie/khaletk2/2019-ca400-khaletk2/tree/master/docs/documentation

7. AI Architecture Lap Time Testing

7.1 Genetic Network

After completing all unit and integration testing between the CarPhysics, Sensor and Genetic classes I started to reconfigure the Genetic Network architecture and attributes. Tweaking the layer sizes, population sizes, mutation rates, sensor lengths and fitness function yielded different lap times and learning rates.

Attributes	Iteration 1	Iteration 2	Iteration 3	Iteration 4
Layers (input, hidden, output)	5,6,2	5,8,2	5,10,2	5, 8, 8, 2
Best Time	81.43	78.62	78.71	78.94
Generation of First Lap	9	6	10	8

I chose 5 as the total amount of sensors as it was an adequate amount of coverage for the path of a car, which was the input layer. The output layer contains 2 nodes, this corresponds to the Engine and Steering value of the car. To decide the number of hidden layers and nodes I first started with the smallest number, then went up recording the lap times and learning rates each time. Here I found that going for a single hidden layer of 8 nodes yielded best results in both the lap times and learning rates. Although the layers {5,10,2} gave slightly better lap times (0.2 seconds), I have ultimately decided to use {5,8,2} because it provided much faster learning rates and similar lap times.

Attributes	Iteration 1	Iteration 2	Iteration 3
Population	10	20	30
Best Time	78.21	77.03	77.18
Generation of First Lap	17	8	6

Population sizes have made a huge impact on the amount of time it takes for a car to complete a lap. Because there is a smaller amount of children to choose from it takes more generations to learn. The times in this instance remain within a margin of error, nevertheless, I found that a population size between 15-20 works best as each generation gets mutated more in the amount of time it takes for all children to die.

Attributes	Iteration 1	Iteration 2	Iteration 3	Iteration 4
Mutation	2	5	10	20
Best Time	75.11	74.17	75.48	77.62
Generation of First Lap	9	6	5	12

Mutation rate is the number of weights that have been mutated from the new set of children that are derived from the Mother and Father car. In this case, I found that 5% and 10% mutation rates yield the best results, although 5% is better for speed and 10% is better for learning rate, both numbers are in margin or error of each other. I noticed that setting this value too high (20%) increases both lap times and learning rates.

Attributes	Iteration 1	Iteration 2	Iteration 3	Iteration 4
Sensor Length (Units)	10	20	30	40
Best Time	77.41	77.84	75.11	79.56
Generation of First Lap	7	7	6	4

Sensor length is the distance at which the sensors on a car are effective. When testing this attribute I found that anything below 25 units makes the car stick to one particular side of the road (in a random fashion). This is due to the fact that the sensors don't cover the entire width of the track. On the other hand, when setting the length to 40 I found that the sensors reach too far ahead and the car brakes too much, which dramatically increases lap times but decreases the learning rate to only 4 generations. I found that 30 was a sweet spot for learning rate and lap times.

Attributes	Iteration 1	Iteration 2	Iteration 3
Fitness	Distance	DriveTime	LapTime
Best Time	74.97	74.34	73.74
Generation of First Lap	5	11	11

The fitness function changes which children are chosen as the mother and father for the next generation. During my testing, I found that the Distance fitness function created the fastest learning model, and had the cars crash least often, making it the safest. DriveTime promoted faster cars but increased the learning rate. LapTime function is very similar to DriveTime but it kills cars after they have completed 2 laps, and then compared other cars which have also completed a lap, choosing faster lap times every time. LapTime was best for making cars go faster with each generation.

Outcome:

Attributes	Fastest Lap Times	Quickest Learning
Layers	5,8,2	5,8,2
Population	20	30
Mutation	5	10
Sensor Length	30	40
Fitness	LapTime	Distance

After all the attribute testing mentioned previously, I have come to the results shown above. Although the differences are minute, both sets of attributes yield very different driving styles. The faster lap time attributes use braking a lot less often and rely on reducing its Engine value to turn around corners faster while taking much longer to learn and shows less safe driving style. The quickest learning attributes use brakes much more often, which subsequently yields a much safer driving style.

7.2 BackPropagation Network

Driver	Best Lap Time	BackProp Attributes	BackProp Best Time
User0 (project owner)	70.78	3%, 3 laps	77.36
User1	96.91	10%, 3 laps	N/A
User2	79.12	5%, 2 laps	N/A
User3	78.86	3%, 2 laps	86.71
User4	72.55	3%, 2 laps	80.83
User5	72.91	5%, 3 laps	N/A
User6	71.73	3%, 3 laps	78.48
User7	74.04	3%, 2 laps	79.96
User8	77.72	0.3%, 3 laps	N/A
User9	83.02	5%, 2 laps	87.02
User10	75.41	3%, 3 laps	79.01

Because of the supervised learning nature of BackPropagation, I needed to test it on other users too, not just myself. First I followed a similar testing process as the Genetic Algorithm mentioned in section 7.1 for the architecture of this network. I found that two hidden layers with 10 nodes in each (5, 10, 10, 2) yielded the best results in terms of efficiency and learning rates. I found that teaching the car for anything more than 3 laps didn't give an increase in lap times, and a smoothing rate of 3% yielded best driving characteristics (minimal swaying, unnecessary braking and maximum acceleration during straights). After testing this algorithm on myself I gave it to other users who have been performing my User Testing after they got sufficient practice with driving a car to reduce crashes during learning.

Outcome:

From the user testing, it was apparent that this algorithm heavily relies on how a user drives and their own lap times are directly correlated to the algorithms times, we can see that there is an average of 5 seconds difference between the drivers best time and the algorithms best time. A smoothing rate of 3% worked best while a smoothing rate of 0.3% and 10% made the cars unable to complete a lap. It was also self-evident that sometimes the algorithm was unable to complete a lap simply because the driving characteristics of the user were not compatible with the algorithm itself, making it inferior to a Genetic Algorithm, yet still much quicker in terms of lap times than the UnityAI.

8. Results & Conclusions

8.1 Results

User Testing Lap Times:

Driver	Racing Game Proficiency	Lap1	Lap2	Lap3	Average Time (3 laps)	Best Time
project owner	5	75.35	74.18	72.23	73.92	70.78
User1	2	123.91	110.18	96.91	110.33	96.91
User2	3	91.99	89.03	81.15	87.39	79.12
User3	4	86.5	84.51	85.72	85.57	78.86
User4	4	76.1	74.14	72.55	74.26	72.55
User5	5	85.21	74.73	76.12	78.68	72.91
User6	5	73.04	71.73	73.11	72.62	71.73
User7	4	86.74	77.39	74.04	79.39	74.04
User8	2	81.1	79.26	77.72	79.36	77.72
User9	2	92.44	88.29	86.94	89.22	83.02
User10	4	81.08	79.62	75.41	78.70	75.41

AI & User Best Lap Times:

Driver	Average Best Time	Best Time
User	77.55	70.78
UnityAI	86.53	86.27
Genetic	75.95	73.74
Backpropagation	81.33	77.36

Results:

When examining the user lap times we can see that most users decrease their lap times with every iteration, which is a good sign that AIRace offers an easy learning curve. Another thing to note is that the proficiency of each user also correlated to the lap times they set. The average best time overall is 77.55, which is quicker than the Unity AI, but slower than the Genetic and BackPopagation AI, proving this project a success. Although some users beat the time of the AI, I found that only 3 users did so, and they claimed to be extremely proficient (self-rating of 5/5) in racing games.

It is evident that the Genetic Algorithm is superior to BackPropagation in terms of lap times, we can see that there is a massive 5 sec/lap difference between the two algorithms. Nevertheless, Backpropagation has shown that it can learn to drive around the track much faster than Genetic Algorithm. When paired with a good driver, Backpropagation can learn to drive around the track successfully after just 2 laps (2.20min). Proving that Backpropagation has its specific use case scenarios.

8.2 Directions for Future Work

Although everything that has been envisioned at the beginning of this project has been implemented, my original goal of creating an interesting and challenging has not changed and I plan to continue working on this project to further my knowledge of game development and machine learning. Further research into this topic will be invaluable for my career in the future, especially with the emergence of self-driving cars and hype around machine learning. Here are some of the features I plan to implement in the future:

- Another popular algorithm used in self-driving cars is Convolutional Neural Networks, during the planning phase of this project it was considered as one of the algorithms for AIRace but, research has suggested the other two algorithms were more applicable for my use case. CNN is a viable option for the third algorithm to be implemented at a future time. It would also be added to Learning Mode where a user will be able to experiment with it as the other two algorithms already implemented.
- A hybrid between the Genetic and Backpropagation algorithm would be interesting to investigate, first we use the backpropagation algorithm to teach the car to drive, then hand it over to the genetic algorithm to perform minor corrections to the existing solution, perfecting the supervised learning approach.
- Currently, the game only works on Windows Computers, but I plan on releasing it to other operating systems and devices. AIRace was developed to be compatible with any device and only small changes around inputs and build directions need to be changed.
- Multiplayer mode will add the ability for multiple users to race against their friends or challenge complete strangers looking for an online race. This will also give me greater insight into networks and how games function in multiplayer modes.
- Ability to customise characteristics of cars, unlocking the possibility for upgrades both cosmetic and performance-based after a player wins a certain number of races.
- An economy system is something that is seen in many games and is a useful skill to have.

8.3 Conclusions

In conclusion, I consider this project a massive success. I have put in a tremendous amount of work into every aspect. I have enjoyed researching and learning the different aspects of this project and I hope to use these skills in the future. Planning, designing, developing and testing my project was both interesting and challenging. The research into machine learning and game development has enabled me to learn a lot of new skills. AI and NPCs are a huge part of gaming nowadays and without good/challenging enemies a player will lose interest which results in a loss of popularity and therefore revenue. The biggest challenge with this project has been the application of machine learning to my car physics scripts, and most of all have the cars perform better or at least on par as human players. I am very satisfied with the results (lap times) that the algorithms can achieve, both algorithms outperform the average time set among all users tested and outperform the best time set by 70% of all users tested. I hope to see this project develop into much more in the future.

Thank you for taking the time reading this document.