National College of Ireland

BSc (Honours) in Computing

Software Development

2020/2021

Jake Horner

16318711

X16318711@student.ncirl.ie

Roadster Run

Technical Report

Contents

Executive Summary

‘Roadster’ is a vehicle-based endless runner, made in ‘Unity Engine’ for mobile-based devices. The core gameplay Loop is to survive for as long as possible in the motorway wasteland, shooting at enemy cars and dodging projectiles. The player will have mobile touch screen-oriented controls such as, rotating the device to control the cars steering. The User will be able to upgrade the car cosmetics such as car colour, body type, wheel type. The player can use Augmented Reality features using the phone’s camera to project their upgraded vehicle into the real world. This will provide an incentive to cosmetically upgrade, as well as user generated content for the product with the ability for users to take photos with their personally customised car. In-game assets for the cars and various models have been made by me using the ‘Blender’ 3D software, and all User Interface and 2D Elements will be created using ‘Adobe Photoshop’, all sounds, and music will have been edited in ‘Audacity’ sound software.

The aim of this software project is to challenge my coding in ‘C#’ through procedural generated road tiles, spawning Artificial Intelligence enemies, gamification of physics-based cars, and save features that will create a consistent car model for the player. I want to improve my complexity with the ‘Unity Engine’ by making my first 3d game, dealing with a whole new game design structure. My artistic skills will also be challenged through 3D modelling and texturing in ‘Blender’, User Interface elements creation with ‘Adobe Photoshop’, and lighting and particle Effects using the ‘Unity Engine’.

I want to create a gaming product that innovates and breathes a new life into the genre of endless runners that have been in staple in mobile gaming. Most endless runner type games have the player running away from enemies and dodging obstacles in a locked 3 lane system. My gameplay loop will turn this established format on its head by not locking the player to any lane, giving freedom of movement down the endless road. My Game design features a gameplay loop that has you facing the enemies within the game head on, with an emphasis on engaging in combat with these enemies.

I want the player to feel that their actions have meaning within the game world, this is reasoning behind the ‘Store’ feature. The store allows the user to customize their personal vehicle with the customisation features they select being visible in each of the scenes of the game.

I wanted to include ‘Augmented Reality’ functionality through the users' device camera. This allows the user to see their customised Vehicle in the real world alongside them. I also see this feature as a great marketing opportunity, where users can take photo with their customised car with some branding overlay. This feature can generate more users if these photos are being shared on social media.

These 3-feature working together will create a robust, new, and enjoyable experience for all mobile gamers of any age. The complexity of the each of the features working together to drive user engagement has not be explored in many apps I have seen on the ‘Google Play Marketplace’.

1.0 Introduction

1.1. Background

Gaming has become the most profitable industry in the Entertainment Sector, with products like 'Grand Theft Auto 5', being regarded as the most profitable entertainment product of all time generating $6 billion in revenue since its 2013 release.

Gaming, as well as being highly profitable, is also seen as an artistic side of computing. Here, creators can express worlds and stories imagined in their mind, displayed on a screen to interact with in a more personal way than that of more traditional screen-based entertainment. Programmers, Artists, and Animators would come together to create one of the most entertaining and interactable media where players can experience stories, mechanics, and challenges that result in outcomes that are unique to the player.

Dublin is home to many game development studios such as the popular mobile developers 'DIGIT', Indie developers 'Pewter Game Studios', Virtual Reality developers 'War Ducks', and up and coming developers 'Vela Games'. Ireland has a very good technology sector, and the gaming industry has a strong presence too, with some of the bigger studios such as 'Activision' and 'Riot' having networking centres based in Dublin.

Gaming has become a great area of interest for myself during my time at NCI. I have completed one game called 'The Amulet of Power' as a Second Year Project, mostly using my skill in the story boarding and art for the game, which my colleagues and myself made in the 'Ren'Py Engine'. I have taken to Game Development as one of my hobbies, teaching myself 'C#', 'Unity Engine' and it many features since my first exposure to game development in Second Year. I have completed another game using 'Unity Engine' and 'C#', this is called 'Blobber', a 2d competitive platformer. I managed to publish this game on the 'Google Play Store' for android mobile devices and integrating some 'Google Play Services' features such as Scoreboards, as well as some monetisation features such as Rewarded Advertisements.

As I now feel more comfortable with 'Unity Engine' and 'C#', I believe I can create a more engaging and entertaining game, that also will have enough complexity for my final year 'Software Project'. I decided to go for a new form of development using 'Unity 3D', adding a whole new dimension to my game development portfolio. I have always drawn towards 3D developed games when playing with friends or alone, and I wanted to use the 'Software Project' to challenge myself into taking the dive into this new medium and art style that comes with its own development challenges. This would require me to use the knowledge I have already gained in a new way, to adapt to a game in 3 dimensions. I also have challenged myself to become better at a hobby I have recently picked up, creating 3d art assets in 'Blender' 3D modelling software. Objects in 3D space are quite different to the sprites and images I have developed games with before. This is using vertices, edges, and faces in a 3d interface to create 3d models on your screen. Although, I am not going into this task blind, but importing and manipulating objects in 3D space with 'Unity' and 'C#' will create new and interesting challenges for me to overcome, as well as display and develop my artistic skill in this 3D medium.

I have chosen to develop this game for mobile platforms as it is a great way to display and share my creation with all types of people, game player or not, as most people own a smartphone. Using smartphones also unlocks an easy way for most people to experience the

relatively new technology of ‘Augmented Reality’. I will be using ‘AR Foundation’ and a google made plugin ‘AR Core’ with ‘Unity’ to display the user’s customised car into the real world as a form of incentive for the player to interact with the game, purchase cosmetic upgrades as they progress with the gameplay. The augmented reality functionality will also be a smart way to create user generated advertising, using photos taken with the application to create unique user generated images to be posted online.

1.2. Aims

The aim of this project ‘Roaster’ is to create a fully advertisable game at an Alpha stage, with the Google play store page being interacted with by internals testers. The ‘Alpha’ stage in game development usually refers to a testing stage that is done in private, usually testing the main functionality of the product. Although I have ambitious ideas for the final release product, I am aware how I can over scope projects in the creation stage. I will outline the main features I will be aiming to complete for my final submission as an Alpha release of the Game on the Google Play Store (Android).

I will be using ‘Unity Engine’ to create this project and its three main pillars:

1. Main Game
2. Store
3. Augmented Reality Car Display

‘Main game’ will consist of a procedurally generated endless road which the player will travel down. This level chunk spawning system will be marginally intelligent, spawning the right level pieces to allow AI enemies to enter the playable road. Enemy characters will drive toward you to damage or destroy your player-controlled car. The Player will have the ability to dodge incoming vehicles and projectiles by moving the car using tilt controls on the mobile device. The player will also have its own weapons to fight back, by clicking using a touch screen button the player will shoot its weaponry at the enemies. The players aim is to destroy enemies and travel the furthest distance on the wasteland road, before receiving the ‘Game Over’ state.

The ‘Store’ will allow the player to visually change their player car in ‘The Cosmetics’ tab, and eventually tweak how the player controller works and its weaponry during the main game through ‘The Upgrades’ tab. ‘The Cosmetics’ section will allow the user to swap out and change their car visually, this includes the changing of car bodies, and car colours.

The Augmented Reality Car Display will allow users to display their customised vehicle edited in the ‘Store’ in the real world. This system will allow the car to be displayed in three lifelike sizes and have ‘AR Core’ light estimation implemented to the model to fit with the environment where it is displayed. The user should have the ability to take images with this Augmented Reality view with the image stored on the mobile device. In processing of the image there should be some branding of the ‘Roadster’ game logo placed on the image.

1.3. Technology

1.3.1. Unity Game Engine

'Unity' is an open-source game engine available for free for hobbyist game developers, most scripts are written in the 'C#' language and uses the IDE of Visual studio. Although 'Unity' is not the only available Engine, 'Unity' is heavily code based that will allow me to display my coding knowledge and practices that are required for any good software project. Unity also allows its users access to cross platform development, this allows users like me to be able to easily test and export for multiple devices, including creating APK's for Android mobile devices. 'Unity Engine' also comes with a lot of features that can aid with development when creating sound effects, particle effects, shaders, and easily created user interface elements. I will be using many of these built-in features to aid in my development and streamline the development process.

1.3.2. Blender

'Blender' is an open-source 3D art and effects program, available for free to hobbyist 3D Model Artists, film makers, CGI Artists, and Special Effects designers. I will be using this software to create all the 3d assets in the game to export to the 'Unity Engine'. 'Blender's' 3d Modelling system works by creating and editing vertices in 3d space along the X-Axis, Y-Axis, and Z-Axis. These vertices are connected along edges, which are then filled in with faces to create the basic structure of the 3d model. Blender has many tools and tricks which I will be using to my full advantage to create interesting and creative art style for the 'Roaster' software.

1.3.3. AR Foundation & ARCore

'AR Foundation' is part of 'Unity's' features but is downloaded in the package manager and provides a way for 'Unity Engine' to create base Augmented Reality features. This API makes it extremely easy for developers to create builds for both Android and IOS mobile devices. The only downside being that Augmented Reality on smartphones is only supported at Android 7.0, and IOS 6 (iPhone 6).

'ARCore' is a Google platform made plug-in, used to allow access to additional Augmented Reality features for Android mobile devices through the connected camera. This plug in is used to create experiences that allows users to interact with digitally made content in the real world through the screen on their phone. 'ARCore' uses Motion Tracking, Environmental understanding, and Light Estimation to blend digital creation into the real world. I will be using these features through a downloaded plugin for my 'Unity' project, to implement Augmented Reality feature for my customised Car display.

2.0 System

2.1. Requirements

2.1.1. Functional Requirements

2.1.1.1. Main Menu

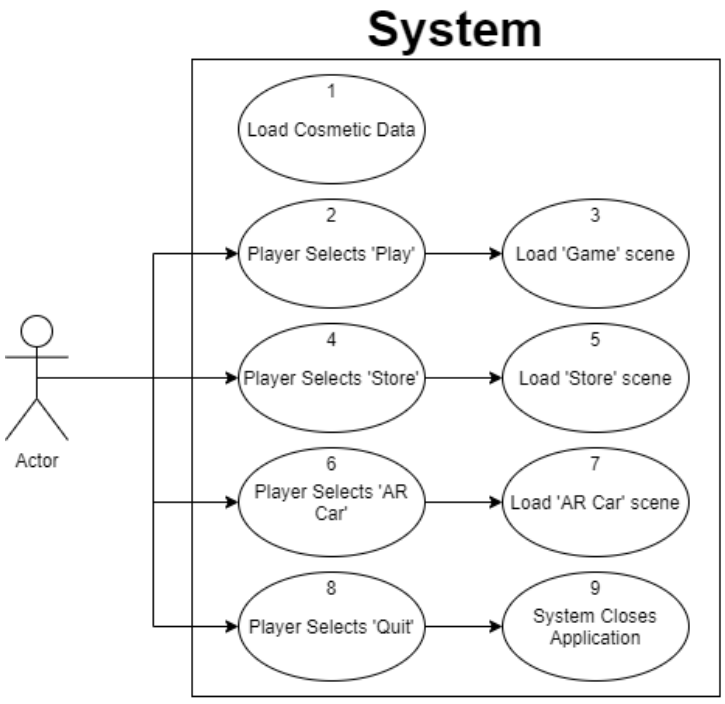
The ‘Main Menu’ will contain the Options that will lead into each flow of each requirement such as ‘Start Game’, ‘Tutorial’, ‘Augmented Reality’, and ‘Store’. This Screen will be Accessible through any state of the application.

Main Flow

1. System Loads ‘Cosmetic data’
2. Player Selects ‘Play’ Option.
3. System Loads ‘Game’ Scene.
4. Player Selects ‘Store’ Option.
5. System Loads ‘Store’ Scene.
6. Player Selects ‘AR Car’ Option.
7. System Loads ‘AR Display’ Scene.
8. Player Selects ‘Quit’.
9. System closes Application.

Alternate Flow

1. Player Exits Game.
2. System Crashes.
3. Failure to Load Scene.



(Main Menu Use Case Diagram)

2.1.1.2. Game

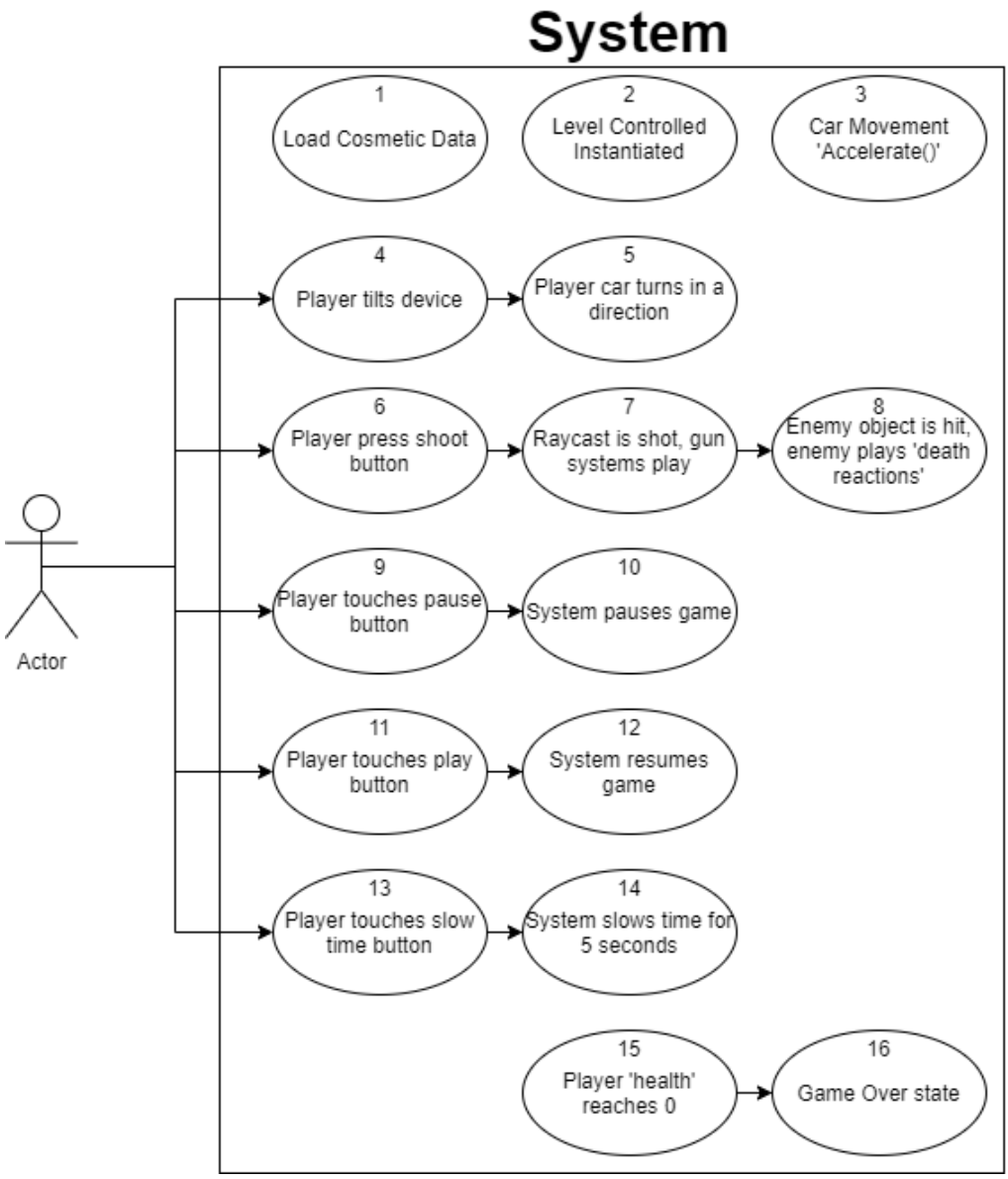
Players should be able to start a ‘Run’ of the main endless runner standard gameplay displaying the user’s previously made changes in the vehicle. This is the main feature of the game and should be most engaging for the end user. The player can access the ‘Main Game’ in the Main Menu, by clicking on the Play Game button.

Main Flow

1. System loads ‘Cosmetic data’
2. System Starts ‘Level Controller’
3. System Starts ‘Car Movement’
4. Player tilts device
5. System moves car object left or right (depending on tilt direction).
6. Player touches ‘Shoot Button’.
7. System shoots out a Raycast.
8. System shoots Object.
9. Player touches ‘Pause Button’.
10. System Pauses Game, Display Pause Menu
11. Player touches ‘Play Button’.
12. System Resumes Game.
13. Player touches ‘Slow Time’ button.
14. System set game speed to 0.7 for 5 seconds.
15. Player health hits 0.
16. System Load Game Over

Alternate Flow

1. Player Exits Game.
2. System Crashes.
3. Failure to Load Scene
4. ‘Level Spawner’ does not respond.
5. ‘Player Movement’ does not respond.
6. System cannot access ‘Cosmetic data’.



(Main Game Use Case Diagram)

2.1.1.3. Store

The Player should be able to select various options from the store tabs, sub tabs, and button display. The buttons contained in button display will set different visual aspects of the player car displayed. This data will then be saved to a file on the user's device 'Cosmetic.data'. The 'Store' Feature should be accessible through the main menu and the 'Augmented Reality Car Display'.

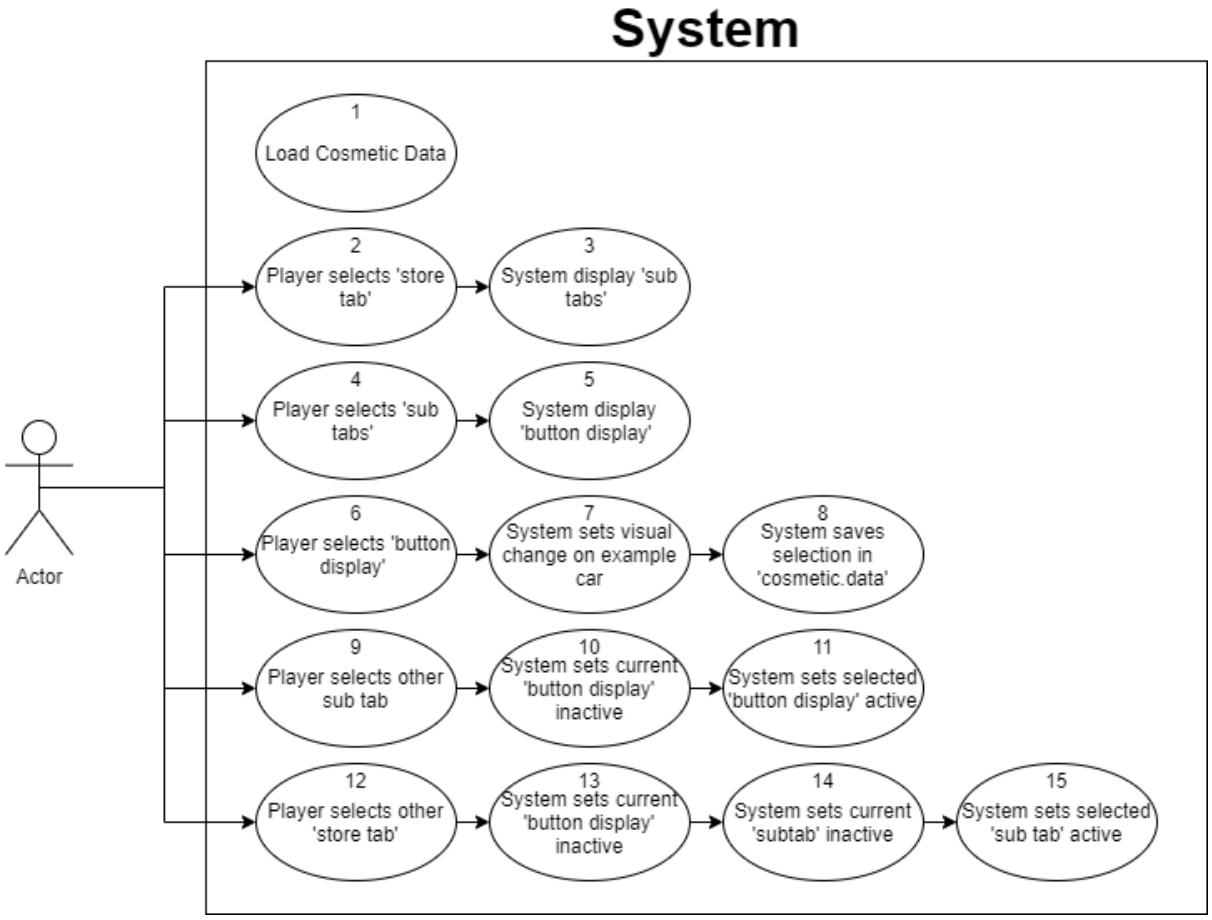
Main Flow

1. System loads 'Cosmetic data'.
2. User selects 'store tab' category.
3. System displays 'sub tabs' in the selected 'store tab' category.
4. User selects 'sub tab'.
5. System displays 'button display' for selected 'sub tab' category.

6. User selects a button.
7. System display button functionality on example car
8. System saves selected option in ‘Cosmetic.data’.
9. User selects different ‘sub tab’.
10. System hides current ‘button display’.
11. System displays ‘button display’ for selected ‘sub tab’ category.
12. User selects different ‘store tab’.
13. System hides current ‘button display’.
14. System hides current ‘sub tabs’.
15. System displays ‘sub tabs’ in the selected ‘store tab’ category.

Alternate Flow

1. System cannot access ‘Cosmetic data’.
2. System Crashes
3. System cannot save to local device.
4. Failure to Load Scene



(Store Use Case Diagram)

2.1.1.4. Augmented Reality Car Display

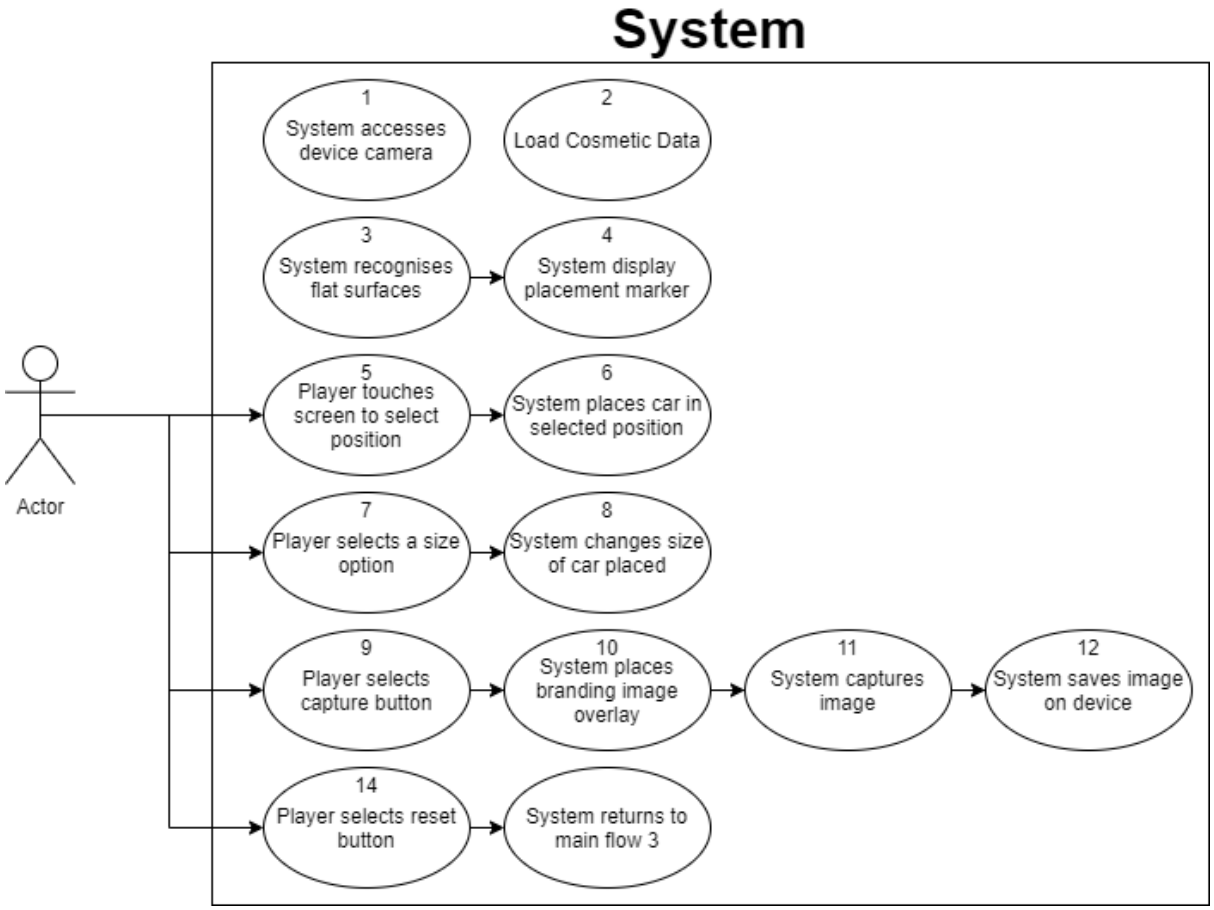
The Player should be prompted to Camera Access for the 'Augmented Reality' features and be able to Display their vehicle through their camera. The user should be able to save a photo of their customised car in the real world. The Augmented reality features will be accessible through the Main Menu.

Main Flow

1. System Accesses device Camera.
2. System load 'Player Car' with 'Cosmetic data'.
3. System Recognises Flat surfaces.
4. System displays 'Placement Marker'.
5. Player touch on screen.
6. System Places 'Player Car' in selected position.
7. Player selects sizing option.
8. System changes size of placed car.
9. Player selects 'Capture Button'.
10. System places branding overlay.
11. System Captures Image.
12. System saves image on local device.
13. Player selects reset.
14. System Recognises flat surfaces etc.

Alternate Flow

1. System cannot access 'Cosmetic data'.
2. Player resets car placement
3. System Crashes
4. System cannot save to local device.
5. System cannot access device camera.



(AR Car Use Case Diagram)

2.1.2. Data Requirements

2.1.2.1. Save States

The 'Save System' should allow the User to set cosmetic options for the player car that will remain the same when the app is terminated. The system will keep track of applied cosmetics from the 'Store' feature, that the player has equipped through previous play sessions. The player will have to play the 'Store' feature previously for this save state to come into effect.

2.1.2.2. Save Images to Device

The Player will be prompted to allow for the application to access the files of the downloaded device, this will allow pictures taken in Augmented reality tab to be stored.

2.1.3. User Requirements

2.1.3.1. Phone Operating System

Augmented Reality features are only available to Android devices over version 7.0; therefore, the app will be inaccessible to phones below this requirement. The Application APK on the

‘Google Play’ store page for ‘Roadster Run’ will be inaccessible to download when the phone does not meet these requirements.

2.1.3.2. Storage Required

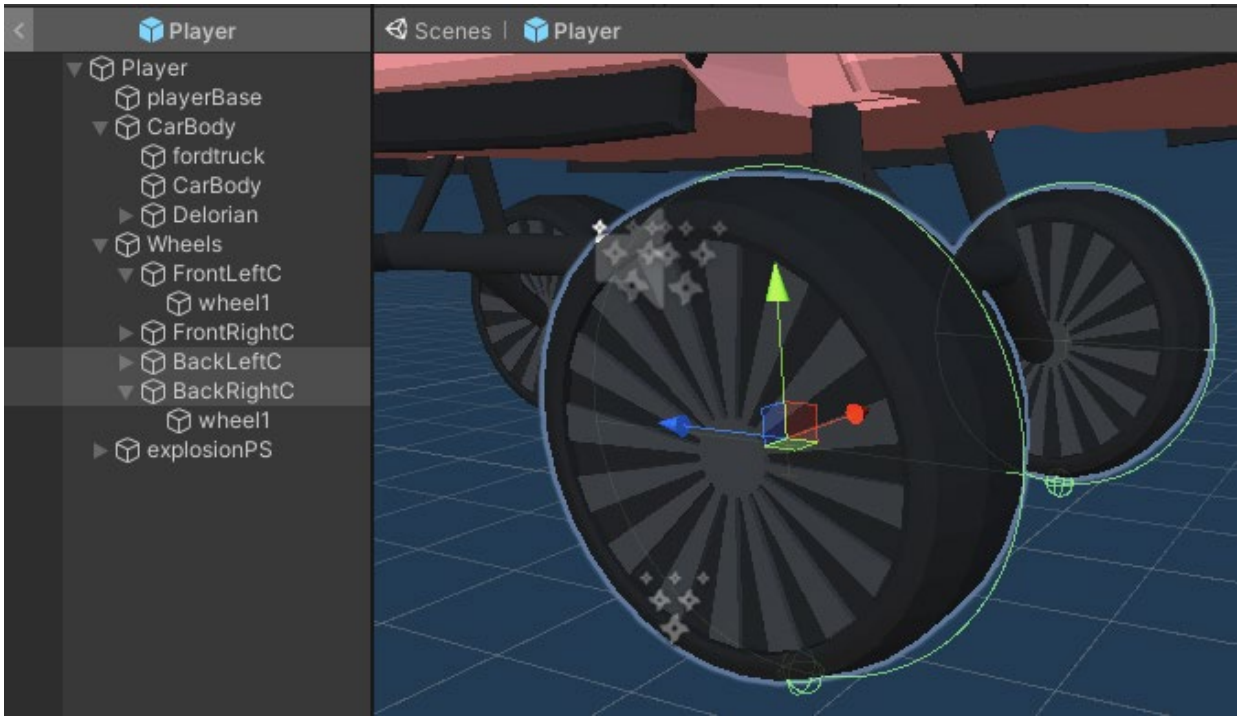
Downloading Apps from the ‘Google Play Store’ requires the device to have a certain amount of storage space on the device's hard drive for installation. The Application ‘Roaster Run’ will also save ‘.png’ image files to the users gallery and save data with the games files when accessing the ‘Store’ feature.

2.1.3.3. Permissions Required

Accessing the devices camera requires permissions from the user when operating these features for the first time. As well as accessing camera permissions of the user’s device will create a prompt for the user to accept upon the attempt to use the augmented reality features.

2.2. Design & Architecture

2.2.1. Player Movement



(Wheel Colliders & Wheel Models)

Player Movement is essential for the main feel of the game, as it is important for the player to be able to control their car in a fun and responsive way, while maintaining the feel of how a car should move. This was achieved using Wheel Colliders and Rigid body forces to simulate responsive and snappy movement from the user's tilt inputs. Torque was applied to the wheel colliders to provoke them spinning which would move the vehicle forwards. As wheel rotation for turning the car seems unstable at higher speeds, I implemented a Rigid body force to apply to the vehicle to turn left or right. This way is not realistically accurate, but I found it made for a fun a responsive way for the player to dodge incoming enemies. This left and right movement is controlled by the accelerometer within the mobile device by turning the device in real time.

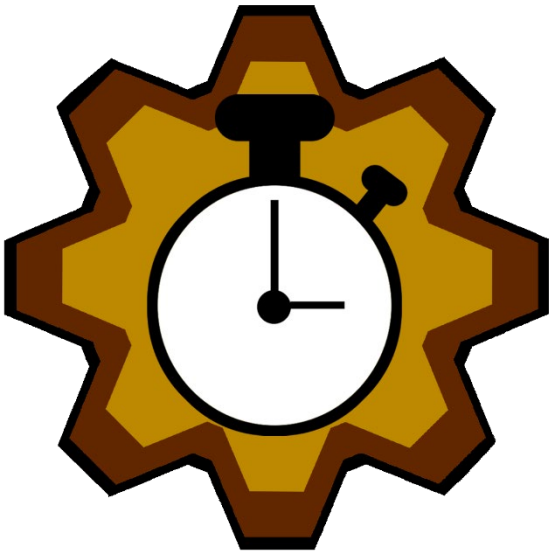
(Unity Editor: Car Movement & Rigidbody)

2.2.2. Player Abilities & Weaponry

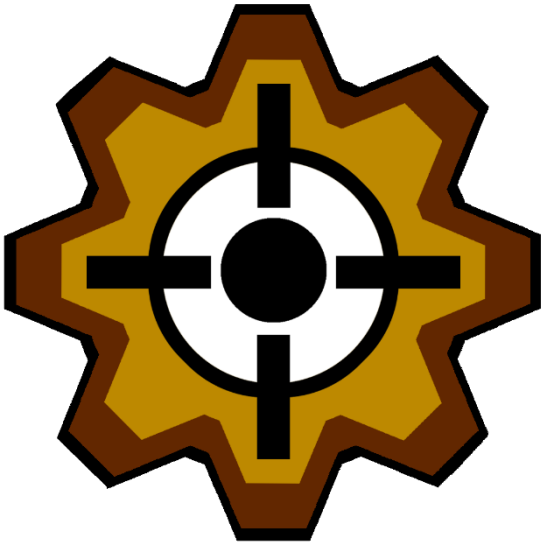
The player will have different abilities for gameplay variation through the main game.

The Ability to slow down time is a useful tool to slow everything down for the player if things are getting difficult. The player can have the ability to press a button in the bottom left of the main game User Interface and slow down time for 5 seconds.

The Player also has the ability to shoot straight ahead. This is used to eliminate enemy car Infront of the enemy clearing the way a prevent a collision game over state.

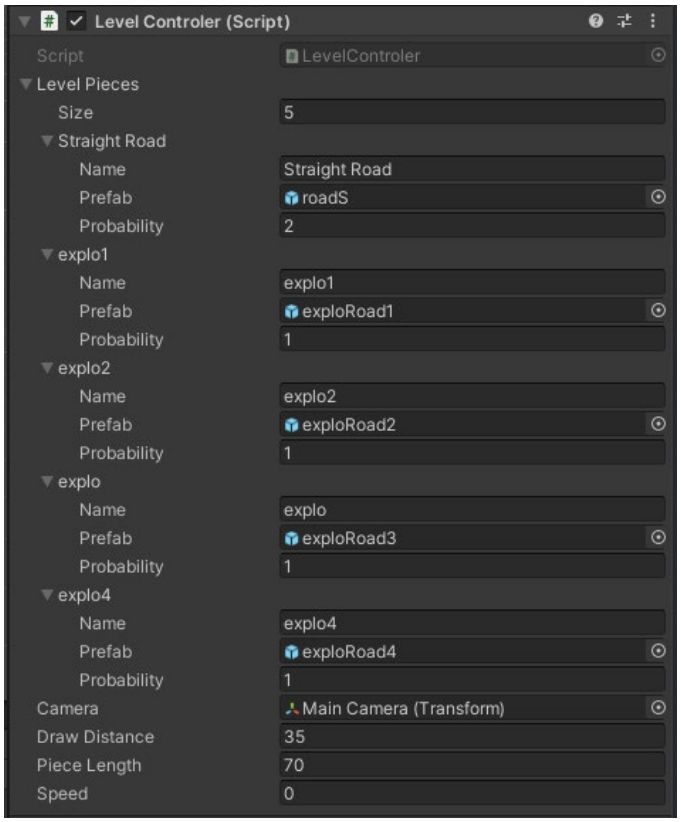


(Time Slow Button)



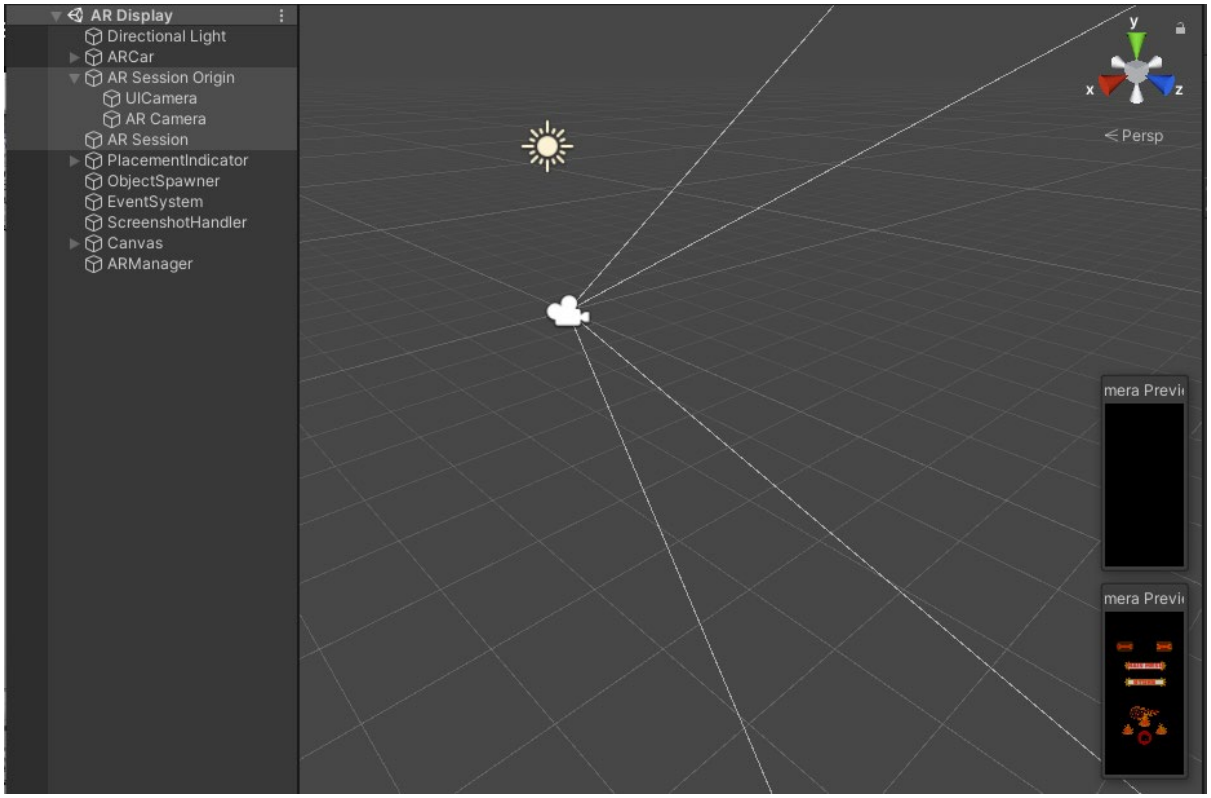
(Shoot Button)

2.2.3. Level Controller



The Level Spawner works by spawning 'LevelPiece' Objects that I have allowed the system to view, this will on game start spawn 35 level pieces in front to create a straight road as a playable area. 'LevelPiece' is a class within the 'C#' script that holds its name, prefab (collection of 1 or multiple objects saved as a collection in 'Unity'), and the probability factor of the level piece. On game start, 35 level pieces will spawn randomly picking each 'LevelPiece' that is contained within an Array. As the car travels forward along the level pieces, directly below the camera position (behind the player car) the level piece will be destroyed, but also provoke a new piece to be spawned 34 pieces ahead of the camera.

2.2.4. AR Car



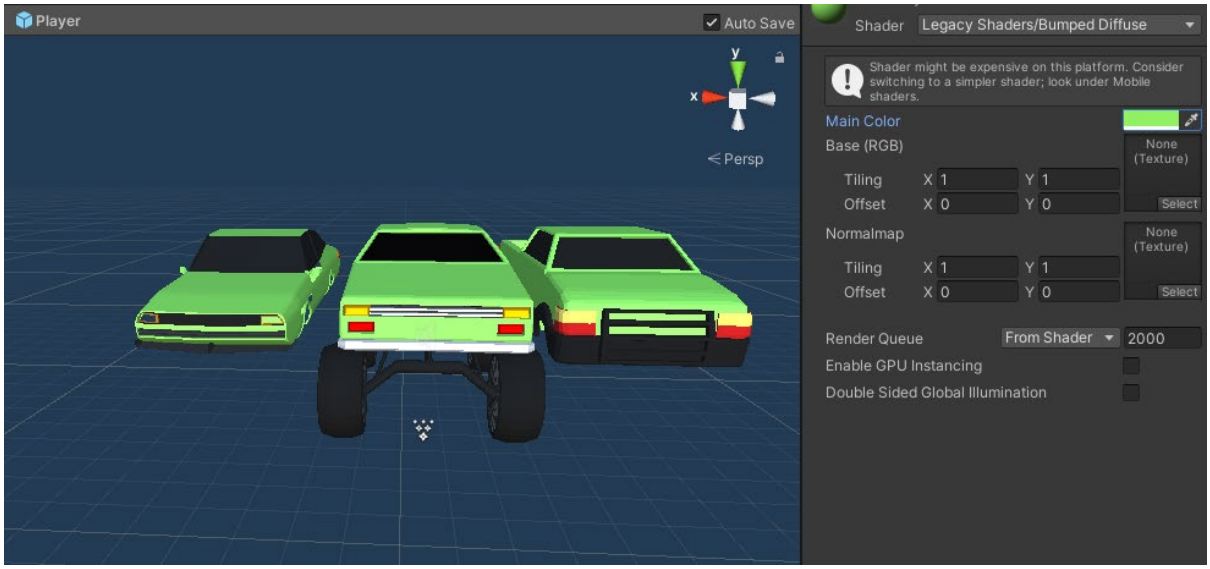
(XR Objects: AR Session Origin, AR Camera, AR Session)

The 'AR Car' scene allows the user to project their customised car into the real world using 'Augmented Reality'. The 'AR Foundation' and 'AR Core' are the plugins that allow the 'Unity Engine' the tools and objects 'XR' (Extended Reality) to display though the devices camera. This works through the system being able to identify flat surfaces and then allowing a model of your choice to be spawned on the shown indicator's point. When the system recognises flat surfaces, an indicator will show in the middle of the screen indicating where the imputed object will be placed. The 'AR Session Origin' object in the scene allows the application to access the devices camera, and the 'AR Session' object allows the camera to track surfaces using feature points commonly used in 'Augmented Reality' technology.

A big reason behind the feature of the augmented reality car display is the branding and marketing purposes. I want the user to be able to have a personal connection with their unique personalised car in 'Roadster Run' and can share their car on social media through capturing images.

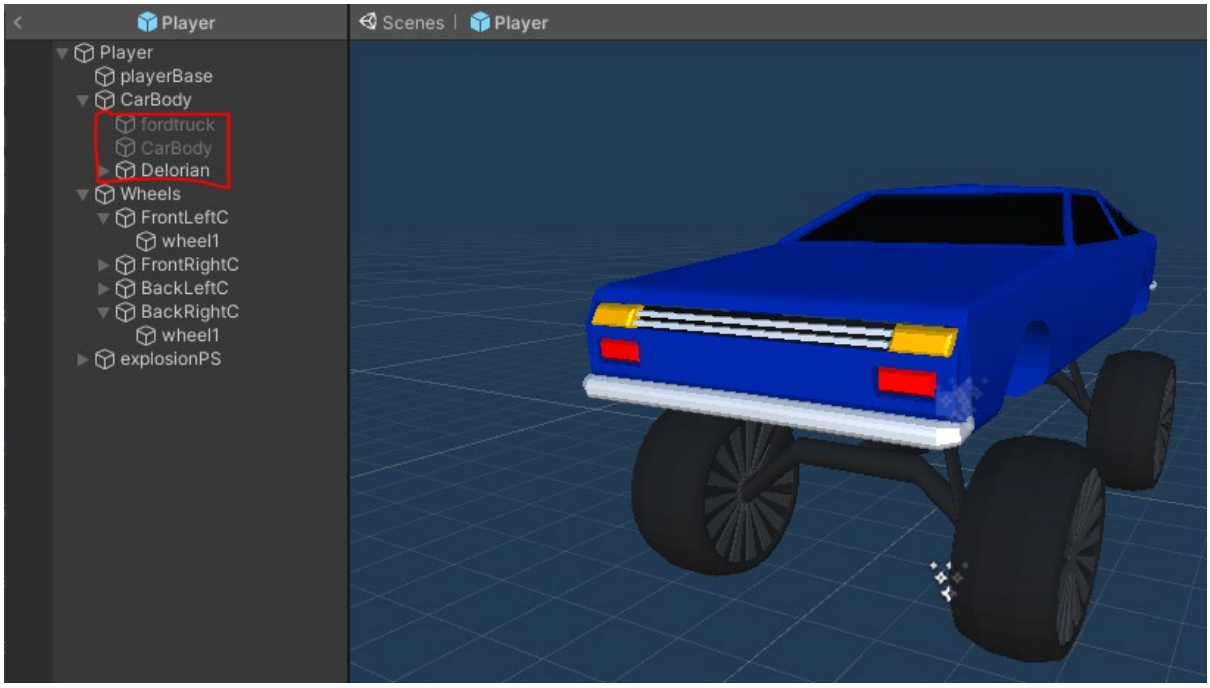
2.2.5. Store: Car Customisation

The 'Store' feature allows the user to customise the cosmetics of the player car, this data is then saved and recalled in every scene load to give a consistent player car though out the game. At this point in development the user can edit the body 'CarBody' and color 'CarColor' of their personalised vehicle. The player car is attached to a bar rig that remain consistent with every combination of cosmetically altered version of the player car, as well as the components and physics based objects are tied to this 3d model.



(Editing Car Color in Player prefab)

The changes in car color by all the car models having the material set as the color for the body of the. This way by changing one material all the car bodies available will receive the same paint.



(Player Prefab, active car bodies)

The ‘CarBody’ system works by setting various object on the player car to active or inactive. When a model is set to inactive it is not visible to the player when running the application.

2.2.6. Sound Design

I have created many scripts that handle the sound of the game. Some of these scripts are set up for a 'Scene' such as 'ShopSounds.cs', and other are set up for features certain features that have a lot of sound queues attached such as 'WeaponSounds.cs'. These are scripts that call the 'Unity Engine' component 'Audio Source' on a particular object, and load different sounds contained in the 'Resources' folder.

2.3. Implementation

2.3.1. Player Movement: Accelerate and Max Speed

The most basic function for a car is to accelerate forward. I wanted to create a semi-realistic car movement as game mechanics grounded in reality are more fun for a player and sell the feeling of driving a car. I decided to create a 'Unity Engine' physics based system using 'Rigidbody' and 'Wheel Colliders' using within the 'Accelerate()' function in the 'CarMovement.cs' script. This 'Accelerate()' function is called within a 'FixedUpdate()', a 'Unity' function, which calls all functions and code within will be run every potential frame defined for the game.

```
if (currentspeed <= maxspeed)
        {
            backLeftW.motorTorque = 1 * motorForce;
            backRightW.motorTorque = 1 * motorForce;
        }
        else if (currentspeed >= maxspeed)
        {
            backLeftW.motorTorque = 0 * motorForce;
            backRightW.motorTorque = 0 * motorForce;
        }
```

'BackLeftW' and 'BackRightW' refer to the cars back wheel colliders which were set as this variable in the 'Unity Engine's Editor. The 'motorTorque' turns the wheel colliders in game which creates the cars movement.

```
        ApplyLocalPositionToVisuals(backLeftW);
        ApplyLocalPositionToVisuals(backRightW);
        ApplyLocalPositionToVisuals(frontLeftW);
        ApplyLocalPositionToVisuals(frontRightW);
```

Now that the wheel colliders are spinning it is important for the actual wheel models to follow the same speed to really sell the realistic car movement. I have set wheel models to child elements of their counterpart 'Wheel Collider'. Using the function 'ApplyLocalPositionToVisuals()' and passing through a 'Wheel Collider' we can achieve the effect of the wheel models moving.

```
public void ApplyLocalPositionToVisuals(WheelCollider collider)
    {
        Transform visualWheel = collider.transform.GetChild(0);

        Vector3 position;
        Quaternion rotation;
        collider.GetWorldPose(out position, out rotation);

        //rotation =rotation * Quaternion.Euler(0f, -90f, 3f);

        visualWheel.transform.position = position;
        visualWheel.transform.rotation = rotation;
    }
```

As the Wheel Model is a child of the Wheel Collider, we grab the current wheel in use using the 'GetChild()' function, and save the object as 'visualWheel'. To have the 'Wheel Collider' and the wheel model moving the same I have taken the current position and rotation of the inputted 'Wheel Collider' using the 'Unity' function 'GetWorldPose()'. This current position and current rotation are then applied to this 'visualWheel'. Due to 'Accelerate()' being called every frame rendered, the 'ApplyLocalPositionToVisuals()' function is also updating the wheel position and rotation every frame.

Without defining a max speed for the player, the player care will infinitely get faster and faster, becoming more uncontrollable, and making other systems harder to work alongside with the player.

```
currentspeed = 2 * 22 / 7 * backLeftW.radius * backLeftW.rpm * 60 / 1000;
```

This algorithm will define a current speed and is based off the formula to get speed from a wheel radius circumference = 2 * π * radius. When we find the circumference of the wheel, we multiply it by the rotation per minute (backLeftW.rpm*60/1000). Although this game does not require real world measurements, this is a good basis to limit a speed. Torque is only applied to the wheels when the 'current speed' is less than a predefined 'maxSpeed'.

2.3.2. Player Movement: Sound Design

Realistic car engines change pitch the higher of a gear they are in, I wanted to simulate this feeling in the player controller. I achieved this using the value of 'currentspeed', in the 'Accelerate()' function within the 'CarMovement.cs' script.

```
if (currentspeed <= 100)
        {
            ac.pitch = 0.5f;
        }
        else if (currentspeed <= 200 && currentspeed >= 100)
        {
            ac.pitch = 0.65f;
        }
        else if (currentspeed >=200)
        {
            ac.pitch = .72f;
        }
```

Here we see I am using the value of 'currentspeed' to change the pitch of the audio source playing. The variable 'ac' is referring to a particular 'Audio Source' component attached to the player car. If the 'currentspeed' is within a certain range the audio is set to a particular pitch.

2.3.3. Player Movement: Move Tilt

As I was developing the 'Player Controller', I found the more traditional method of rotating the front wheels of a car by a certain number of max degrees, to result in quite an uncontrollable car when travelling down a straight road. This made the player constantly be readjusting the car's angle to drive straight. I thought of more ways to achieve switching lanes using rigid body forces, but this would cause the issue of the car rolling over if too much force was applied. Using Rigid body Constraints, I was able to achieve a more arcade style left and right movement scheme, while disabling the constraints when no force is being applied. The Component of 'Rigidbody' attached to the player was accessed using the variable 'rb'.

```
 void MoveTilt() {
        float temp = Input.acceleration.x;
        if (0.001 < temp || temp < -0.001) {
            rb.velocity = new Vector3(temp * handling, rb.velocity.y,
rb.velocity.z);
            rb.constraints = RigidbodyConstraints.None;
            rb.constraints = RigidbodyConstraints.FreezeRotationY;
            rb.constraints = RigidbodyConstraints.FreezeRotationZ;
            rb.constraints = RigidbodyConstraints.FreezePositionY;
        }
        else if (0.001 > temp || temp > -0.001)
        {
            rb.velocity = new Vector3(temp * -handling, rb.velocity.y,
rb.velocity.z);
            rb.constraints = RigidbodyConstraints.None;
            rb.constraints = RigidbodyConstraints.FreezeRotationY;
            rb.constraints = RigidbodyConstraints.FreezePositionX;
            rb.constraints = RigidbodyConstraints.FreezePositionY;
        }
}
```

A big part of my control scheme is the tilt controls, this was achieved using 'Input.acceleration.x', which took values from the user's device being rotated on the X-Axis. 'Input.acceleration' method would output positive and negative values depending on if the device was rotated left or right. This allowed me to quantify the devices rotation and store it as the variable 'temp'. If 'temp' is a value greater than 0.001 the car would move right, and if it was under -0.001 the player car would move left, the 0.002 difference between the two values was an easy way for the player to stabilise their car to drive straight. Using the function 'rb.velocity(x,y,z)', I was able to pass in a 'handling' variable that could be changed in the 'Unity' editor. Multiplying this 'handling' value by the 'temp' input value, would allow the player to turn left or right faster depending on how much the have rotated their phone.

2.3.4. Player Feature: Slow Motion Time

I created a 'Slow Motion Time' there are two main function for this feature. 'Slow()' is a function that handles the slowing down of the game speed, and 'Normal()' oversees setting the time speed back to normal. The feature must be always cancellable if the user does not want to stay in slow motion time, as well as the player not being able to abuse the system by constantly playing in slow motion time.

```
public void SlowmoSwitch(){
        if(slowmoable == true && rechargetime == false){
            Slow();
        }
        else if(slowmoable == false && rechargetime == false){
            Normal();
        }
        else if(slowmoable == false && rechargetime == true){

        }
    }
```

This is the function that the 'Time Slow button' will activate. Is 'slowmoable' and 'rechargetime' are set to true the user will be able call the 'Slow()' function.

```
private void Slow(){
        slowmoable = false;
        Time.timeScale = 0.7f;
        coroutine = waitfornormalTime(slomolength);
        StartCoroutine(coroutine);
    }

    IEnumerator waitfornormalTime(float _waitTime) {
        yield return new WaitForSeconds(_waitTime);
        Normal();
    }
```

The slow function first sets 'slowmoable' to false so user can cancel the action if they press the button again. The line 'Time.timescale = 0.7f' set the games time to 70%, this percentage is based off feeling of how it effects the player movement. 'Slow()' then calls a 'coroutine' timer function 'waitfornormalTime()' ,passing in the variable 'slowmolength' which is a set amount of time that can be changed in the editor (defaults is at 2 seconds). When 'waitfornormalTime()' wait for 5 second the 'Normal()' will be called.

```
public void Normal(){
        Time.timeScale = 1;
        coroutine = rechargeSlowmo(rechargelength);
        StartCoroutine(coroutine);
    }

    IEnumerator rechargeSlowmo(float _waitTime) {
        rechargetime = true;
        yield return new WaitForSeconds(_waitTime);
        slowmoable = true;
        rechargetime = false;
    }
```

The 'Normal()' will set the 'Time.timescale = 1' which is the normal speed of the game and start a 'coroutine' timer called 'rechargeSlowmo' passing in the timer variable 'rechargelength' (defaulted to 5 seconds). Recharge time is set to true which disables the player from pressing

the ‘Time Slow Button’ and activating ‘Slow()’, and after the wait time is complete the user is free to activate ‘Slow()’ again.

If the player is in ‘Slow Motion Time’ they are able to press the button again will enable the ‘Normal()’ function early to exit this time state and return to back to normal.

2.3.5. Player Feature: Weapons Logic and Sniper

I wrote two scripts to handle the shooting within ‘Roadster Run’, I designed it this was to allow to scalability for many weapons to be added in future updates. The ‘WeaponsLogic.cs’ script is what handles the ‘Shoot Button’ that allows a gun to shoot.

```
void Start(){
       guntype = "sniper";
       pos = player.transform;
       sniper = GetComponent<Sniper>();
}

void FixedUpdate(){
        transform.position = new Vector3(pos.position.x, pos.position.y + 0.5f
, pos.position.z + 5);
}
```

The ‘Start()’ function run this when the scene is started, here we have set a starting gun on the variable ‘guntype’ and retrieved the script ‘Sniper.cs’ to access it functions later. The ‘pos’ value is set to the player cars position; this is very important as the gun object is not a child of the Player car object. In the ‘FixedUpdate()’ function we move the gun object alongside the player car, so they are in the same place at all times.

```
public void ShootTouch(){
        if(guntype == "sniper"){
            sniper.canShoot();
        }
    }
```

This ‘ShootTouch()’ is called by the ‘Shoot Button’ and depending on the ‘guntype’ will call a function in the particular guns script.

The sniper weapon uses a ‘Raycast’ to shoot and damage incoming enemy cars, this can be thought of like a light ray that will recognise object, therefore making impact instant selling the feel of a sniper type weapon. The Sniper work though a ‘Shoot()’ and ‘waittoReload()’ functions.

```
public void canShoot(){
        if (shootable == true){
            Shoot();
        }
        else{

        }
    }

private void FixedUpdate()
    {
        ray = new Ray(shootingpoint.transform.position,
      -shootingpoint.transform.forward);
    }
```

The ‘ShootTouch()’ function within ‘WeaponsLogic.cs’, calls upon the ‘canShoot()’ function within ‘Sniper.cs’. This ‘canShoot()’ must have the ‘shootable’ set to true to allow the player to ‘Shoot()’. The ‘FixedUpdate()’ function is what sets up the ray from a defined ‘shootingpoint’ and sets its direction using ‘new Ray(position,direction)’. This new set ‘Raycast’ is then set to the variable ‘ray’.

```
public void Shoot()
    {
        shootable = false;
        WeaponSounds.PlaySound("snipershot");
        RaycastHit hitInfo;
        if (Physics.Raycast(ray, out hitInfo, range))
        {
            Debug.DrawLine(ray.origin, hitInfo.point, Color.red);
            Debug.Log(hitInfo.collider.name);
            Health target = hitInfo.transform.GetComponent<Health>();
            if (target != null)
            {
                target.takeDamage(damage);
            }
        }
        else
        {
            Debug.DrawLine(ray.origin, ray.direction * range, Color.green);
            Debug.Log("false");
        }
        coroutine = waitandReload(0.5f);
        StartCoroutine(coroutine);
    }
```

When ‘Shoot()’ is run ‘shootable’ is set to false to allow time between the players shots, the sound is also provoked called ‘snipershot’. The proceeding if statement, will finally draw the ‘Raycast’ ray, using ‘Physics.Raycast()’ and passing in the created ‘ray’, the range of the ray

predefined by the weapon variable ‘range’, and the output info of the ‘Raycast’ to ‘hitInfo’. Collecting the object, we have hit using the ‘Raycast’ in ‘hitinfo’, will allow us to access this particular object’s ‘Heath.cs’ script component and store it in the variable ‘target’. If the ‘target’ variable contains an object’s ‘Heath.cs’ script, we call that script to run the ’takeDamage()’ function passing in the predefined damage value of the ‘Sniper’ weapon.

To Stop the Player from abusing the ‘Shoot()’ function, and to clearly communicate with the player why they cannot shoot I have created timer functions such as ‘waittoReload()’, ‘reload()’, and ‘waittoShoot()’.

```
    IEnumerator waitandReload(float _waitTime) {
        yield return new WaitForSeconds(_waitTime);
        reload();
    }

    public void reload() {
        WeaponSounds.PlaySound("reload");
        coroutine = waittoShoot(0.5f);
        StartCoroutine(coroutine);
    }

    IEnumerator waittoShoot(float _waitTime) {
        yield return new WaitForSeconds(_waitTime);
        shootable = true;
    }
```

After the 0.5 seconds it takes for the sound ‘snipershot’ to play, the system will begin the ‘reload()’ function. This function will play the sound ‘reload’, and start another timer ‘waittoShoot()’ that is 0.5 seconds long to allow the sound to play. Before finally setting the ‘shootable’ variable back to true, so now the player can press the ‘Shoot Button’ to run the ‘Shoot()’ function once again.

2.3.6. Components: Health Script

A modular reusable health system is very important for all cars of the ‘Roadster Run’ game, so this component can be used on multiple car objects within the game. The various health scripts allows players to use the weaponry such as the ‘Sniper’ to be able to damage incoming enemy cars, as well as the player being able to achieve a game over state through loss of health. As an example script, I will be using the enemy health script ‘Health.cs ’ for the ‘Explo’ enemy, which will be reused and edited for all interactable cars in the game.

```
// ExploEnemy.cs
public void Awake(){
    GetComponent<Health>().health = 50f;
}
```

For any object using the ‘Health.cs’ script it is important to set the health amount on instantiation within the object’s main controller, in this case the ‘ExploEnemy.cs’ script gives

the ‘Health.cs ’ it’s starting health amount. This approach to setting health in the object’s main controller, will help me assign different enemies different health values for future updates.

```
public void takeDamage(float inputDamage)
    {
        health -= inputDamage;
        if (health <= 0)
        {
            DeathReations();
        }
    }
```

The ‘takeDamage()’ function within the ‘Health.cs’ component is the most important for scaling with all enemy objects, as the gun system for the player works by calling this function. The ‘inputDamage’ variable is passed though when a damage system has impact with the object and is taken away from the overall ‘health’ variable. If the ‘health’ variable is less than or equal to 0, the function ‘DeathReactions()’ is then called.

```
public void DeathReations(){
        Physics.IgnoreCollision(player.GetComponent<Collider>(),
GetComponent<Collider>());
        Physics.IgnoreCollision(deathColisionObj.GetComponent<Collider>(),
GetComponent<Collider>());

        flip();
        explosion();
        coroutine = waitandRemove(3f);
        StartCoroutine(coroutine);
}
```

When an enemy object has ran out of health, it is important that the object cannot then hurt or destroy the player. This is achieved using ‘Physics.IgnoreCollision(object to ignore, current object)’, I have ran this function twice for two player objects, the main player and the death collision object (this system will be explained in ’Game Over: Collision Death’).

```
public void flip(){
        float strength = Random.Range(30f,50);
        this.transform.Rotate(strength, 0.0f, 0.0f, Space.World);
        rb.velocity = new Vector3(rb.velocity.x, rb.velocity.y*strength,
rb.velocity.z);
}
```

Next the ‘flip()’ function is called, this randomly generates a ‘strength’ variable to pass through a ‘transform.Rotate()’ function to rotate the current object, as well as a ‘rb.velocity()’ to send the object flying into the air. This randomly generated strength value creates some variety between each car death which is more pleasing than watching the same animation for every destroyed car.

```
public void explosion(){
        boom.Play();
        ex1.Play();
        ex2.Play();
        ex3.Play();
}
```

The ‘explosion()’ function handles playing of the particle system I have created to render and explosion type visual, to sell impact of destroying a car to the player. Each of the ‘boom’, ‘ex1’, ‘ex2’, and ‘ex3’ are particle system type variables that are set in the ‘Unity Editor’, and are set to ‘Play()’.

```
IEnumerator waitandRemove(float _waitTime){
        yield return new WaitForSeconds(_waitTime);
        Destroy(gameObject);
}
```

The ‘waitandRemove()’ will wait 3 seconds and then completely destroy the object to make to free the system from tis instantiated object.

Most interactable object with a health variable act in the same way, there is a few changes for the ‘PlayerHealth.cs’ script, such as the ‘waitandRemove()’ being replaced by a ‘waitandGameover()’ function, that stops the game and shows a ‘Game Over’ state.

2.3.7. Game Over: Collision Death

I wanted all frontal collisions from to the player car to enemy cars to create an instant ‘Game Over’ state. I wanted the player to be able to hit into the backs of the enemy cars and not be penalised with a game over, so I had to create a separate object called ‘CollisionDeath’. All the cars movement is based off physics systems, therefore the model and objects attached to the car will bounce a move around a lot due to its suspension on the ‘Wheel Colliders’, and it’s ‘Rigidbody’ weight. It was much easier to create a separate object that follows in front of the player car object rather than a child element object of the player car.

```
 void Start()
    {
        pos = player.transform;
    }

    void FixedUpdate()
    {
         transform.position = new Vector3(pos.position.x, pos.position.y
-1f, pos.position.z + 5.5f);
    }
```

On ‘Start()’ function the system will find the player location. The ‘FixedUpdate()’ will make the ‘Collision Death’ object follow the players position each frame, with some off set on the y and z axis to position it in front of the player.

```
void OnCollisionEnter(Collision col){
        if(col.gameObject.tag == "Enemy" ){
            float damage = player.GetComponent<PlayerHealth>().health;
            player.GetComponent<PlayerHealth>().takeDamage(damage);
            col.gameObject.GetComponent<Health>().DeathReations();
        }
}
```

An 'OnCollisionEnter()' 'Unity' function allows collisions between box colliders to be recognised, with the 'col' variable input being the 'Collider' component of the other object. The 'if statement' is to check to see if the tag of the game object in the 'Unity' editor was set as 'Enemy'. If the object is tagged 'Enemy', the system will take the health value from the 'PlayerHealth.cs' script and store it as 'damage'. This 'damage' variable is then passed through the 'takeDamage()' function on the 'PlayerHealth.cs' script, provoking a 'DeathReaction()' on the player car. The 'DeathReaction()' is also provoked on the 'Enemy' tagged collision object.

2.3.8. Level Controller: Level Pieces

'Roadster Run's Level is straight road, where piece of the level is generated based of the player car position. For smooth framerate and good performance, the game scene cannot be overpopulated with too many objects. The 'LevelController.cs' script handles this functionality by keeping the 'LevelPiece's in the scene to a defined 35 amount.

```
public class LevelPiece {

    public string name;
    public GameObject prefab;
    public int probability = 1;
}
```

Each 'LevelPiece' is defined by a class, by a name, a game object, and number rating its probability factor.

```
private void Update()
    {
        currentCamPos = (int)(_camera.transform.position.z / pieceLength);
        if (currentCamPos != lastCamPos) {
            lastCamPos = currentCamPos;
            DestroyLevelPiece();
            SpawnNewLevelPiece();
        }
    }
```

The 'Update()' function is a unity function that will run every frame rendered for the game. Each new 'LevelPiece' instantiated and deleted is based of the position of the players camera represented in a number of pieces passed ('currentCamPos'), and the position of where the last piece was deleted ('lastCamPos'). On the provoking of creating and destroying a 'LevelPiece' the 'lastCamPos' is update to this position (represented as a number of pieces).

```
public void SpawnNewLevelPiece(){
        int pieceIndex = probabilityList[Random.Range(0,
      probabilityList.Count)];

        GameObject newLevelPiece = Instantiate
      (levelPieces[pieceIndex].prefab, new Vector3(0f, 0f, (currentCamPos +
      activePieces.Count) * pieceLength),
      transform.rotation * Quaternion.identity);

        activePieces.Enqueue(newLevelPiece);
}
```

The 'SpawnNewLevelPiece()'is responsible for spawning the a 'LevelPiece' in the correct position in the scene. A piece is selected by choosing a random number between 0 and the 'probabilityList.Count', this is the total sum of all the probabilities that are defined in the 'LevelPieces'. A Random 'LevelPiece' is selected and instantiated with the right position based of the current cam position and the number of active pieces multiplied by the 'piecelength' (how long the pieces are in the unity editor), and the rotation is set as the normal rotation which I imported the asset in to the 'Unity Engine' with using 'Quaternion.identity'. now that the piece is spawned it is added to the 'activePieces' object queue stack.

```
 void DestroyLevelPiece() {
        GameObject oldLevelPiece = activePieces.Dequeue();
        Destroy(oldLevelPiece);
    }
```

This 'DestroyLevelPiece()' is also provoked at the same time as 'SpawnNewLevelPiece()'. This function takes the bottom of the queue stack and dequeues it while setting the piece as 'oldLevelPiece'. This 'oldLevelPiece' is the destroyed in the game scene.

2.3.9. Component: Sound Scripts

I have created many sounds scripts, as an Example I am going to explain the code used in the 'ShopSounds.cs' script. This script handles all the sounds that can be played for the store by calling a function in any script.

```
void Start()
    {
        audioSrc = GetComponent<AudioSource>();

        paintspray = Resources.Load<AudioClip>("paintspray");
        drill = Resources.Load<AudioClip>("drill");
        carInstall = Resources.Load<AudioClip>("carInstall");
    }
```

On the 'Start()' function, we get the 'Audio Source' component attached to the current scripts object using the variable 'audioSrc'. Here we can also set variables that contain the .wav audio files contained in the 'Resources' folder.

```
public static void PlaySound(string sound){
        switch (sound)
        {
            case "paintspray":
                audioSrc.PlayOneShot(paintspray);
                break;
            case "drill":
                audioSrc.PlayOneShot(drill);
                break;
            case "carInstall":
                audioSrc.PlayOneShot(carInstall);
                break;
        }
}
```

Here we have the function 'PlaySound()' which accepts a string input from any script attached to the same object. This 'switch statement' will play the right sound depending on the input, with the statement 'audioSrc.PlayOneShot()' which will play the sound file once.

```
//CosmeticColors.cs
ShopSounds.PlaySound("paintspray");
```

This is how a script 'CosmeticColors.cs' attached to the same object as the 'ShopSounds.cs' script can initiate a sound.

```
//CosmeticColors.cs
public void playsound(){
        if(soundAvaliable == true){
            soundAvaliable = false;
            ShopSounds.PlaySound("paintspray");
            coroutine = waittoplay(1f);
            StartCoroutine(coroutine);
        }
    }

    IEnumerator waittoplay(float _waitTime){
        yield return new WaitForSeconds(_waitTime);
        soundAvaliable = true;
    }
```

When sound designing for store item buttons that can be infinitely pressed, it is a good practice to define when a sound can be played by the user. I have done with the 'playsound()' function in the 'CosmeticColors.cs' script by making a Boolean switch 'soundAvaliable', which locks the user to only play one sound per second.

2.3.10. Store: Set Cosmetic Car Visuals

This system is used in both tasks of setting the color and the body type of the player car. The 'CosmeticColors.cs' script handles setting the material color for each of the available cars. This

script is accessible to change the colors within the ‘Store’ scene, where it will pass on the set data to the ‘SaveSystem.cs’ script.

```
public class ColorType {
    public string name;
    public Color color;
}
```

The different colors available in the ‘Store’ are stored in a class ‘ColorType’, containing a string ‘name’ and a Color (RGB value) ‘color’.

```
public void Start(){
        colortypes = new List<ColorType>();
        settingColorsInList(colortypes);
        data = SaveSystem.LoadData();
        carcolor = data.CarColor.ToString();
        Debug.Log("car color loads :" + data.CarColor);
        setColor(carcolor);
}
```

On the ‘Start()’ function a new Arraylist ‘colortypes’ is created with object type as ‘ColorType’. This ‘colortypes’ is then passed into the ‘settingColorsInList()’ which will prepare all available color available to the player. The next few lines are to do with the ‘SaveSystem.cs’ ‘LoadData()’ function for setting the correct color on each car model (this system will be explained in ‘Save System: Save Cosmetic data’).

```
public void settingColorsInList(List<ColorType> colortypes){

        ColorType blue = new ColorType();
        blue.name ="blue";
        blue.color =new Color(0/255f,35/255f,142/255f);
        colortypes.Add(blue);

        ColorType darkblue = new ColorType();
        darkblue.name ="darkblue";
        darkblue.color  = new Color(2/255f,21/255f,80/255f);
        colortypes.Add(darkblue);

        ColorType babyblue = new ColorType();
        babyblue.name ="babyblue";
        babyblue.color = new Color(0/255f,35/255f,142/255f);
        colortypes.Add(babyblue);

        //etc with more colors…
}
```

In the ‘settingColorsInList()’ function we prepare the input array list with all the possible colors available to the player. There is a new ‘ColorType’ created and given a ‘name’ and RGB value Color type set as the ‘color’. This is then stored within the ‘colortypes’ arraylist passed into the ‘settingColorsInList()’ function.

```
public void changeColor(string c){
        if(colortypes.Count == 0){
            Debug.LogError("Elements wount add");
        }
        else{
            for(int i =0; i < colortypes.Count; i++){
                if(colortypes[i].name == c){
                    playerColor.color = colortypes[i].color;
                    carcolor = c;
                    playsound();
                    SaveSystem.dataBuilder();
                }
            }
        }
    }
```

This ‘changeColor()’ function will be provoked by each of the ‘Color Buttons’, each passing in their corresponding color name when pressed. The first ‘if statement’ checks to see if the ‘ColorType’s have added to the ‘colortypes’ array list. If the ‘colortypes’ arraylist is populated, a ‘for loop’ will cycle through each of the ‘ColorType’s in the arraylist. If the name of the ‘ColorType’ object matches the input from the ‘ColorButton’, that ‘ColorType’ objects ‘color’ input will be set as the ‘playerColor.color’ variable. The ‘playerColor’ is a material type object that was set in the ‘Unity’ Editor, we can access the color variable of this material in ‘C#’ using the ‘.color’ extension. The ‘carcolor’ variable is now set to the same as the ‘ColorType’ name and the input, this is to prepare the data for the ‘SaveSystem.cs’ Script. The ‘playsound()’ function is run to play a noise for player feedback (this system is explained in Component: Sound Scripts). Then the ‘SaveSystem.dataBuilder()’ function is provoked to save the changed data.

```
public void changeBody(string b){
        if(bodytypes.Count == 0){
            Debug.LogError("Elements dont Exist");
        }
        else{
            for(int i =0; i < bodytypes.Count; i++){
                if(bodytypes[i].name == b){
                    bodytypes[i].body.SetActive(true);
                    carbody = b;
                    playsound();
                    SaveSystem.dataBuilder();
                }
                else{
                    bodytypes[i].body.SetActive(false);
                }
            }
        }
    }
```

There is a very similar system used in the ‘CosmeticCarBody.cs’ script. A class is defined ‘BodyType’, and objects are added to an arraylist. The ‘Body Buttons’ will then pass in a string to ‘changeBody()’. By matching the name of the ‘BodyType.name’ object with the input, the right ‘BodyType.body’ game object is set as active in the scene ‘bodytypes[i].body.SetActive(true)’. The name of the ‘BodyType’ is then stored in ‘carbody’ to so ‘SaveSystem.dataBuilder()’ function can save the changed data.

2.3.11.Save System: Save Cosmetic Data

The ‘SaveSystem.cs’ allow the system to use binary formatting to build a file ‘cosmetic.data’ to save the players requested car color and car body selected in the ‘Store’ scene. This is a static class so can be always accessed in all scenes without attaching it to a game object.

```
public class Cosmeticdata
{
    public string CarColor;
    public string CarBody;
    public string CarWheel;
}
```

The data that is stored is represented in a class ‘Cosmeticdata’, which stores three string values, ‘CarColor’, ‘CarBody’, and ‘CarWheel’ each representing an option selected in the ‘Store’ scene.

```
public static void dataBuilder(){
        grabthedata = GameObject.Find("StoreHandler");

        Cosmeticdata dataCos = new Cosmeticdata();
        dataCos.CarColor= grabthedata.GetComponent<CosmeticColors>().carcolor;
        dataCos.CarBody = grabthedata.GetComponent<CosmeticCarBody>().carbody;
        dataCos.CarWheel = "Wheel1";

        SaveData(dataCos);
    }
```

The ‘dataBuilder()’ function’s purpose is to gather all the data pieces from each script to build a ‘Cosmeticdata’ object. First to grab the data we ‘SaveSystem.cs’ needs access to the ‘StoreHandler’ game object which contains the ‘Store’ scene scripts. A new ‘Cosmeticdata’ object is created called ‘dataCos’. We can then append the ‘carcolor’ string from the ‘CosmeticColors.cs’ script and store it as the ‘dataCos.CarColor’, also the ‘carbody’ string from the ‘CosmeticCarBody.cs’ script can be stored as the ‘dataCos.CarBody’. This format of gathering the data from different scripts can be used also for the ‘dataCos.CarWheel’ when the feature is implemented, for the moment it has the placeholder of ‘Wheel1’. Then the created an filled out ‘Cosmeticdata’ object ‘dataCos’ is passed through the ‘SaveData()’ function.

```
public static void SaveData(Cosmeticdata dataCos){

        BinaryFormatter formatter = new BinaryFormatter();
        string path = Application.persistentDataPath + "/cosmetic.data";
        FileStream stream = new FileStream(path, FileMode.Create);

        formatter.Serialize(stream, dataCos);
        stream.Close();
        Debug.Log("Saved Color :" + dataCos.CarColor);
        Debug.Log("Saved Body :" + dataCos.CarBody);
        Debug.Log("Saved Wheel :" + dataCos.CarWheel);
    }
```

First we need to access the Binary formatter that is an imported into the script witch was gathered with 'using System.Runtime.Serialization.Formatters.Binary;' at the top of the script. I then declare a new instance of a 'BinaryFormatter' object and name it 'formatter'. I define a path for that data file to be saved using 'Application.persistentDataPath', this defines a place within the games files on installation. A 'File Stream' is then declared that defines the operation of the 'path' of the file and it objective to create a file with 'Filemode.Create'. The data is then finally created into a binary file using the 'formatter' ('BinaryFormatter') to serialise the 'Cosmeticdata' object created in the 'dataBuilder()' function to save on the path defined in the 'stream' variable. The stream must be closed 'stream.Close()' after the data stream to stop any interference with the input data.

2.3.12. Save System: Load Cosmetic Data

Now that the 'Cosmetic data' has been saved in the 'cosmetic.data' file created this saved data need to be applied to the many versions of the player car throughout the application when opening a scene. For an example I will be using the 'Car Body' setting function in 'CosmeticCarBodys.cs' script, but all loading functions in 'Roadster Run' will work in a similar way.

```
//CosmeticCarBody.cs
public void Start(){
        data = SaveSystem.LoadData();
        carbody = data.CarBody.ToString();
        Debug.Log("car color loads :" + data.CarBody);

        setBody(carbody);
}
```

On the 'Start()' function of the 'CosmeticCarBody.cs' script, the 'SaveSystem.LoadData()' function is called to store the data from the 'cosmetic.data' file created in the previous section.

```
//SaveSystem.cs
public static Cosmeticdata LoadData(){
        string path = Application.persistentDataPath + "/cosmetic.data";
        if (File.Exists(path)){

            BinaryFormatter formatter = new BinaryFormatter();
            FileStream stream = new FileStream(path, FileMode.Open);
            Cosmeticdata data = formatter.Deserialize(stream) as Cosmeticdata;
            stream.Close();
            Debug.Log("Loaded Color: " + data.CarColor);
            Debug.Log("Loaded Body: " + data.CarBody);
            Debug.Log("Loaded Wheel: " + data.CarWheel);

            return data;
            //Debug.Log("Found data");
        }else{
            Debug.LogError("Issue finding data");
            return null;
        }
}
```

In the ‘SaveSystem.cs’ script, the ‘path’ is found by inputting the same data path from where we save the ‘cosmetic.data’ file, using ‘Application.persistentDataPath’. If the path contains a file, we create a new instance of ‘Binary Formatter‘ and create another ‘FileStream’ ‘stream’ inputting the ‘path’ and declaring an open operation using ‘FileMode.Open’. The ‘stream’ is the deserialized and converted back into a ‘Cosmeticdata’ object, and stored into ‘data’. The ‘data’ is then returned to the instance calling the ‘SaveSystem.LoadData()’ function.

```
//CosmeticCarBody.cs
public void setBody(string b){
        if(bodytypes.Count == 0){
            Debug.LogError("Elements wount add");
        }
        else{
            for(int i =0; i < bodytypes.Count; i++){
                if(bodytypes[i].name == b){
                    bodytypes[i].body.SetActive(true);
                }
                else{
                    bodytypes[i].body.SetActive(false);
                }
            }
        }
    }
```

The ‘CometicCarBody.cs’ script calls the ‘setBody()’ in the ‘Start()’ function, passing the ‘data.CarBody’ variable received from the ‘SaveSystem.LoadData()’. This ‘setBody()’ function work very similar to the ‘changeBody()’ function seen in the ‘Store: Set Cosmetic Car

Visuals' section, the key differences being the lack of playing a sound and saving the data that is being applied to the player car.

```
public void setColor(string c){
         if(colortypes.Count == 0){
            Debug.LogError("Elements wount add
        }
        else{
            for(int i =0; i < colortypes.Count; i++){
                if(colortypes[i].name == c){
                    playerColor.color = colortypes[i].color;
                }
            }
        }
    }
```

Likewise, the 'setColor()' function works in a very similar way to the 'changeBody()' function seen in the 'Store: Set Cosmetic Car Visuals' section. The data being passed into the 'setColor()' function is the 'data.CarColor' variable received from calling the 'SaveSystem.LoadData()' function.

2.3.13.Augmented Reality: AR Foundation and AR Core

The 'AR Car' scene allows the user to use their mobile devices camera to project their customized car into the real world. Using the 'Unity' made plug in 'AR Foundation' and 'AR Core' so we can access base augmented reality functions for within the 'Unity' editor.

The 'PlacementIndicator.cs' job is to read the 'AR Sessions Origins' data collected through the 'AR Camera' object on where placeable surfaces are. This can be tested by placing a material prefab into the 'Plane Prefab' section on the 'AR Plain Manager' component, that is attached to the 'AR Session Object'.

```
//PlacementIndicator.cs
public void Start()
    {
        rayManager = FindObjectOfType<ARRaycastManager>();
        visual = transform.GetChild(0).gameObject;
        visual.SetActive(false);
    }
```

On the 'Start()' function of the 'PlacementIndicator.cs' script, we need to access the 'AR Raycast Manager' compent on the 'AR Session' object. We also need to gather the material prefab that is set as a child of the current 'Placement Indicator' object to use later to indicate placement of the AR car. I set this material prefab as not viewable by using 'visual.SetActive(false)'.

```
public void Update()
    {
        List<ARRaycastHit> hits = new List<ARRaycastHit>();
        rayManager.Raycast(new Vector2(Screen.width / 2, Screen.height / 2),
hits, TrackableType.Planes);

        if (hits.Count > 0) {
            transform.position = hits[0].pose.position;
            transform.rotation = hits[0].pose.rotation;

            if (!visual.activeInHierarchy) {
                visual.SetActive(true);
            }
        }
    }
```

On the ‘Update()’ function, there is an array list of ‘ARRaycastHit’ type object called ‘hits’ instantiated , which stores various information about the recognised surfaces using the ‘AR Session Object’ such as position and rotation. Then the a ‘Raycast’ initiated form the center of the screen (or device camera), the information is then stored in the ‘hits’ arraylist, where the Raycast will look for planes using ‘TrackableType.Planes’. If the ‘hits’ has received entrys, we update the position of the current object to the position and rotation of the last added ‘hits’ entry. If the ‘visual’ plane (indicator) is not visible ‘!visual.activeInHeirarchy’, we set it to visible ‘visual.setActive(true)’.

So now that we have an indicator on you devices camera displaying the potential place we can place an object. Now we need to instantiate an object on that given ‘hits’ entry position.

```
public void Start()
    {
        allowSpawn= true;
        placementIndicator = FindObjectOfType<PlacementIndicator>();
    }
    public void Update()
    {
        PlaceCar();
    }
```

On the ‘Start()’ function we set a switch that limits the objects to spawn to one, with ‘allowSpawn’, and find the ‘PlacementIndicator.cs’ component. In the ‘Update()’ function we run the ‘PlaceCar()’ function to initiate spawning the player car.

```
public void PlaceCar(){
        if (Input.touchCount > 0 && Input.touches[0].phase == TouchPhase.Began
 && allowSpawn == true) {
            obj = Instantiate(objectToSpawn,
placementIndicator.transform.position, placementIndicator.transform.rotation);
            allowSpawn = false;
            placement.SetActive(false);

            manager.GetComponent<ARManager>().placedCar();
        }
    }
```

The ‘PlaceCar()’ function recognises touches from the user on the touch screen and will instantiate the ‘AR Car’ object. The variable ‘objectToSpawn’ is set to a prepapared ‘AR Car’ object in the ‘Unity’ editor. The ‘objectToSpawn’ object is instantiated in the position and rotation of the latest ‘Placement Indicator’ position and saved as ‘obj’. Then ‘allowSpawn’ is set to false to prevent the user from abusing the object spawner, as well as the ‘Placement Indicator’ objects functionality removed to stop viewing the ‘Placement Indicator’ plane. The ‘ARManager’ component is called to set the UI correctly for the situation.

```
public void RemoveCar(){
        Destroy(obj);
        allowSpawn = true;
        placement.SetActive(true);
        manager.GetComponent<ARManager>().deletedCar();
    }
```

The user should be able to remove the saved object by pressing a reset user interface button which calls the ‘RemoveCar()’ function. This destroys the instantiated ‘AR Car’ object ‘obj’ and sets the ‘allowSpawn’ Boolean switch to true. This sets the Placement Indicator object back to active so the device camera can once again recognise surfaces with the indicator image. The ‘ARManager’ component is called to set the UI correctly for the situation.

```
public void changeSize(string size){
        if(size == "small"){
            obj.transform.localScale = new Vector3(0.02f, 0.02f, 0.02f);
        }else if(size == "medium"){
            obj.transform.localScale = new Vector3(0.05f, 0.05f, 0.05f);
        }
        else if(size == "large"){
            obj.transform.localScale = new Vector3(1f, 1f, 1f);
        }
    }
```

I wanted the user to be allowed to set the ‘AR Cars’ size using the features user interface when the object was instantiated. Three buttons will run this ‘changeSize()’ function, each passing their own string ‘small’, ‘medium’, or ‘large’ resetting the local scale of the ‘AR Car’ in the devices camera.

2.3.14.Augmented Reality: Capture Image

An important part of the ‘AR Car’ scene is the capturing of images saving to the user devices file system.

```
public void TakeAShot()
    {
        manager.GetComponent<ARManager>().imageSetup();
        StartCoroutine ("CRSaveScreenshot");
    }
```

This ‘TakeAShot()’ function is called by the image capture button that shows when an ‘AR Car’ object is instantiated through the user’s device camera. First the ‘ARManager.cs’ script is called to run the ‘imageSetup()’ function hide the user interface and display the logo the logo.

```
IEnumerator CRSaveScreenshot()
    {
        yield return new WaitForEndOfFrame();

        string myFileName = "RadsterRun" + System.DateTime.Now.Hour +
System.DateTime.Now.Minute + System.DateTime.Now.Second + ".png";
        string myDefaultLocation = Application.persistentDataPath + "/" +
myFileName;
        string myFolderLocation =
"/storage/emulated/0/DCIM/Camera/RoadsterRun/";
        string myScreenshotLocation = myFolderLocation + myFileName;

        if (!System.IO.Directory.Exists(myFolderLocation))
        {
            System.IO.Directory.CreateDirectory(myFolderLocation);
        }

        ScreenCapture.CaptureScreenshot(myFileName);

        yield return new WaitForSeconds(1);

        manager.GetComponent<ARManager>().imagedone();

        System.IO.File.Move(myDefaultLocation, myScreenshotLocation);

        AndroidJavaClass classPlayer = new AndroidJavaClass("com.unity3d.
player.UnityPlayer");
        AndroidJavaObject objActivity = classPlayer.
GetStatic<AndroidJavaObject>("currentActivity");
        AndroidJavaClass classUri = new AndroidJavaClass("android.net.Uri");
        AndroidJavaObject objIntent = new AndroidJavaObject("android.content.
Intent", new object[2] { "android.intent.action.MEDIA_MOUNTED", classUri.
CallStatic<AndroidJavaObject>("parse", "file://" + myScreenshotLocation) });

        objActivity.Call("sendBroadcast", objIntent);
    }
```

Accessing the file system ‘Photo Galley’ of an android phone using Unity was similar to the Save System where ‘System.IO.File’ was used to access the devices internal folder for storage. The images was taken on the next farme after calling the ‘CRSaveScreen()’ function using ‘WaitForEndOfFrame()’. Detail of the saved image were then set up for storage such as the file name ‘myFileName’, the image location ‘myDefaultLocation’, the devices gallery location ‘myFolderLocation, and the combining both these locations using ’myScreenshotLocation’. Next we need to check if the specified folder in the gallery section already exists using an if statement that declares ‘!System.IO.Directory.Exists()’ and passing in ‘myFolderLocation’. If this directory doesn’t exist we create one using ‘CreatDirectory()’ and passing in ‘myFolderLocation’. Now we capture the screen shot using ‘ScreenCapture.CaptureScreenshot()’ and store the file as ‘myFileName’, and wait for 1 second for theimage to be captured. We then set back the UI using ‘ARManager.imageDone()’ and move the file to its proper location using ‘System.IO.File.Move()’ and passing in the location of the gallery file ‘myDefaultLocation’ and the screenhot location ‘myScreenshotLocation.

Next, we have to refresh the photo gallery for the users device for the image to show up in the gallery. We create an ‘classPlayer’ which sets unity to access the android filing system in ‘C#’, the ‘objActivity’ is set to th ecurrent activity to control the Android background processes. The ‘classUri’is set to the current android device class, and we set an object intent to ‘objIntent’ that specifies the refresh of the correct ‘myScreenshotLocation’. Finally, all these set up objects come to gether and the current activity ‘objActivity’ calls the ‘objIntent’ which will run the android os process of refreshing the users image gallery with the newly taken image.

2.4. Graphical User Interface (GUI)

All the graphical user Interface elements logic is handled by a ‘manager’ scripts. I have created a ‘manger’ script for each scene such as ‘MenuManager.cs’, ‘GameManger.cs’, ‘StoreManager.cs’ and ‘ARManager.cs’. These Manger scripts usually handle the graphical user interface elements of a scene, like when elements should be seen and actions that allow other user interface element to be active.

2.4.1. Main Menu

(Main Menu)

The ‘Main Menu’ is the portal that allows the user to play the ‘Main Game’, access the ‘Store’ to change the car appearance, and is also used access the ‘AR Car’ scene, used to display the player's customised car with Augmented Reality functionality.

2.4.2. Main Game

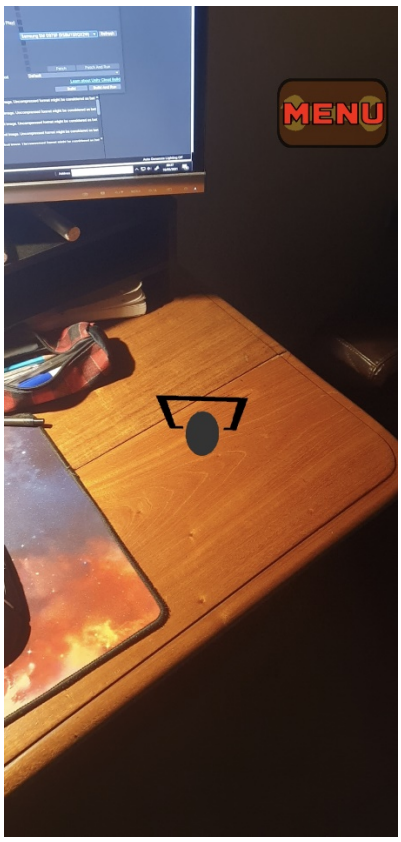
(Main Game)

The User Interface I have chosen for the Main game is very minimal and will rely more on player intuition than a complex and cluttered User Interface blocking the view from the gameplay. The player's left and right movement will be handled by the device's rotation. There will be three buttons tucked away in the side corners of the screen. First button is for pausing that game that will mainly allow the user to quit to the ‘Main Menu’ located in the top left. The second button is to fire the weapon located in the bottom right. The third button is the ‘Slow

Time' button which access the feature of slowing down the in-game time, located in the bottom left corner.

2.4.3. AR Car

The user interface for the 'AR Car' feature come in two phases. The first phase is when the system is trying to recognise flat positions, and the user is trying to figure out the best way to display the car using the 'placement indicator'. There is also a navigation button that will access the other scene of the game such as 'Main Menu' and 'Store'. The second phase is when the user has placed the car in the correct position. This phase allows the use to change the sizing of the placed object with three fire bolt buttons ladled 'S' small, 'M' medium, and 'L' for large. The capture button is on the bottom of the screen this will hide all the current user interface and place a logo to capture an image. The 'Reset' button is also in the top right, this will remove the placed 'AR Car' object and return the user interface to the first stage. The player can also rotate the vehicle to view different angles by touch swiping on the vehicle.



(Phase 1: Car Placement) **(Phase 2: Car Framing)** **(Image Captured)**

2.4.4. Store

The 'Store' scene has the most complex user interface. There are two tab buttons labelled 'Cosmetics' and 'Upgrades' which will open sub category buttons depending on the option selected. Each of these subcategory buttons will set active a set of buttons that are related to the subcategory that will make changes to the player vehicle. The player can also rotate the vehicle to view different angles by touch swiping on the vehicle.

(Store: Cosmetics > Bodys > Button Display)

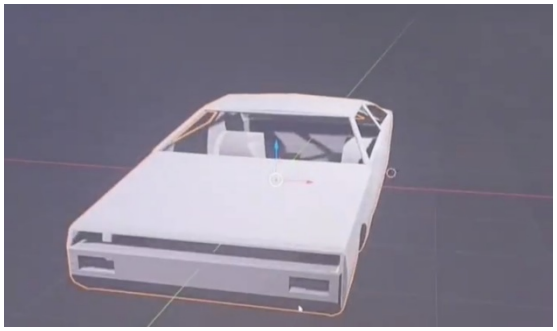
2.5. Creative Visualisation

A huge part of a player's interactions with the game is based off the visual elements of the project. This is essential for the user to be able to clearly identify the information displayed on the screen, and in the games marketing to attract users to try out the project.

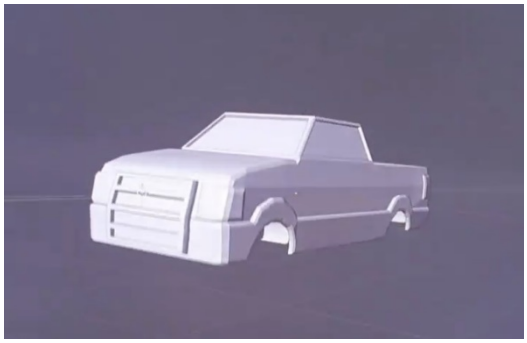
My personal interest in visual creating art, lead me to tackle the task of creating all the visual elements on display in 'Roadster'. I used a 3d modelling software 'Blender', for all the in-game art assets the user will interact with, also the using features in 'Blender', combined 'Unity Shaders' to create the texturing of these assets. Adobe Photoshop was used to create my own personal GUI elements. These elements clearly indicate actions to the user as well as matching the art style of the rest of the elements.

2.5.1. 3D Model Assets

Blender was a big part of my workflow when creating assets for the game 'Roadster run'. I wanted to make a wide range of cars for the player to choose from to edit to their liking, as well as have intriguing and interesting looking enemies to face off against.



When designing a model, I usually will gather reference images image for a frontal and side view of the target car I was inspired from. I set these as plane is the 'Blender' 3D scene and mould a cube until it looks similar to the reference images.



I then continue to add other objects to the car such as lights, bumpers, and windows.

This model I have created is painted with texture I think will suit all the added elements. The body of the car is given a base paint that will be replace later in the unity editor.

(Blender work environment)

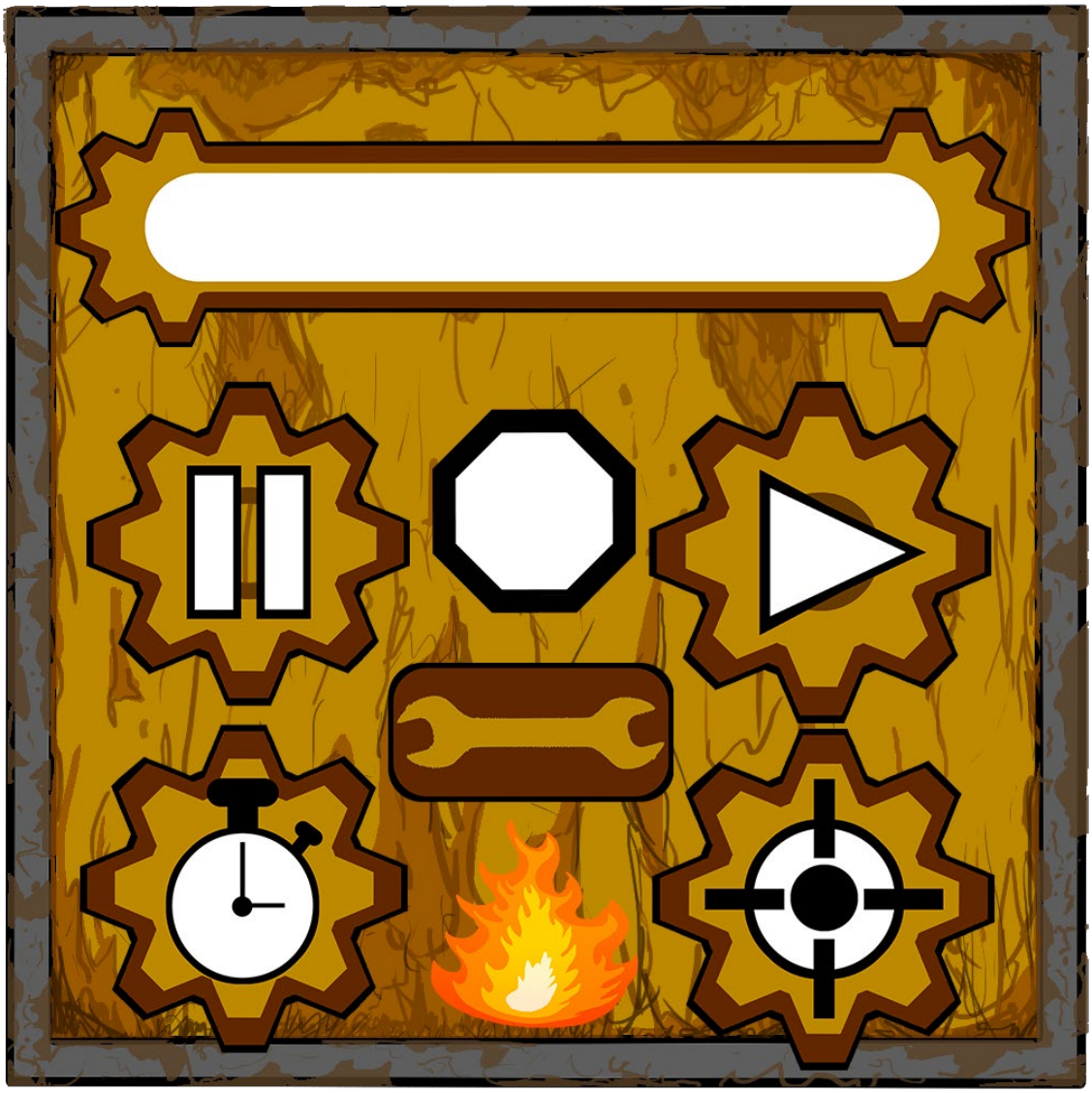
Then a newly created model is brought into another blender project to compare sizes of all the created element to reduce fixing models within Unity. When they are the correct size they are exported into '.fbx' files, and brought into the 'Unity' project in the 'Import FBX' folder

2.5.2. Unity Assets

When bringing blender objects to the 'Unity' editor the materials need to be replaced to make them look nicer in the game engine. I usually used the 'Legacy Shaders' which gave a nice shine to a lot of the object in the game. This was implemented after the midpoint presentation where you can see a clear change in graphics processing within the game.

2.5.3. GUI Imagery

I created all the graphical user interface element in ‘Adobe Photoshop’. Once I selected a theme for a rusty kind of wasteland, I just created object that would suit this atmosphere I was trying to create.



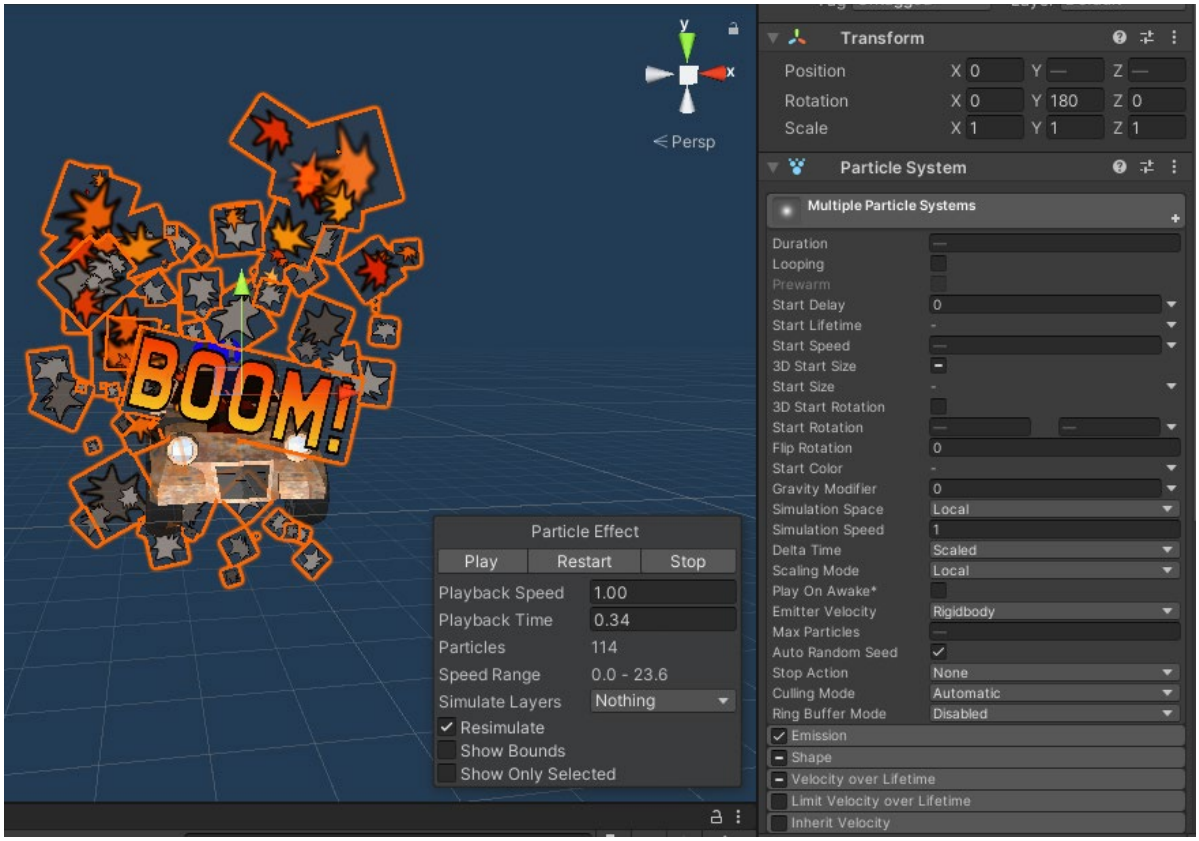
(GUI Elements)

2.5.4. Particle Effects

I used ‘Unity’s particle system to create various particle effects for better game feel and a game world that is more grounded.

(Unitys Particle System, Smoke Effect)

The effects I created and used was the explosion effect and the smoke effect. 'Unity's particle system allowed me to change 2d assets into materials that can be emitted from a certain object point, the system is very robust and has many settings for tweaking.



(Unitys Particle System, Explosion Effect)

2.6. Testing

I have set up a 'Google Play' application that is unfortunately still in review. This did give me the functionality to set up 'Internal testers' to download and test the application. I asked three of my close friends to download the app and see if they can navigate and play the game at a smooth frame rate so I could test on multiple devices.

2.6.1. Unit Testing

I created a function with the 'CarMovement.cs' script called 'Movebuttons()', this allowed me to test the 'Game' scene and it function as they were being created as I could use my left and right arrow keys to move the car to dodge obstacles.

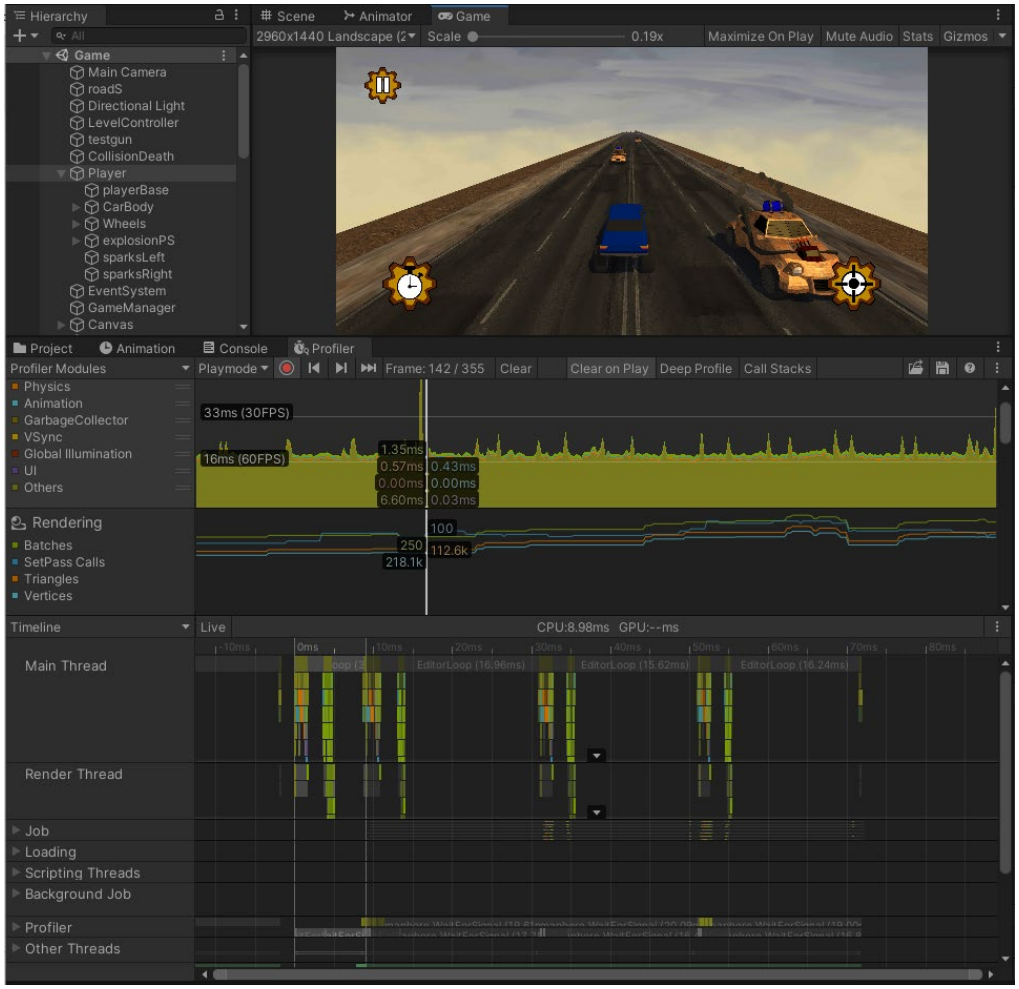
The 'Store' could also be tested on the development computer by clicking the various buttons.

2.6.2. Performance Testing

I would be able to install the build of the game on my personal mobile device through 'Unity' Build settings. This allowed me to test out the mobile specific features of the application such as the 'AR Car' feature. As my script for the augmented reality did not work with computer connected video devices. I was able to test the AR features and the save to gallery photo capture feature.

2.7. Evaluation

Evaluation of the testing techniques employed on target devices was done my assessing the gameplay and the frame rate of each of the devices when running the game.



(Unity Profiler)

Issues of these nature were resolved using the 'Unity Profiler'. This feature allows the developer to see the frame rate of the of the project when running to identify bottle necks in the code or certain graphics processes that are intensive on the system.

2.8. Google Play Store

I have created a 'Store page' on the google play store, and have been using the Internal test tract since February 2021. This internal test tract allows me to share my new update with a specified list of internal testers.

I have moved to the Open testing tract to allow lecturers or people who view this document to download the app on their android device. Unfortunately, Google hare having trouble and are expecting higher than usually review time as of writing this. But it should be accessible within the week of 17th of May 2021.

Link for download:

https://play.google.com/store/apps/details?id=com.Hornerific.RoadsterRun

If there issues with this link, I can personally add an email to the list of internal testers to access the unreviewed version of 'Roadster Run'.

Please sent me your google account email at:

x16318711@student.ncirl.ie

3.0 Conclusions

I enjoyed developing this project. I find all the features engaging and complement each other well. I challenged myself in an already familiar framework of the 'Unity Engine' and now feel more confident when developing games with 'Unity', coding with the language 'C#', and developing Android applications.

I think I have underestimated the time it takes to create a project of this scale, while also managing other projects for college. When planning for 'Roadster Run', I promised a lot of features that could not be delivered if I wanted all the current features developed to be polished and work smooth with the rest of the systems.

Although I did not deliver on some content that was planned, I hope it is clear to anyone who views this 'Alpha' version of the game that the framework and systems design is already completed. I think 'Roadster Run' is in a very good position to be ready to scale up and meet the missing these features mentioned in the project plan, from more options in the store to customise the player controller, as well as other enemies and other player abilities to be added.

One part of the project I wish I could have done more for is the main game. I feel the complexity and framework are there to add a more engaging combat loop as well as more sophisticated enemies. I felt as the game grew in complexity the more 'game design' work that needed to be done. This work is hard to display with a technical report. A script with 100 lines of 'C#' could have gone through 5 different iterations to get the right combination to get a satisfying game feel, which takes up a lot of time that could be spent working on core functionality.

This project has also made me realise that projects of this scale usually get developed by more than one person or at least with some any help. Through my development process I have created all my own 'C#' scripts, 3d models, 2d art assets, particle systems and any other tweaking and editing that is done within the project. The 'Unity' asset store is commonly used by other developers, from big companies and small indie teams using the 'Unity' engine. I think I will be exploring resources that could make my workload less when creating future projects and making the development process easier for me.

In the end, I am happy with the outcome of this project I find it really satisfying how all the features in the project compliment each other. I like how I can make a personalised car system and have it display in both the game and the Augmented Reality Features. I think this Alpha version of 'Roadster Run' show the potential of the framework already in place.

4.0 Further Development or Research

I think I will find it very easy to scale up the Store feature to include the choice of car wheels and various gameplay tweak upgrades. A lot of variable in the scripts of the main game have be purposely set up to be able to receive input from a save file.

I really wanted the player to be able to select enemy tagged objects in the main game and for the weaponry to lock on to these enemy to shoot. I have done research on how lock on turrets in other games are created and will be implementing this feature very soon.

I wanted more enemy variation withing the game as the 'Explo' enemies are not very challenging to the player. I already have the art assets created for a 'Rocketere' enemy mentioned before. I would also like a scoring system for the player based off the distance travelled and the number of enemies defeated. I want this scoring to be converted into a currency to unlock the items within the 'Store' feature.

I will continue to develop 'Roadster Run', with my newfound time after graduating. My estimated time for a fully released game is this Autumn.

5.0 References

Blender to Unity: How to Import Blender Models in Unity (2020) *All3DP*. Available at: https://all3dp.com/2/blender-to-unity-how-to-import-blender-models-in-unity/ (Accessed: 16 May 2021).

Build new augmented reality experiences that seamlessly blend the digital and physical worlds (no date) *Google Developers*. Available at: https://developers.google.com/ar (Accessed: 16 May 2021).

Foundation, B. (no date) 'blender.org - Home of the Blender project - Free and Open 3D Creation Software', *blender.org*. Available at: https://www.blender.org/ (Accessed: 16 May 2021).

Technologies, U. (no date a) *Unity - Manual: Audio Source*. Available at: https://docs.unity3d.com/Manual/class-AudioSource.html (Accessed: 16 May 2021).

Technologies, U. (no date b) *Unity - Manual: Wheel Collider*. Available at: https://docs.unity3d.com/Manual/class-WheelCollider.html (Accessed: 16 May 2021).

Technologies, U. (no date c) *Unity - Scripting API: Collider*. Available at: https://docs.unity3d.com/ScriptReference/Collider.html (Accessed: 16 May 2021).

Technologies, U. (no date d) *Unity - Scripting API: ParticleSystem*. Available at: https://docs.unity3d.com/ScriptReference/ParticleSystem.html (Accessed: 16 May 2021).

Technologies, U. (no date e) *Unity - Scripting API: Rigidbody*. Available at: https://docs.unity3d.com/ScriptReference/Rigidbody.html (Accessed: 16 May 2021).

Technologies, U. (no date f) *Unity - Scripting API: ScreenCapture.CaptureScreenshot*. Available at: https://docs.unity3d.com/ScriptReference/ScreenCapture.CaptureScreenshot.html (Accessed: 16 May 2021).

6.0 Appendices

6.1. Project Plan

National College of Ireland

Project Proposal

Roadster

01/11/2020

BSc (Honours) in Computing

Software Development

2020/2021

Jake Horner

16318711

x16319811@student.ncirl.ie

Contents

1.0 Objectives

The main objective of the project 'Roadster' is to create a more modern take on the endless runner style game. Technologies being used are 3d graphics created using *Blender*, UI elements using *Adobe Photoshop*, and *Unity Engine* with *C#* scripts develop and program the game. I plan to have this widely available for all types of people by aiming for release on Android Devices.

The game Roadster is an endless runner, car combat style game, with a range of customizable features such as cosmetics and car upgrades. The player will earn currency while playing the main game to spend on the in-game shop for these upgrades. Augmented reality features will be implemented using the camera of the mobile device to display the player's customizable car. This acts as an incentive for the player to buy cosmetic upgrades, as well as a way to provide social media marketing through the sharing of images of the player's car, with the 'Roadster' branding overlay. With these features, I have broken down my project into 3 key features: Main Game, Upgrade shop, AR (Augmented Reality) Display.

Main Game Objectives:

- An intuitive and enjoyable player controller.
- Fun and diverse set of tool to interact with the world and its AI members
- Intelligent and challenging AI enemies to interact with.
- The procedural based road system that interacts with the states of the game.
- Resource management of player Health and Fuel.

Upgrade Shop Objectives:

- Upgrades to the player's car and current tools.
- New Upgrades to equip the player Controller.
- Cosmetic upgrades to the car using various assets for body and wheel.

AR Display Objectives:

- Display the player's Customised car through the camera.
- Have Roadster Branding Overlay.
- Ability to Capture photos and store on devices.

Visual Design Objectives:

- Create all art assets used in-game using *Blender*.
- Create all User Interfaces and HUD elements in photoshop.
- Particle effects using Unity's particle system.

2.0 Background

Due to the current situation with COVID-19, many people have had no choice but to stay inside for extended periods of time. During Lockdowns, many people have turned to games to keep occupied for these difficult times. This has caused a huge increase in gaming popularity, and the industry is flourishing as a result. I think the gaming industry is more important than ever to allow people to connect with their friends over a game or to give experiences to talk about with other players.

The reason I would like to pursue this project is that I would be very keen on joining the gaming industry after I have completed my degree. Dublin is home to many game development studios such as the popular mobile developers 'DIGIT', Indie developers 'Pewter Game Studios', Virtual Reality developers 'War Ducks', and up and coming developers 'Vela Games'. Ireland has a very good technology sector and the gaming industry has a strong presence too, with some of the bigger studios such as 'Activision' and 'Riot' having servers based in Dublin. It would be one of my lifelong dreams to work for such a creative entertainment industry. I believe video games are the next new expression of art with a way to engage a viewer like no other. It is a perfect mix of computer science and artistic vision that can become engaging for any kind of person if they find a genre they like.

I've decided to develop 'Roadster' in 3D, I have never made a game with 3d mechanics, and I have recently been developing my skills as a 3d artist. This will challenge me both technically and artistically over the course of my development. The Player will be challenged to drive down a wasteland highway in a customisable car, trying to survive managing fuel and health. Combatting with various AI controlled enemies, with vehicle mounted guns. The Player will generate a score based on their performance in this main game loop, which will be converted into a currency that can be used to purchase upgrades and cosmetics for your car. I have taught about the purposes for the cosmetic purchases, why would the player be enticed to these options if they can improve their gameplay performance with upgrades. My Augmented reality feature will allow the player to display their personally customised car in the real world using their mobile camera device. This feature will also allow for user generated advertising through social media through the posting of images.

For Development I will be using 'Unity Engine', as I am quite comfortable with this software and believe it holds many features and helpful resources to aid me in my development cycle.

I think long term strategies for a project like this could be quite profitable, if there are internal structures built from the beginning that will support a monetization model. These plans will be implemented after my time with NCI, and therefore will be only commenced after the deadline in May when I have a fully working and advertisable product. I plan to create this currency used in the in-game shop to be purchasable through Google Play, that will allow the player to bypass the time required in the main game, to have enough currency to purchase any upgrades and cosmetics they would like. Also, rewarded Advertisement watching would be implemented, that would multiply the players score in the main game. I will use Google Play ads and various other social media advertising platforms to spread word of the product. This monetization scheme will allow passive income from various players that are only willing to watch ads, as well as a more direct source of income from currency purchasing customers.

3.0 Technical Approach

Brief description of the approach to be followed (Max. 1 Page), Research, literature review, requirements capture, implementation etc…

In preparation for my Project, I have done research in each of the main core features of my design. As well as having Previous Knowledge of Unity and its various tools.

For the Main game I researched level chunking which allows level pieces to be instantiated in the world when a parameter is met. This also allows me to Destroy Level pieces to reduce strain on the loaded in level as a whole as it only will be dealing with a certain amount of pieces at one time. Aswell, research into physics based car systems that are commonly used in Unity such as 'Wheel Collider' and 'RigidBody'.

The Store will be mainly UI based, which I am familiar with from other projects. I have researched into creating cars in an multiple Object based system that will allow me to set them in a class to load the right objects in all instances of the Car.

ARCore is a plugin I have been researching for the Augmented Reality car display. This is a powerful system created by google that can allow environment reactive 3d models and animation to be displayed in the real world through a device with a camera.

I have taught myself 'Adobe Photoshop' and I am currently at an Intermediate level of 3d modeling and animations with 'Blender'. I have some experience with Unity's Particle system features but will be developing my skill further to elevate my products visuals.

4.0 Special Resources Required

I will be making the game in the 'Unity Engine', which is available for free, only requirement is that a small splash screen plays displaying the engine at the start of the game launch.

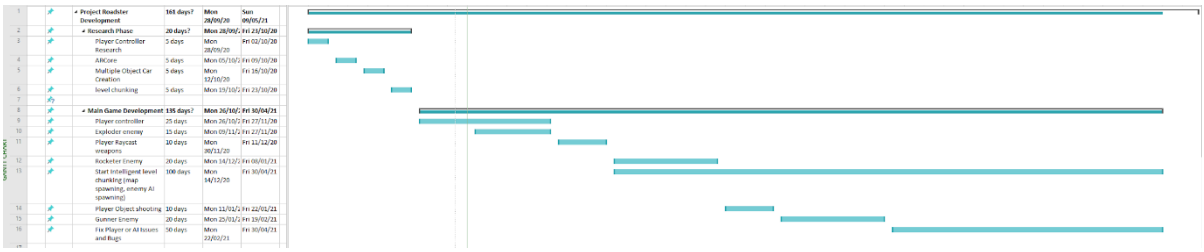
I will also be using 'Blender' to create 3d art assets, which is also free software. I have acquired a licence for Adobe's Suite of programs such as Photoshop and Illustrator to create User Interface and other elements.

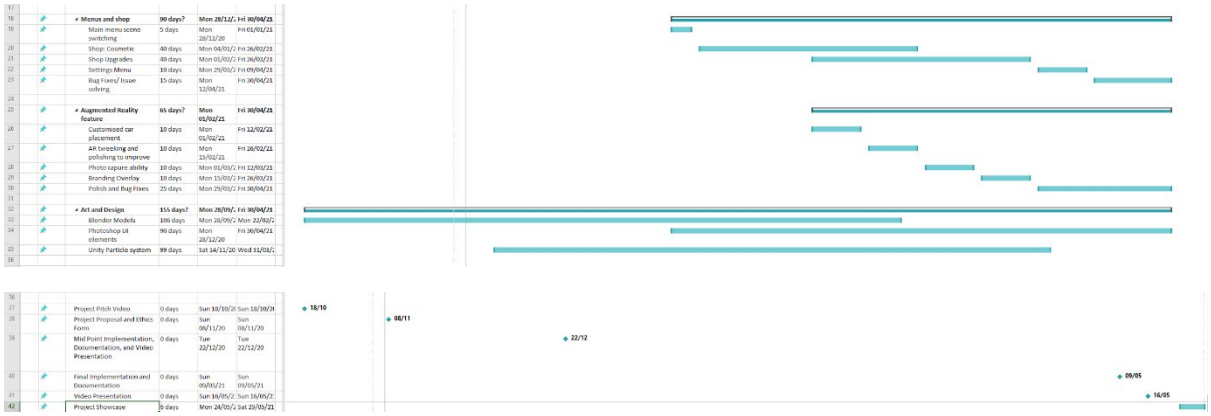
ARCore is a free Plugin developed by Google that will allow me to implement Augmented Reality features.

I will make use of 'Unity Documentation', 'ARCore' website's Development section, and 'Youtube' resources to gather the knowledge required to make 'Roadster'.

5.0 Project Plan

Gantt chart using Microsoft Project with details on implementation steps and timelines.





Here we have a closer look at the tasks involved:

1		**Project Roadster Development**	155days?	09/28/2020	04/30/2021
2		**Research Phase**	20days?	09/28/2020	10/23/2020
3		Player Controller Research	5days	09/28/2020	10/02/2020
4		ARCore	5days	10/05/2020	10/09/2020
5		Multiple Object Car Creation	5days	10/12/2020	10/16/2020
6		level chunking	5days	10/19/2020	10/23/2020
7		**Main Game Development**	135days?	10/26/2020	04/30/2021
8		Player controller	33days	10/26/2020	11/27/2020
9		Exploder enemy	19days	11/09/2020	11/27/2020
10		Player Raycast weapons	12days	11/30/2020	12/11/2020
11		Rocketer Enemy	26days	12/14/2020	01/08/2021
12		Start Intelligent level chunking (map spawning, enemy AI s	138days	12/14/2020	04/30/2021
13		Player Object shooting	12days	01/11/2021	01/22/2021
14		Gunner Enemy	26days	01/25/2021	02/19/2021
15		Fix Player or AI Issues and Bugs	68days	02/22/2021	04/30/2021
16		**Menus and shop**	90days?	12/28/2020	04/30/2021
17		Main menu scene switching	5days	12/28/2020	01/01/2021
18		Shop: Cosmetic	54days	01/04/2021	02/26/2021
19		Shop Upgrades	54days	02/01/2021	03/26/2021
20		Settings Menu	12days	03/29/2021	04/09/2021
21		Bug Fixes/ Issue solving	19days	04/12/2021	04/30/2021
22		**Augmented Reality feature**	65days?	02/01/2021	04/30/2021
23		Customised car placement	12days	02/01/2021	02/12/2021
24		AR tweeking and polishing to improve	12days	02/15/2021	02/26/2021
25		Photo capure ability	12days	03/01/2021	03/12/2021
26		Branding Overlay	12days	03/15/2021	03/26/2021
27		Polish and Bug Fixes	33days	03/29/2021	04/30/2021
28		**Art and Design**	155days?	09/28/2020	04/30/2021
29		Blender Models	148days	09/28/2020	02/22/2021
30		Photoshop UI elements	124days	12/28/2020	04/30/2021
31		Unity Particle system	138days	11/14/2020	03/31/2021
32	Project Pitch Video		1day	10/18/2020	10/18/2020
33	Project Proposal and Ethics Form		1day	11/08/2020	11/08/2020
34	Mid Point Implementation, Documentation, and Video Presentation		1day	12/22/2020	12/22/2020
35	Final Implementation and Documentation		1day	05/09/2021	05/09/2021
36	Video Presentation		1day	05/16/2021	05/16/2021
37	Project Showcase		6days	05/24/2021	05/29/2021

You can download the Project file here:

https://drive.google.com/file/d/1vWEYcdnGJpUDDocyA6n8jK2h3hFL7ARr/view?usp=sharing

6.0 Technical Details

The programming language I am planning to use is 'C#', which is one of the languages on offer from 'Unity Engine' to interact with their environment. The IDE in which I will write my scripts for the project is 'Microsoft Visual Studio'.

A plugin I will be using to aid with development on the Augmented Reality is ‘ARCore’ developed by google.

7.0 Evaluation

For Evaluation of the game and its features, I have a small group of friends that tested as internal testers for my previous project that was released on the Google Play store. I will be using Google Developer Console to push APKs to my internal testers group where I will be able to receive comments on and bugs, and performance issues they are having

For Performance Issues ‘Unity’ provides a great feature called ‘Unity Profiler’. This will help with what is causing issues with low framerate or high memory usage. Optimising a mobile game to perform efficiently is very important because of the range of devices on over that can download the game.

6.2. Reflective Journals

Reflective Journal: October

As part of my final year of study at NCI, I had to present a pitch for the idea of my final year project. The idea for this project is ‘Roadster’, an endless runner car combat game. I decided on it because it incorporates all that I have learned during my time here in college whilst still challenging me. The most important aspect of this project is that is one that I am genuinely interested in. Throughout my time in college, I have studied and experimented with game development with tools such as the 'Unity Engine'. Game design is something I am passionate about as it allows me to combine my creative interests such as animation with the skills that I have learned.

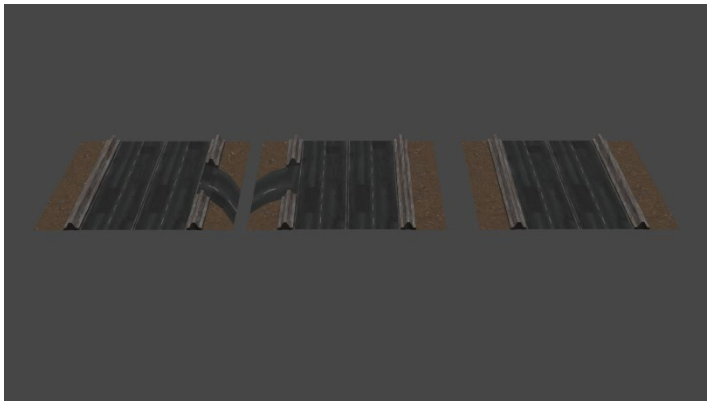
My original intention was a game where the player would drive a car down a long road while fighting different types of other vehicles which I wrote down over summer. This design seemed to be an idea that kept popping up in my head when thinking about my final year project. However, I decided to write a more in-depth game design document and came up with a nice feature that allowed for the inclusion of augmented reality to display the player's customized car. This allowed me to combine my artistic hobbies such as ‘Blender’ into my project, as I would need models to customize the car. I decided this game would aim for release on Android phones as I already have experience with Google Developer Console.

‘Roadster’ is the development name, and the three main features/pillars of the game are:

1. Endless ‘Runner Style’ main game.
2. Shop to purchase upgrades and car customization.
3. Augmented reality car placement

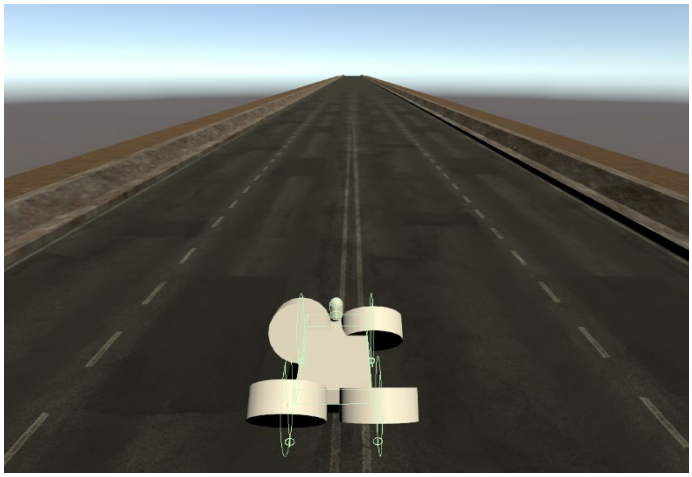
I researched how I can implement each of these key features.

For endless runner style games, the map needs to be generated in chunks. I created three road pieces that can be used for this purpose. I wrote a script that when started will generate thirty pieces ahead of the player so that it looks like an endless road. I also wrote a script that would delete level pieces to reduce rendered objects in the scene, due to the player constantly moving forward.



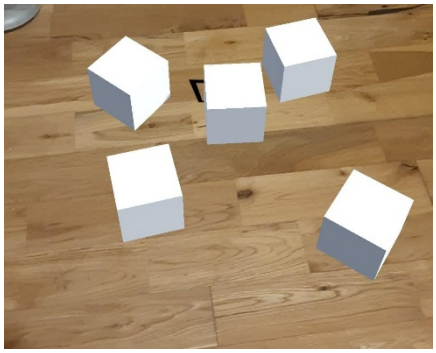
3d models of roads In made in blender (3d software)

I researched the ways that cars have been made by others using the 'Unity Engine'. I wanted the car to feel somewhat realistic by implementing physics tools that were provided by unity. I learned about collider objects called 'wheel colliders' that when force is applied to them they rotate and cause friction against the box collider to simulate a more realistic movement for the car. I have used 'Ridgidbodys' in the past which does not cause the same effect, as it moves a static object as a whole. I made two cars, one using an asset pack that I found on the Unity store which acted as my tester. The other was blocking it out in simple shape objects to be replaced after I create the art assets. The test scripts that I wrote will apply the force to the wheel to make them spin and to make them turn a certain amount of degrees to allow for turning.



The gameplay of test car made with shapes (Wheel Colliders)

I learned that there were many plugins I could use to achieve augmented reality on a phone. “ARCore” seemed to be a very good option as it is made by Google and had a lot of Documentation. I jumped right in for research and I made a phone app that would spawn cubes in my environment. It works by recognizing the flat surfaces and placing an object in a static position.



ARCore Test running on a phone made with Unity

For my first month of developing this project, I think I have gained a good grasp of the technologies and concepts that I will be using throughout the year to aid me in the creation of this software project. For my next month, I want to work on improving the player controller and importing models to this block layout I have created as the car. I would also like to get started on the AI of the enemies to complete the main game playable prototype.

Reflective Journal: November

For the second month, in my development of the Project ‘Roadster’, I decided to focus on the player controller as it is very important to the overall feel of the game. As well as starting on 3d modeling I would have to do for the various car customization in Blender.

Player Controller

The system I had researched for the car movement was using wheel colliders in unity. I had one issue with turning, as I was using wheel colliders being rotated to angle the car in its direction it wasn’t very precise. This caused many issues when playing the game, The user would have to steer the opposite way when they wanted to stop turning and straighten the player car as they were already angled in this direction. This especially affects the wheels when going at high speeds, the car would ofter flip due to a sudden turn, or the car would become uncontrollable when trying to straighten up the vehicle.

First I added a speed limiter, this was achieved by using a theorem that calculates speed based on the radius of a wheel and the rpm(rotations per minute), ‘Unity’ stores these two variables

when you add a wheel collider object. The theorem is: 2*π*Radius*rpm, which I converted into C# scripting with:

current speed = 2 * 22 / 7 * backLeftW.radius * backLeftW.rpm * 60 / 1000;

I stored the current speed and compared it to a variable called max speed, when the current speed exceeded the max speed value I stoped applying force/rotations to the wheel.

The Second issue was the car turning not being very precise, this made me scrap the turning of the wheels to move left to right. I decided on a Rigidbody method where I would move the car using a precise force left and right. This cause an issue of the car wanting to roll over as I was applying this movement, this was easily fixed using Rigidbody Constraints that had the ability to freeze rotations and movent along a certain axis. I also implemented this into the car's basic movement to make the car always drive perfectly forward.

Car Customisation Precursor

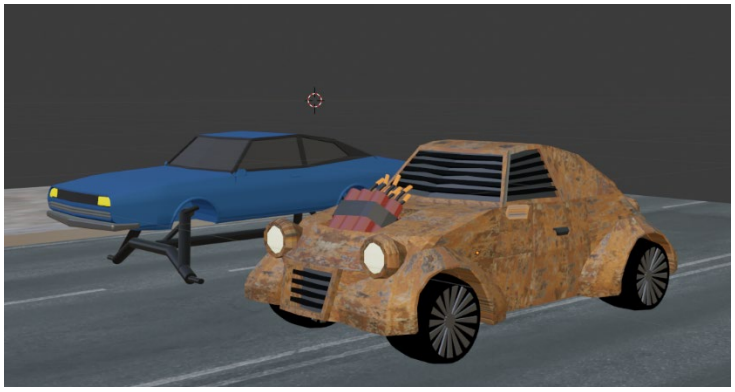
I decided that in order to save a lot of fiddling around with placement settings in the future when handling the car customization. I would put in a Bar rig base that can commonly be seen in 'Monster Trucks', This would allow me to always put the wheels and car body in the same place as the rig would remain consistent in all different car customizable setups.

Player Car with Rig Base (Unity)

Blender, and Import to Unity

Although I have previously taught myself enough of this software to be proficient, to create the art I need is a big undertaking but one I quite enjoy. Blender works by creating shapes with different vertices in 3D space, I manipulate these shapes using extrusion, movement, rotations, beveling, and many other techniques I have acquired. Something that was completely new to me was the importing of 3d made assets into the 'Unity' game engine. This caused some issues with the first model I had created as it had holes in the mesh when importing into 'Unity. The second model which I have created seems to be perfect I assume this issue was something to do with double vertices or unconnected faces within the first mesh. There also was an issue with the 3 axes (X, Y, and Z) when converting to 'Unity'. I fixed this issue with a plugin I found online that would allow a check box in on the FBX export file that would flip the axis to match the 'Unity' set up.

Two Personally created model in Unity (Blue hasProblem)



Two cars in Blender (No Problems)

Reflective Journal: December

For the third month, in my development of the Project ‘Roadster’, I was preparing for the mid-point project presentation mobile build. This for me meant having a playable prototype on a mobile phone and adding menu screens to change between the two already prototyped features.

Menus

Unity has some really helpful tools to aid with an easy and impressive looking user interface. Firstly, I had to make a canvas object that covers the edges of the camera viewport object, I also made the canvas scale to different camera sizes to aid with multiple mobile different device screens. I added three buttons one for each of the features, Play for the ‘Main Game’, Store for the ‘Cosmetic and Upgrades Store’, and AR Car for the ‘Augmented Reality Car Display’. I added a script name Mainmenu manager that will handle all logic from this screen and allowed the ‘Play’ and the ‘AR Car’button to lead to their respective unity scenes. I also added a title

and moved around some previously made assets to make the main screen more engaging.

I then had to make a pause screen to exit the 'Main Game', back to the 'Main Menu'. I achieved this by adding another canvas and attaching it to the camera in the game. There is a pause symbol button on the top left side of the screen. When pressed the pause button will hide and be replaced by a play symbol button, there is also a button centered labeled 'Go to Menu'. So the Pause game can be added to later I have set up the necessary features such as setting the time scale to 0, so the game doesn't continue playing when the pause button is pressed. Likewise when the play button is pressed the 'Go to Menu' is hidden the pause button replies to the play button, with the time scale now being set back to 1.

Player Controller

As I wanted to have the tilt controls working for my mid-point presentation, I worked on making those rigid body forces that move the player car left and right assigned to the accelerometer of the mobile device. When the phone is tilted on the x-axis left, the values are positive and they are negative when the player's device is tilted right. I multiplied this output value by the handling that was involved in the previous script by the output accelerometer value, this allowed the car to move faster from side to side the more tilted the player's mobile device was.

Sound

As I made a speed limiter, my script is aware of the rpm of the player vehicle's wheels, I decided as I wanted more realistic feedback to the car speed with the sound of the engine. This is made in the 'Player Controller' script, where depending on the rpm of the wheel the engine noise will have a higher pitch as if the driver is changing gears.

I also added two-track to the game made by Lucas Bonzzi, one for the Main Menu and another for the Main Game. These are played on the start-up of the particular scene and loop when the song has ended.

This was achieved using Unity's Audio Listener and Unitys Audio source modules.

AR Car Display

I implemented a similar setup as the first-month (Reflective Journal: September) cube spawning prototype. This time I have replaced the object to be the player car model. This caused some issues with sizing, but now I am more comfortable with allowing the user to edit their model for display in the Augmented Reality feature.

Reflective Journal: January

For the fourth month of my development for the project 'Roadster', I had been learning more about 'Unity's' particle system, UI element creation, 3D model for 'Rocketeer Enemy',

shooting logic with ray casting methods, and Health system to be implemented to both the player and enemy AI.

Prototyping: Ray cast Shooting and Enemy Health

Ray casting is a common method used in game development, where an object will shoot out a ray to recognize other objects in the scene's 3D space. I use this system to create a prototype of a sniper weapon for the player car which will, in turn, allow me to test the health scripts. An object that follows the player car using a script called 'Weapons Logic' allows the weapon to follow the car as if it was a child element but will not be affected by the physics on the cars twists and turn from the suspension, which is important to make the weapon aiming as easy as possible. The 'Weapons Logic' script will also handle the changing of weapon types as development proceeds. The 'Sniper' Script handles shooting out a ray cast to recognize objects on the press of a button. If the ray cast is a positive hit a function is run on the receiving objects 'Health' script's 'taking damage' with the sniper's damage input. This damage input is taken away from the health of the object. I designed the 'Health' Script to be attachable to all objects, with each initial instantiation defining the amount of health for that particular object.

Unity Particle System: Engine Smoke & Explosion

Engine smoke is a particle system created using a 2d sprite asset that I created in 'Adobe Photoshop' and emitted in different variations using 'Unity's Particle System'. This particle system is then emitted from the areas on object models for an extra set of visual flair to the game. This Particle system is currently attached to each of the 'Explo' Enemies tails pipes. The sprite will emit from the pipes at different sizes and rotations and will eventually fade away the further it raises from its creation point.

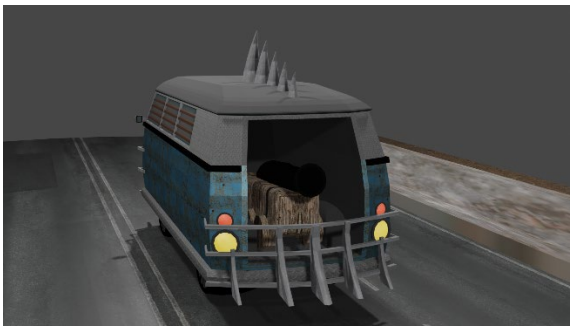
The explosion was a particle effect using more than 4 different particle systems, one text sprite, and 3 variations of an explosion cloud. The Text spite in the 'Boom' Particle system will change size suddenly through the course of its life to have a sudden cartoon-style effect, with a random rotation over a life-time to add more variation to each instance. 'Explosion Small' will emit 50 spites at different size and rotation variations, in either a red or orange color. 'Explosion Mid' and 'Explosion Big' works similarly with different spites, size, and rotation variation are applied but the color change from a dark color at the center of the explosion while fading to a light color as time passes since emission. I put a lot of variability within this particle system as it will be played in multiple instances when each car is destroyed are destroyed.

New Art Assets

I started to create new UI elements for the in-game canvas GUI using 'Photoshop'. The two currently displayed are for shooting and time slow.

‘Shooting’ ‘Time Slow’

I also created a new 3d model for the new Enemy ai the ‘Rocketeer’ using ‘Blender’



Reflective Journal: February

For the fifth month of my development for the project ‘Roadster’, I implemented Player health and Collision Death. This all came together to provide a ‘Game Over’ animation and state when the player has a front-end collision with an enemy car. I have also implemented touch screen buttons for the ‘Slow Time’ mechanic and the prototype ‘Sniper’ gun. I have also done precursor work in ‘Blender 3D’ on creating a new object for the player car to swap between.

I have also updated my unity version to2019.4.18f1, this seems to destroy some of my apk building plugins such as ‘Gradle’. This prevented me from playing new changes on phones until I updated and had to do a lot of error fixing. Also, My Visual Studio 2017 stopped working alongside the unity project and have installed a better IDE closer to traditional ones ‘Visual Studio Code’. These two factors definitely hindered my progress in the project this month but I have now fixed all the errors and am back on track

Collison Death

I wanted the player to receive a game over screen when they had any front collisions with any of the enemy cars, but still be able to hit into vehicles without taking any damage on any other types of collisions. This is an object attached to the front of the Player car that contains a box collider and script to follow the player car. When the box collider ‘collision death’ would be hit by another collider object with the tag ‘enemy’ it would access the ‘Player Health’ script and find the health value, and call the taking Damage function passing in the collected health number.

Player Health

The ‘Player Health’ script works similar to ‘Health’ (attached to the enemies) with the taking damage function being called by a certain object when the player loses health. The difference is when the death animation is over it will call a ‘Game Over’ state from the ‘Game Manager’ script. The Game over function will allow me in the furniture to tally up the score and apply it to the plates wallet as currency for the ‘Store’ customization features.

Touch Screen Buttons (Slow Time & Shoot)

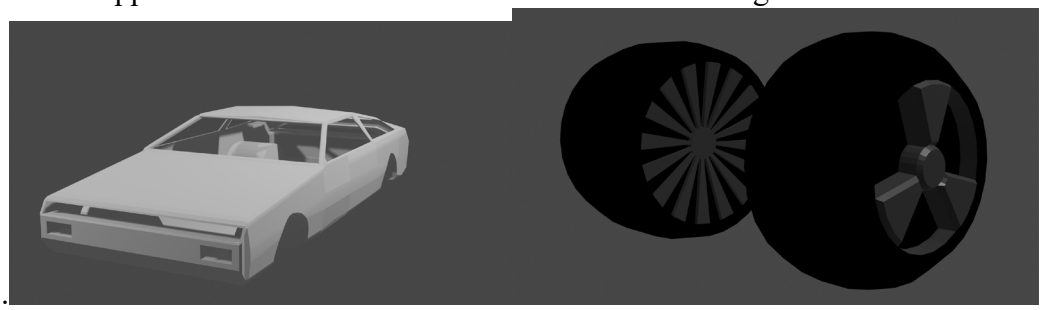
‘Slow Time’ is a new mechanic that allows the user to slow down time to make it easier to dodge income projectiles and enemy cars. The script will set time to 0.5 speed for 5 seconds and then it will be set back to normal 1.0 speed. The user will have to wait for a timer to use this feature again, as well as the user being able to cancel this feature during the slowed time state. This is the tied to a button in the button left of the screen (Figure 1).

The Prototype gun remains the same but can now be triggered with a button on the bottom left of the screen (Figure 1).

(Figure 1, Slow Time and Shoot Buttons)

New Player Models

I have created a car body that has no textures, as precursor work so material can be applied in unity using C# and applied in the Store menu. I have also been working on more variations of wheel types.



(Figure 2.1, new Player car option) **(Figure 2.2, two new wheels)**

Reflective Journal: March

For the month of March, I had a lot of precursor work necessary to flesh out each of the sections I am to complete for the final upload, as well as more game feel particle systems being added, and whole new design GUI to sell the feel of the rusty waste land of 'Roadster'.

Store Logic, Touch Spin, and Cosmetic Car Colors

I started to create the store scene which will allow the user to change the look and abilities of the player-controlled car for all of the features in the game. The first store tab I have started to implement is the 'Car Colors' section as well as the store tabbing system logic that will remain consistent among the whole 'Store' feature.

Each button set contains an Array List of button type objects. Each section button would loop though the specified button list set, and make the contents visible while, looping through all other button list sets and setting them invisible. This was achieved by two functions 'SetButtonSetVisible' and 'SetButtonSetInvisible' in the 'StoreLogic' script. The functions would accept a List<GameObject> type. This makes each of the button functions simpler to write, as I will only have to specify button sets and which ones to make visible based on what I would like the user to see.

Touch Spin allows the user to rotate their car using touch to get a better view of the model. This is used during the 'Store' Scene so the user can better view the changes being made to the vehicle, as well as during the 'Augmented Reality Car Display' so the user can position their car in their desired position.

Setting car colors was achieved using my 'CosmeticColors' script. This script defined a new Object called ColorType where it would define a name and new RGB color value. The script would create objects of this type with my specified RGB values and store them in a list. Based

on a button press specifying the set name of the color, a function ‘changecolor’ will loop through the specified objects and search for the name associated with the color type. Upon finding this name, the script would set the color value for the material on the player car to match the corresponding color type. Although this functionality works in the Unity editor, or in the app until the user closes the system. I need to add a save system so these colors and other values can be applied on the player car when the user opens the app.

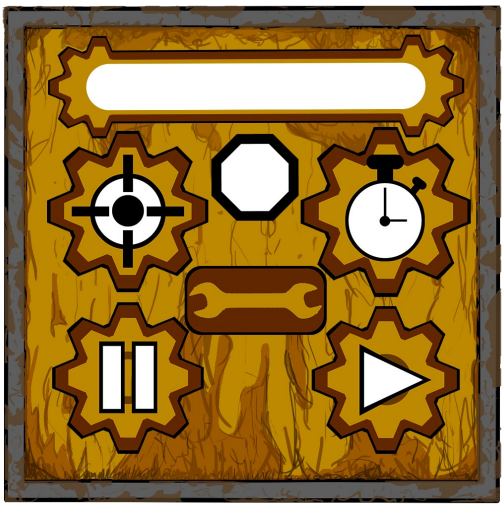
(Figure 1.1, Store Scene with Colors tab open)

Save System

I have started to use Binary formatters to save the state of the player cars visuals. So far I have managed to save data to a generated file, using my function ‘dataBuilder’, and a created Object called ‘Cosmeticdata’ which stores strings for ‘CarColor’, ‘CarBody’, and ‘CarWheel’. ‘CarBody’, and ‘CarWheel’ were added in preparation for future cosmetic features being implemented in April. The function of Load data is finding the right data, but I am running into issues when trying to use similar methods to ‘CosmeticColors’. This will be corrected by the creation of a new ‘DataLoader’ object in each of the necessary scenes.

New UI Assets

I wanted to create more of a personalised look to the menus as they were previously created assets for another project. I imported a new font that suits the style I was hoping to achieve as well as creating container elements for the new scene and main menu.



(Figure 2.1, Collection of new UI assets) **(Figure 2.2, Title screen with new assets)**

New Car Models

The 'DeLorean' model has been created and added to the game (figure 2.2). New model, 'ford truck' has been created.

(Figure 3, new ford truck model)

Reflective Journal: April

For the month of April, I have finalised the Binary formatting save system that will store the customised car for each feature of the game, as well as started on UI functionality for the 'AR Car' feature.

Save System of Cosmetic.data

Cosmetic data is the name of the file that is saved on to the user's device which stores three pieces of data, 'CarColor', 'CarBody', and 'Car Wheel'. So far I have been able to save data for 'CarColor' taking data from 'CosmeticColors.cs', and 'CarBody' taking from 'CosmeticCarBody.cs'. Functionality for CarWheel is not in place yet but all necessary structure and logic will be structured similarly to saving data for 'CarBody'.

(Figure 1.1, Store Scene with Colors tab open)

Cosmetic Colors

On Start of the Scene Cosmetic Colors will create an ArraylIst of 'ColorTypes', this is defined by name and a RGB color value. All objects of this Array are defined by me for my specific color choices. When the player selects an Item from the store the button will call a function

passing through the name of the color that the button is assigned to. The function ‘changeColor’ will loop through the ‘ColorTypes’ Arraylist and find the object with the matching name of the input color string. Upon finding the correct color the system will play a painting sound, as well as run ‘SaveSystem.dataBuilder()’ and set the material color set to the car's body as that defined RGB value . The function ‘dataBuilder’ will then proceed to take each defined value of ‘CarColor’, ‘CarBody’, and ‘Car Wheel’ from their respective scripts.

(Figure 1.1, Store Scene with Bodys tab open)

Cosmetic CarBody

On Start of the Scene ‘CosmeticCarBodys’ will create an ArraylIst of ‘BodyType’, this is defined by name and a GameObject. All objects of this Array are defined by me by naming the object and selecting the incense object of each of the car bodys. When the player selects an Item from the store the button will call a function passing through the name of the Car Body that the button is assigned to. The function ‘changeBody’ will loop through the ‘BodyTypes’ Arraylist to find the object with the matching name of the input car body string, for anyobject that do not match are set to invisible in the game’s scene . Upon finding the correct object the system will play a sound, as well as run ‘SaveSystem.dataBuilder()’ and set the gameobject as visible in the scene. The function ‘dataBuilder’ will then proceed to take each defined value of ‘CarColor’, ‘CarBody’, and ‘Car Wheel’ from their respective scripts.

Setting Car in each Scene

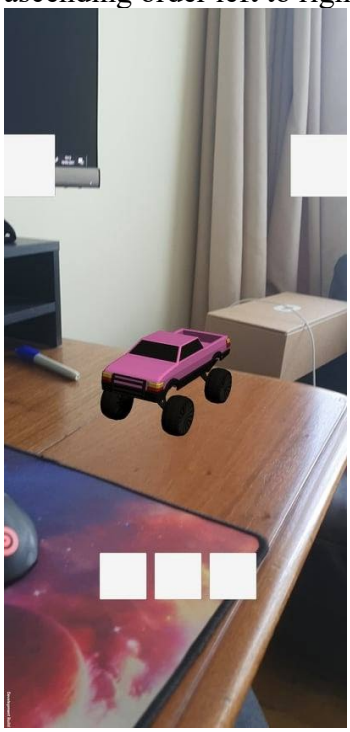
I created a Game Object called ‘Cosmetic Handler’, this object contains all the Cosmetic scripts contained in the Store Scene. On start up of each of these scripts they will run ‘SaveSystem.LoadData()’ and set the output to a variable called data. Each script will take their corresponding value (CosmeticColor will take data.CarColor), and pass the variable through a set function (CosmeticCarBody will run setBody(data.CarBody)).

These set functions work similar to their change function used when saving the data from the Store scene. But it doesn't play a sound or save the data again as this data can only be saved from the store scene when interacting with the button options. This is important as all the player vehicles displayed look the same while achieving different purposes.

AR Car functionality and UI

Firstly, I fixed the bug that allowed the user to spawn an infinite amount of cars and implemented a reset button that will delete the object and allow the user to place a new car (top right button). I have also allowed the user to return to the ‘Main Menu’ or the ‘Store’ to change

their car (top left button). The three buttons on the bottom appear when the car is placed, and allow the user to control the sizing of the car in the display, small, medium, and large in ascending order left to right.



(Figure 2.1, AR Car placeholder UI)

1.1. Other materials used

All scripts and art assets within the 'Roadster Run' project has been created by myself, Jake Horner. Two music tracks are included in the Alpha version for 'Roadster Run', 'Wasteland', and 'Cyberpunk Inspired' produced by Lukas Bonzzi. I have gained permission to use these tracks from Lukas (http://lukasbonzzi.com/).