horizontal line

Grand Traffic Auto

Software Document

**24th October 2019**

**Project Team**

|  |  |
| --- | --- |
| **Name** | **Student Number** |
| Robert Pennefather | 21511164 |
| Daniel Gonzalez | 22024722 |
| Martin Porebski | 21498791 |
| Jacqueline Soon | 21719676 |

## 

## 

[**1.0 Introduction**](#_38avwhq4qgty) **2**

[**2.0 Game Development**](#_xgf9k7p8myjd) **2**

[2.1 Start Screen](#_vq2ftiq42vvm) 3

[2.2 Game Screen](#_u5vaokxvv8ch) 4

[1.2.1 Session Manager](#_ej5ykg3ghe5o) 5

[1.2.2 Database](#_8yphebems7gi) 7

[2.2.3 Phone](#_6tj0srwvziau) 8

[2.2.4 Cars](#_dx5gwz39j79n) 9

[2.2.5 Minimap](#_j11t9az5iptz) 10

[2.3 End Screen](#_fjyipdptsrzo) 10

[**3.0 Web Server Development**](#_7blptqu0bxx9) **10**

[3.1 Website](#_qlpsglaxlax) 10

[3.2 Server](#_8nqjvc300wdp) 10

## 1.0 Introduction

This document provides an overview of the software design and structure behind the Grand Traffic Auto project. The project consists of three main software components; a web server, a game and a database. The database is hosted in the cloud and is called using an API, while the other two components have been developed by our team. Section 2 describes the design of the code in the Unity game while section 3 explores the software design behind the web application.

## 2.0 Game Development

The game was developed using Unity. This was chosen as it is a popular game engine, there is a lot of documentation and support for using the software and it abstracts away a lot of the technical details of developing our own game engine. Unity comes with various UI, audio and graphical elements pre-built in, plus includes an asset store where more packages can be purchased (sometimes for free), saving our team from modelling our own models or creating our own audio. All of this was to allow the team to focus on coding and developing the features required by the client.

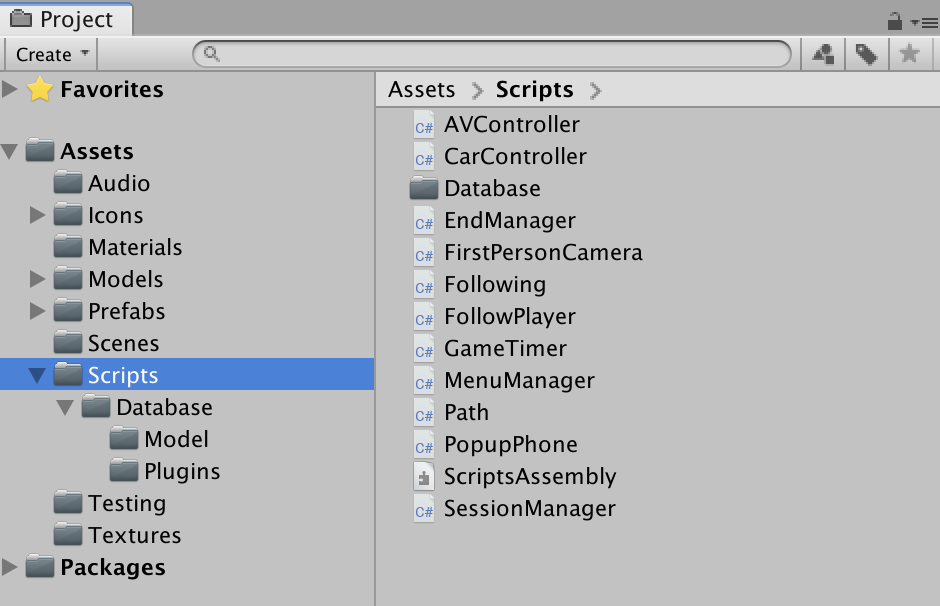
Unity splits a game into Scenes which act as different stages of the game. This Unity project was divided into three Scenes:

* Start Screen
* Game Screen
* End Screen

The first and last Scenes were created using Unity’s UI elements and act as the start and end menu for the game. Both Scenes required minimal scripting due to the limited user input. The Game Scene includes the bulk of the code, it handles environmental elementals, a dynamic HUD and AI-based traffic systems for the NPCs and player.



All of the code was written in C# and stored in the Unity Project in the main *Assets* folder under the *Scripts* subdirectory. C# was chosen as it’s compatible with the Unity game engine.



### 2.1 Start Screen

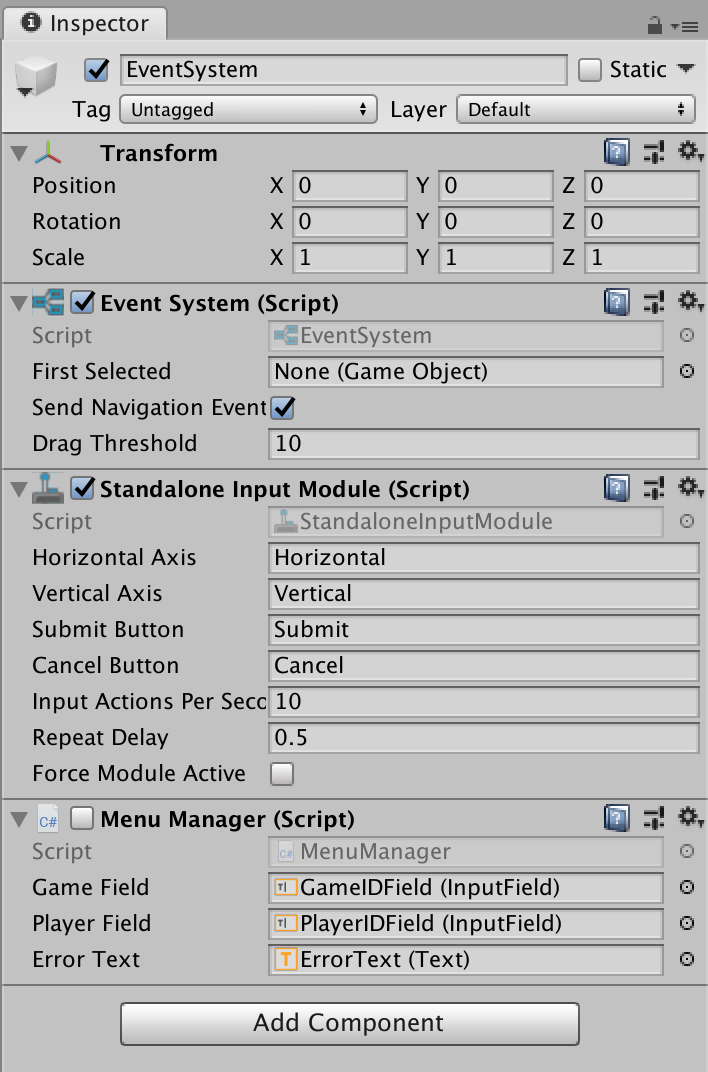
In the initial screen called *LoginScreen* the player is presented with a menu made of Unity’s UI elements.



This screen is comprised of multiple text and canvas elements, all of these are formatted and customised in Unity’s GUI. There are additionally two text fields (*GameID* and *PlayerID*) for the user to input a string into, plus a button that the user can interact with (*PlayButton*), these elements again are formatted in the GUI but their interactivity is scripted in the *MenuManager.cs* file. The Figure below shows the hierarchy structure of elements in the *LoginScreen*.



The *MenuManager.cs* file checks to make sure the two strings entered are valid, if not a text element will appear explaining the error, if the inputs are valid the next scene is loaded and the two values in the text fields are passed in. The *EventSystem* element links the *MenuManager* script to the *GameField*, *PlayerField* and *ErrorText* objects. This can be seen in the Figure below.



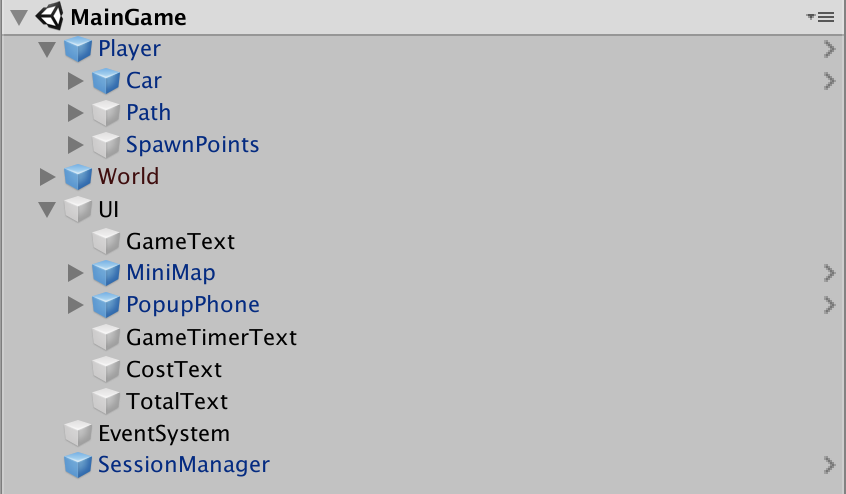
### 

### 2.2 Game Screen

The main game scene, named *MainGame*, is where the player will spend the majority of the game. The player is given a first person POV from the passenger seat of a car. The car is driven around the world by a driver or autonomous vehicle taking the player to their required destination. A phone appears in the players HUD after a set interval each round. The phone asks the player if they want to pick up another passenger before reaching their destination. Picking up the passenger decreases the cost of the trip for the player, but does lead to an increase in the length of their trip.



There are several scripts being used in this Scene. Firstly the *SessionManger.cs* which controls the sequence and order of the game, this script updates the UI elements, controls when the phone pops up and when the car drives. Each of these objects; the phone and car, both have scripts attached to them to control their behaviour and read user input. Lastly there is a minimap to give the player a birds eye view of where they are in the game world. The Figure below shows the hierarchy for elements in the *MainGame* scene.

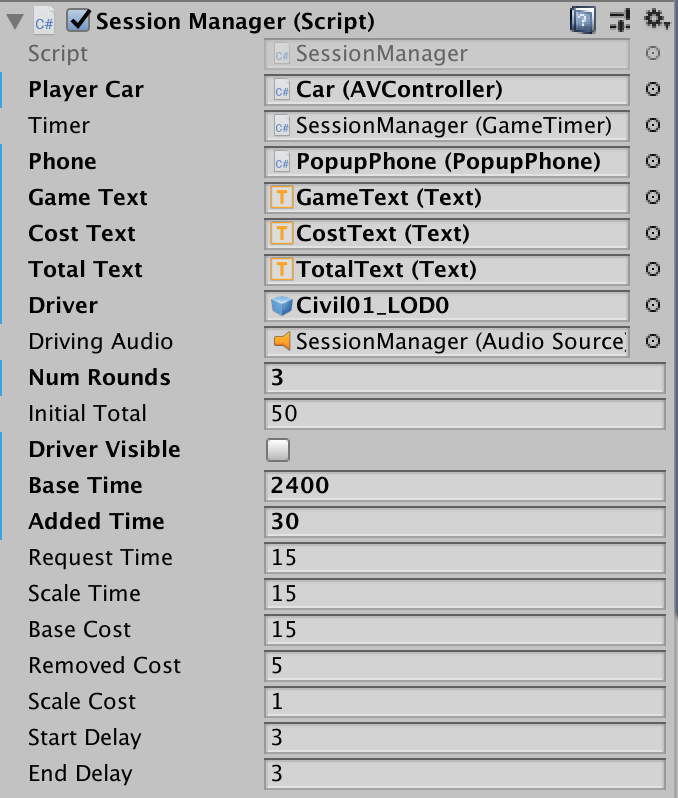


The Game Screen consists of most of the scripting. There are five main modules in the game which control the cars, phone, database, sessions and minimap. A breakdown of each module and the scripts it contains can be seen below. Each component will be discussed in detail within the following sections.



#### 2.2.1 Session Manager

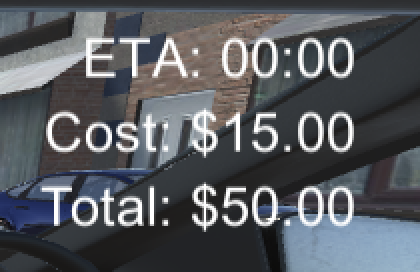
The *SessionManager.cs* script controls the sequence of how events occur throughout the game. The game ID given by the player will determine several factors including; the number of rounds, the time/cost changes each round and the visibility of the driver each round. These configuration parameters are encoded in the string that the player initially puts in the starting textfield, by decoding the string we can setup the game with the required parameters. The script is attached to the *SessonManager* element, parameters can then be seen in the component inspector box as shown below:



The script starts by initialising a game loop. The game loop ends once the number of rounds set has been reached. For each round there are three more game loops; *RoundStaring()*, *RoundPlaying()* and *RoundEnding()*. Both the *RoundStarting()* and *RoundEnding()* game loops stop the car and display some text about the progress of the game. The *RoundPlaying()* loop goes the longest and includes the movement of the car and the phone pop up.

Firstly the *RoundStarting()* loop begins by displaying text stating the round number to the player for three seconds. At this stage the player’s car is not moving, and it is simulating the experience of initially being picked up by a driver/AV. After the 3 seconds passes the *RoundPlaying()* loop begins.

Once the *RoundPlaying()* loop starts a timer is begun. This timer is indicating the time it will take for the player to reach their destination, and is shown in the top right corner of the screen as the ETA. This length of this timer is predetermined in the configuration code. Once the timer reaches zero this loop ends. The timer is updated every frame refreshing the displayed time, this number is scaled based on the configured time scale factor. For example, 15 minutes in game time may only be equivalent to 1 minute in real time.



After some predetermined amount of time has elapsed the *SessionManager* script will call the phone to appear. The phone will open up and give the player the option to pick up an additional passenger. Ignoring the request and responding with ‘No’ will remove the phone from the UI. Selecting ‘Yes’ will extend the player’s trip time, causing the timer to be increased, it will also lower the cost of the trip. The decision is sent back to the *SessionManager* from the *PopupPhone* script, and the *GameTimerText* and *CostText* UI elements are updated in the top right corner.

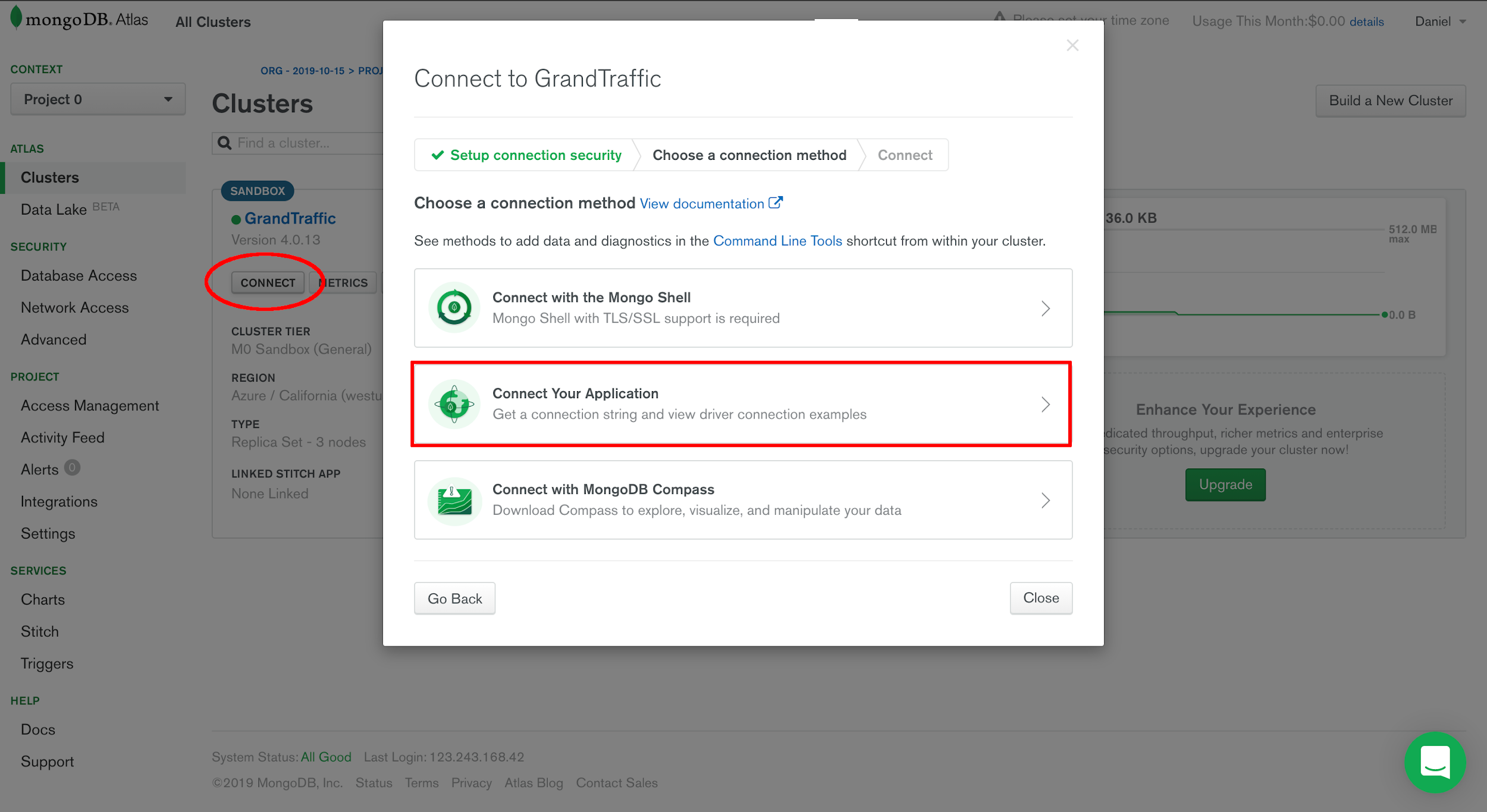
When the game timer reaches zero the *RoundPlaying()* loop ends and the *RoundEnding()* loop beings. The *RoundEnding()* loop lasts for three seconds. In this loop the car stops as it has arrived at the player’s destination, and text is displayed to tell the player that their trip is over. At this point the *SessionManager* script will update the player’s total money by reducing it by the cost of the trip, now that the trip is complete. This is indicated to the player by updating the *TotalText* object. The *SessionManager* checks if the configured number of rounds has been completed. If not the *RoundStarting()* game loop will begin again. If so the final Scene will be loaded, the *EndScreen* Scene. Information about the user’s inputs and final score are sent to the database at this point.

#### 2.2.2 Database

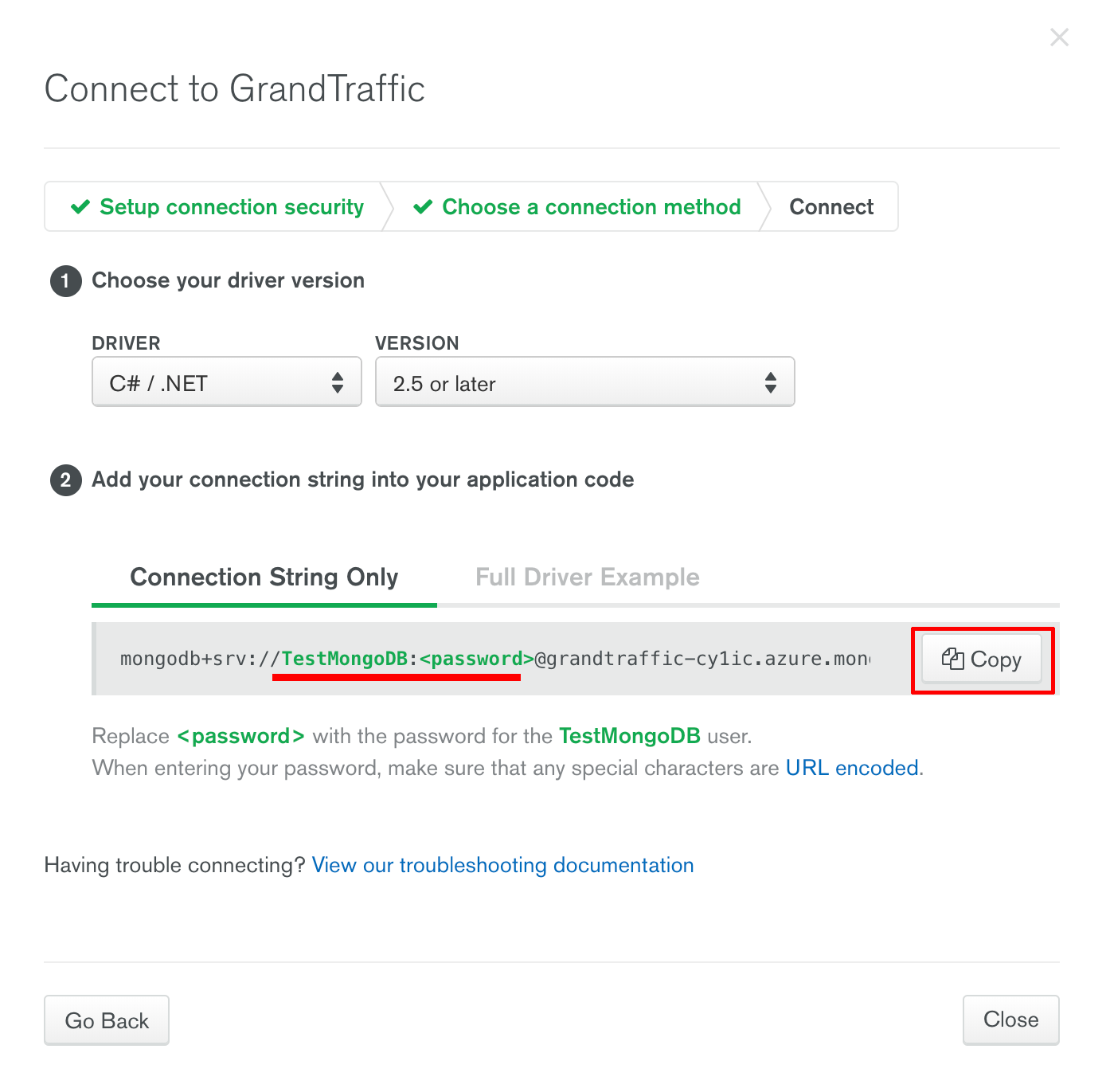
The connection to the MongoDB database is managed by the *Mongo.cs* script and the format of the objects sent to the database is determined by the *Mongo\_Config.cs* file.

Once the player has completed the game, a connection to the MongoDB database is created using the initialise function in the *Mongo.cs* script. The script is used to configure the connection to the server, to do so, the URI from MongoDB has to be included. This can be found on the MongoDB Atlas dashboard under Atlas > Clusters. Clicking the button “Connect” will prompt the following:

The connection is opened about the players choices and game and the final end screen will be loaded, passing in the total amount of money the player has made.



After choosing “Connect Your Application”, the connection string can be copied from the next dialog box. This string must be modified to contain login details for a user with read and write permissions attached to the project.

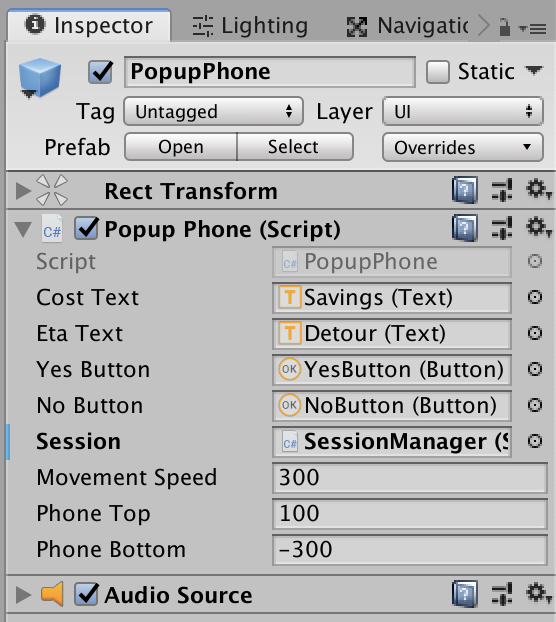


The object sent to the database is structured in the *Model\_Config.cs* file. The fields and variable types are set there. The *Mongo.cs* file then creates an instance of the *Model\_Config* object, assigning the parameters and uploading it to the database. The *Mongo.cs* file additionally has a read function for testing purposes or future features.

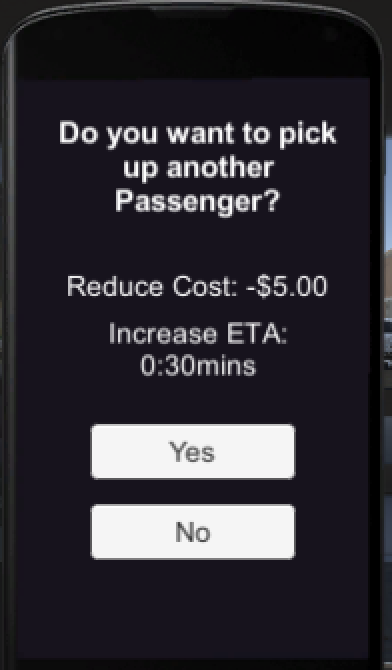
The *Model\_Config* object contains the decisions made by the player as well as the time it took to make each decision. This information as well as the game ID and player ID is sent to the database. This can then be processed by the researchers from their MongoDB Atlas accounts.

#### 2.2.3 Phone

The phone script is attached and interacted with through the *SessionManager.cs* file. The *PopupPhone* object uses the *PopupPhone.cs* script. This file controls the animation and appearance of the phone, along with the notification audio sound and the user input of the button presses. The script is attached to the two buttons on the phone to read the player’s decision, and the UI text elements to update them based on the changes to the ETA and Cost being made by picking up an additional passenger.



The *SessionManager* can call a public function called *openPhone()* when the input from the user is required. This makes the phone appear and increases its y position to move it into frame. Additionally, the appearance of the phone triggers the audio source causing the notification sound to occur. Once the phone is at its max height the buttons become intractable. The *SessionManager* also passes in information about the increase in time and decrease in cost associated with the user’s input, this is displayed through the text on the phone and updated using the *PopupPhone* script.



Once one of the buttons has been pressed by the user, they become uniteractable and the phone’s y position decreases. When it reaches the set minimum height the phone is made inactive, which makes it disappear from the scene altogether. This is important so that the audio source is triggered when it is made active again.

In the case that no decision has been made once the player gets to the end of the trip, a public function called *fastClosePhone()* is called by the *SessionManager* to quickly set the phone to inactive and send it to its minimum y position to remove it fully from frame. This is to avoid the phone blocking other UI elements once it is no longer needed, and to reset the position and state of the phone for the next trip.

#### 2.2.4 Cars

All the cars in the gameworld use the same script, *AVController.cs*. Each car object has the script attached to it. The script connects to the wheel objects of each respective car, the steering wheel, the driver and the path of the vehicle.

The path of the car is determined by a *Path* game object. Each *Path* contains multiple ordered *Nodes* that form a cyclic path around the game world. Every car has a path and an initial starting node number that it will drive to, once reached it will increment the node number and move to the next node. Any car without a path or node object will remain stationary, such as the cars parked on the side of the road or stuck in traffic.

The front wheels drive and turn each car and carry the rest of the physical object of the car body with it. The speed is calculated based on the circumference of the wheels and the revolutions per minute of the wheels. The wheels’ visuals are updated each frame and rotated based on the speed of the car. The wheels’ angle is such that they will always point towards the next node along the path, hence taking the shortest distance to the node.

The steering wheel works by measuring the rotation of the car between two frames. If the rotation difference is greater than a threshold in a particular direction the steering wheel will rotate. The wheel is rotated along its local plane where it sits on the car frame. The rotation will continue if the car keeps turning until a maximum hardcoded angle. If the car is not turning and the wheel is not in its central position, the wheel will start spinning backwards in the opposite direction towards the initial centre point.

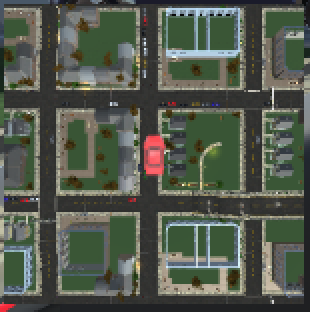
Finally each car has an option to be randomly spawned into the world using the list of spawn points. A point is randomly selected from the list and the corresponding spawn and starting node is set for the car. Currently this is only being used by the player’s car however this could be added to the NPC cars in the future to provide more randomness to their driving.

#### 2.2.5 Minimap

The minimap is the most basic scripted element in this scene. It uses the *Following.cs* and the *FollowPlayer.cs* scripts. The *FollowPlayer* script is attached to an icon and camera that is positioned vertically above the player’s car at some hard coded distance. The icon is set so it rotates with the car along the x plane while the camera does not, such that the map always points north.

|  |  |
| --- | --- |
|  |  |

The calculations for these positions are done in the *Following.cs* file which is attached to the *FollowingPlayer* script. This separation is important for testing purposes. The canvas UI element displays the map and is simply positioned with the Unity GUI at a margin away from the bottom left corner of the screen.

****

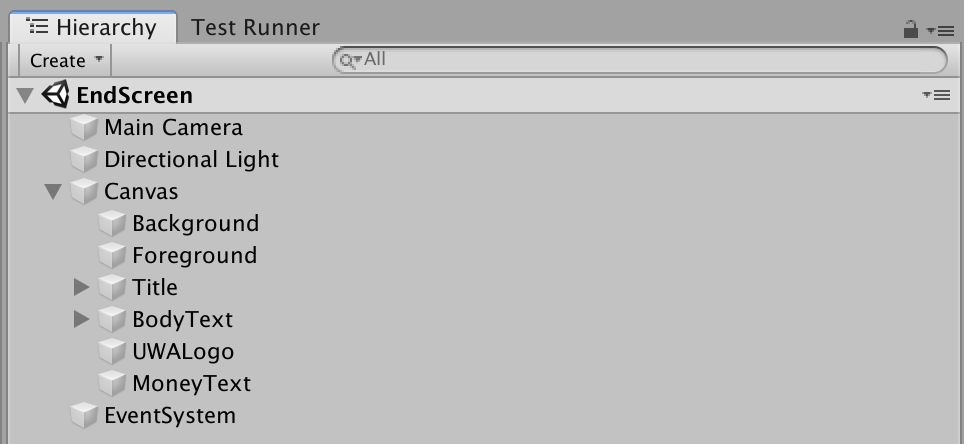
### 

### 2.3 End Screen

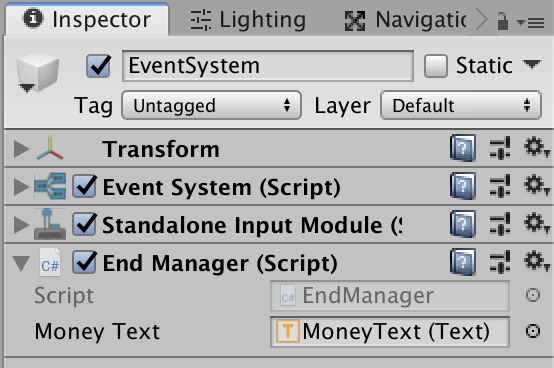
The final scene of the game, named *EndScreen*, is again another heavy UI screen much like the first. This screen is only comprised of text and canvas elements which have all been formatted and customised in Unity’s GUI.



A script called *EndManager.cs* is used to update the text in the *MoneyText* object, this displays the player’s final total money. The value of the money is stored in Unity’s player preferences object from the previous Scene. The player preference object allows developers to pass variables between Scenes. The Figure below shows the hierarchy structure of elements in the *EndScreen*.



The *EndManager.cs* script gets the total money from the player preference object, converts the float to a currency, and displays the text in the *MoneyText* object. The *EventSystem* object is used to connect the *EndManager* script to the *MoneyText* object such that the value can be shown to the user, as shown in the Figure below.



## 3.0 Web Server Development

The website was developed using the ASP.NET framework and it was hosted using Microsoft's Internet Information Services (IIS) on a Windows 10 VM.

### 3.1 Website

The ASP.NET framework was selected for the web development process as it allowed for quick creation of advanced websites using the Model-View-Controller (MVC) approach. The website serves the required survey and configuration forms using a Bootstrap template in the view component. The controller and model handle the data communication between the input from users and the game or database. Furthermore, given that games built in Unity require a knowledge C#, the selection of ASP.NET allowed us to keep all the tools used in this project within the Microsoft suite. This aimed at making it easier to take over the development of Grand Traffic Auto for any future programmers as the overall framework which was built using the same language is expected to require less training time.

The website consists of the following key elements which are represented in code by the same directory names:

* View - using Bootstrap templates, the survey (entry page) and the configuration forms were built. These handle the user input and pass it down to the *Controllers*.
* Model - stores the information from the *Views* and *Controllers*.
* Controller - handles the operation of the pages based on user input. It verifies any data and passes it to the *Models*.
* wwwroot - stores website’s assets such as images, CSS and JavaScript code
* WebGL - contains the latest deployment of the Unity-built game, which was described in the previous section.

The configuration (*config.htmls*) page accessible to the admin contains several HTML fields styled with the CSS stylesheet with the corresponding name. JavaScript was applied to verify user input “on the go” using the *onFormSubmit* and *onFocusOut* in-built functions. Using Regular Expression, the script verifies that the strings entered are numeric or alpha-numerical combinations (when relevant) and that they are of appropriate length. The output of the configuration form is encoded in the corresponding *Controller* and displayed to the user on the next page. The homepage (*home.htmlcs*)contains a form built in the same manner which aimed to serve as an alternative method of collecting information about participants. When correctly filled and submitted, the page redirects to the WebGL game.

### 3.2 Server

The client (Chao Sun) offered his Windows 10 VM to be used as a server to host the website. Hence, to keep all of the tools used in the project within the Microsoft suite (as described in section 3.1) IIS was selected as a HTTP web host. The VM was configured to handle only traffic that originated from UWA, which was based on the connecting IP address. IIS provides an interface for an easy management of the service and for more details see section 4 of the Maintenance Manual.