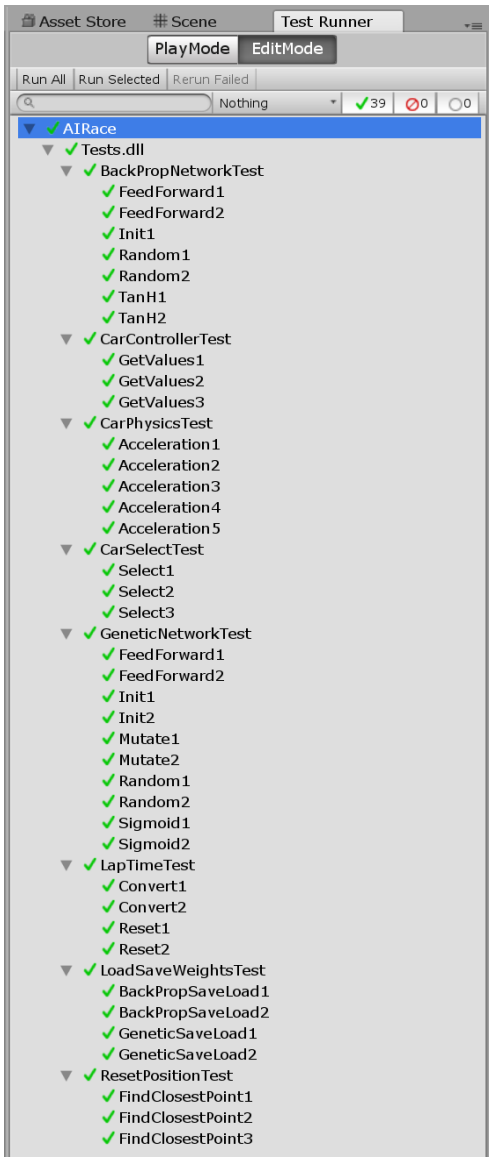
Testing Document

CA400

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

1. Unit Testing

I unit tested every feature after its completion, this meant 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). Another benefit of writing the unit test before the feature is that I can always check that this will give the correct output even when the code was modified since my unit tests directly referenced the actual code. In order for my scripts to run in the Unity Testing Environment I first I had to make an assembly reference to all my other classes used in AIRace. Similarly, I made references to any other scripts that were used in other parts of the project. Making an assembly reference, allows me to call other classes and methods from the testing environment. Although AIRace contains no user-generated input, which significantly reduces any possible errors that might occur when playing the game it is still important to unit test places where the transfer/manipulation of important data is occurring. Other GUI and physics-based testing where there is no transfer of data can be seen in Section 2. For this section I am using a version of NUnit, it is an open-source unit testing framework for Microsoft .NET. It serves the same purpose as JUnit. Below is each section I Unit tested, not all code is shown below, the full set of Unit test code is located at:
https://gitlab.computing.dcu.ie/khaletk2/2019-ca400-khaletk2/tree/master/src/unit-test-scripts



1.1 Reset Position Test

```
[Test]
public void FindClosestPoint1() {
    // ARRANGE
    var reset = new ResetPosition();
    var body = new Rigidbody();
    var pos = new Vector3(0,0,0);
    var p1 = new GameObject();
    var p2 = new GameObject();
    p1.transform.position = new Vector3(0, 5, 0);
    p2.transform.position = new Vector3(0, 10, 0);
    GameObject[] points = new GameObject[2];
    points[0] = p1;
    points[1] = p2;
    var expected = p1;

    // ACT
    var closest = reset.ResetCurrentCar(body, pos, points);

    // ASSERT
    Assert.That(closest, Is.EqualTo(expected));
}
```

The ResetPosition class finds the closest point to a given Rigidbody, in order to respawn that car to that point if the car is stuck. Before writing the class I set up a Unit test to correspond to the class, this saved me time actually starting the game up to check if it would spawn at the correct place. The test creates a Rigidbody and sets it to position 0,0,0. Then I made two points (p1 & p2), one of which is further away than the other. Lastly, I assert that the expected point p1 is equal to the point which is returned by the ResetCurrentCar() method. Similarly, I made two more tests, but with different values and points to make sure the method was working as expected.

1.2 Load Save Weights Test

```
[Test]
public void BackPropSaveLoad1() {
    // ARRANGE
    string path = Application.dataPath + "/Test/testWeights1.text";
    List<double[][]> weights = new List<double[][]>();
    double[][] w1 = new double[][] {
        new double[] { 1.0, 3.450, 5.34, 7.0, 9.4 },
        new double[] { 0.2, 2.0, 4.0, 6.0, 11.12}
    };
    weights.Add(w1);

    // ACT
    SaveWeights.WriteToFile(path, weights);
    BackPropWeights prop = new BackPropWeights();
    prop.layers = new int[] {5, 2};
    prop.lengthLayers = 2;
    List<double[][]> weightsSaved = prop.LoadWeights(path);

    // ASSERT
    Assert.That(weightsSaved, Is.EqualTo(weights));
}
```

In order to test the Read and Write methods for Backpropagation networks, I wrote the test shown above. It creates a “testWeights1.text” file which is populated with a predefined set of weights. These weights are then written to that file using the SaveWeights class. After they are written they are then

retrieved from a file using the BackPropWeights() class which uses BackPropNetwork as a base class. The weights retrieved from the file are then compared to the original weights I set up earlier, the method above asserts that both sets of weights are equal. I made another method with different weights and parameters to further test it. Similarly, I also tested the GeneticWeights method against a different set of weights this can be seen in the testing files.

1.3 Lap Time Test

```
[Test]
public void Convert1() {
    // ARRANGE
    var time = new LapTimeManager();
    string expected = "08";
    string expected2 = "10";

    // ACT
    int x = 8;
    int y = 10;
    string num = LapTimeManager.Convert(x);
    string num2 = LapTimeManager.Convert(y);

    // ASSERT
    Assert.That(num, Is.EqualTo(expected));
    Assert.That(num2, Is.EqualTo(expected2));
}
```

When the Lap timer is counting the lap times the singular numbers must start with a 0 in order to keep the format of a timer and make it more visible (eg. 01.23:09). The method above runs the Convert() method inside the LapTimeManager class to assert that the output is in the correct format. The Lap Time test also checks the ResetTimer() method which asserts that all the values have been reset during a lap, or else when the race ends. This can be seen further in the test files.

1.4 Car Select Test

```
[Test]
public void Select1() {
    // ARRANGE
    var select = new CarSelectController();

    // ACT
    Cars = new List<GameObject> { EVO, GTO, Z66, GTX };

    // ASSERT
    Assert.That(select.Cars, !Is.EqualTo(Cars));
}
```

The CarSelectController script assigns a car to each of the different drivers, User, UnityAI, BackPropDriver and GeneticDriver. It takes an original List of objects, assigns one to a user, depending on what he/she selected and the other three are randomly shuffled and assigned to a different algorithm each time. The Shuffle() method uses the Fisher-Yates shuffle in order to rearrange the List in a random order. The Unit test above checks that the original List doesn't equal to the new (shuffled) List. Similarly, I performed more tests with different arrangements of the original List to see what the List is being rearranged, I logged the order of each output using Debug.Log() to also visually see what order each List came in.

1.5 Car Controller Test

```
[Test]
public void GetValues2() {
    // ARRANGE
    var controller = new CarController();
    controller.GeneticEngine = 1;
    controller.GeneticSteering = 0.8;
    controller.BackPropEngine = 0.1;
    controller.BackPropSteering = 0.4;
    var expectedGE = 1;
    var expectedGS = 0.8;
    var expectedBE = 0.1;
    var expectedBS = 0.4;

    // ACT
    var GE = controller.GeneticEngine;
    var GS = controller.GeneticSteering;
    var BE = controller.BackPropEngine;
    var BS = controller.BackPropSteering;

    // ASSERT
    Assert.That(expectedGE, Is.EqualTo(GE));
    Assert.That(expectedGS, Is.EqualTo(GS));
    Assert.That(expectedBE, Is.EqualTo(BE));
    Assert.That(expectedBS, Is.EqualTo(BS));
}
```

The CarPhysics script uses GeneticDriver or GeneticController class and BackPropDriver or BackPropController class depending on whether the game is in Learning Mode or Racing Mode. To make the code more neat I created a Getter/Setter class called CarController, this method takes the output of whichever class mentioned above is live at the time. To check that this method worked correctly I made the test shown above. It sets a value to each of the variables and then gets those variables, the test method then asserts that the expected values are equal to the values retrieved from the class. Similarly, I made other test methods using different values.

1.6 Genetic Network Test

The main testing of this segment was done in section 7 of the Technical Specification were I apply the code to an actual car and see if it performs as intended with the correct set of values and outputs, although there were still some methods in this class that I needed to check and make sure they outputted the correct lengths and types of data.

```
[Test]
public void Init1() {
    // ARRANGE
    int population = 10;
    GeneticNetwork[] genetic = new GeneticNetwork[population];
    int expectedlength = 10;

    // ACT
    for (int i = 0; i < population; i++) {
        genetic[i] = new GeneticNetwork();
    }

    // ASSERT
    Assert.That(genetic.Length, Is.EqualTo(expectedlength));
```

```
        for (int i = 0; i < population; i++) {
            Assert.That(genetic[i].GetWeights().GetType() == typeof(List<double[][]>));
        }
    }
```

The GeneticNetwork() method inside the GeneticNetwork class works as an initialiser for the class, the code initialises a variable number of Genetic networks and fills them with a random set of weights. Since I have already checked the GetRandomNumber() method (shown in section 1.9) I only need to assert that it is a correct number of Networks and that they are all of type List<double[][]>.

```
    [Test]
    public void Sigmoid1() {
        // ARRANGE
        GeneticNetwork genetic = new GeneticNetwork();
        List<double> sums = new List<double> {1.0, 6.3, 2.2, 0.00005};

        // ACT
        List<double> outputs = genetic.Sigmoid(sums);

        // ASSERT
        foreach (var output in outputs) {
            Assert.That(output, Is.InRange(0f, 1f));
        }
    }
```

The Sigmoid method works by changing a List of doubles into a set of numbers between 0 and 1. The test above asserts that all numbers contained in the original List are now between 0 and 1, after being run through the Sigmoid method in GeneticNetwork class.

```
    [Test]
    public void FeedForward2() {
        // ARRANGE
        GeneticNetwork genetic = new GeneticNetwork();
        double[] inputs = { 0.6, 0.4, 0.2, 0, 0 };

        int expectedlength = 2;

        // ACT
        List<double> outputs = genetic.FeedForward(inputs);

        // ASSERT
        Assert.That(outputs.Count, Is.EqualTo(expectedlength));

        foreach (var output in outputs) {
            Assert.That(output, Is.InRange(0f, 1f));
        }
    }
```

The FeedForward method in the GeneticNetwork class takes the inputs of the Sensor class and feeds it through the layers of the class, multiplying them and activating them using the Sigmoid() method. The test above takes a set of inputs and runs it through a random set of weights in order to see that the output is both the correct length (steering and engine) and the correct range of values.

1.7 Car Physics Test

```
[Test]
public void Acceleration1() {
    // ARRANGE
    float currentSpeed = 100;
    float scaledTorque = 1;
    int dropSpeed = 150;
    int maxSpeed = 300;

    // ACT
    if (currentSpeed < dropSpeed) {
        scaledTorque = Mathf.Lerp(scaledTorque, scaledTorque / 3f, currentSpeed / dropSpeed);
    } else {
        scaledTorque = Mathf.Lerp(scaledTorque / 3f, 0, (currentSpeed - dropSpeed) / (maxSpeed -
dropSpeed));
    }

    // LOG
    Debug.Log(scaledTorque);
}
```

One of the things I needed to Unit test was the acceleration curve and that it was modifying the scaledTorque as it should. Instead of doing an assert I simply used a Log system to check that the values are increasing/decreasing correctly based on acceleration curve found in the technical spec in section 4.1, I then compared the values in the Log to the desired acceleration curve.

1.8 BackPropagation Network Test

```
    [Test]
    public void Init1() {
        // ARRANGE
        BackPropNetwork backprop = new BackPropNetwork();
        int expectedlength = 3;

        // ASSERT
        Assert.That(backprop.GetWeights().Count, Is.EqualTo(expectedlength));
        Assert.That(backprop.weightsDer.Count, Is.EqualTo(expectedlength));
        Assert.That(backprop.layerInputs.Count, Is.EqualTo(expectedlength));
        Assert.That(backprop.layerOutputs.Count, Is.EqualTo(expectedlength));
        Assert.That(backprop.layerLoss.Count, Is.EqualTo(expectedlength));

        Assert.That(backprop.GetWeights().GetType() == typeof(List<double[][]>));
        Assert.That(backprop.weightsDer.GetType() == typeof(List<double[][]>));
        Assert.That(backprop.layerInputs.GetType() == typeof(List<double[]>));
        Assert.That(backprop.layerOutputs.GetType() == typeof(List<double[]>));
        Assert.That(backprop.layerLoss.GetType() == typeof(List<double[]>));
    }
```

The BackPropNetwork() method inside the BackPropNetwork class works as an initialiser for the class, it fills the class with a random set of weights. Since I have already checked the GetRandomNumber() method (shown in section 1.9) I need to assert that it is a correct number of layers has been created based on the architecture of the network, and that each of the variables in the network is of the correct type.

```
[Test]
public void TanH2() {
    // ARRANGE
    BackPropNetwork backprop = new BackPropNetwork();
    List<double> inputs = new List<double> {0.0, 0.7, -0.4, 0.1, -0.2222};

    // ACT
    List<double> outputs = new List<double>();
    foreach (var input in inputs) {
        outputs.Add(backprop.TanH(input));
    }

    // ASSERT
    foreach (var output in outputs) {
        Assert.That(output, Is.InRange(0f, 1f));
    }
}
```

The TanH method works similar to the Sigmoid method mentioned above, but instead of a List like before it only changes one value. It works by changing a double into a number between 0 and 1. The test above asserts that all numbers contained passed through the TanH activation method are in a range of 0 and 1.

```
[Test]
public void FeedForward1() {
    // ARRANGE
    BackPropNetwork backprop = new BackPropNetwork();
    double[] inputs = { 0.2, 0, 0, 0, 0.2 };
    int expectedlength = 2;

    // ACT
    double[] outputs = backprop.FeedForward(inputs);

    // ASSERT
    Assert.That(outputs.Length, Is.EqualTo(expectedlength));

    foreach (var output in outputs) {
        Assert.That(output, Is.InRange(-1f, 1f));
    }
}
```

The FeedForward method in the BackPropNetwork class takes the inputs of the Sensor class and feeds it through the layers of the class, multiplying them and activating them using the TanH() method. The test above takes a set of inputs and runs it through a random set of weights in order to see that the output is both the correct length (steering and engine) and the correct range of values. Further testing of the actual Network and Lap times can be found in section 7 of the Technical Specification.

1.9 Random Number Test (BackPropagation/Genetic Network)

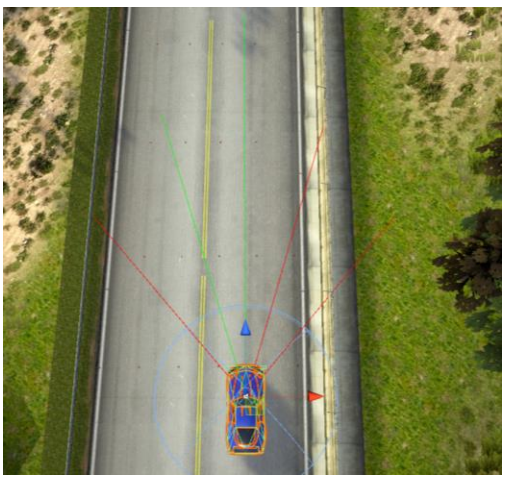
```
[Test]
public void Random1() {
    // ARRANGE
    GeneticNetwork genetic = new GeneticNetwork();
    BackPropNetwork backprop = new BackPropNetwork();

    // ACT
    double random = genetic.GetRandomWeight();
    double random2 = genetic.GetRandomWeight();

    // ASSERT
    Assert.That(random, Is.InRange(-1f, 1f));
    Assert.That(random2, Is.InRange(-1f, 1f));
}
```

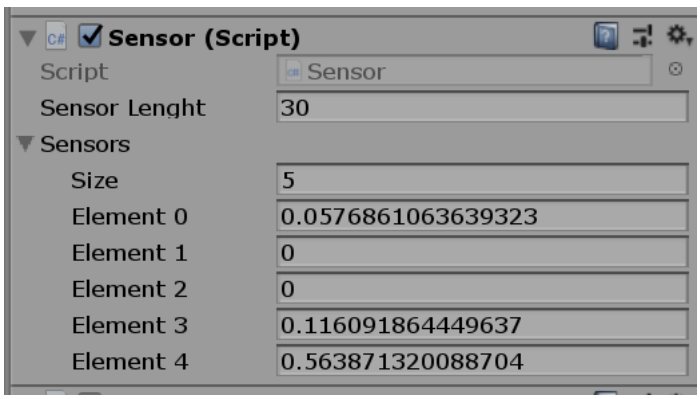
To test the random number generator contained in both Networks I used the test shown above. It asserts that the random number returned by the GetRandomWeight() method of both classes is between -1 and 1.

2. Physics, GUI & Integration Testing

2.1 Integration Testing

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. After performing a unit test on both features I have created, I begin to integrate them together, to do this I attached both features to the object in question and ran the game in Editor Mode, if the game starts is the first sign that the features are working as intended, then I check the values in the inspector and console, to see if they match the expected output.

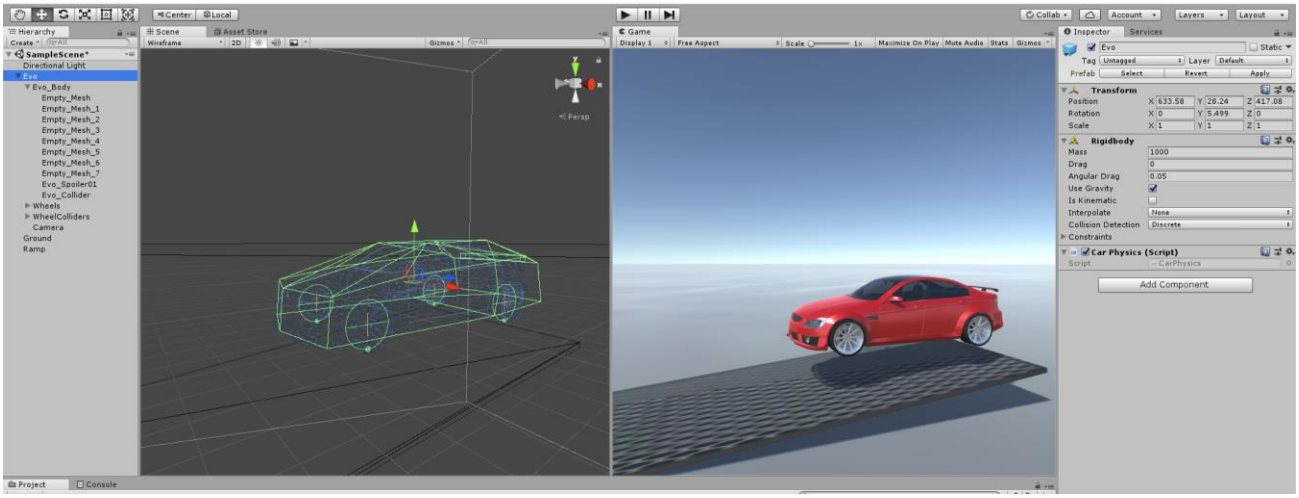





```
Debug.DrawRay(transform.position, sensor * length, Color.red, 0, true);
Debug.DrawRay(transform.position, sensor * length, Color.green, 0, true);
```

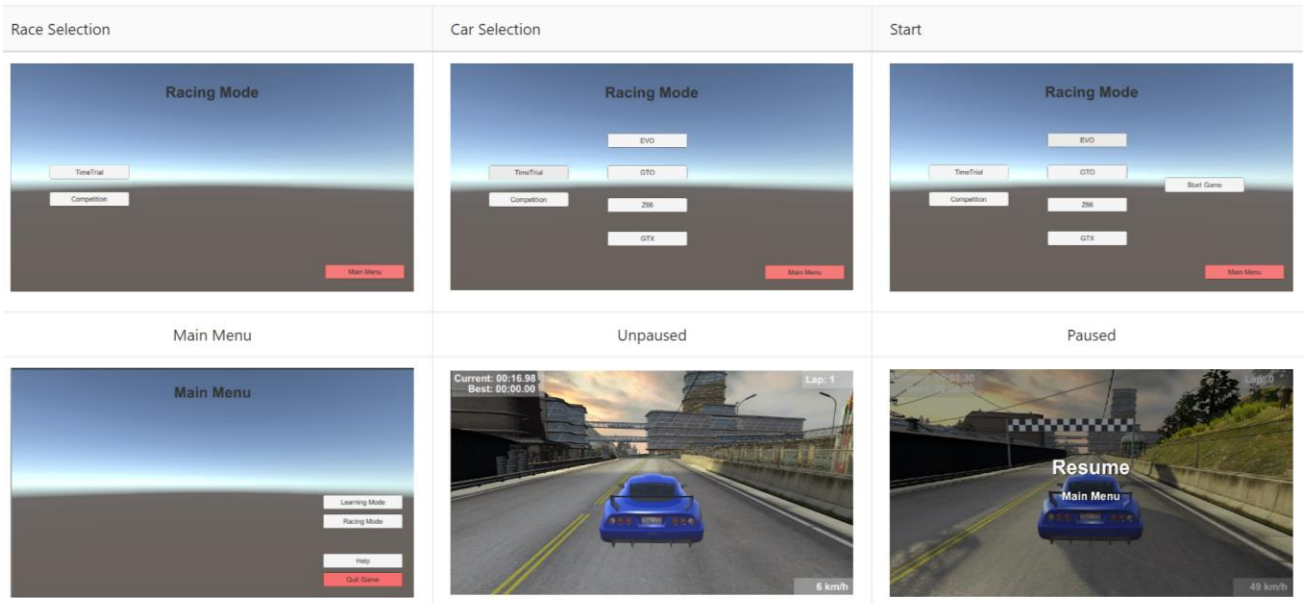
For example, when I was testing whether the Sensor and CarPhysics class worked together I attached them to a car Rigidbody and used a combination of Inspector and Debug.DrawRay in order to find that the the correct values, and angles are maintained by the car sensors.

2.2 Physics Testing



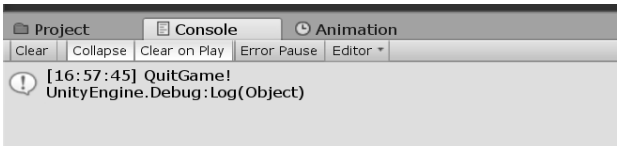
Because some of the physical movements within the game cannot be translated into a unit test and some of them can be quite subjective, the only way to test them was to interact with them in game, and see if they performed as expected. In the early stages of the project, I set up a “Test Scene”, this test scene acted as a testing environment to how the car reacts to its surroundings. For example, one of the first things I did was making the wheel colliders and models spin together. To test that it was working I made a ramp in the test scene and made sure that the wheels would roll as expected. Similarly, camera movements/angles, collisions, engine and steering were all done in test scenes in order to make sure they worked when they were put into actual scenes which are in the game right now. When I set up a particular object correctly and I checked that it works as intended I then saved that object as a “prefab” so that it could be reused again, this prefab would then be run through an integration test mentioned in 2.3. Due to the subjective nature of some of the physics in the game, I have also taken user testing into account for this section. The qualitative and quantitative review and opinions of other people gave me an insight into whether certain parts need to be tweaked.

2.3 GUI Testing

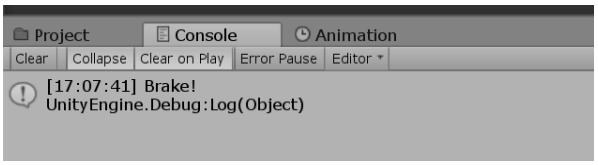


For this section, I used a combination of Logs, Test Scenes and User analysis. As shown in the screenshot above, before creating a good looking user interface I first made it functional with all basic buttons and backgrounds to make sure each button worked as intended. Although some parts of the project were still to be completed I could make sure that they worked by utilising logs. When the UI was polished I gave it to users to evaluate it in terms of accessibility, looks and functionality, this helped me get the opinion of others as looks can be mainly subjective. After collecting a series of user reviews I was able to base my analysis and tweak it accordingly. More information on User Testing can be found in section 3.

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



For example, when in scene editor mode the action that pertains to quitting the game will not work as the game is technically not running, therefore I added a log message to the button that quits the game. The screenshot on the left shows the message in the console that corresponds to when I press “Quit Game” in the scene editor.



Similarly, when testing the BrakeLightScript class which only involves graphical representation of code, I first set up a boolean within the script to check that the script works as intended (when User or AI brakes). When I checked that the script works as intended I attached it to a graphical object, and visually tested that it performs as intended, as shown in the screenshot above.

2.3 UnityAI

Although the Unity car AI is an import, I had to make some changes to it in order to be integrated with my car controller. After I set it up and created my own pathfinding helper class for the Unity AI I set out the nodes on the track and began testing it by watching the car going around the track, as there was no other way of unit testing this physical aspect. I made it better by tweaking the position of the nodes and the Unity Car AI algorithm (shown in orange in the screenshot above). 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. After setting it up tweaking the position of the nodes my lap times went down from 1.45 on average to 1.26. Although this value is still hugely larger than my own Genetic and Backpropagation algorithms.

3. User Testing (Acceptance Testing)

Before performing test I received ethical approval (located in this repository). After having the users sign the consent form I got them to do a series of tests and fill out a sheet to rate AIRace after playing/using it.

3.1 Lap Time Testing

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	110.3333333	96.91
User2	3	91.99	89.03	81.15	87.39	79.12
User3	4	86.5	84.51	85.72	85.57666667	78.86
User4	4	76.1	74.14	72.55	74.26333333	72.55
User5	5	85.21	74.73	76.12	78.68666667	72.91
User6	5	73.04	71.73	73.11	72.62666667	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	89.22333333	83.02
User10	4	81.08	79.62	75.41	78.70333333	75.41

After explaining how to navigate and control cars in AIRace I got the users to drive around in TimeTrial mode using the GTO. The first lap was a warmup lap, thereafter I recorded the first 3 running laps, and following gave users unlimited attempts to set the best time. All times are shown above in seconds. Please refer to the Technical specification for an analysis of the times shown above.

3.2 User Interface & Game

Driver	UI Quantitative Test	Computer Proficiency	UI	Sound Design	Help Section	Physics	Challenging Opponents	Graphics	Game Overall Score
project owner	9.05	5							
User1	13.77	4	5	4	5	5	5	5	5
User2	10.81	5	5	4	4	4	5	5	5
User3	10.68	5	5	4	5	4	4	5	5
User4	10.98	4	5	4	5	4	4	4	5
User5	9.29	5	4	5	5	4	4	3	5
User6	10.41	5	5	5	5	4	3	4	5
User7	11.01	4	5	4	5	4	5	5	5
User8	10.5	5	5	5	5	4	5	5	5
User9	9.88	3	4	5	5	4	5	5	5
User10	10.11	5	5	5	5	4	4	5	5

The rating test yielded the results shown above. I am very happy with these results as most are in the range of 4-5 (higher is better). To remove any inherent biases in the user test I performed quantitative tests. I asked users to do a series of actions and recorded the time they performed doing these tests. The average time performing these tests were 10.744 seconds, which is less than 2 seconds from the reference time set by myself. This proves that the UI is easy to use and accessible to a variety of different users. I asked users to find issues or anything they would like to suggest, these were:

- Hide mouse pointer if inactive when in game.
- Reset position of cars that are stuck in various modes.
- Time goes too fast when you increase the timescale in “Genetic Learning Mode”.
- Lap tracker going up when other cars cross it.

After the user tests were complete I fixed any issues found and added some of the suggested features the users have brought forward.

4. Shneiderman’s 8 Golden Rules

Strive for consistency

Throughout AIRace, the same theme is applied, the game uses transparent panels to overlay text over background images, these panels appear on the screen depending on the users navigation and what he/she interacts with. Initially, the user is met with the main menu screen which has an overlay on it with text (buttons) which allow the users to navigate throughout the user interface. All buttons stand out with good contrast ratios and simple colour schemes. These features are consistent across all screens, utilising the same font, size, placement, style and sound (when pressed). All buttons which bring the user back are red in colour. This uniform way of undoing one's actions is a key part of the UI design. This consistency throughout the app will aid the user in achieving what they want to do as quickly and easily as possible and is vital to gain the user's trust so that they know they are getting the same experience throughout using the app.

Enable frequent users to use shortcuts

AIRace aims to create as minimal interaction with the UI, enabling users to jump straight into a game as quick as possible. Although there aren't any shortcuts for frequent players, if a user wants to play a game it requires only 3 clicks to start either a “TimeTrial” or “Competition”. During the user testing phase, it was found that users required an average time of 10.744 to begin driving around the track.

Offer informative feedback

Throughout AIRace, all buttons, dropdown menus, sliders, and are labelled in accordance with what they represent. When a user changed a value on a slider the integer value corresponding to that value changes beside the label of that slider in order to provide the user with informative feedback. When the user hovers over a button it is highlighted. Similarly, when a user presses anything through AIRace the clickable objects will make a “clicking” sound and the button will become highlighted to a higher factor than it was when a user only hovered over it.

Design dialogues to yield closure

It is important that users have a clear path to their goal. For instance, if a user wants to start a game, sequences of actions are organized into groups with a beginning, middle, and end. Such as the selection of a racing mode and car in the “Race Mode Selection” screen. When a user wants to start a game he/she simply presses the “Start Race” button after which they will be presented with a loading screen which will inform the users how long it will take for a game to load, this stops the users from thinking that the game is “frozen”.

Offer simple error handling

AIRace contains no user data input only predefined selections of previously tested variables, therefore, there shouldn't be any errors occurring with a fully tested and compiled game. Nevertheless, if any unforeseen error occurs the user is notified through a simple dialog/notification that the action they are trying to perform cannot be done.

Permit easy reversal of actions

As soon as a user launches the game they are greeted with the main menu screen which offers users to “Quit Game”, this is a simple one-click operation which terminates the game, this button is also red in colour in comparison to all others buttons which are white. When in other screens such as “Racing Mode” or “Learning Mode” the user is provided with a “Main Menu” button (also red in colour) which returns users back to the main menu. When driving around the track the user can pause the game at any point using the “Esc” key on their keyboards. From the pause menu, the user can either resume the game, go back to the main menu or even quit the game entirely (a feature which is sometimes not even seen on AAA games).

Support internal locus of control

It is important that a user feels that they are in control. A user has to be the initiator of each action rather than reacting to the UI. Users should never ask themselves “How did I get to this screen?”. With each button and action well labelled, it is hard for a user to make a mistake in what they are trying to achieve.

Reduce short-term memory load

All buttons are intuitively laid out. All the major menu actions start on the left-hand side and the relevant submenu items progress towards the right-hand side of the screen. The UI is simple and easy to understand so that the learning curve is not steep. A user should be able to use and play AIRace on their first attempt no matter what their previous knowledge of user interface is.

5. Interface Testing

Throughout the testing of AIRace, the game was tested on 12 different computers. It was worked flawlessly on Windows 7, 8 and 10. The loading times and boot times remained at a constant time. Furthermore, it was tested on a magnitude of different screen resolutions. Although it is locked at a 16:9 screen ratio it worked as intended on these screen resolutions:

- 1176x664
- 1280x720
- 1360x768
- 1366x768
- 1600x900
- 1920x1080
- 2560x1440
- 3840x2160

It is important to note is that it worked very well on a 4K resolution monitor, which is very good because a lot of people are running computers with 4K monitors nowadays, and this number will only increase with time, making AIRace compatible with this resolution is very beneficial.