OCR GCE A

COMPUTER SCIENCE PROJECT

H446-03

Name : <JAMES SHARROCK>

Candidate Number : <INSERT CANDIDATE NUMBER>

<North Halifax Grammar School> : <INSERT CENTRE NUMBER>

Title of Project : <CHICAGO COIN SPEEDWAY>

Final Product : Video File

H446-03 – PROJECT CONTENTS

TABLE OF CONTENTS

A. Analysis

Initial Ideas

I am making a replica/remaster of the 1969 arcade game, Chicago Coin Speedway. Made by Chicago Coin Machine Co, only 317 machines were ever made. Since then, the game has vanished into obscurity with only a handful of collectors finding the machines desirable. There are 32 known instances of working machines across the world that need to be regularly maintained with sought after spare parts. Arcade machines that are still for sale retail online across various sites at more than $1,000 and with considerable shipping fees making the game inaccessible. I believe I can make a game free to download by all.

Stakeholders

I would like to aim towards the arcade collectors societies and those interested in playing vintage games. The expectation of the game would be to provide the same mechanics as the original with similar graphics. In all cases possible I aim to create my game as close to the original as I can however with brighter colours and better sounds to improve on the game in a way that I believe the creators themselves would have wanted.

The target platform for the game to be presented on will be Mobile. The game can therefore be easily portable as it requires very low hardware to run on, will run smoothly and on a mobile system will make it easier to trend between users. On mobile, inputs can be simple and intuitive such as rotating your device the same way as you might a wheel and driving down the speedway or tapping the sides of your screen to determine which way you'd like to move. To keep the game exciting when a player achieves a high-score they will be able to enter their name on the leaderboards and compare where they stack up against other users allowing for friendly competition.

The nature of the graphics will be retro meaning there are no associations with real-life. There will be no language barriers in the game excluding text. Ages required to play the game will be low meaning it can qualify for a low age-rating. The game will be played on mobile as it is portable and easy to play. It doesn't require extensive hardware to run and controls can be changed from touchscreen to the use of arrow keys or WASD. This allows the target audiences ages to be anything from 3 and up maximising a target market.

A similar game to Chicago Coin Speedway is Sega Grand Prix (1969) with the same use of a spinning background and cars jostling with one another for position; the games are nearly identical and both fan-favourites with the arcade community. With their use of a rotating background, simple car models and tracking laps they have simple logic and therefore the logic when creating them should not be extremely complicated.

HOW THE PROBLEM CAN BE SOLVED BY COMPUTATIONAL METHODS

This problem can be easily solved by a combination of algorithms. The main challenges will be to design hitboxes and create algorithms that can figure out if hitboxes overlap or touch and penalise the user for crashing. However, the project will be able to be abstracted allowing for more efficient development by focusing on the problem. Decomposition can be used to split the problem into a number of functions and they can be developed in isolation and then tested how they interact with one another. The following examples explain this further:

THINKING ABSTRACTLY

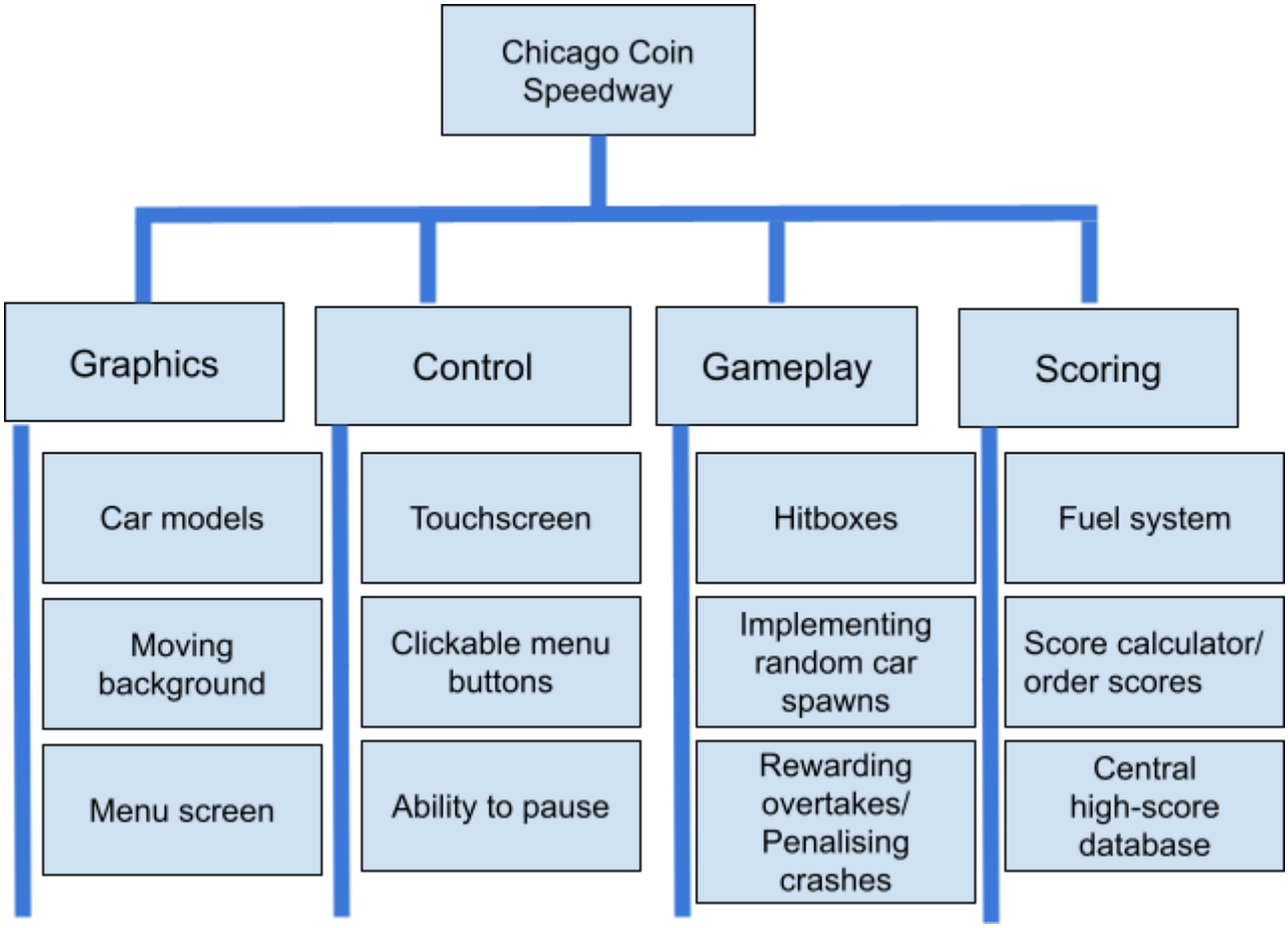
The key components of the game:

- Restrict the game to 2 dimensions
- A moving background that moves at a similar speed to cars to create an illusion of movement
- Retro graphics and car models
- Simplified audio effects (If I am able to gain access to original audio from the Chicago Coin Speedway 1969 this will be implemented for realism.
- Evaluate each game and provide a score as a number after a game.
- Add a menu to allow the user to change the colour of their car.
- Optional: Add a shop that the user can spend their score in to buy different paint colours for cars as well as advantages such as a larger tank or faster engine.

THINKING AHEAD

- I plan to use … to build algorithms and code for my program.
- Inputs for the game will use touch controls to steer the car.
- Outputs will be sound effects, animations and a score will be generated.
- A score system and shop will provide a user incentive to play the game.
- If possible, embedding Google Play Games Services into the game will allow users to compete amongst one another for leaderboards.

Breaking Down the Problem



The game can be broken down into 4 main problems:

Graphics:

These will determine how the end product looks to a user. I aim to create car models and backgrounds as similar to the originals as possible. I need to also create an intuitive and aesthetically pleasing menu screen to allow for easy usage. With minimal text it would make it easier to allow support for multiple languages.

Control:

Using touchscreen controls will allow the user to control the car by moving it along the road. A system that pulls the car back to the bottom of the screen when not under acceleration will create an illusion of movement. The ability to pause from the menu screen will mean users can pause a highscore and continue whenever they want.

Gameplay:

The use of hitboxes is integral to the game. It is important in making sure that car models don't overlap one another. It will mean that crashes can be registered between models and penalise the user for colliding with the other car models reducing in a reduction of time. Implementing random cars spawning above the top of the screen along with a moving background and the user car automatically returning to the bottom of the screen will create an illusion of real driving.

Scoring:

Using distance that the car has continued travelling will create a score rewarding the user for being competitive and continuing to improve their score. Using a database or Google Play leaderboards a highscore system can be created that can track who is doing the best at the game leading to increased user engagement.

RESEARCH INTO SEGA GRAND-PRIX AND CHICAGO COIN SPEEDWAY 1969

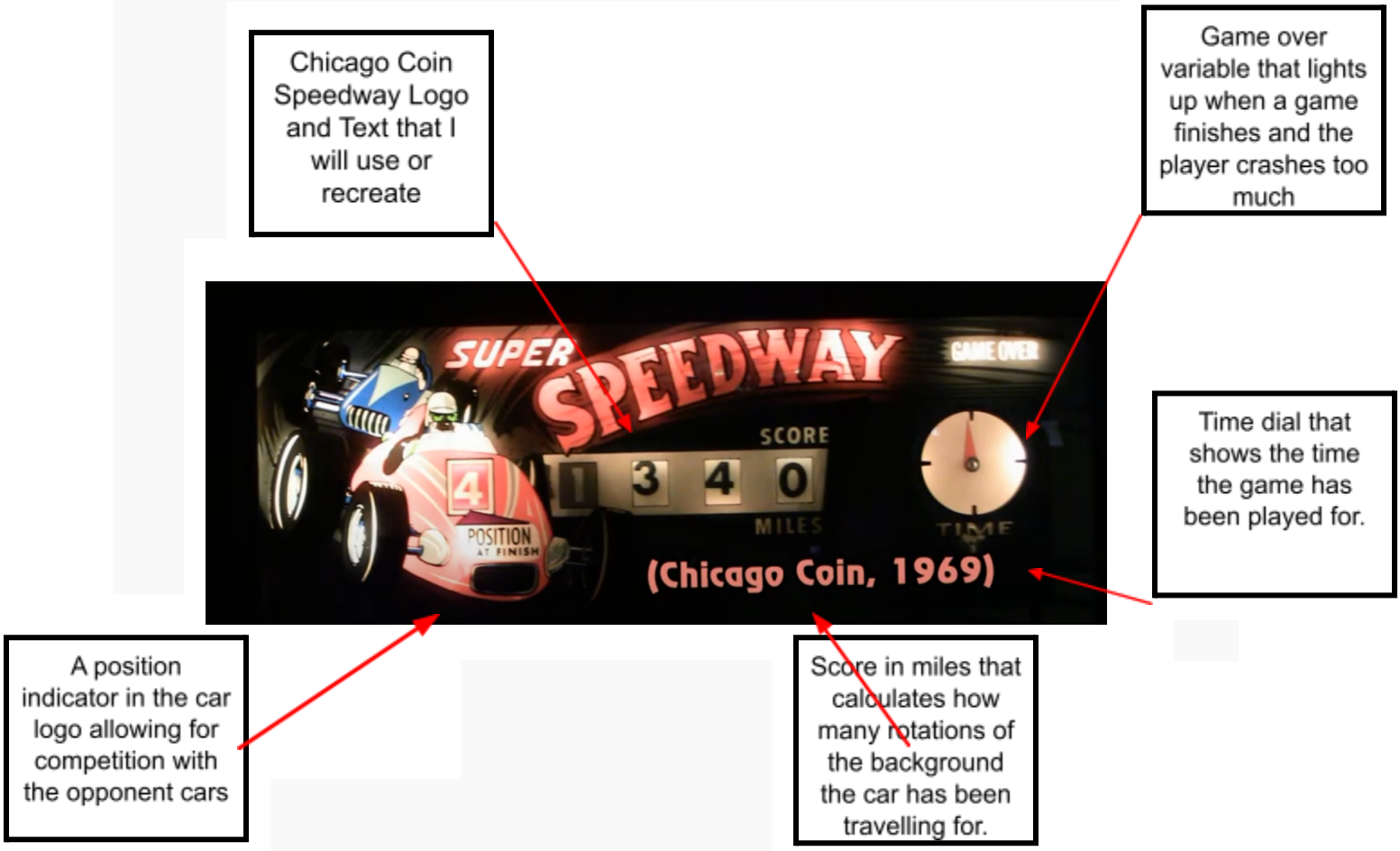
Grand Prix is a 1969 electro-mechanical arcade racing game produced by Sega. The player has control of a racing car in competition and can speed up or slow down by using the gas pedal at the cabinet's base. The object is to avoid and pass other cars without colliding into them. Collisions are accompanied by a crash sound and impact movement by the steering wheel.

It was a video projection game, with a first-person view and forward-scrolling road projected on a screen. It also had electronic sound, and a dashboard with a racing wheel and accelerator. Its projection display system is also similar to Sega's Duck Hunt.

The game relied on electro-mechanical components to produce sounds and images rather than a CRT display. They used rear video image projection to display moving animations on a video screen.

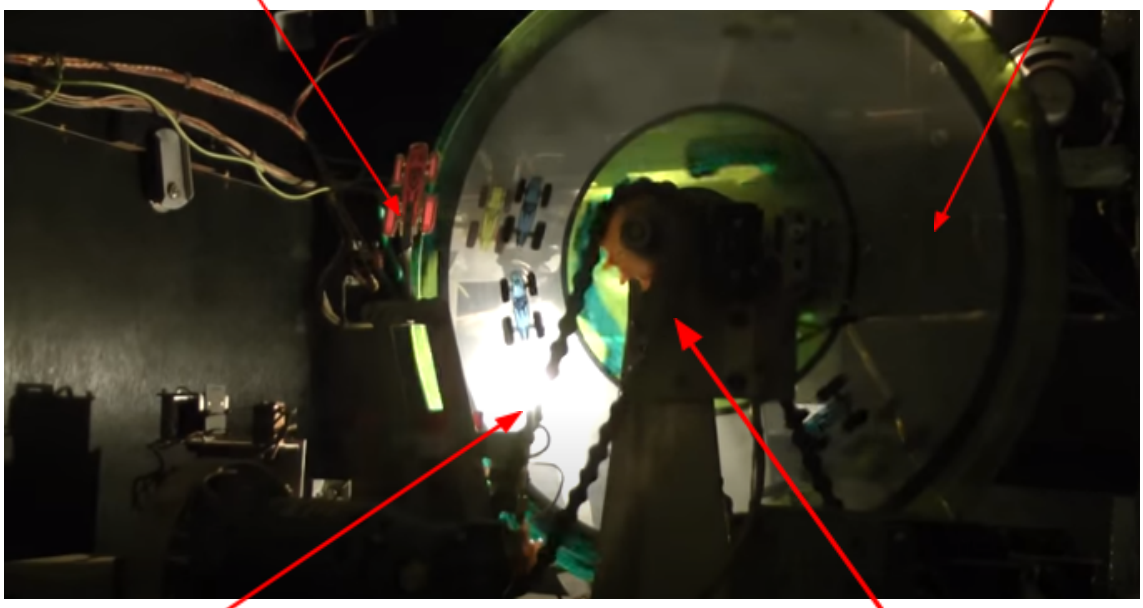
How to play the game

In the game each round has a mile counter that records how long the player has driven for. The user will have 5 lives available in which he can make 5 crashes or accidents with other cars. Once this amount has been reached the game will end and the odometer (mile counter) will be the game score. Enemy cars appear from the top of the screen and the player has to dodge these cars and overtake them. If a player crashed into enemy cars there would be vibration and force feedback through the wheel and a noise of a solenoid crashing into a metal plate.



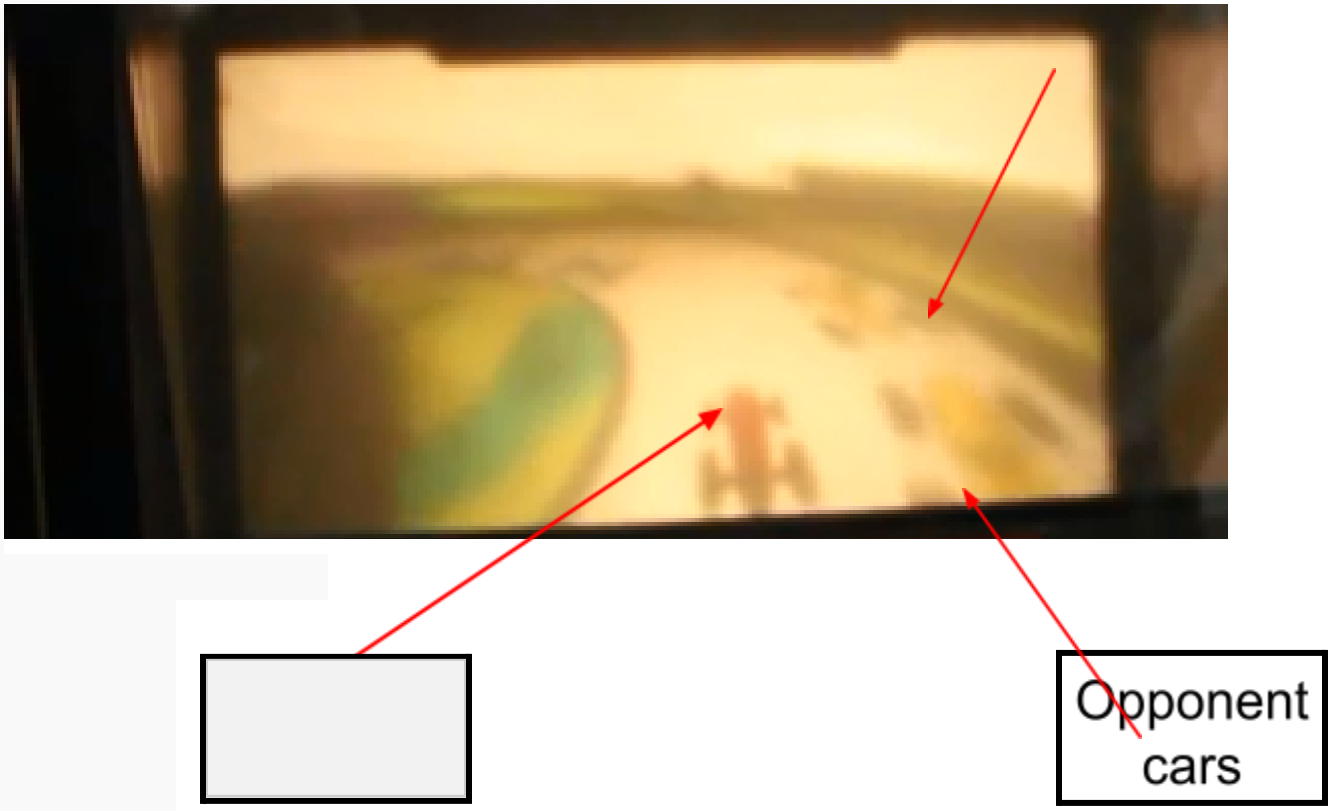
On the top of the arcade machine the user can see:

- The classic Chicago Coin Speedway logo.
- A Gameover light that turns on when the user runs out of fuel in the game.
- A time dial showing the time remaining until there are 5 collisions
- A score in terms of miles that resets after every game.
- A position indicator for competing with the other cars.



The inside workings of the game

Inside the machine there is a rotating disc containing opponent cars. The disc rotates creating a rotating background and the illusion of opponent cars driving. A backlight is shone through the disc and this creates a projection on the screen. When the user steers a mechanical car attached to a metal rod is also projected. When these two overlap a solenoid hits a metal and creates a smash noise, vibrations echo up through the wheel and the screen flashes red and says , 'ACCIDENT'. Every period of time a small disc rotates which increases the miles driven by the player.



When there is an accident recorded, a red light flashes under the timer and the miles freeze. An accident is then recorded and after 5 the game ends. This discourages users from attempting accidents to increase their distance. Opponent cars then drive off the screen and the user loses control of their car for a short amount of time to penalise them before the game once again continues.

When the game ends the Gameover light lights up and the miles freezes until the next game is played. The screen then also goes black as backlights are turned off.

HOW DIFFERENT FEATURES WILL AFFECT GAMEPLAY

Menu/Dashboard:

Displays important information including position of the car, the miles travelled and if there is an accident taking place. This is important for providing information to the user as they are driving.

An accident:

Loud noises and physical vibrations produce a shock factor from the game and acts as a deterrent for crashing along with the freezing of miles travelled.

Sound:

The sound of a simple engine can be implemented and the use of arcade noises means that playing the game on a mobile phone should still reproduce the same feeling as the original game.

Moving background:

This provides for the ability to create an illusion that the car is driving in the old arcade game without the use of a screen. In my game this will work similarly with a background continually being the same rotation to reduce the hardware needed for the game.

Menu:

Chicago Coin Speedway didn't have a menu, only a slot for coins to be entered and then for the machine to begin playing a race. I will implement a simple menu that allows for the pausing and quitting of a game to go back to the original screen.

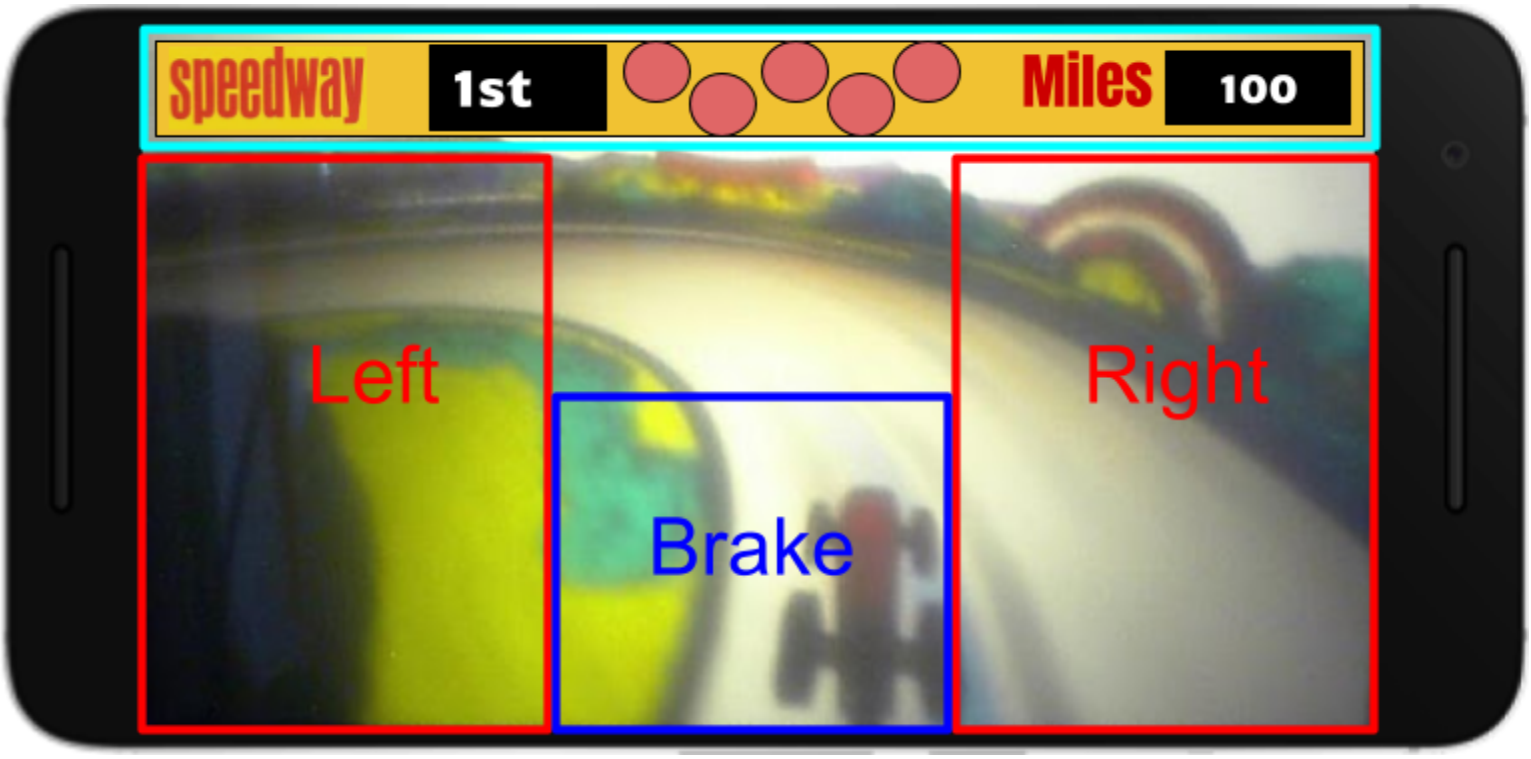
Visuals:

Particle effects can also be used when crashing into an opponent car to further simulate damage to the car. Tyre marks and further visuals can be implemented to provide a fun light-hearted sense of realism yet that is still suitable for all.

Controls:

- The game will be played on mobile devices such as mobile phones and tablets. Therefore the game will be controlled using a touchscreen input device.
- To control the direction of car travel pressing on either side of the screen will control the direction of travel.
- Acceleration will be automatically controlled by the game and cannot be changed by the player.
- To brake the player can press the center bottom of the screen to allow for acceleration to stop and for the car to slow avoiding accidents.

A visual representation of the game controls:



Hardware Limitations

Hardware Requirements:

- 1GB+ RAM

- Supported hardware devices must meet or exceed Google's Android Compatibility Definition limited to the following Device Types:

1. Handheld (Section 2.2)
2. Television (Section 2.3)
3. Tablets (Section 2.6)

- Hardware must be running Android OS natively. Android within a container or emulator is not supported.

Must have a screen that can at least support 720p resolution. These are common in mobile devices and tablets.

These are the listed Unity requirements needed to run the game on an android device. It is extremely difficult to get an app onto the app store and so this game will only be produced for android, initially as an apk for testing and after testing onto the Google PlayStore.

Android	OS 4.1 or later ARMv7 CPU with NEON support or Atom CPU OpenGL ES 2.0 or later.

A newer Android OS is important in meaning that it has recent security updates and has a relevant version of the PlayStore with which to access the game properly. The 1GB RAM Requirement is a minimum requirement for Unity to be able to run and a significant majority of mobile devices meet this requirement making it available to a large group.

The Nokia 1.3 a suitable device costs £65 and provides the minimum requirements showing that the game can run on economical software.

DISPLAY	Type	IPS LCD, 400 nits (typ)
	Size	5.71 inches, 81.4 cm^2 (~77.6% screen-to-body ratio)
	Resolution	720 x 1520 pixels, 19:9 ratio (~295 ppi density)
PLATFORM	OS	Android 10 (Go edition), Android One
	Chipset	Qualcomm QM215 Snapdragon 215 (28 nm)
	CPU	Quad-core 1.3 GHz Cortex-A53
	GPU	Adreno 308
MEMORY	Card slot	microSDXC (dedicated slot)
	Internal	16GB 1GB RAM
		eMMC 5.1

The Device initially needs a connection to the internet to Download and Install the game but after that the internet is not required to play the game. However the internet free version will have some disadvantages such as not having the ability to show highscores of other players as there cannot be any contact with any highscore saving servers.

Requirements

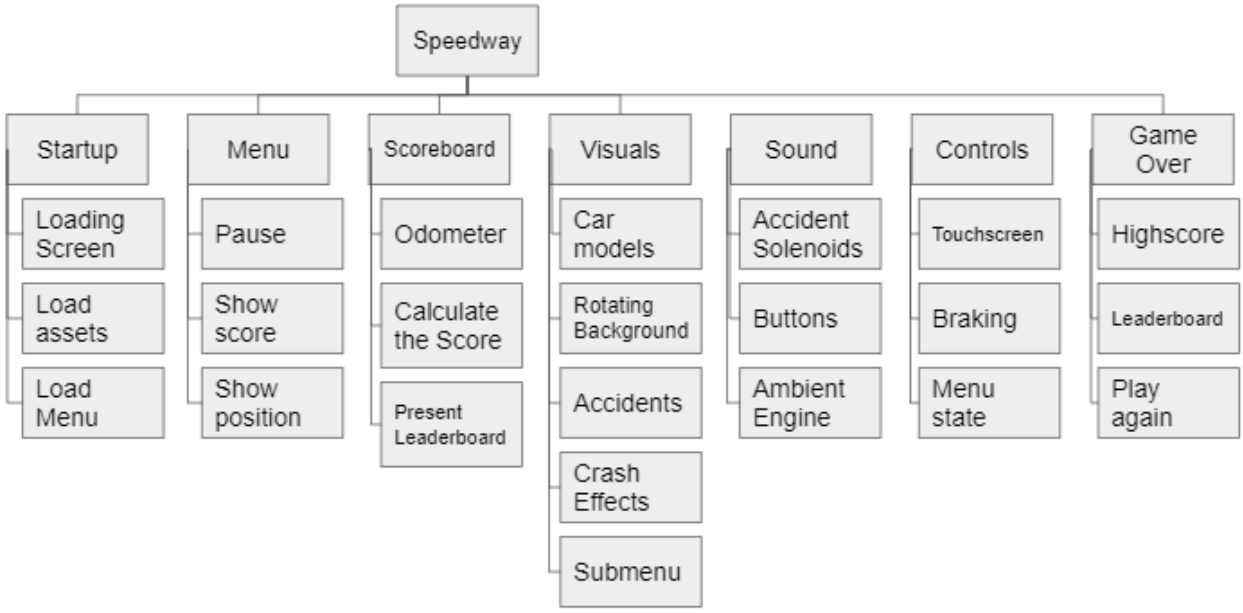
No.	Requirement	Description and Explanation	Reference
1	Position Indicator	This acts as an indicator for the user to recognise where they are amongst the opponents. A better position will mean a greater score.	A feature from Chicago Coin Speedway.
2	Odometer	This provides the information for how long the car has driven for. This will be the score that allows the user to know when they are close to beating a high score.	A feature from Chicago Coin Speedway.
3	Accident Indicator	This will use mobile phone vibration feedback along with flashing red circles in the top menu to allow the driver to recognise when there is an accident and then their Accident counter will increase.	A feature from Chicago Coin Speedway.
4	Moving Background	Provides the illusion of movement and distance travelled. Also means with a moving background the game will seem more dynamic and interesting.	A feature from Chicago Coin Speedway.
5	Opponent Cars	Opponent Cars will be deployed to provide obstacles to the player and difficulty in overtaking. This will mean that there will need to be skill involved in committing clean overtakes and not causing accidents.	A feature from Chicago Coin Speedway.
6	Accidents	When there is an accident Opponent Cars will continue off-screen. This means that after an accident the user can begin driving again without the worry of being in a constant accident.	A feature from Chicago Coin Speedway.

7	Menu	This provides an interface for the user to enter the game, pause and look at their highscore. It will also provide interesting data such as how many miles need to be travelled to reach the high score.	A feature from Chicago Coin Speedway.
8	Graphics	Retro Graphics will make the game look similar to how it looked in 1969. With car shapes and a moving background in a retro art style it can keep a large target audience and be suitable for all.	A feature from Chicago Coin Speedway.
9	Sound	When there is an accident there will be a vibration and the noise of metal colliding. There will be a retro engine sound and sound effects and audio will be suitable for all meaning the target audience remains large.	A feature from Chicago Coin Speedway.
10	Game Over Screen	This will provide an interface with information from the Odometer, Position Indicator and Accident Indicator. It will produce a high score and allow the user to look at how much they improved or how close they were from beating their highscore. It will then allow the player to replay the game.	A feature from Chicago Coin Speedway.
11	High Score Table	Users will be able to save their highscores to a central database where they can be ranked. This means a user can compete against themselves but also against others creating some competitiveness and competition.	A feature from Chicago Coin Speedway.
12	Controls	Mobile touchscreen controls will be used to steer and brake the car. This means that anyone can play and there will be instructions at the beginning of the game.	A feature from Chicago Coin Speedway.
13	Driving Sound	A Sound to play in the	A feature from Chicago Coin

		background whilst the car is driving. This makes the game have some ambience.	Speedway.
14	Pause	The ability to access the menu and pause a game to be continued at a later time.	Suggested Improvement.

B. Design

Systems diagram



By using a top-down module design it makes it possible to identify each sub-problem that needs to be solved.This allows for me to think ahead for the needed inputs and outputs of the game and from there begin to design flowcharts which can help identify algorithms that will need to be implemented .

Decomposition of the design

Startup:

Loading Screen -

- There will be a simple black background, game states and sprites will be loaded in in the background from their files. Each separate object in the game will have an identifying name with which it will be stored as.
- The name of the game will first be loaded slowly as it fades into view from the background.
- From then this will then become the menu state and you will be able to play the game

Load Assets -

- Sprites, backgrounds, menu art which are needed will all be loaded into memory. There is not much of this art so this won't be too intensive and take up large parts of memory. This will mean that data can be cached before use.

Load Menu -

- Menu art will be loaded and then displayed on the screen
- As this happens all variables will be reset such as the odometer and all gameplay variables. This then means the game is playable from this menu state and all variables have been reset and art has been loaded ready for implementation.

Menu:

Pause -

- The menu screen will then load and all gameplay will stop so the screen will be dark with just the logo and an unpause button
- All gameplay variables will be frozen and appended to an array where they can be safely stored as the game remains in the background.
- Once the game is unpaused variables will be fetched and once that has taken place the game can then begin from this saved state.
- Displays the current high-score so that the user can see what they need to beat.
- This will all be in a function that uses a button to pause and unpause and then changes the game state from the menu state to the playing state. It will be important here to remember the positions of where the cars were as the game was paused so when the game is unpaused the cars can return to their original position. This can be implemented with an array or list which acts as a temporary store of data.

Show Score -

- The current odometer and position will be displayed at the top of the screen in a submenu that shows important information. This function that displays the scores will be iterative and will check to see if the score value has changed. If the score changes then the new score will then be displayed. This checking for changes will mean that the score always remains live and upto date for the user.

Show position -

- The position will also be displayed in the same way at the top of the screen in the submenu, it will check for updates and if that's the case then the new value will be outputted.

Scoreboard:

Odometer -

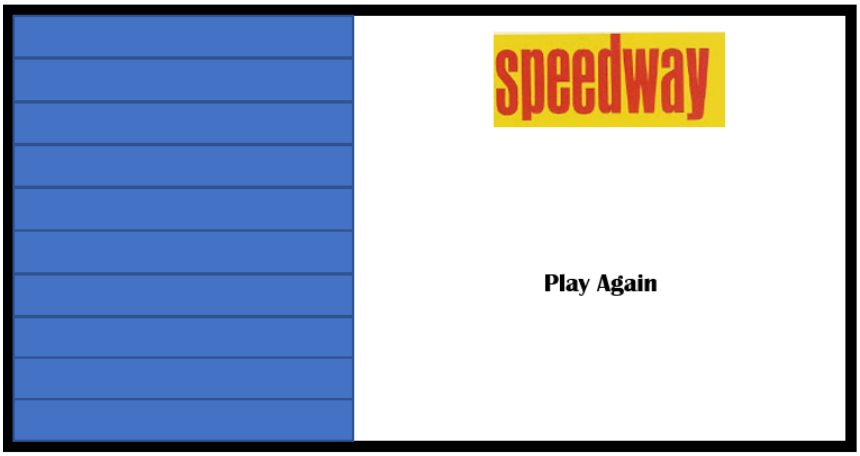
- The odometer will be the scoring system used and means that it can be used for a highscore system.
- This score will be created based on a time system. The user will gain 1 point per second of gameplay.
- The odometer will be frozen and the variable not changed whilst the game is in a pause state or whilst there is an accident where cars collide with one another.
- This will always be displayed in a submenu at the top of the screen
- The pause menu will display the odometer highscore

Calculate the Score -

- The score will be calculated by the odometer value minus the number of collisions.
- Each collision corresponds to a negative 5 points and means that clean-driving is rewarded by not being penalised.
- This value will be displayed in the pause screen along with the high-score.
- The score can then be used in a leaderboard text file meaning that users can play both against themselves and against others on the same mobile device.

Present the Leaderboard -

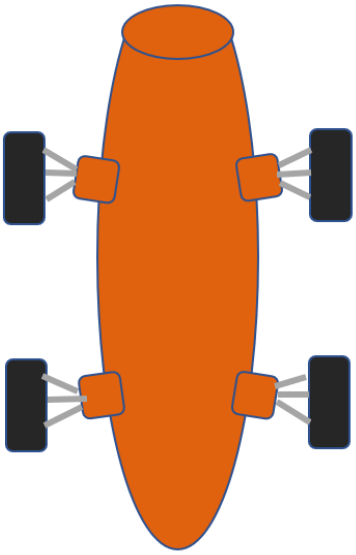
- The leaderboard layout will be similar to this below.
- The leaderboard will be shown after there are 5 collisions and the game will enter a menu state displaying the leaderboard.



Visuals:

Car models -

- Car models will be created as simple 2d sprites that will move based on movement from touchscreen inputs unless there are collisions that take place.



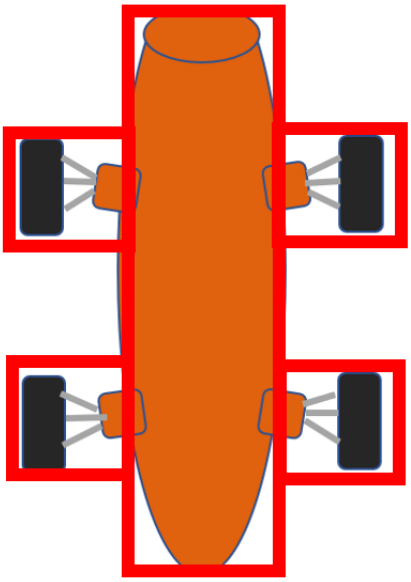
- Car model will include a simple composite shape that embodies a 1960s F1 Car shape.

- It will consist of a body with outlets where wheels are and then silver wheel tethers that connect the wheels to the body of the car.

- 4 Thin tyres will then float at the side of the body

- This simple car model will be reproducible and colours can be changed simply meaning that opponent cars can be generated simply without requiring many sprites. This also makes it possible for colour randomisation.

- The car hitbox can also be shown. This will be used when needing to calculate collisions



- This car hitbox surrounds the car as tight as possible whilst still overlapping certain areas of the car such as the wheel tethers and the front and back of the car. This means that it is conservative in making sure that there are no false negative collisions.

- However, it is still possible for there to be a false positive collision in which car hitboxes could overlap and collide without the cars graphically colliding. The chances of this happening are slim and due to efficiency of having simple hitboxes that risk will have to be taken in the game and will be an important part of my testing in the future.

Rotating Background -

- A simple background consisting of trees and a background of mountains can be introduced that rotates through over and over.
- This could be implemented as a sprite that constantly loads at the top of the screen and then moves down to the bottom whilst this iterates.
- However, an idea to explore would be to use a repeating gif that repeats seamlessly with itself so a gif can keep the background continuing.
- This part of the program will use recursion to save on resources and make it easier to program.

Accidents -

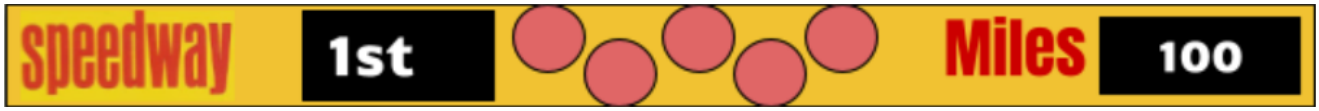
- An algorithm or function will be used to constantly check for overlapping hitboxes between the user and opponent cars. If there is an overlap then it will call upon further functions that will freeze the odometer, pull opponent cars off screen and reset the player's car. It will also increment an accident counter.
- 5 of these accidents and the game ends and the user's score is then saved and available for use in a high score leaderboard and sharing with friends.

Crash effects -

- Simple particle effects such as sparks and tyre lines can be implemented to add to make it clear when there is an accident between two cars. As this happens then there will also be audio punishment.
- This hopes to associate the player with crashes and when there are sparks put them off from the game.

Submenu -

- The submenu will contain: The Game logo, Position Indicator, Accident counter and Odometer



- This submenu design is not finalised as I may think of a better way to show the Accident Counter that can be more clear to the user and take up less space.

Sound:

Accident Solenoids -

- In the original Chicago Coin Speedway game when there was an accident between cars a solenoid would crash into a plate of metal and the steering wheel that the player used would vibrate. This noise and the vibration acted as a shock factor that made it seem as though you had just physically crashed.
- To keep the game as similar to the original as possible then keeping the same noise is important and if the sound is off then I will implement vibration instead. There will be a choice to disable both of these in the menu/settings for those who want to play silently without vibration for example when on public transport or other situations where silence would be liked.

Buttons -

- Simple audio can be played when buttons are pressed such as clicking to open the menu, pause the game etc.
- This will be used to make buttons seem responsive and make it seem as though they are physically pressed.

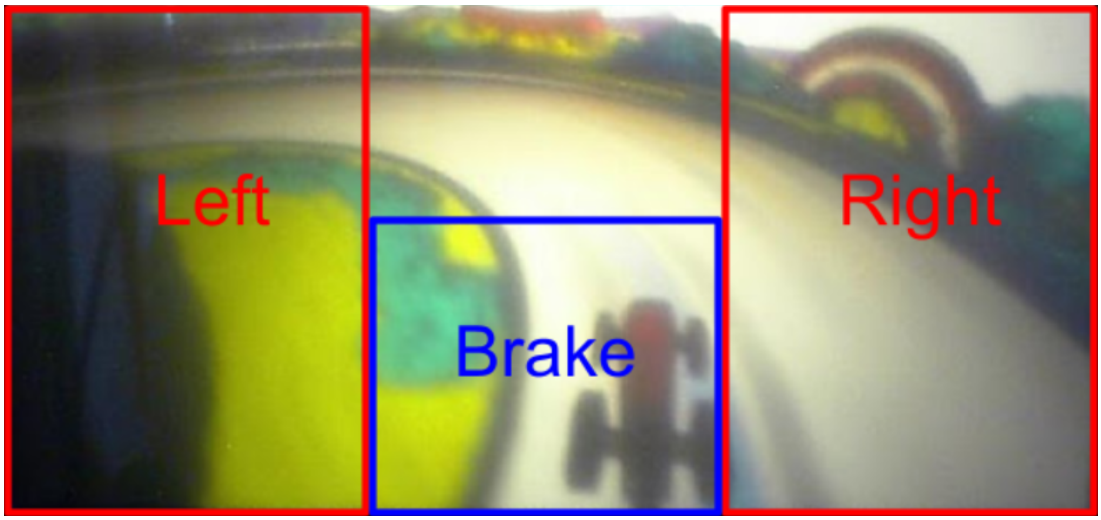
Ambient Engine -

- Engine sounds may play whilst the car is driving and the game is being played to provide a feeling of driving.
- This will continue when there is an accident because the car will keep moving.

Controls:

Touchscreen -

- The touchscreen will be used as the input for steering as almost everyone has a mobile phone meaning there is a large market for the game to be played and touchscreen is a portable input meaning that people can play the game at any time.
- The touchscreen layout will look like this:



Braking -

- Braking is an important part of the game as it helps to avoid accidents meaning that the player can continue driving. I considered the best places to put the touch controls but i believe having the steering either side of the screen is intuitive in choosing the steering.
- The braking however, I decided to place in the bottom center to symbolise that there is no steering at this point and that this will cause the car to go back on the screen as the game is shown at a constant speed. By that I mean, the background constantly rotates so the illusion of movement has to be kept.

Menu State -

- The menu state will use touch controls for buttons and use audio to also show the impact of buttons.

Game Over:

Highscore -

- The Highscore system will use the odometer to come up with a score. From there the value from the odometer along with the user information such as the player # and other hidden variables that differentiate between users. They will be stored in separate arrays so that if you were to output the third item of each list then the outputs would correspond.
- These outputs will then be ordered and placed into a central text file or database in which the highscores can be stored and then fetched when showing the leaderboard.
- If possible using Google Play Services it would be good to use the built in leaderboards and order Google accounts based on miles travelled.

Leaderboard -

- (The same as Present Leaderboard -) this will show the highscores and players from the Highscore file. Highscores will be saved in an Array. This will store only the top ten highscores and then when there is a new highscore the Array can then be shuffled and the lowest amount removed. The Array should remain the same size throughout.

Play again -

- After seeing the highscores and after the game has ended a button will be added to the menu which allows the user to then click to play again and at this point the game will begin from the start again. All variables will then be wiped.

Key Variables

Method	Key Variables	Data Type	Local / Global	Validation	Data e.g.
Important Global Variables					
Variable	player#	integer	Global	Doesn't need validation as it isn't entered.	24
Variable	playername	string	Global	Whitespace, 3-15 Characters, Simple list of banned words like profanity and racism.	JDS3
Startup					
Game State	loadingState	Boolean	Global	Black screen with logo where assets are loaded in the background and the game cannot be played.	True
Game State	backgroundState	Boolean	Global	The game is at a part where it is running and the background will continue cycling and controls can be inputted.	False
Procedure	title	N/A	Global	The game title loads in onto the black screen as an image.	

Variable	time	Integer	Global	Used for the odometer and when using sleep to keep a uniform loading screen.	
Sprite	logo	.jpeg	File	Shows the game logo	
Menu					
Game State	menuState	Boolean	Global	The game is at a point where the user can choose game options and the menu is displayed.	True
Procedure	play	N/A	Global	Links the play button and the Game State	
Procedure	controls	N/A	Global	Links the play button and the game allowing control inputs again.	
Playing					
Game State	playing	Boolean	Global	The game is actually in a state of play.	False
Procedure	initialise	N/A	Global	The game is initialised and game variables are reset.	
Procedure	pause	N/A	Global	This links the playing state to the paused state.	
Procedure	odometerIncreases	N/A	Global	The odometer increases adding a mile per second whilst there are no accidents taking place.	
Variable	odometerValue	Integer	Global	The odometer value which is used for the game score.	386

Variable	accidentValue	Integer	Global	The accident value which is used for calculating when the game is over.	2
Variable	odometerReset	Integer	Global	The value the odometer is reset to when accidentValue = 5	0
Variable	accidentReset	Integer	Global	The value the accident counter is reset to when the game is restarted.	0
Sound-file	engineSound	.wav	File	The sound of an engine plays	
Sound-file	tyreScreech	.wav	File	The sound of tyres screeching plays	
Sound-file	solenoidSmash	.wav	File	The sound of a collision plays	
Sprite	userCar	.jpeg	File	The user's car appears	
Sprite	enemyCar	.jpeg	File	The opponent's car which can be reproduced	
Submenu					
Procedure	odometerMenu	N/A	Global	Links the odometerValue to showing up at the top of the submenu	
Procedure	positionMenu	N/A	Global	Links the positionValue to showing up at the top of the submenu	
Sprite	logo	.jpeg	File	Shows the logo	
Sprite	redLight	.jpeg	File	Shows a red light that is used to indicate the number of accidents that	

				have occured.	
Sprite	positionIndicator	.jpeg	File	Position sprite where the text then appears	
Sprite	odometerIndicator	.jpeg	File	Odometer sprite where the text then appears	
Controls					
Game State	gameControlsState	Boolean	Global	The game allows control inputs.	
Procedure	escapeControls	N/A	Global	The controls state changes back to the menu state. (For use when paused or game over).	
Position Calculator					
Variable	positionValue	Integer	Global	The value of the position that is then used and shown in the submenu.	
Procedure	hitboxPass	N/A	Global	If the car hitboxes pass one another this is an overtake or the player has been overtook.	
Accident Calculator					
Procedure	hitboxCollision	N/A	Global	If car hitboxes collide then there is an accident	
Accident Collision					
Procedure	runSolenoidSmash	N/A	Global	Calls upon solenoidSmash to be played	
Procedure	odometerPause	Boolean	Global	Pauses the odometer from rising	

Procedure	escapeControls	N/A	Global	Controls cannot be used as they act as though it is a menu state	
Procedure	incrementAccident	N/A	Global	The accidentValue is incremented by One.	
Procedure	runRedLight	N/A	Global	The redLight file is loaded into the submenu.	
Game Over State					
Game State	gameOverState	N/A	Global	There have been Five accidents and the game has finished and the game is over..	
Procedure	gameOverDisplay	N/A	Global	The Game over screen is displayed to the player.	
Procedure	leaderboard	N/A	Global	Links the leaderboard and Highscore to the Game Over State.	
Ordering System					
Variable	player#	Integer	Global		
Variable	odometerSaved	Integer	Global	The highest saved value that the odometer has ever had for the individual	
Procedure	odometerOrder	N/A	Global	Orders odometer values of others and ranks	
Procedure	runLeaderboard	N/A	Global	Loads the leaderboard state with these ordered values and shows the Top 10 scores.	

Leaderboard					
Sprite	goldMedal	.jpeg	File	Gold Medal	
Sprite	silverMedal	.jpeg	File	Silver Medal	
Sprite	bronzeMedal	.jpeg	File	Bronze Medal	
Play Again					
Procedure	variableClear	N/A	Global	Clears all game variables that need to be reset for the game to play again.	
Procedure	menu	N/A	Global	Links to the menu state .	
Procedure	play	N/A	Global	Links to the game state.	

Algorithms:

Gameplay Algorithms:

Collision Algorithm -

- I need to implement a collision algorithm so that I can tell when two car hitboxes overlap and if so then an accident will have occured and then the accident procedure can take place
- I want the physical simulator to act as a posteriori (that is where it will detect a collision after the collision actually occurs. This is also known as discrete). This is the preferred method as unlike a priori it needs only be fed simple information as it only requires a list of bodies and checks for overlaps.
- Optimisation then becomes another issue because if you are constantly checking for collisions between every car on the screen then it will take a lot of time.
- In two-dimensional games, in some cases, the hardware was able to efficiently detect and report overlapping pixels between sprites on the screen - Wikipedia
- This would be my preferred method as designing a hitbox collision algorithm would take too much time along with the rest of programming. Therefore I will need to find an already designed algorithm which I can then use and implement in Chicago Coin Speedway. The original game didn't have this issue as it would work with physical cars.
- Therefore I have decided to use the Axis-Aligned Bounding Box method which uses rectangles as hitboxes.

Axis-Aligned Bounding Box Algorithm -

- The algorithm works by ensuring there is no gap between any of the four-sides of the rectangles. Any gap means a collision does not exist.

Pseudocode -

```
var rect1 = {x: 5, y: 5, width: 50, height: 50}
var rect2 = {x: 20, y: 10, width: 10, height: 10}

if (rect1.x < rect2.x + rect2.width &&
   rect1.x + rect1.width > rect2.x &&
   rect1.y < rect2.y + rect2.height &&
   rect1.y + rect1.height > rect2.y) {
    // collision detected!
}

// filling in the values =>

if (5 < 30 &&
    55 > 20 &&
    5 < 20 &&
    55 > 10) {
    // collision detected!
}
```

Explaining the above pseudocode:

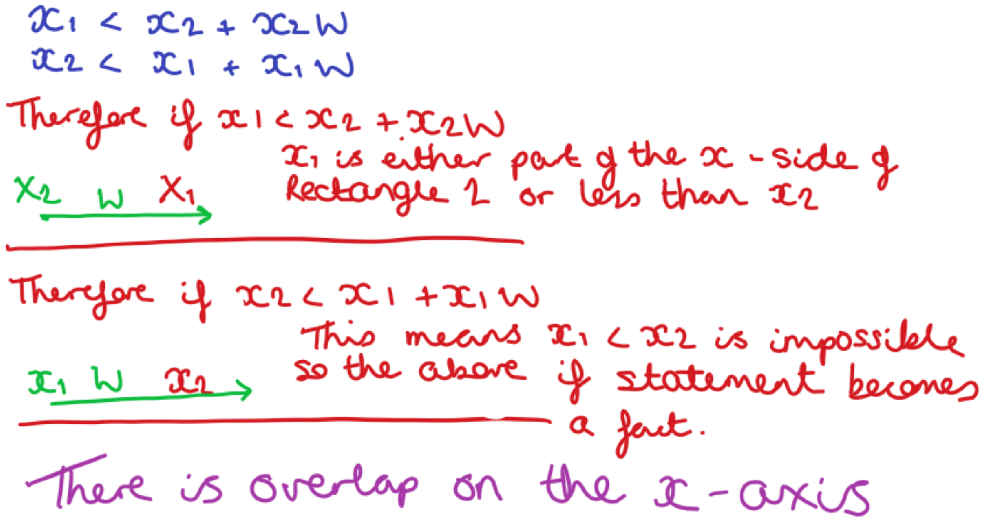
- In the first two lines the rectangles are defined. Here the x and the y act as co-ordinates and then they are paired with the width and height
- Here they set up the if statement w = width h = height

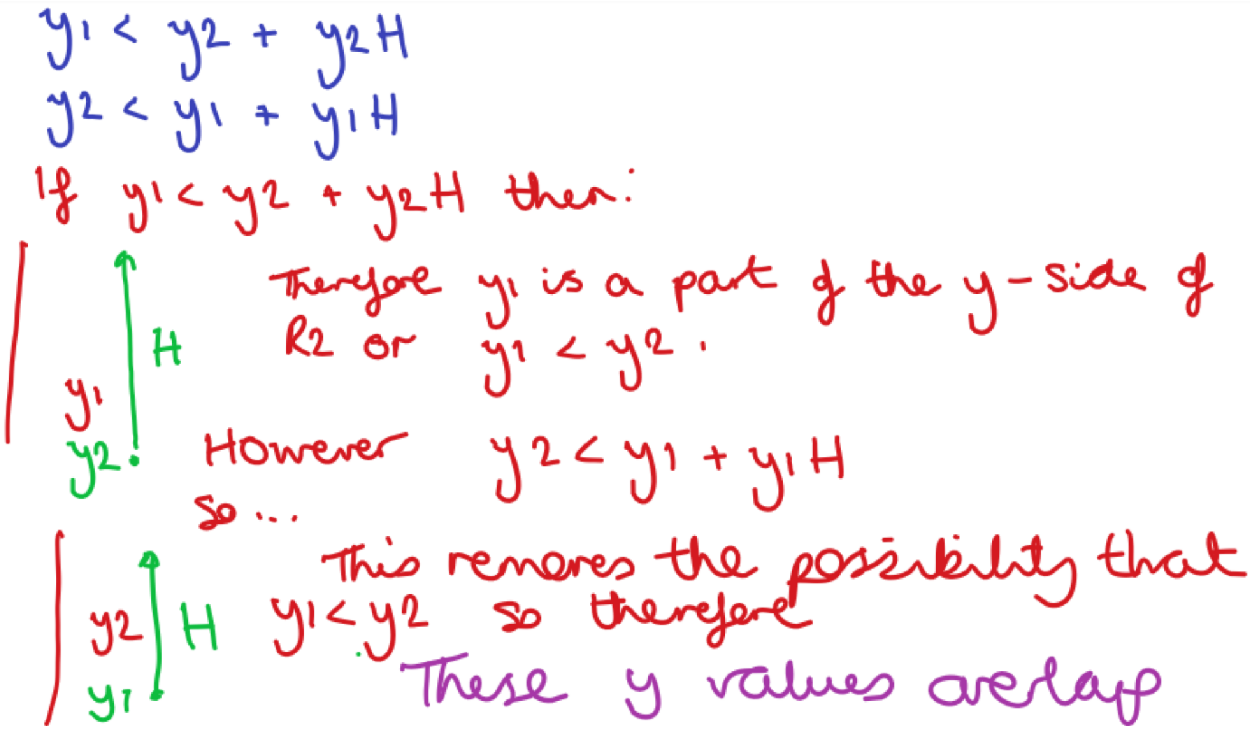
If x1 < x2 + x2w and x2 < x1 +x1w and y1 < y2 + y2h and y2 < y1 + y1h

- This first if statement contains 4 statements:

x of Rectangle 1 is smaller than the largest x value of Rectangle 2
x of Rectangle 2 is smaller than the largest x value of Rectangle 1
y of Rectangle 1 is smaller than the largest y value of Rectangle 2
y of Rectangle 2 is smaller than the largest x value of Rectangle 2

- The second if statement here is just a simplified version of the first if statement using numerical values.





Therefore as I have proven with these diagrams these rectangles overlap on the x-axis and on the y-axis. This means that there are overlapping coordinates and at least 2-sides on each Rectangle are colliding with 2 from another. So there will be some sort of collision or hitbox overlap which needs to be resolved.

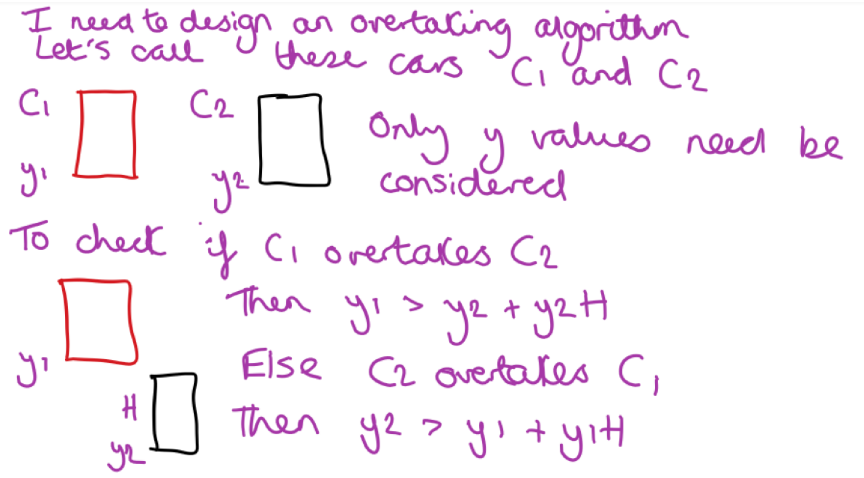
Using this algorithm I will be able to find if there is a collision. From there I can then run the Accident Procedure.

This seems to be the simplest way of calculating if there is a collision however it then introduces new problems because this is a posteriori hitbox collision algorithm. That means that for it to detect a

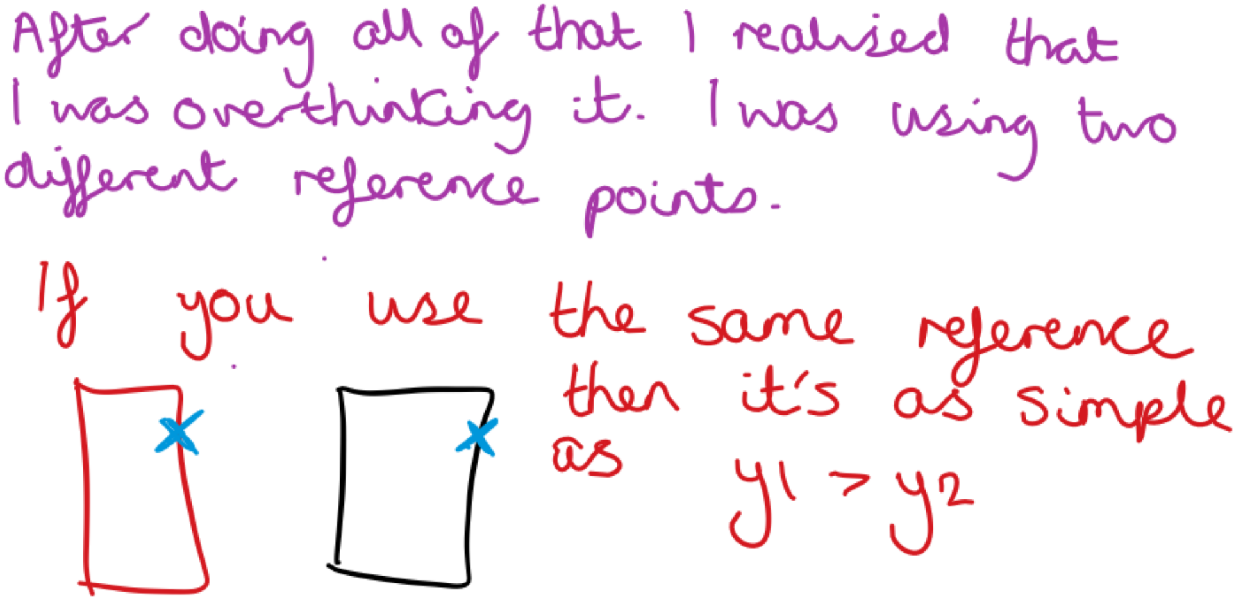
collision a collision must have already happened. Therefore I need to separate these overlapping cars and send these sprites somewhere where they will no longer be overlapping.

Overtaking Algorithm:

- If i can use the algorithm above to find if two shapes overlap then I can repurpose this to find out if the player's car has overtook an opponent's car. This time it will only need the use of the y-axis and not the x-axis.



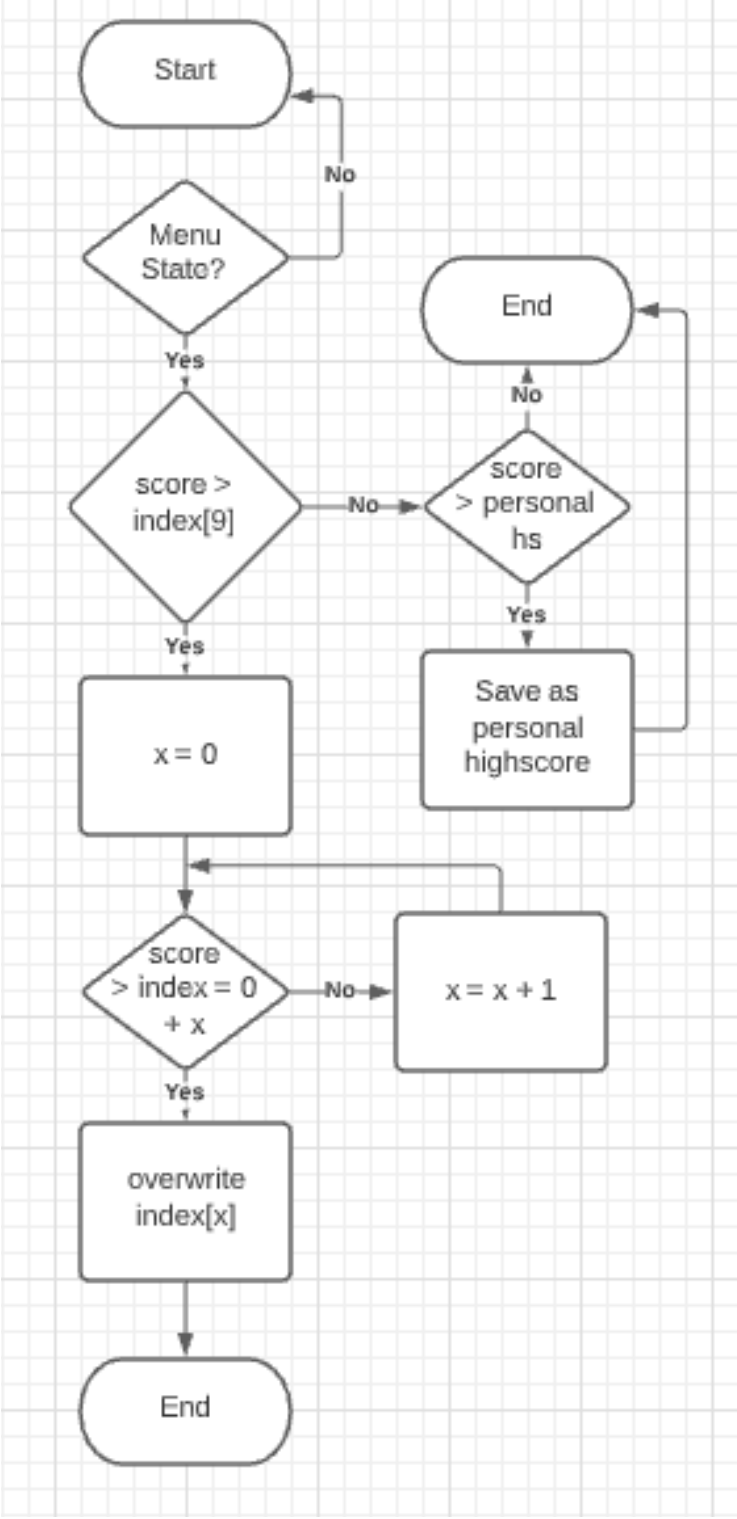
As you can see from the diagram this is much simpler as it only requires one inequality. However there are multiple outcomes this time so there will need to be many elifs. An overtake will only count if cars fully pass according to my algorithm and not if one is slightly ahead of the other so we need to modify this further. So I will do that now. This will need to think about the height of the overtaking car as well.



Flowcharts:

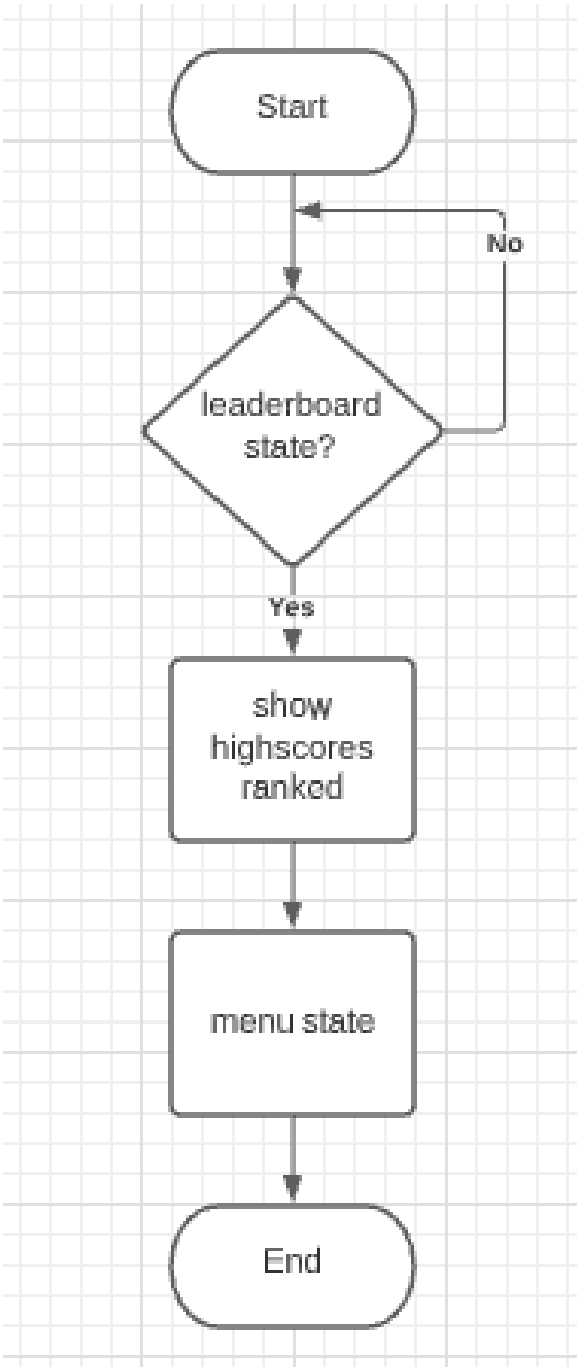
Leaderboard -

Highscore Checker -



Here the High Score checker flowchart firsts checks if the game is currently in the menu state. If it isn't then it will attempt this again and continually iterate until it is in the menu state. This may be updated each frame. Once the game is in the menu state the score is checked to see whether it is above the 10th highscore. If it isn't then this score won't be part of the top 10 scores. It then checks whether this score is above the user's local highscore. If not the program ends as this wasn't a highscore. However if it is above the user's personal high score this will then replace the user's personal highscore. If the score was above the 10th score in the top 10 high score array then a counter variable named x is used. This is used to check the position of this new score by finding the position that it is larger than. It will then replace this position and the rest will move down one.

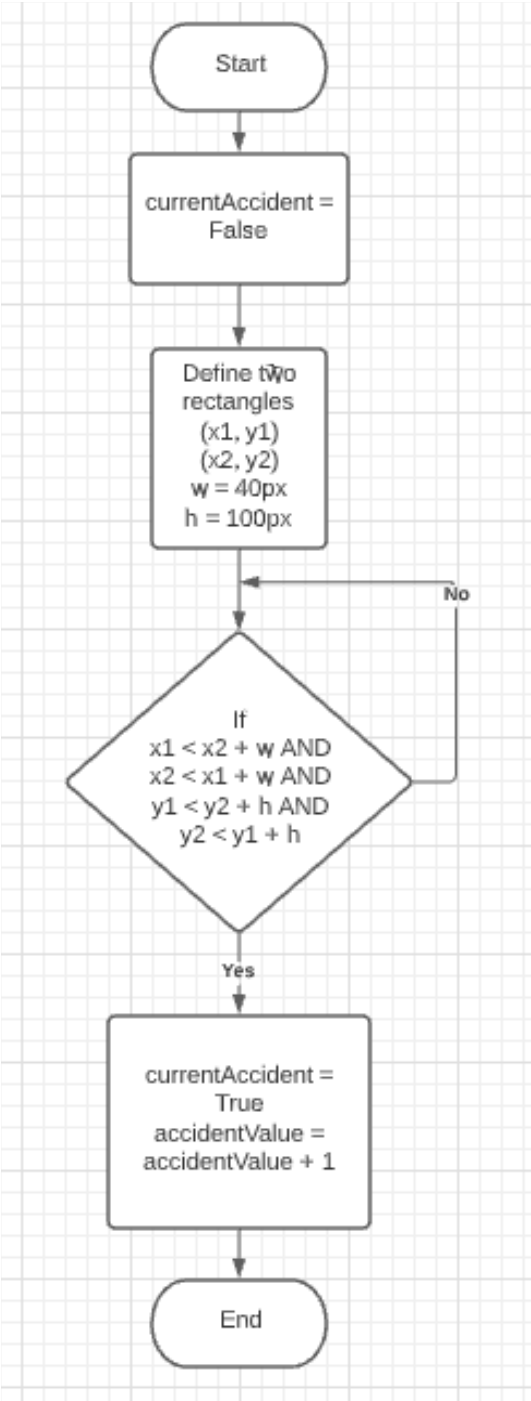
Displaying -



To display the flowchart above the program then checks if this is in the leaderboard state. If not it will keep checking on each updated frame. If yes then it will show the high score array and this new score will be implemented in there if it fits the requirements of the above flowchart. Once done it will then go back to the menu state.

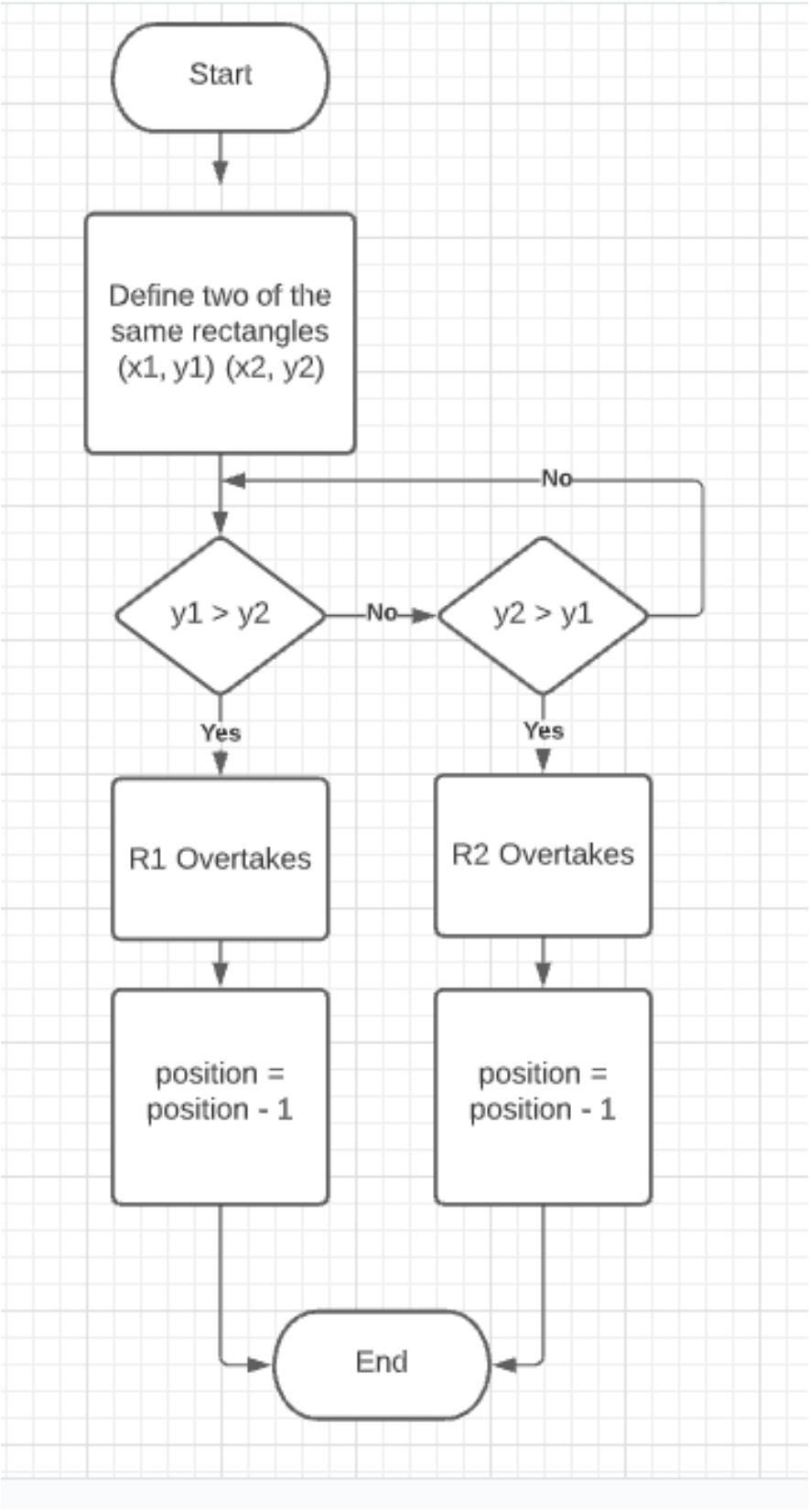
Whilst Playing:

Accident Checker -



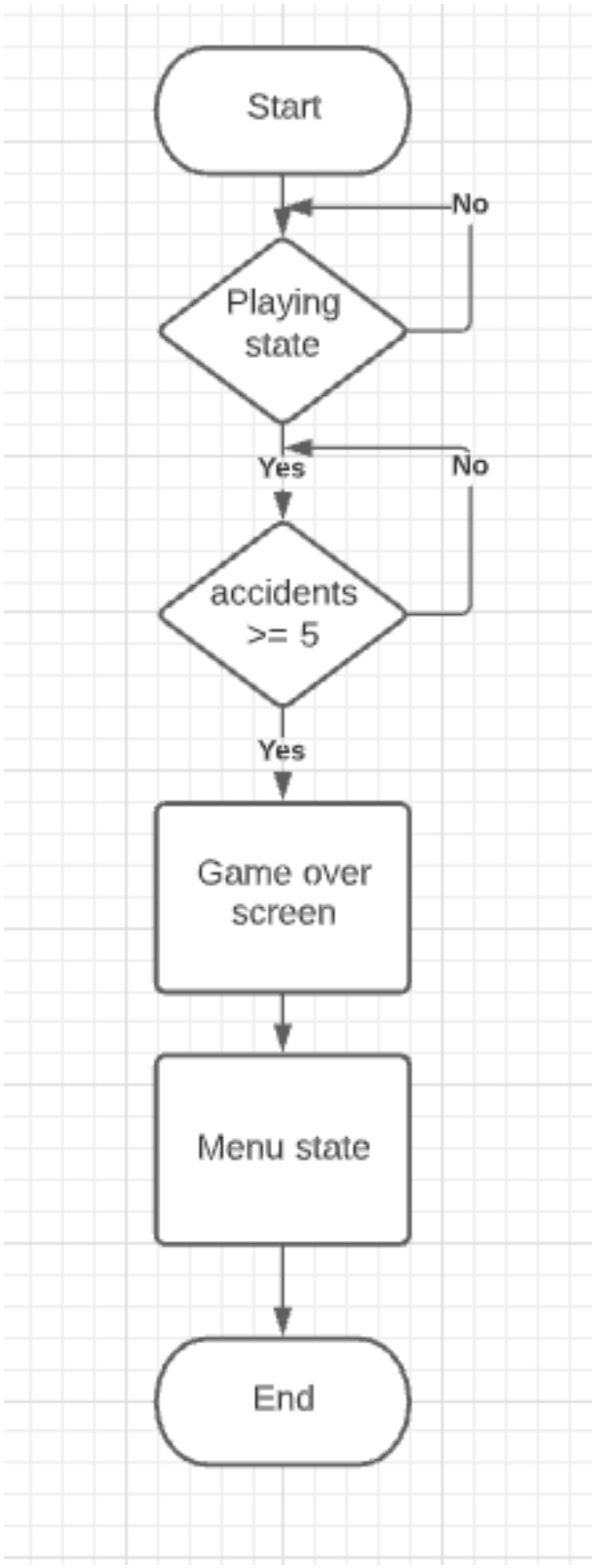
To check for an accident the program begins and first checks if an accident is already taking place. This is important as it stops multiple accidents being recorded for one event. Otherwise if there was an accident lasting 5 frames this would then result in 5 accidents. Once it finds that an accident isn't currently taking place it then uses the variables that are defined with the two sprites. From here it then uses the Axis-Aligned Bounding Box algorithm to compare the coordinates of the sides of each and find any overlaps or collisions. It checks for collisions every updated frame. If there is a collision the currentAccident becomes true meaning that it won't duplicate the accident counter. The accidentValue is then incremented.

Overtake Checker -



From here we check if a spite overtakes another sprite. y values are compared to see which sprite is in front of the other on the screen. First the user's car is checked whether it is in front of the opponent. If yes then he overtakes and gains a position. His positioncounter decrements. However if this is not the case then the same check is made for the opponent car. If he overtakes his position counter is then decremented and the player's position counter will increment. If neither of these take place then there are no overtakes and so the program will iterate until there is an overtake.

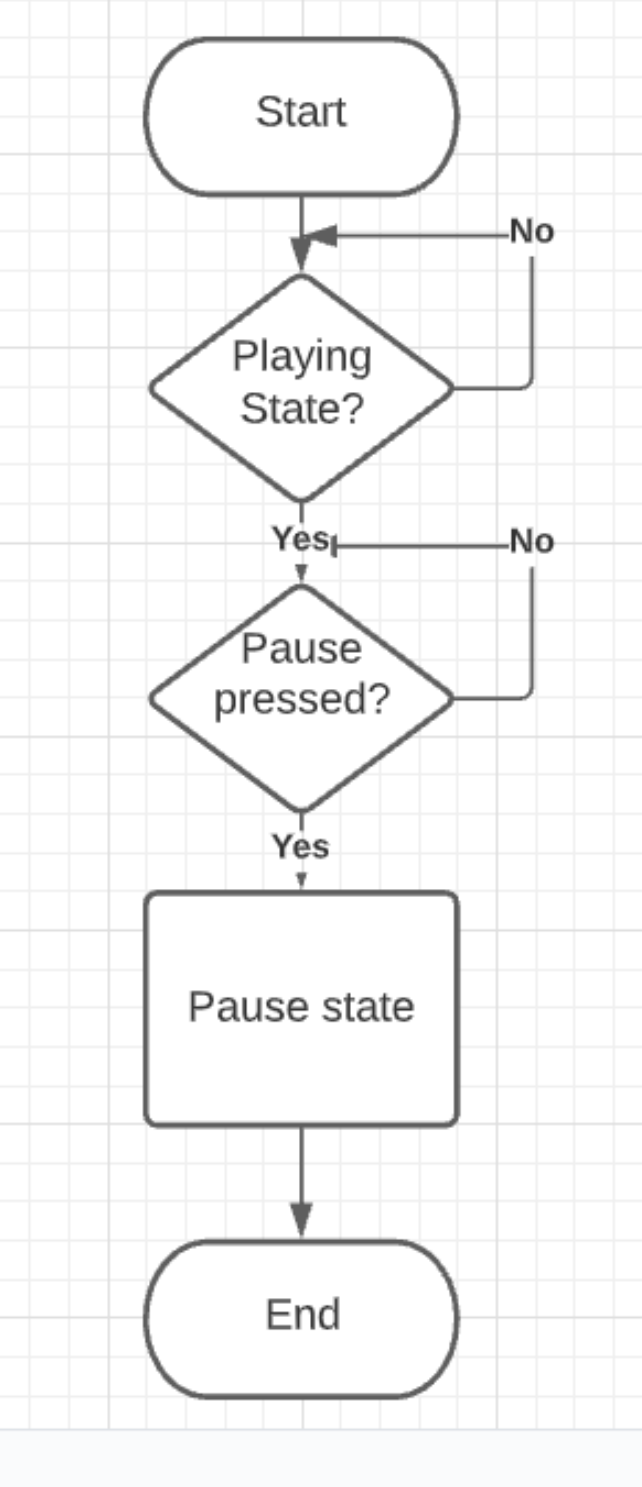
Game Over -



Here we check if the game is over. If the game is in the playing state then we check if the current number of accidents is equal to or exceeds 5. If there are 5 accidents then the game over screen will then be shown and the menu state will be entered.

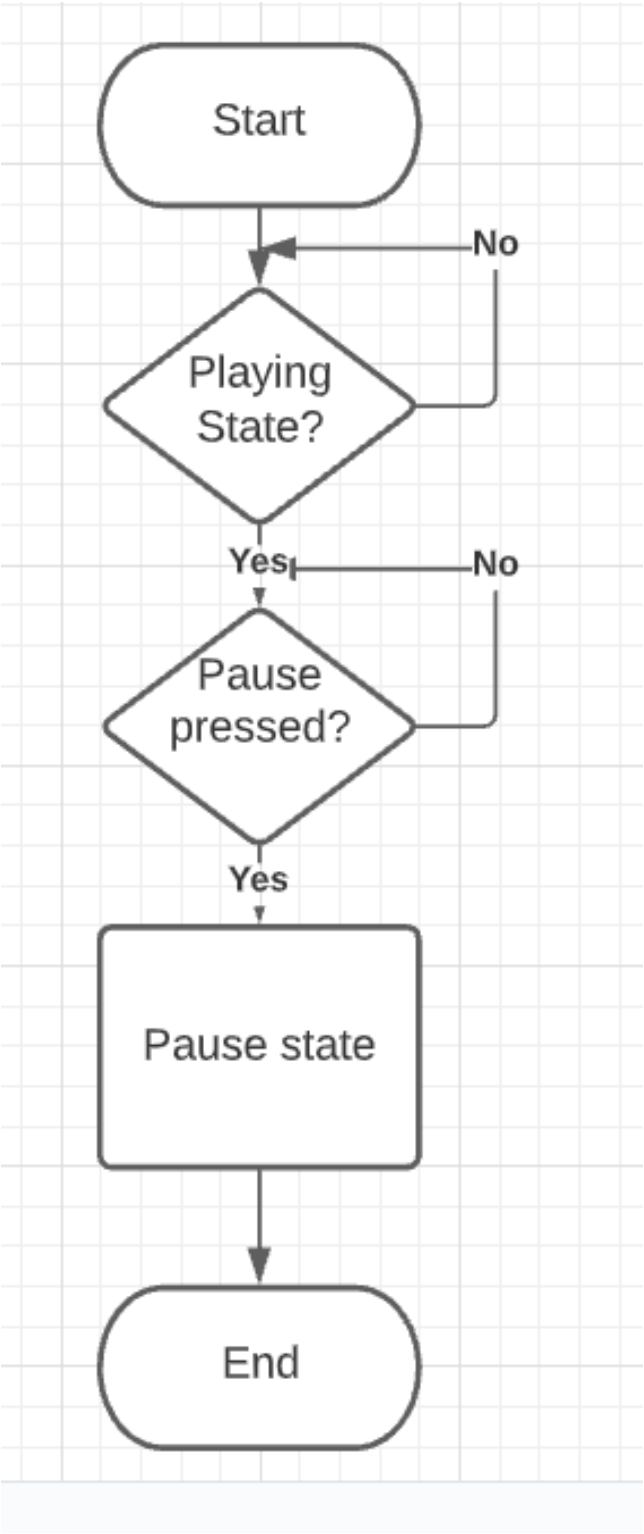
Paused

When pausing -



Here we check if the game is paused. If the game is in the playing state and the Pause button is pressed then the pause state is entered. Both of these need to be the case for the game to enter the pause state. If not in the playing state or the pause button hasn't been pressed it wont enter the pause state.

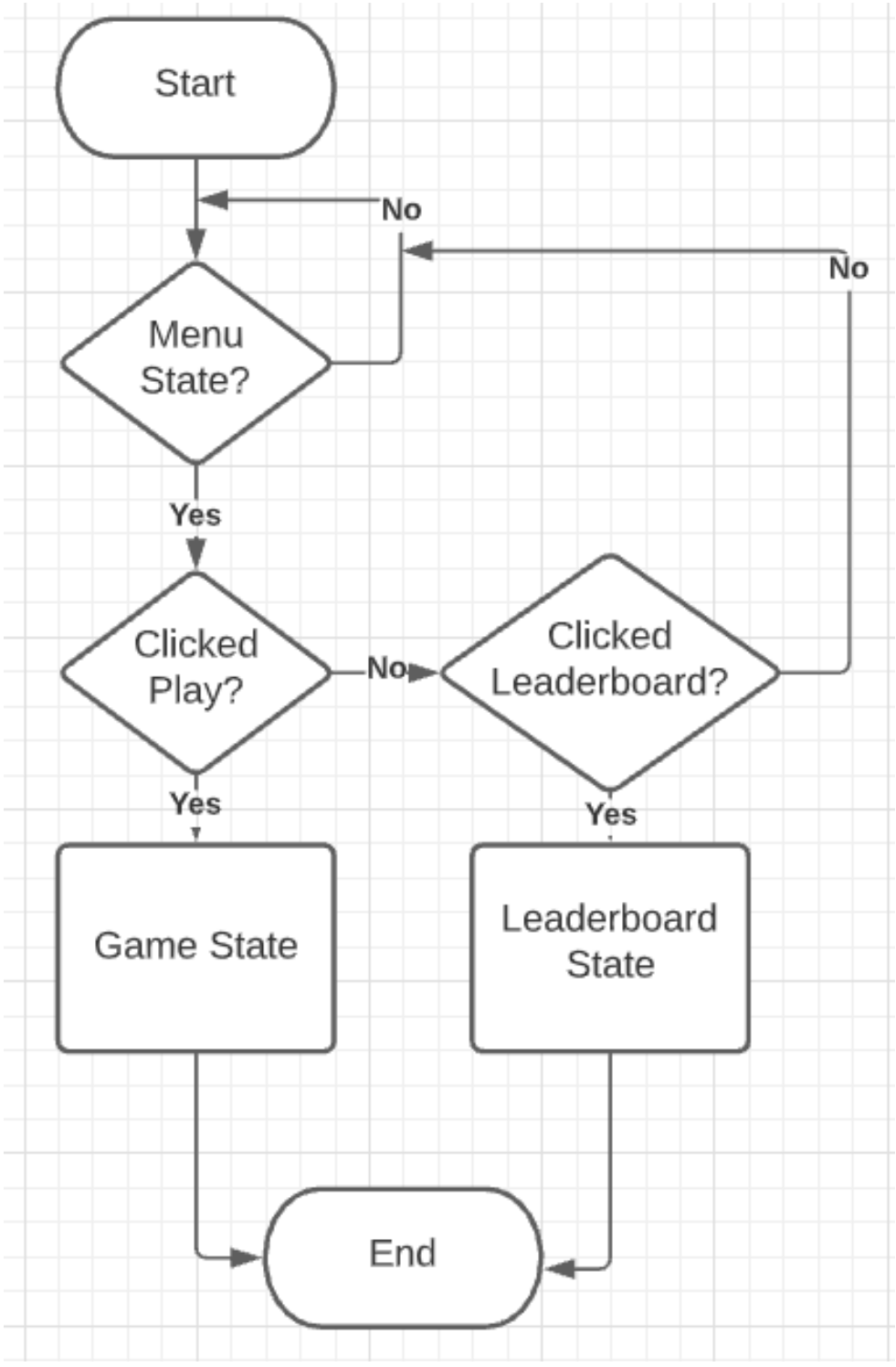
When unpausing -



This works the opposite of above When in the pause state and the unpause button is pressed the the game will be unpaused and enter the playing state.

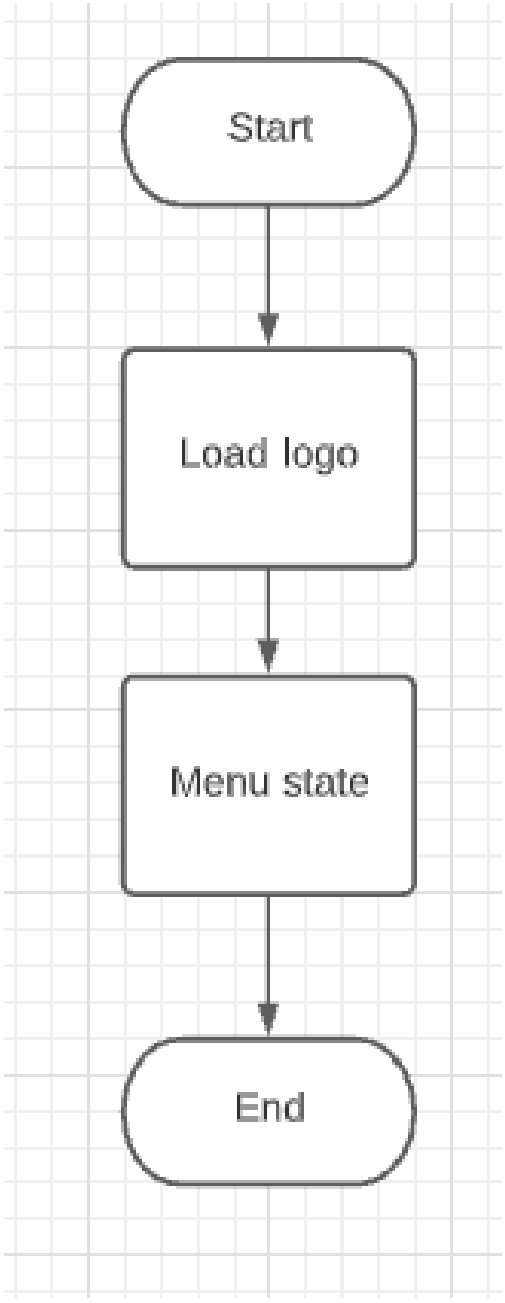
Menu:

Menu -



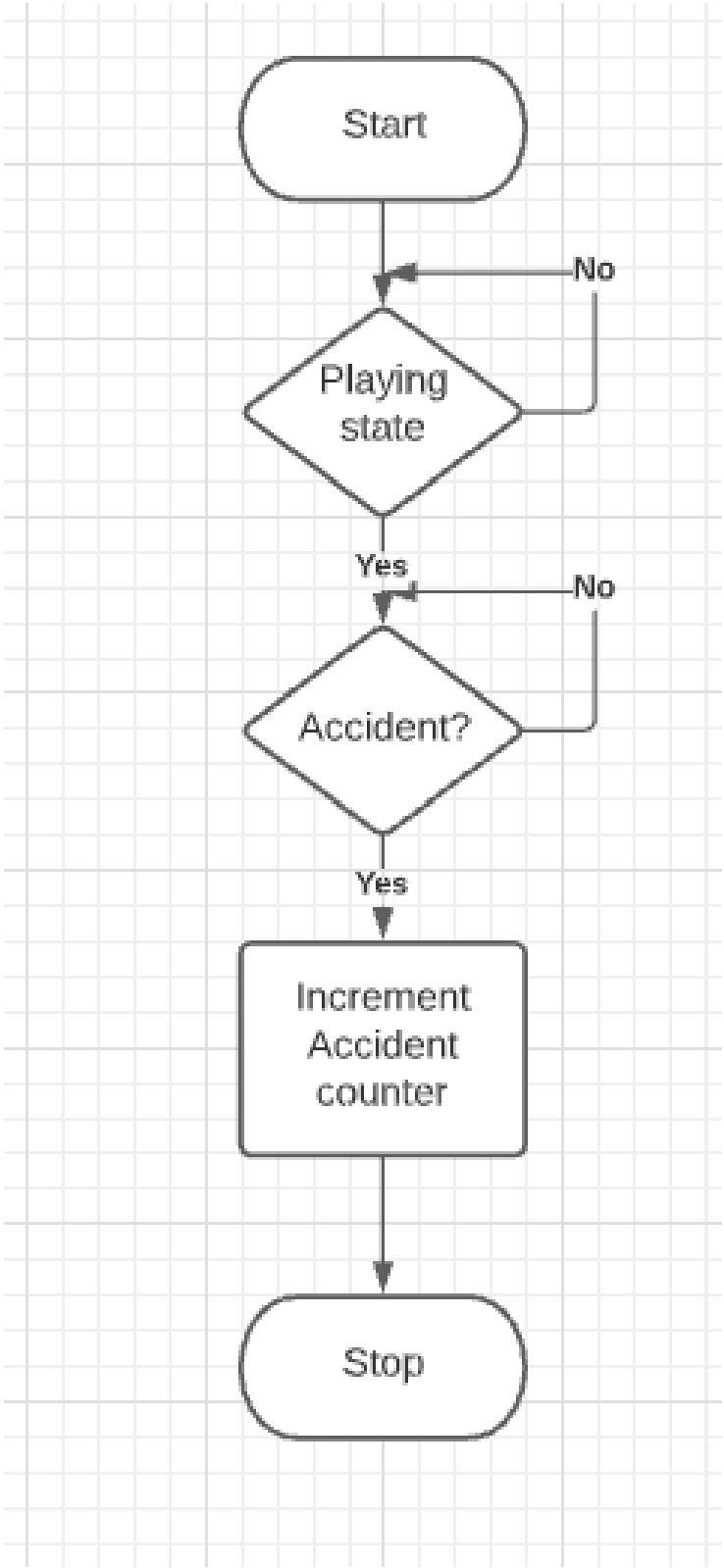
Here we check if we are in the menu state. This will update once a frame to check if we are in the menu state. If in the menu state and the play button has been pressed then we enter the game state. If in the menu state and the leaderboard has been pressed then we enter the leaderboard state. However if neither of these are pressed we iterate until one of the buttons are pressed. The flowchart then ends.

Startup:



To startup the game we use sequence to first load the logo, then the menu state and once this has been done the game is loaded and ready to begin and therefore we can then end the flowchart.

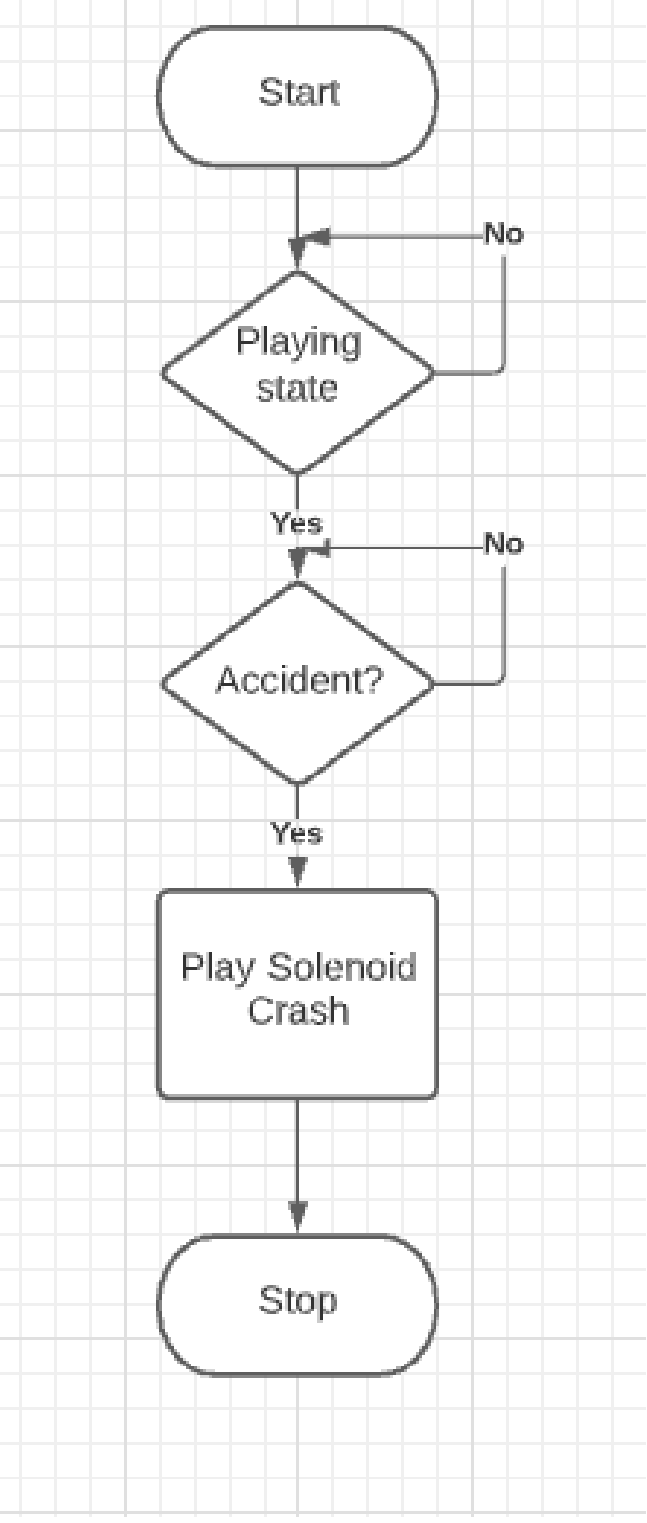
Accidents:



If in the playing state and an accident has occurred then increment the accident counter.

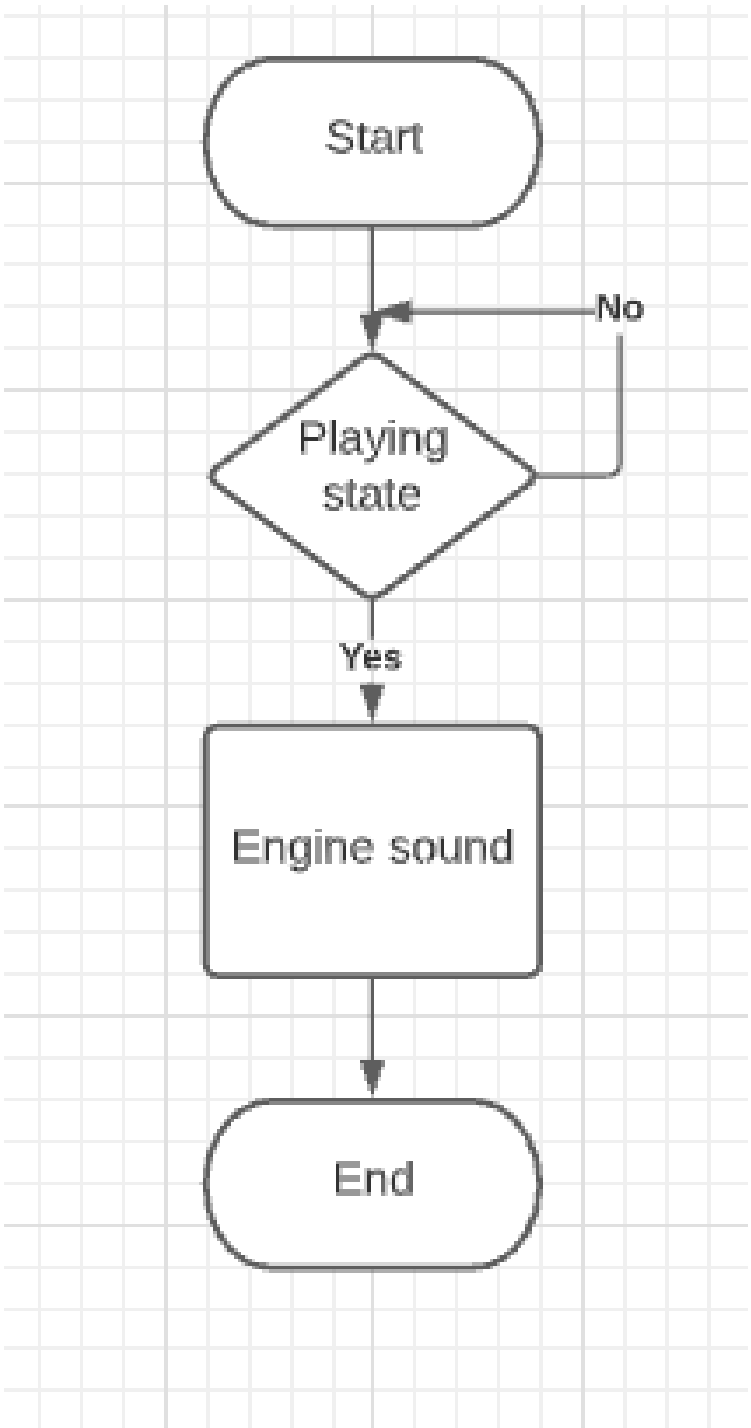
Sound:

Solenoid -



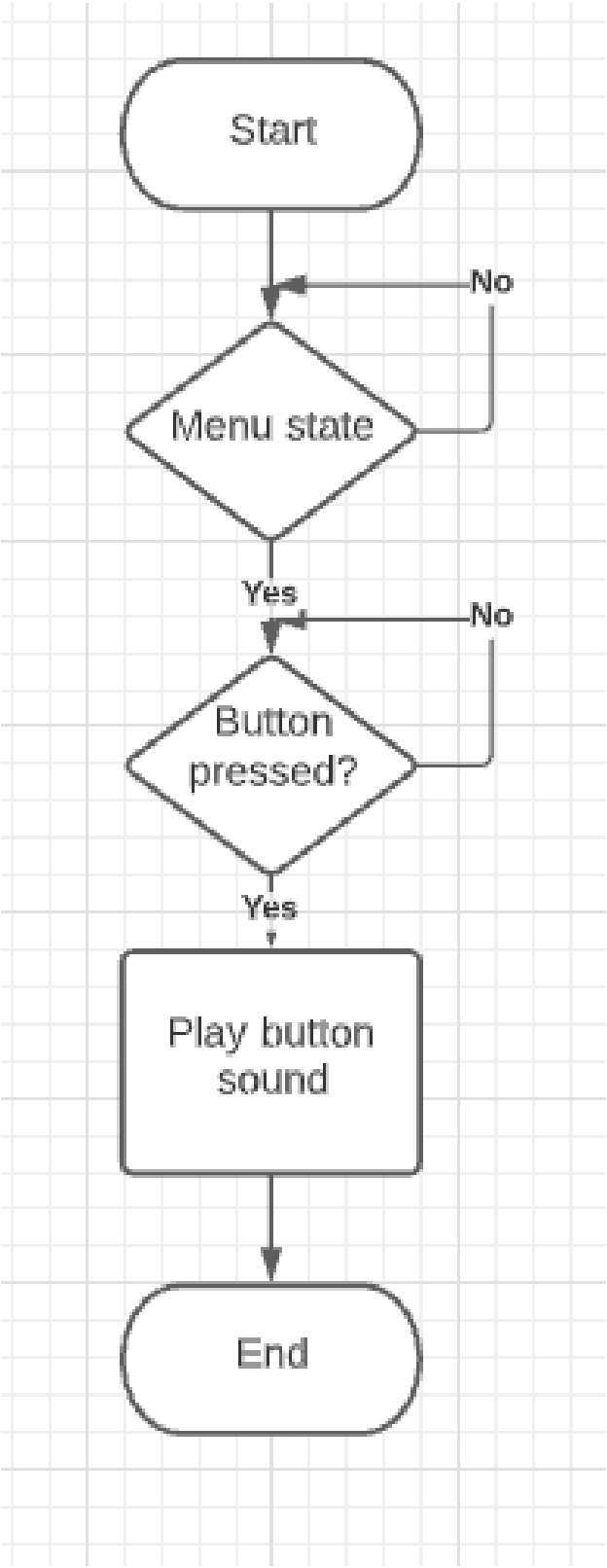
If in the playing state and an accident occurs then play the solenoid crash recording.

Engine sound -



If in the playing state then play the engine sound audio file. This creates an ambient environment when the sound is on in the game.

Button sound -



When in the menu state and a button is pressed play the audio that comes with a button being pressed. This will make the menu feel more interactive.

Usability Features:

- The entire game will be played on touchscreen and available on mobile devices meaning that it is available and usable to as large an audience as is possible. It also means the cost of hardware needed to play is low. Mobile devices are also owned by a significant amount of people meaning that many players will have access.

- The menu screen implements a feature to pause your game at any moment meaning it can then sit suspended until the player then wants to resume the game. This then means it can be played for a short amount of time without having to lose progress.

- Clear measures of the players score will be shown to them in menus and whilst they play so that they can see their number of accidents and for how long they have been driving for. This attempts to make sure that the user is provided with everything they need as they play without having the need to enter further menus or scrounge around in settings.

- All buttons will be clearly outlined and with text providing a description of the use of each button. This is important to make sure that the game is playable and useful for everybody. Unfortunately the limiting factor here is language. This game will only be in English and so other users may struggle with this. Therefore, a solution is to provide diagrams for the use of buttons.

- Play/Pause will be implemented using the > and || notation. This transcends language barriers. A Trophy can then also be used to access the high score screen. This is known as symbolic communication.

- Whilst in the playing state if car hitboxes collide then an accident will be shown by the accident counter incrementing and a vibration or solenoid noise depending on the level of device volume.

Validation Required:

- The only entry of data needed is to enter a name when there is a high score. This highscore will then be saved next to the name. Global highscores will be seen by many and due to the game being suitable for all ages all profanity will have to be unavailable for use. For this a profanity filter will need to be added for this input. There are large libraries of profanity available online for use in such a filter.

- Whitespace validation will also be added as a security measure for the databases of usernames and scores. Whilst none of this is harmful information it is still better to keep this as safe as possible as many users will value their privacy. If anyone is able to access the database it is also possible that they would attempt to add themselves in as the highest score. This isn’t wanted as then users will be able to cheat.

- Username length will need to be validated to make sure that people don't put too long a string.

- Character validation will be added so that special characters such as :\# cannot be added into a username. Only numbers and letters will be allowed to be entered here.

Alpha Testing

Test No	Test Details	Requirement No	Test Data	Expected Outcome
1	Testing the High Score leaderboard	11	newScore > highScore	New highscore
2	Testing the High Score leaderboard	11	newScore < highScore	No new highscore
3	Username Validation	Design - Validation	No profanity	Allowed
4	Username Validation	Design - Validation	Profanity	Not allowed
5	Username Validation	Design - Validation	Whitespace	Not allowed
6	Pausing the Game	14	Clicking Pause	Pauses the Game
7	Collision Detection	6	Cars collide	Accident counter increments, solenoid audio plays, Car resets to original position
8	Controls	12	Clicking Left of Center	Car turns to the left
9	Controls	12	Clicking Right of Center	Car turns to the right
10	Controls	12	Clicking the Center Bottom	Car brakes
11	Checking Hitboxes	N/A	Driving to the edge of the road	The car will stay within either kerb
12	Accident Counter	6	5 Collisions	The game finishes

				and goes to the GameOver state
13	Position Counter	1	Pass a car	Position decrements
14	Position Counter	1	Passed by a car	Position increments
15	Pause State	14	Clicking unpause	The game enters the game state once again
16	Driving Audio	N/A	Whilst playing and not in an accident	Car produces engine audio
17	Odometer	2	Whilst Accidents < 5	Odometer increases by a set value

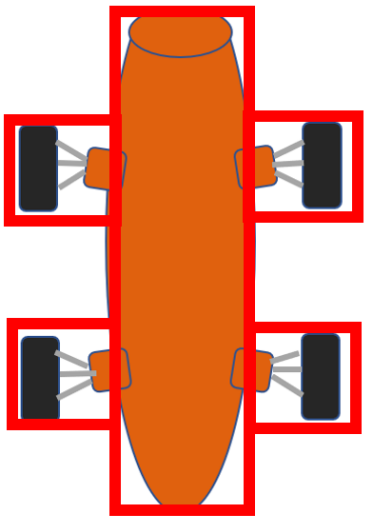
C. Developing the coded solution ("The development story")

Beginning Sprite designs and Menu layouts

Car Models:

Due to the previous section in which I began to look at algorithms that could detect if hitboxes were to overlap it seemed that this was most simple for rectangular hitboxes. Therefore the initial car design of a 1960's racer provides some problems.

Option One: Using multiple rectangles would result in something like this below. This reduces the amount of hitbox that isn't part of the car resulting in false positives of a collision even though there has been no visual overlap.



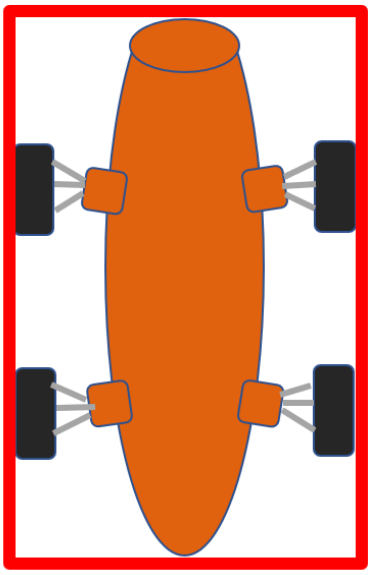
Advantages: Quite easy to be able to code as the only difference would be more rectangle variables needed.

Disadvantages: Not too sure how a sprite could be saved this way or if the sprite and the hitbox could operate independently. This

also comes with the problem that whilst simple there are still chances for false positives in some cases.

Option Two:

The most simple and efficient hitbox as just one rectangle. Very easy to code and when testing there are less variables to have to consider working meaning that the code is more bulletproof.

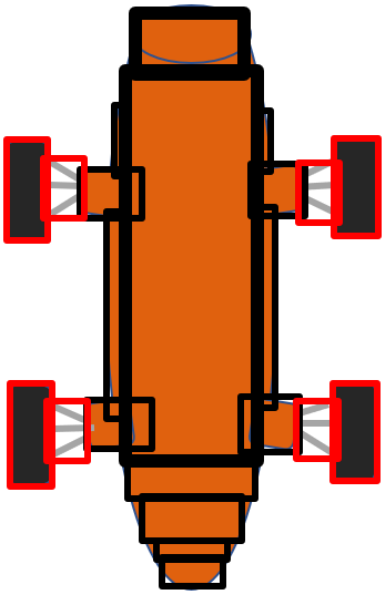


Advantages: Durability and simplicity.

Disadvantages: High chances of false positive collisions may make the game seem outdated and bugged. Likely to be frustrating to play with.

Option Three:

Almost a polygon this hitbox uses many rectangles together and will need large amounts of code to check for collisions between each and every rectangle here. It however, will result in a greater finished product.

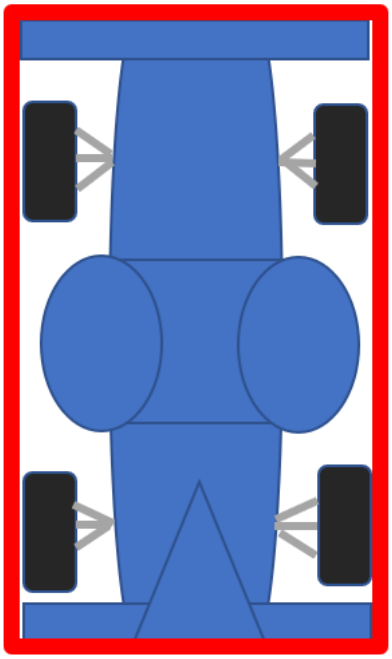


Advantages: Most accurate and reduces the chances of false positives almost to 0.

Disadvantages: Requires many boxes to make and therefore lots of comparisons all the time to check for collisions. This could make the game slow to run on devices that are at the minimum hardware requirements.

Option 4:

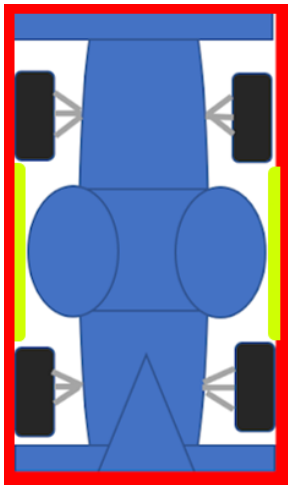
Instead of modifying the hitbox the car could also be modified. This would need to be into a way within which it could fit the shape of a rectangle. For example this could be something like below. This car could be refined and made into a sprite. Whilst not fitting the original Chicago Coin Speedway game this still does remain historically correct as it has the same widebody and large wings as a 1969 Formula 1 car.



Advantages: Both wings act as barriers for collisions meaning that the chances of false positives are almost zero. This is also done without the use of the many rectangles needed for there to be little chances of false positives such as in Option Three.

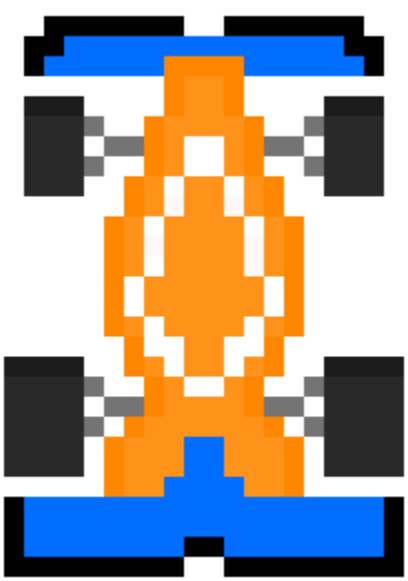
Disadvantages: Not directly the same as the cars from the original game. The changes here are the addition of the wings.

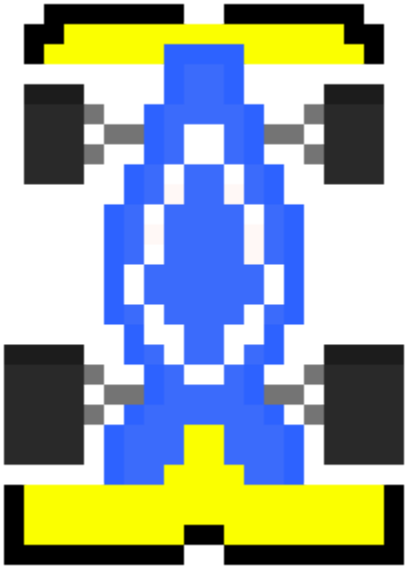
Choosing Option 4 seems the most beneficial here in both an ease of coding and how the finished product will feel to the user. It reduces the amount of operations needed and means that the game would likely run better on a low-end mobile device. It also creates less chances of false positives making the game feel polished.



Here the neon yellow represents the possibility of there being a false positive. There are very small chances of there being a false positive here due to the wings acting as barriers for the cars.

I then began to create my sprites using the online site piskelapp.com which allows you to create animated sprites which can then be imported into unity.





Here I created the car model which uses a Formula 1 inspired design in implementing the front and rear wings to act as barriers for a simple hitbox which can then be used in Unity. It is then possible to change tyre colours and create the illusion of movement. This gif is at 16fps with 6 frames total each with a different stage of tyre rotation. This creates subtle movement. Changing the colours of the car then allow for it to be used as an enemy car.

Testing the Collision Detection Algorithm:

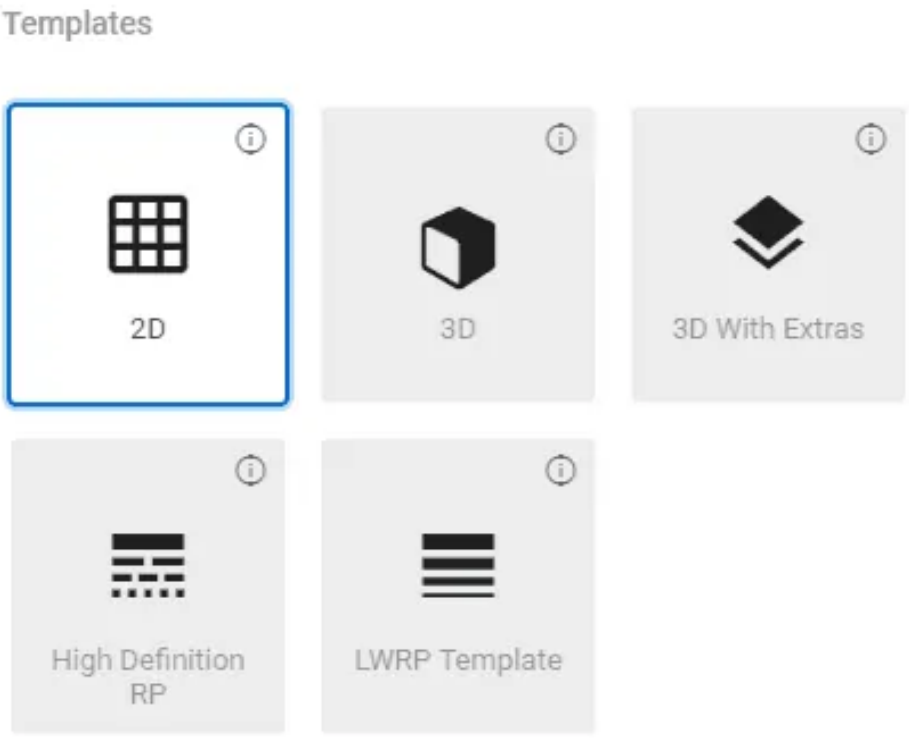
Using python I began to check that the algorithm would work before I began to think about implementing it. Checking my logic seemed important here.

```
def aabbcollision():
    #rect = [x, y, width, height]
    rect1x = int(input("Enter Rect1 x value: "))
    rect1y = int(input("Enter Rect1 y value: "))
    rect1width = int(input("Enter Rect1 width value: "))
    rect1height = int(input("Enter Rect1 height value: "))
    rect2x = int(input("Enter Rect2 x value: "))
    rect2y = int(input("Enter Rect2 y value: "))
    rect2width = int(input("Enter Rect2 width value: "))
    rect2height = int(input("Enter Rect2 height value: "))
    rect1 = [rect1x, rect1y, rect1width, rect1height]
    rect2 = [rect2x, rect2y, rect2width, rect2height]
    if rect1[0] < (rect2[0] + rect2[2]) and rect2[0] < (rect1[2] + rect1[0])
    and rect1[1] < (rect2[1] + rect2[3]) and rect2[1] + (rect1[3] > rect1[1]):
        print("Collision Detected")
aabbcollision()
```

It would only detect a collision under the parameters that I wanted. This was when multiple sides overlapped as is in the AABB Collision Algorithm.

18/03/2021:

Before finalising more sprites I decided to begin learning C# and Unity to find initial limitations of my idea and to create a roadmap of reaching the solution. I would use placeholder sprites whilst I develop the algorithms and logic and become comfortable with using Unity and then complete the graphics and audio to smooth the rough edges. This also seems easier to test when using placeholders in some situations and then further testing can be done once the visuals and audio are added. I installed Unity and created a blank 2D Game.



Importing placeholder assets and beginning to create Assets:

Here as a first step I began to create the scrolling background that would flank either side of the road and give the illusion of movement on the screen as the cars stay stationary. I needed to create a C# script to allow for the background to repeatedly scroll. I found similar cases online and modified their own programs to fit my own. This reduced programming time and then enabled me to continue developing.

Below is a script for a floating background asset that I can modify to make the road and the scenery scroll down the screen.

Implementing a Scrolling Background 04/04/2021: (Requirement No.4)

This was saved as ScrollBackground.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ScrollBackground : MonoBehaviour
{
    Renderer rend;
    public float scrollSpeed = 10.5f;

    void Start()
    {
        rend = GetComponent<Renderer>();
    }

    void Update()
    {
        float offset = Time.time * scrollSpeed;
        rend.material.SetTextureOffset("_MainTex", new Vector2(0, offset));
    }
}
```

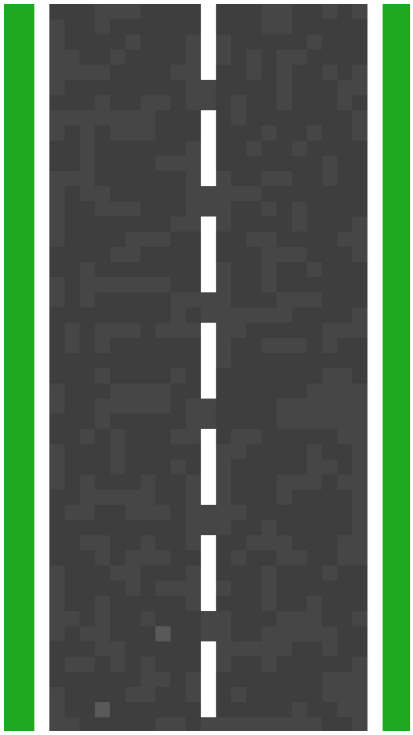
" First we declare a renderer which will give us access to our Quads(Asset) material. Then we set up a scrollSpeed and add a xAxis boolean. This is so we can control which direction we will scroll in. In the start method we will get our renderer component and assign it to rend. Then finally in our update method we will calculate our offset which takes our time and multiplies it by our scroll speed."

This script works by finding the distance that the background sprite has travelled, calculating this with the scrollSpeed and time variables. ScrollSpeed is a variable that can be modified when within unity as it is defined as public. This makes it possible to find the most realistic scrolling speed and easy to modify the background. This script can then be applied to an empty quad. This is important because it means that a sprite can be used to make the material of the quad and sprites can easily be interchanged. This is important as it means if further changes are made to the visuals of the game in the future the actual gameplay integrity won't be changed.

From here I then had a choice to make. In my Analysis and Design I had imagined a landscape driving game however when importing sprites into my quad I began to realise that due to changing aspect ratios among mobile devices it would introduce another layer of complexity. My phone has a 19.5:9 aspect ratio. This means that there will be a considerable amount of blank screen bars on each side.The combination of these problems made me realise that a solution here was to make the game be used as a portrait. This also creates a feeling of the game being casual and more likely to be played often. A

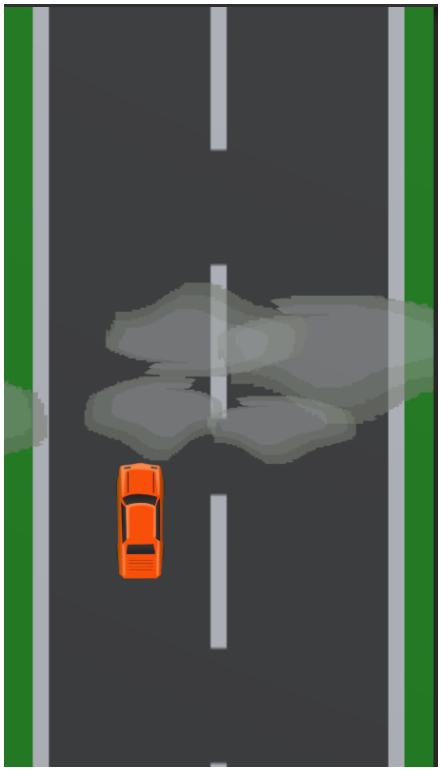
portrait driving game then also creates another thought in terms of controls. A casual yet competitive portrait game can often be played using only one hand unlike two on landscape to either be used as controls or support the device.

Portrait Background Sprite:



Produced for a 16:9 Portrait screen which is the standard background aspect ratio for a mobile device. This makes it usable for a large majority of mobile users. If users with other aspect ratios wanted to play the game then this could be elongated. This also creates more issues as they would have more of an advantage as they would see the opponent cars earlier if their screen is longer. This may mean that although highscores are local it will be easier to achieve a higher score.

How this looks in Unity:

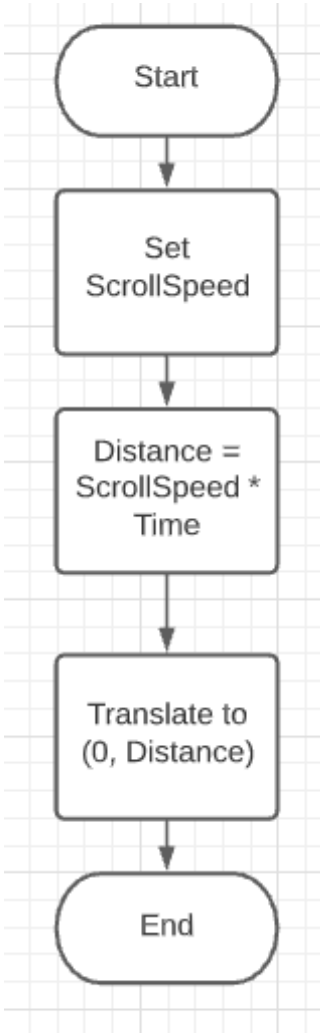


How this looks when played:

Testing the Scrolling Background Functions:

Test No	Test Description	Expected Outcome	Outcome	Success
1	When applied to a quad does it Scroll	It will scroll	It scrolled	☑
2	When scrollSpeed is 0	No scroll	It didn't scroll	☑
3	When scrollSpeed is negative	Scrolls upward	Scrolls upward	☑

Flowchart explaining the Scrolling Background:



First we set a public ScrollSpeed variable which determines how fast the background will scroll. The distance that it will have scrolled is then calculated and then the background translated to that distance every frame. (In the update function)

Creating the User's Car 06/04/2021

First I inserted my car sprite and resized it to fit the road. From there I gave it it's necessary physics properties to be able to move. This included:

- A RigidBody 2d(Kinematic) which allows a sprite to behave in a convincing way by applying forces. It also helps in problems such as collision detection.
- A BoxCollider 2d which uses a rectangular hitbox to detect collisions. This means you can create scripts that make use of the collision detection to make instructions be performed on an event of a collision.

Once these were added I used this C# Script saved as Player.cs In the code it saves the Rigidbody2d as a variable named rb. It then defines a public movement speed variable that can be changed when inside Unity allowing for gameplay balancing in the future.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5
6   public class Player : MonoBehaviour
7   {
8       public float CarSpeed;
9       Vector2 position;
10
11      void Start()
12      {
13          position = transform.position;
14      }
15
16      // Update is called once per frame
17      void Update()
18      {
19          position.x += Input.GetAxis("Horizontal") * CarSpeed * Time.deltaTime;
20          position.x = Mathf.Clamp(position.x, -2f, 2f);
21          transform.position = position;
22      }
23      void OnCollisionEnter2D(Collision2D col)
24      {
25          if (col.gameObject.tag == "Enemy")
26          {
27              Destroy(gameObject);
28              SceneManager.LoadScene("Menu");
29          }
30      }
```

In the code first we make sure to import the UnityEngine.SceneManagement to allow the changing between scenes in Unity. This means that when there is an accident we can go back to the main menu. We create a public float variable known as CarSpeed which is used to define the speed of the Player. This is also changeable as it is public. We then set Vector 2 as a variable called position. A Vector2 is a simple way to map coordinates in Unity as having an x and a y value and is used in 2d programming.

In our Start function which runs at the beginning of the Main Scene then we set the position of the Player to its original position. This is important as it means that every time the game is restarted the car will return to its original position and therefore the scene will be reset.

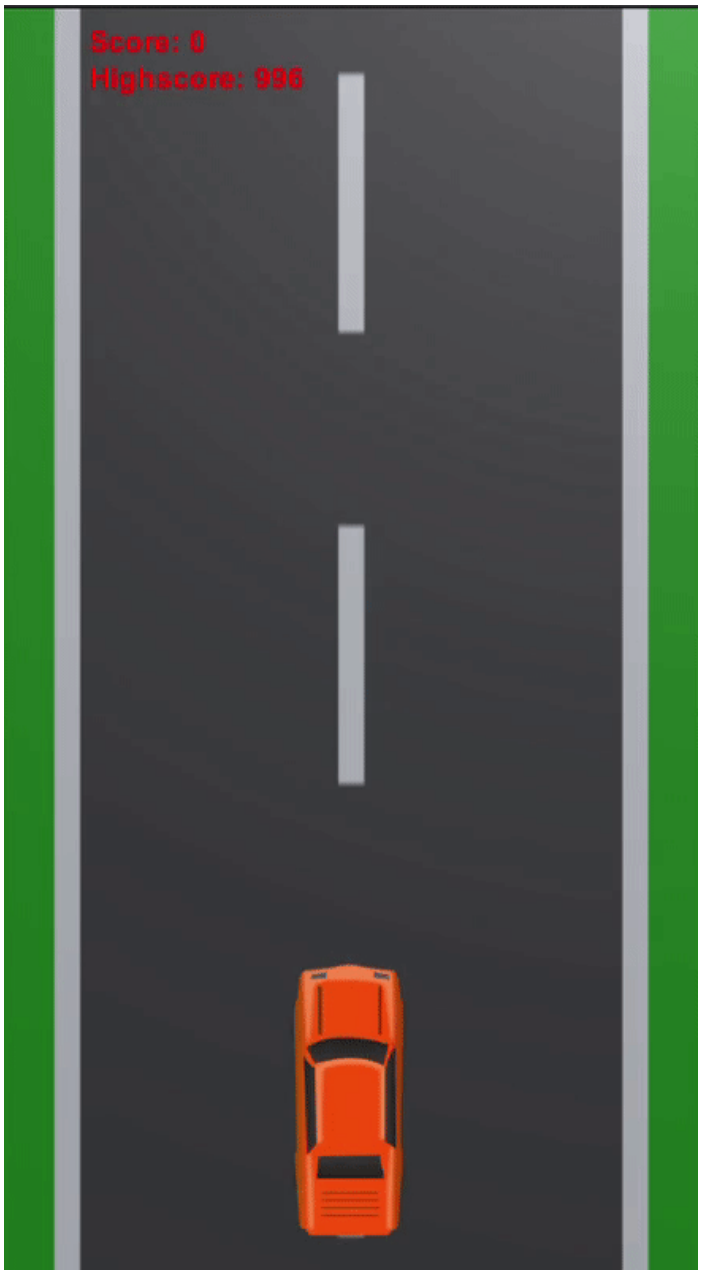
In our update function which runs once a frame (more than 40 times per second) we set the x coordinate of the Player gameObject by using GetAxisRaw as a method to find inputs for left and right. This enables me to use the arrow key controls whilst in development. I then clamped this position by using the Mathf function which is used in this context to keep this GetAxisRaw value between the two bounds -2f and 2f. The purpose of this is to stop the user from exiting the screen with their movement. transform.position = position is important as this then sets this coordinate as the new position so then it can be used again the next time the Update() function runs.

To calculate an accident we use the Box Collider 2d applied to the Player's car. When there is a 2d collision between a gameObject with the tag "Enemy" then the Player's gameObject will be destroyed(meaning it can be reset for the next game) and the game will change states using the SceneManagement library to go back to the Menu screen. This allows the user to then have a break and enter another game. Testing the player's movement as a video.

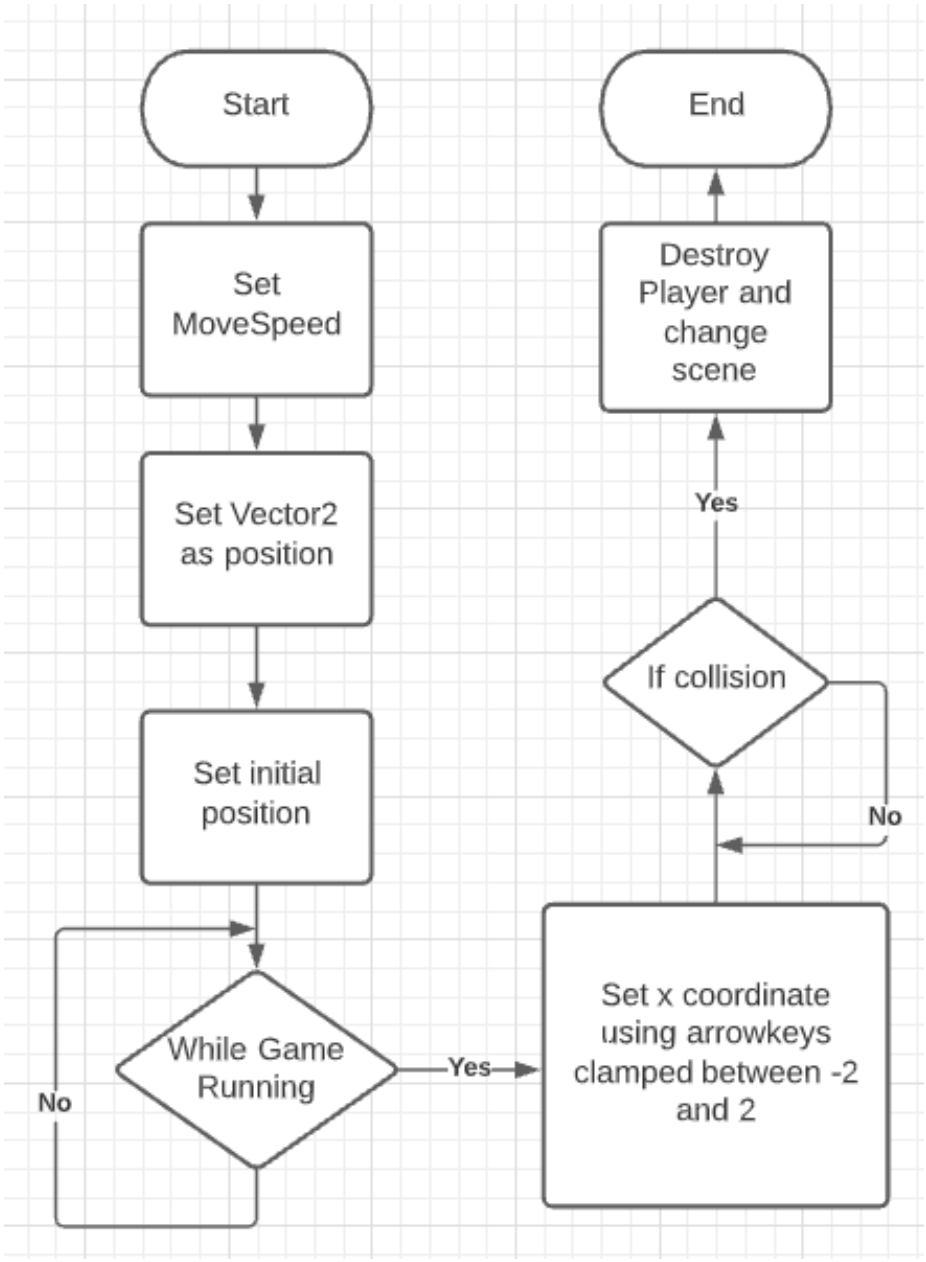
Test No	Test Description	Expected Outcome	Outcome	Success
1	Test with Left Arrow	Moves Left	Moved left	☑
2	Test with Right Arrow	Moves Right	Moved right	☑
3	Left and Right Arrow	No net movement	Didn't move	☑
4	Up Arrow	No net movement	Didn't move	☑
5	Down Arrow	No net movement	Didn't Move	☑
6	Testing Wall boundaries	Stays within camera	Stays within Camera	☑
7	Collision with an Enemy	Goes to the Menu Scene	Goes to the Menu scene	☑

This was successful as the car stayed within the expected bounds and replied to the tests as expected. It's movement is simple, reducing the chances of further problems, complications or bugs.

Video Evidence:



Flowchart explaining the logic behind the movement:



I had to evaluate my project during development. Requirements had to change and you can see these below.

Time Constraints 07/04/2021:

At this point I came upon my first problem due to time constraints. In my requirements, analysis and design I had visualised an amount of enemy cars that would continuously come down from the top of the screen and for there to be a position indicator and a set number of enemy cars. However this creates a number of issues:

1) If you have a set number of cars if the user can pass these cars then they can no longer have an accident and then the game continues infinitely.
2) If you have a position indicator it will quickly become obvious that the player will soon get into 1st place for the majority of the game if they have the required skill to dodge cars. If this is the case then the need for a position indicator doesn't exist.
3) Having 5 accidents raises issues like above but also creates an idea where the game becomes long and tedious. Having 5 lives in an endless racing game means that many players could make games last for a long amount of time and quickly become boring. This also means that there isn't likely to be a replayability factor.
4) If I was to have a global high score this would mean using Google Play Leaderboards or a central high score database. These both require access to the internet whereas all the other components of the game are created in such a way that this is an offline endless racer.
5) Because of all these problems I felt that coding such things would both drain a large amount of time for the project on functions that provide little or no effect to the actual game itself other than providing an identity crisis. Time would be better spent on making sure that there is good gameplay and then from there working more on sprites to create a professional looking game. This also creates more time available for testing which is more important in my eyes than complicating the gameplay.

Because of this I feel that some of the requirements will have to be cut or modified and more time spent on other sub-requirements. Many sprites I aimed to make myself may have to be shelved and other created assets used. This means I aim to create the logic of the game. Sprites can be swapped in at a later date after development. This means the game will be working to the set requirements.

From here I changed some of my requirements and adapted to a smaller amount of time.

No.	Requirement	Description and Explanation	Reference
1	Odometer	This provides the information for how long the car has driven for. This will be the score that allows the user to know when they are close to beating a high score.	A feature from Chicago Coin Speedway.
2	Player Controller	Give the user movement on screen to enable them to dodge enemy cars for as long as possible	An updated controller for a portrait orientation.
3	Accident	When there is a collision the game will go back to the menu scene in which the user can restart the game.	A feature from Chicago Coin Speedway.
4	Moving Background	Provides the illusion of movement and distance travelled. Also means with a moving background the game will seem more dynamic and interesting.	A feature from Chicago Coin Speedway.
5	Opponent Cars	Opponent Cars will be deployed to provide obstacles to the player and difficulty in overtaking. If you collide with an enemy car then the game ends.	A feature from Chicago Coin Speedway.
6	Menu	This provides an interface for the user to enter the game.	A feature from Chicago Coin Speedway.
7	Graphics	Retro Graphics will make the game look similar to how it looked in 1969. With car shapes and a moving background in a retro art style it can keep a large target audience and be suitable for all.	A feature from Chicago Coin Speedway.
8	Sound	Ambient Music	A feature from Chicago Coin Speedway.

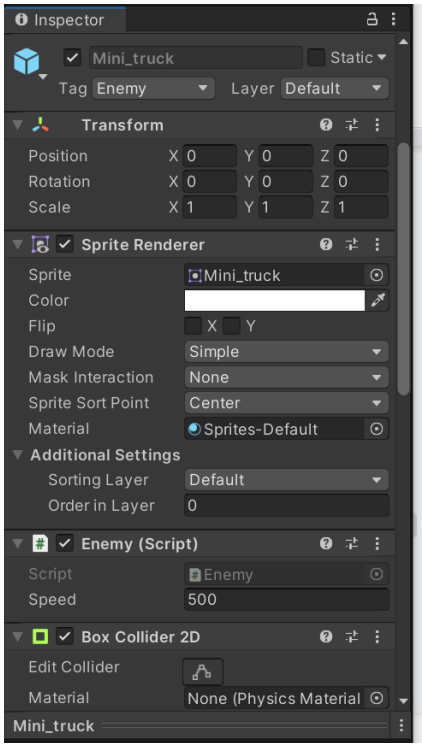
In these revised requirements the importance is placed on gameplay and is where the majority of the time will be placed.

OPPONENT CARS 11/04/2021 (REQUIREMENT 5):

Here I planned to use enemy car sprites and a modified version of the scrollBackground code to place many enemy cars off the camera and have them travel down at slow scrollSpeeds so that the user can dodge them. I then need to find the correct balance of difficulty. I may then also create another script in which as the time goes on the scrollSpeed of these enemy car's increases until the driver crashes.

I needed to create two scripts here. One for the creation and random spawn of the Enemy cars and One for the movement of those cars. Before I could create these scripts because I would be using multiple of the same sprite to act as the Enemy I had to create a prefab.

Prefab:



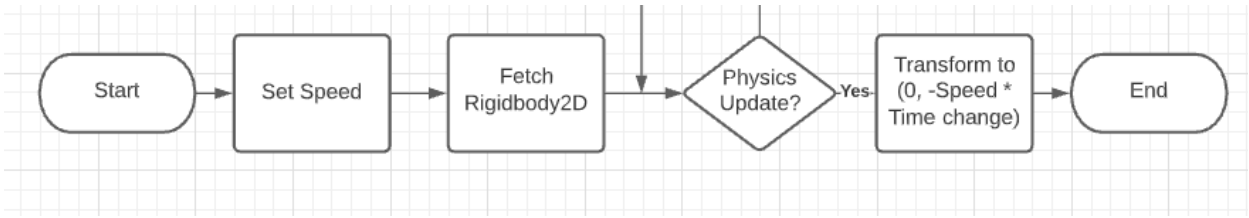
A prefab is a special type of component in Unity that allows a gameObject to be instantiated multiple times whilst having the same properties. This means that when I apply the Enem tag to this prefab all instances of the prefab will have this same tag. This also applies to scripts and components. This allowed me to create an Enemy script and to add a Box Collider 2d and a RigidBody 2d.

The first thing I did was create the Enemy.cs script. This is used to act as the movement of the Mini_truck prefab.

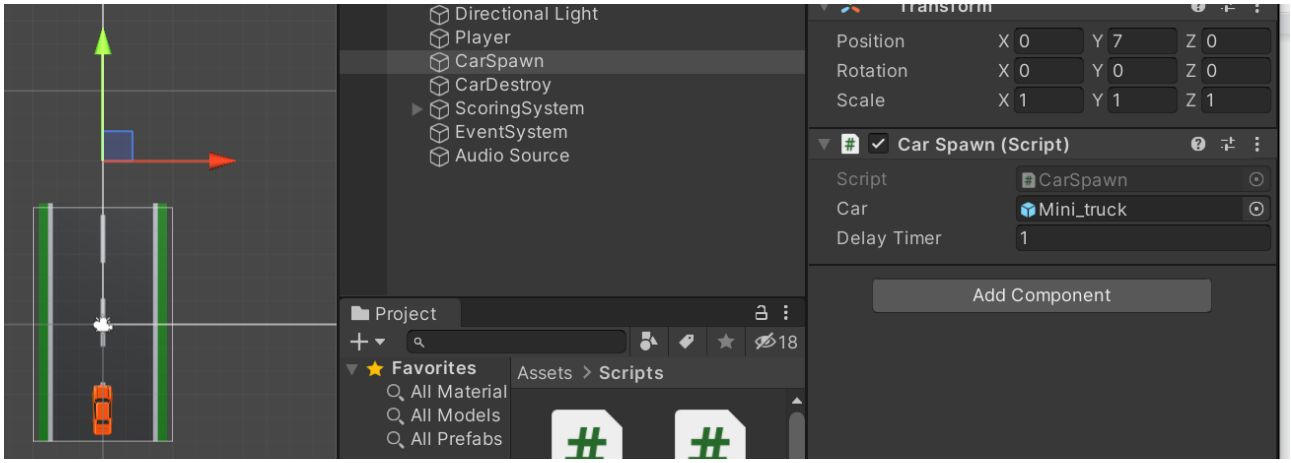
```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Enemy : MonoBehaviour
6   {
7       public float Speed;
8       private Rigidbody2D rb;
9       // Start is called before the first frame update
10      void Start()
11      {
12          rb = GetComponent<Rigidbody2D>();
13      }
14
15      void FixedUpdate()
16      {
17          rb.velocity = Vector2.zero;
18          rb.velocity = new Vector2(0, Speed * Time.fixedDeltaTime * -1);
19      }
20  }
```

First I created a public float Speed which is the Speed of the Enemy cars. This is public so it can be changed from within Unity. In the Start function which is used at the beginning of the Main Scene then we access the RigidBody2d component applied to the Enemy car. This means that all kinematics and velocity are then available for use.

In the FixedUpdate function (Similar to the Update function however the Update function runs once a frame whilst the FixedUpdate function is used in Physics manipulation and runs every time the Physics is changed. This makes the code more efficient as it isn't running for no purpose.) We set the velocity of the object to zero and then make it transform in the y-axis by using the Negative Car Speed (So it scrolls down) multiplied by Time.fixedDeltaTime. Time.fixedDeltaTime means the time that has passed since the last Physics Update. This means every physics update the car travels down at a constant velocity.



Then I needed to spawn the Mini_truck Prefab with the Enemy script applied. So I created two empty objects in the hierarchy to act as reference points. I named one CarSpawnPosition which would have the CarSpawn.cs applied to it. Once an Enemy car prefab spawned then the Enemy Script would make it scroll down.



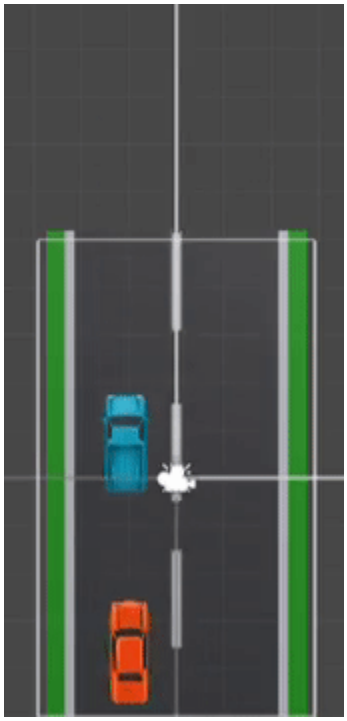
The CarSpawn empty object was placed above the screen and given the Car Spawn Script applied to it.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CarSpawn : MonoBehaviour
{
    public GameObject car;
    public float DelayTimer;
    float timer;

    void Start() {
        timer = DelayTimer;
    }

    void Update() {
        timer -= Time.deltaTime;
        if (timer <= 0)
        {
            Vector3 CarPos = new Vector3(Random.Range(-2.0f, 2.0f), transform.position.y, transform.position.z);
            Instantiate(car, CarPos, transform.rotation);
            timer = DelayTimer;
        }
    }
}
```

In CarSpawn.cs I set a public GameObject as the car which is important as it means that you can drag in the prefab that will be spawned in as you can see above from the CarSpawn gameObject. I then made a public float variable named DelayTimer that will act as the Delay between the spawn of each car. This means it can be used to change the difficulty of the game. I then made an empty float variable called the timer.

When the game starts I set the timer equal to the DelayTimer meaning a car will not spawn initially. In the update function each frame then the timer value will have the time that has passed between each frame taken away from it. The purpose of this is to count down until the DelayTimer reaches 0.

Then there is an if statement where if the timer reaches 0 (The DelayTimer has counted down to 0.) then the position of the car is defined as a Vector 3 known as CarPos which spawns the car between x values -2 and 2. This keeps it within the boundaries. The rest then makes sure it spawns at the CarSpawn gameObject location. The timer is then reset to the DelayTimer and will count down again. This means cars randomly and regularly spawn every DelayTimer between -2 and 2x.



As can be seen here the Enemy Car spawns above the Main Camera using the CarSpawn.cs and then scrolls down using the Enemy.cs script.

I then ran into a problem though that when these cars were being created they wouldn't be destroyed and would continue moving with the Enemy.cs script downwards. This meant that they were still loaded in and so a long game with many car instances would begin to lag and slow down. This means that the minimum hardware needed for my project would increase massively. Therefore I needed to create a way to destroy the Enemy cars.

Testing the opponent cars:

Test No	Test Description	Expected Outcome	Outcome	Success
1	Does the car spawn where it is expected to.	Car spawns off-screen	Car spawns off-screen	☑
2	Does the car travel at a set speed and can this be changed in UnityHub	The car travels at a set moveSpeed public variable	The car travels at this moveSpeed public variable	☑
3	Does it despawn or teleport once it has passed the bottom of the screen.	It will teleport back up to the top of the screen.	It does not yet do this and I will need to create destroy.cs script further on in the project.	☑

Menu Scene 09/04/2021 (Requirement 6):

First I created another scene and labelled it Menu. The Gameplay scene had been saved as a scene named Main.

1) I Imported a canvas UI element into the scene and resized it to fit the main camera. (What the screen sees)
2) I then imported a button UI element into the scene. This will have the purpose of being a button that says (Start Game) and then onClick will open up the Main scene.
3) I edited the Button Text and resized and placed the button in the centre of the screen.
4) From here I placed a placeholder sprite onto the Canvas which will later be replaced with a start menu screen. It will have the Chicago Coin Title and pictures of cars racing behind the text and button.

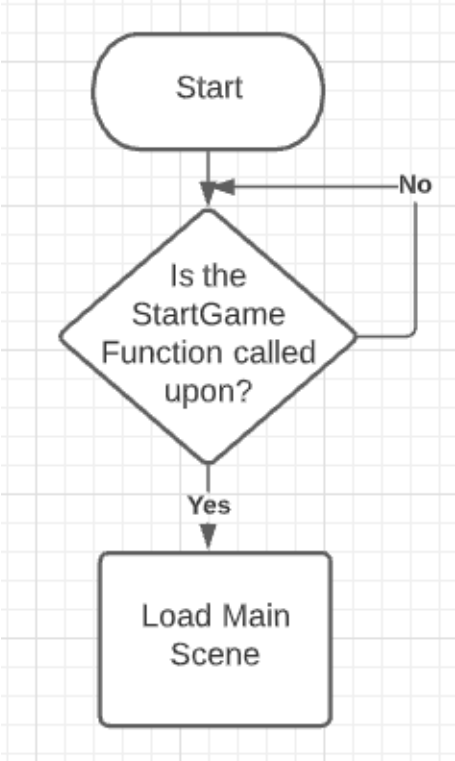
Here I created a C# Script that will be used to make sure that when the button is clicked the Main scene will be loaded. This is saved as UI.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5   using UnityEngine.UI;
6
7
8   public class UI : MonoBehaviour
9   {
10
11      public void StartGame()
12      {
13          SceneManager.LoadScene("Main");
14      }
15  }
```

This code works by clicking the Start Game button it loads the Main Scene which means that gameplay then begins. From here I then assigned this script to the canvas and assigned it to the button.

At this point the game has a moving car, scrolling background and Start Menu. The most important parts before I test and finalise the Graphics and Audio are the Odometer, Accident and Opponent Cars.

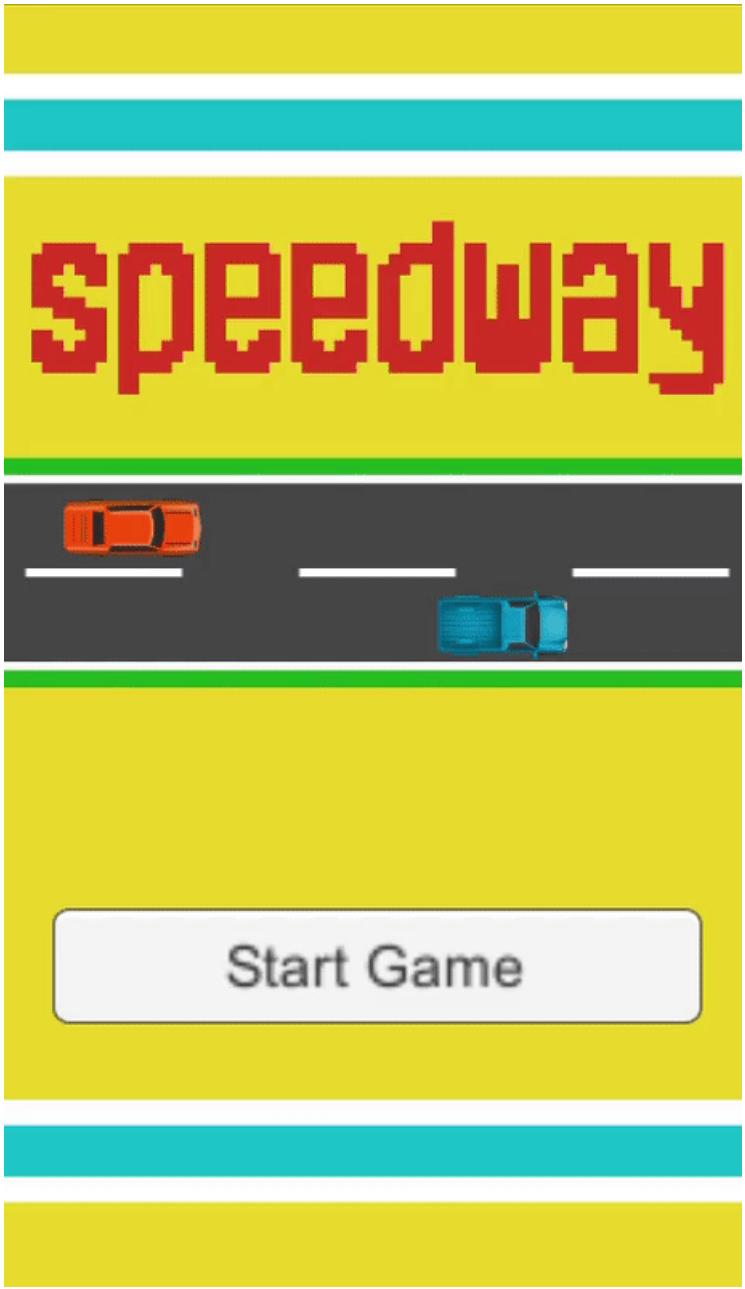
A Flowchart explaining the UI.cs Script:



Testing the Menu Scene:

Test No.	Test Description	Expected Outcome	Outcome	Success
1	Clicking the Start Game button	The Main Scene loads	The Main Scene loaded	☑
2	Clicking on a Boundary of the Start Game button	The Main scene loads	The Main Scene loaded	☑
3	Not Clicking on the Start Game button	The game stays in the Menu scene	The game stays in the Menu scene	☑

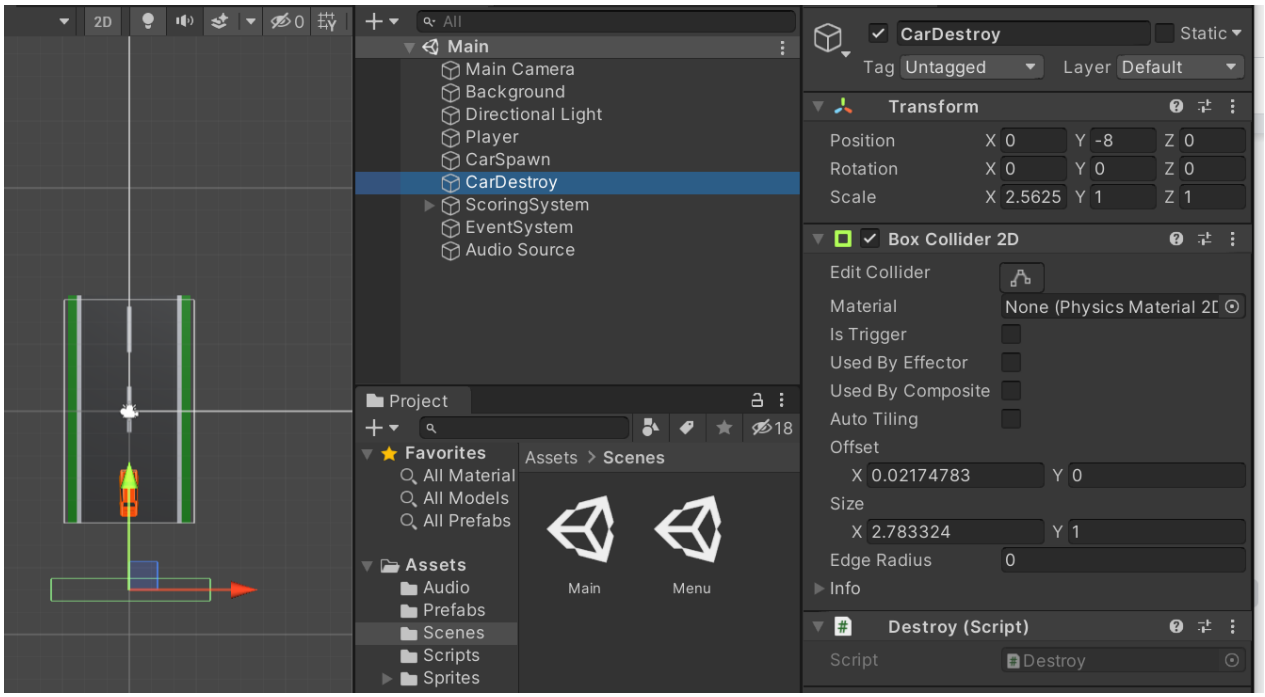
How the Menu looks to the player and it's usage:



When the Start Game button is clicked then it enters the Main scene and the game begins to be played again.

Opponent Cars Revisited Under Changed Requirements 20/04/2021 (Requirement 5):

I had the Enemy Cars spawning and travelling down however they weren't despawning and would still be taking up memory allocations. Therefore I created another Empty Object similar to the CarSpawn gameObject and called it CarDestroy



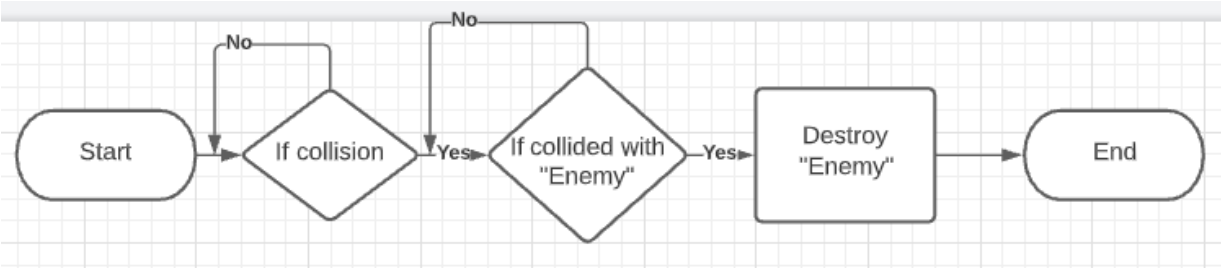
Here I gave it a Box Collider 2D for the purpose of being able to find out when an Enemy car collided with the CarDestroy gameObject. I placed this just below the Screen to minimise the resources needed for the game. The green outline simulates the Box Collider2D hitbox. Therefore this was wide enough to cover the area at which the enemy cars spawn in.

I then created a script called Destroy.cs with the purpose of using the BoxCollider2D On collision event to destroy the Enemy gameObject.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Destroy : MonoBehaviour
6   {
7
8       void OnCollisionEnter2D(Collision2D col) {
9           if(col.gameObject.tag == "Enemy")
10          {
11              Destroy(col.gameObject);
12          }
13      }
14  }
```

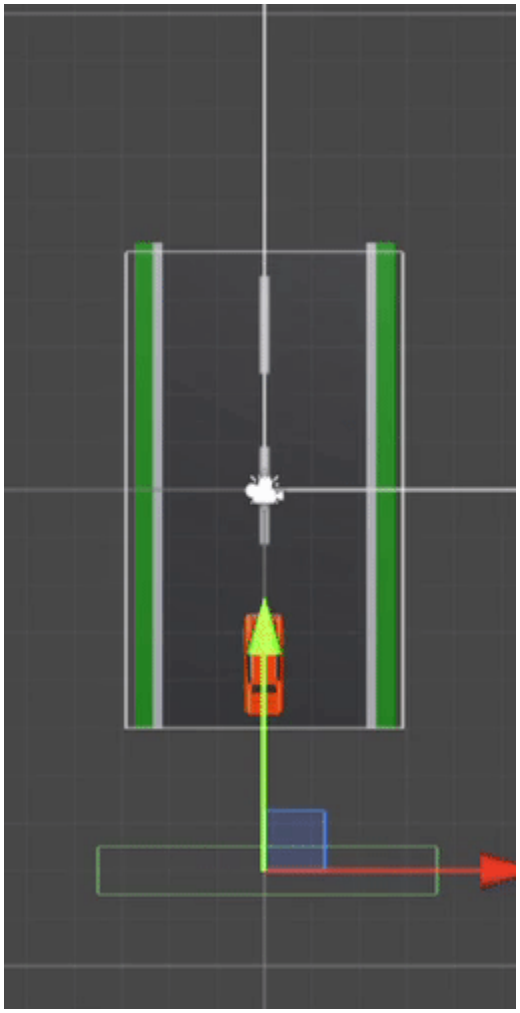
This code uses the OnCollisionEnter2D function which only runs when there is a collision. If the collision is between the CarDestroy gameObject and between any gameObjects with the tag “Enemy” then it will Destroy the gameObject which collides with the CarDestroyObject.

Flowchart -



Test No	Test Description	Expected Outcome	Outcome	Success
1	Cars get destroyed when colliding with CarDestroy	“Enemy” gets destroyed	“Enemy” gets destroyed	☑

The Opponent Cars Finalised -

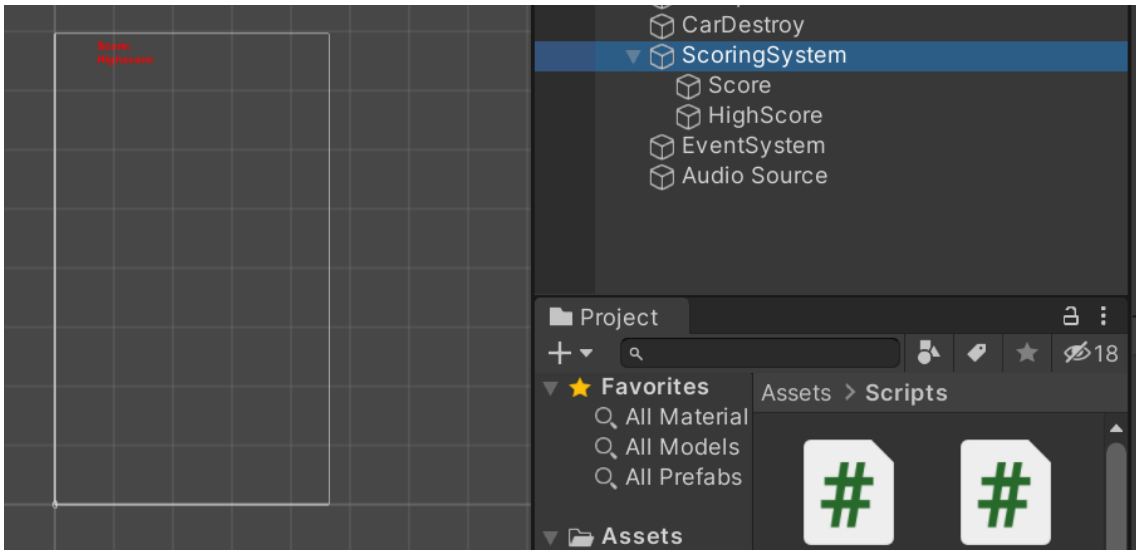


The Enemy Car prefabs now spawn in, scroll down and despawn making my game efficient by reducing the amount loaded in outside of the main camera.

ODOMETER 22/04/2021 (REQUIREMENT 1):

Whilst the game is in the game state a static timer will begin that resets every time the game state is accessed. This timer will continue until there is an accident and the score at that point will then freeze. If this score is above the current saved highscore then it will replace the high score. Whilst the game is being played the score will be visible in the top left of the screen.

To use text I would need to create a UI element. Therefore I created a canvas named ScoringSystem and added two pieces of Text. One named Score and the other named HighScore. These will be used to show the score and highscore to the user in the Main state.



From here I then created a script called Scoring.cs -

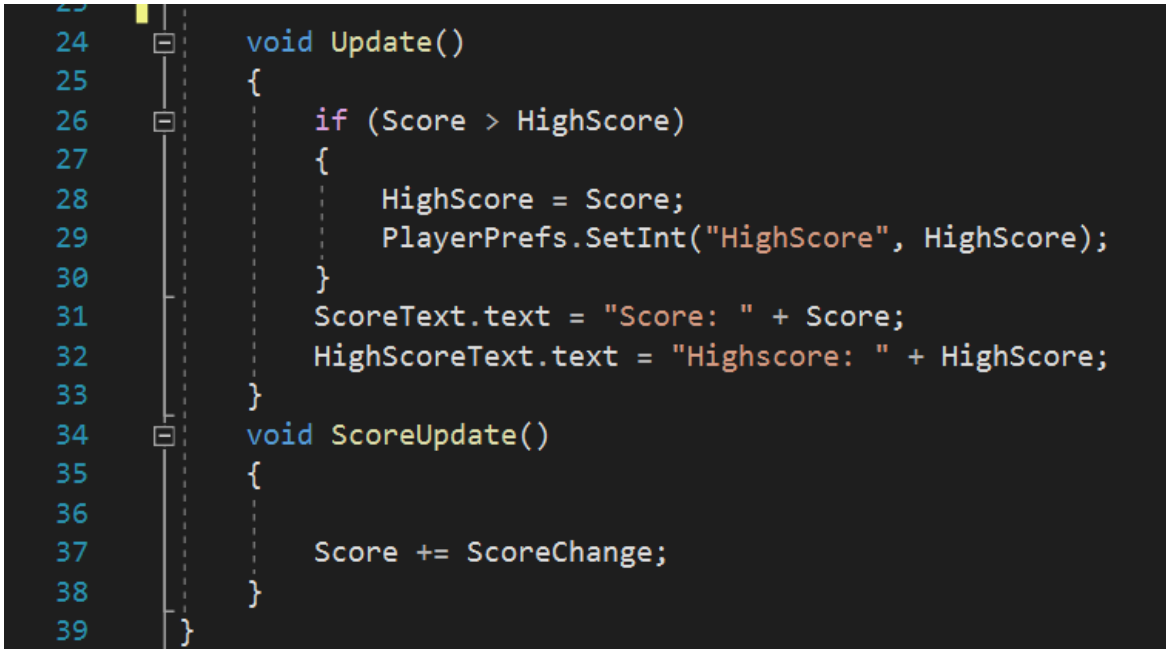
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Scoring : MonoBehaviour
{
    public Text ScoreText;
    public Text HighScoreText;
    int Score;
    int HighScore;
    public int ScoreChange = 4;

    void Start()
    {
        if (PlayerPrefs.HasKey("HighScore"))
        {
            HighScore = PlayerPrefs.GetInt("HighScore");
        }
        Score = 0;
        InvokeRepeating("ScoreUpdate", 0.5f, 0.25f);
    }
```

Here I create two public Text variables called ScoreText and HighScoreText which allows me to then specify which text will be allocated to which through unity. Then I create two private integer variables for the Score and HighScore values.

PlayerPrefs is a built-in Unity way of storing information between scenes and when the program isn't running. This means it can be used to store the HighScore value.

In the start function we first check if a HighScore has already been saved. .HasKey is a method that checks whether data with the name HighScore has already been created. If so then it will fetch that data to enable me to use the current HighScore. If not (In the case the game is being played for the first time) Then the HighScore will not be fetched and will be 0.

At the beginning of the game the score will be reset to 0 and after 0.5 seconds every 0.25 seconds the ScoreUpdate() Function will run.

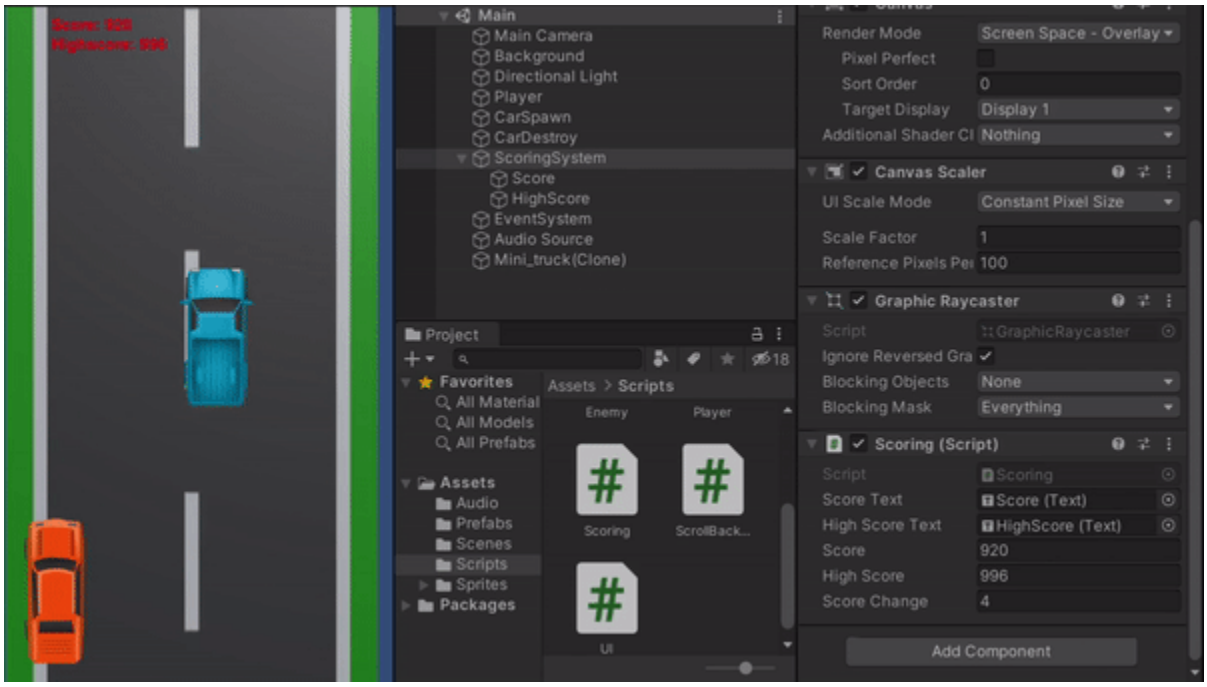


```
24      void Update()
25      {
26          if (Score > HighScore)
27          {
28              HighScore = Score;
29              PlayerPrefs.SetInt("HighScore", HighScore);
30          }
31          ScoreText.text = "Score: " + Score;
32          HighScoreText.text = "Highscore: " + HighScore;
33      }
34      void ScoreUpdate()
35      {
36
37          Score += ScoreChange;
38      }
39  }
```

Every frame it will check if the Score value is higher than the HighScore value. If it is then the HighScore value will be changed and the new HighScore will be saved using PlayerPrefs.SetInt meaning Store an Integer Value and name it HighScore. After this is done then the ScoreText.text and HighScoreText.text just make sure that this will be shown to the screen.

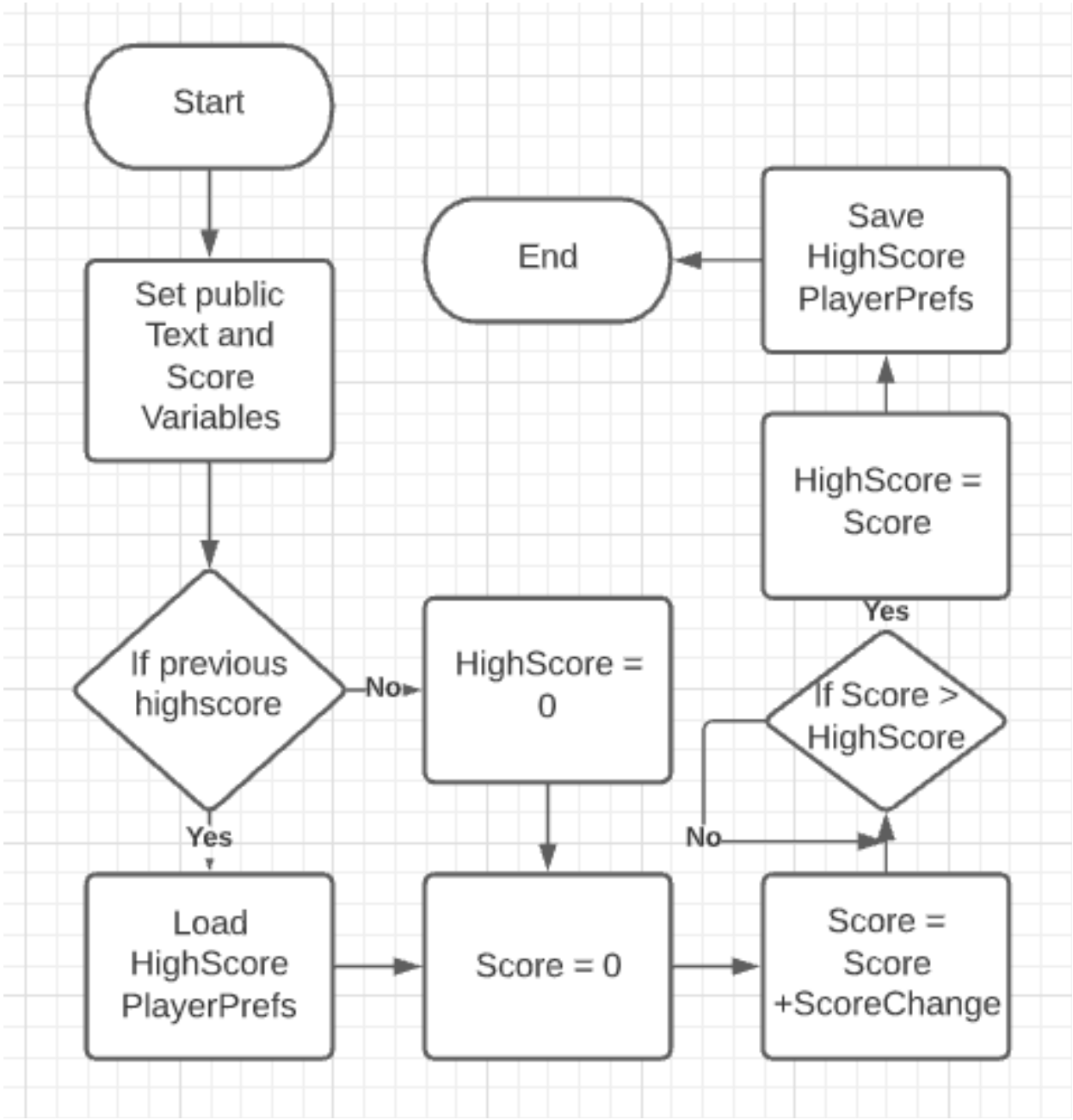
The ScoreUpdate() Function is used to make the Score increase by adding the ScoreChange value every time it is called.

The Score in function -



Test No	Test Description	Expected Outcome	Outcome	Success
1	The Timer starts at 0 when you begin the game	The Score begins at 0	Score began at 0	☑
2	The Timer increases by the expected amount regularly	The Score increases at a set rate of 4 every second	Score increases by 4	☑
3	When there is a crash the timer freezes	The Score will stop when there is an accident	Score stops when an accident	☑
4	When you play again the timer will start again from 0	When you replay the game the Score will be at 0	Score starts at 0 again	☑
6	The first score will be given as the high score	The first score ever will be set as a high score	First Score is Highscore	☑
7	A score higher than the high score will be the new highscore	A score exceeding the old highscore is now the new highscore	New Highscores are saved	☑

The Scoring System Flowchart -



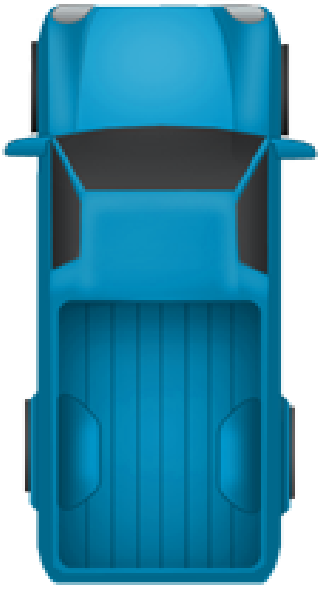
Graphics and Finalising Sprites 24/04/2021 (Requirement 7):

This is the Menu Sprite that will be placed on a canvas in the Menu State once there is an accident. The bottom provides a space for the user to click to Start the game again. Once this button is pressed the game will enter the Main state. The Text and overall design is created to seem similar to the original Chicago Coin Speedway poster.

Speedway Text

Player and Car Sprites:

The Player's sprite. This sprite is a royalty free licensed sprite that is free for use in creating games. The colour is orange to allow for contrast between the car and the road surface. It still retains a rectangular shape making it useful when having a rectangular collider2d applied to it.



The Opponent's sprite. This sprite is a royalty free licensed sprite that is free for use in creating games. The colour is blue to stand out from the orange car as much as possible. It has a rectangular shape for a logical hitbox. The use of a truck here is to maintain the logic that the player's muscle car will be much faster than the trucks and that is why he will continue to overtake and pass them.

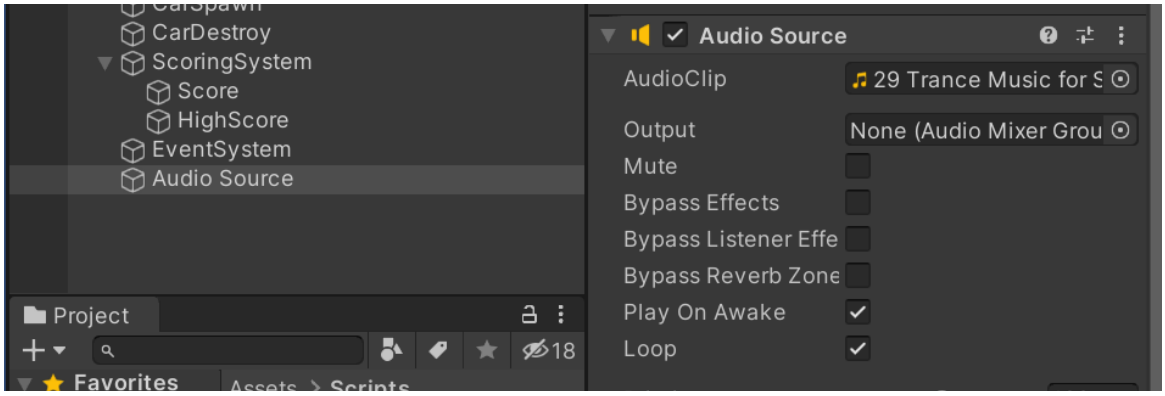
Audio 26/04/2021 (Requirement 8):

Creating my own audio wasn't a priority. I found Royalty Free Music online to play in the background of the Game.

Audio File: Skateboarding Trance Music

Applying Audio:

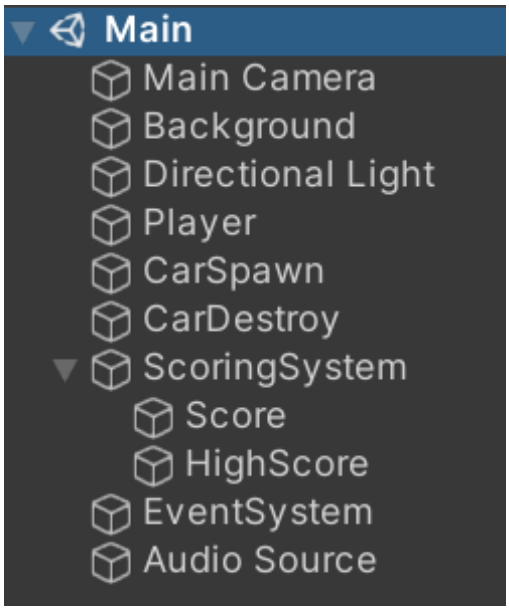
I added an Empty Object and gave it the Component Audio Source. I then applied my audio file into the Audio source and in the settings set it to loop. This means it will continue infinitely in the Main state until the user dies where it restarts when the Main scene is reloaded.



Test No	Test Description	Expected Outcome	Outcome	Success
1	Audio does not play in the menu state	The audio doesn't play in the menu state	Audio Doesn't Play	☑
2	Audio plays in the main state	The audio plays in the main state	Audio Plays	☑

ATTRIBUTES OF THE PROGRAM:

Main Scene -



Attribute Hierarchy. (Most of this is placed in the order that the camera will see from a plane view purely for the reason of being easier to understand for me and to make sure that the z values assigned to the sprites and quads are appropriate.)

Main camera - Shows the boundaries of what the screen is able to see. This is useful for reusing assets for example the opponent cars being offscreen initially.

Background - This is a Quad to make the Scrolling Background road illusion. Has the ScrollBackground.cs script assigned.

Directional Light - Used to make sure that the scene is bright and uniform and make sure that the sprites and art will look good on a mobile device

Player - The user's car sprite and what they control when they move. User's car sprite with the Player.cs script, Rigidbody2D and BoxCollider2D assigned.

CarSpawn - The location of where the enemy car prefabs will spawn in. Empty Gameobject with the CarSpawn.cs script assigned.

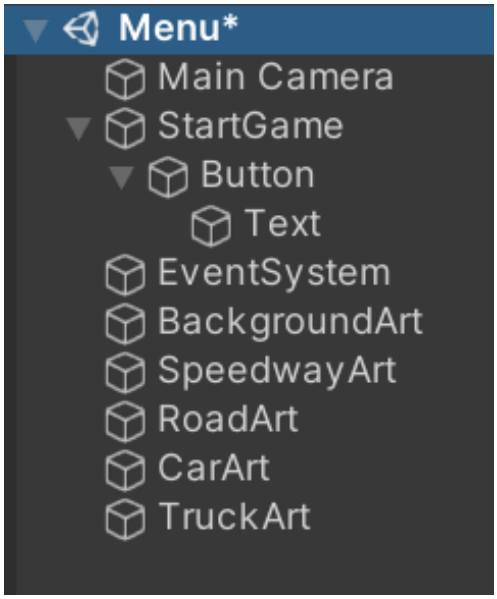
CarDestroy - The location of where the enemy car prefabs will be destroyed. Empty Gameobject with the Destroy.cs script and a BoxCollider2D assigned.

ScoringSystem - The Canvas containing the Score and HighScore Text values. Has the Scoring.cs script assigned.

EventSystem - Part of the built in Unity code that automatically creates this when any UI is used.

Audio Source - The audio component for the music. Empty GameObject with the Audio Source component added.

Menu Scene -



Attribute Hierarchy -

Main Camera - Shows the boundaries of what the screen is able to see. This is useful for reusing assets for example the opponent cars being offscreen initially.

StartGame - Canvas with the Button that is used to Start the game and transition to the Main Scene. Has the UI.cs script assigned.

EventSystem - Part of the built in Unity code that automatically creates this when any UI is used.

BackgroundArt - The yellow background sprite

SpeedwayArt - The red speedway text sprite

RoadArt - The road sprite

CarArt - The player sprite

TruckArt - The truck sprite

Annotated Scripts:

ScrollBackground.cs -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ScrollBackground : MonoBehaviour
{
    Renderer rend;
    public float scrollSpeed = 10.5f;

    void Start()
    {
        rend = GetComponent<Renderer>();
    }

    void Update()
    {
        float offset = Time.time * scrollSpeed;
        rend.material.SetTextureOffset("_MainTex", new Vector2(0, offset));
    }
}
```

- All Unity imports and libraries that are needed are used at the beginning of the script.
- I then created a public class called ScrollBackground. Monobehaviour is the base class for Unity.
- A public variable with the float data type named scrollSpeed is created. Making this public means this value can be edited from within the Unity Hub.
- Before the first frame update it renders the sprites as 2D.
- Every frame it will then call upon the update function. An offset variable (float data type) calculates the time that has passed multiplied by the scrollSpeed. This is used to calculate the distance that has been moved. This sprite is placed at a vector (0, offset) meaning that it will create an endless scroll vertically.

Player.cs -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class Player : MonoBehaviour
{
    public float CarSpeed;
    Vector2 position;

    void Start()
    {
        position = transform.position;
    }

    // Update is called once per frame
    void Update()
    {
        position.x += Input.GetAxis("Horizontal") * CarSpeed * Time.deltaTime;
        position.x = Mathf.Clamp(position.x, -2f, 2f);
        transform.position = position;
    }
    void OnCollisionEnter2D(Collision2D col)
    {
        if (col.gameObject.tag == "Enemy")
        {
            Destroy(gameObject);
            SceneManager.LoadScene("Menu");
        }
    }
```

- All Unity imports and libraries that are needed are referenced at the beginning of the Script.
- A public class named Player is created.
- A public CarSpeed variable is created (float data type) that can be edited when in Unity Hub to decide the movement speed of the user. This enables me to find a good balance of difficulty in testing.
- A Vector2 variable is created called position
- At the beginning of the game the initial position is set
- Once per frame the direction variable will use GetAxisRaw to check for arrow key inputs. It will check this once per frame and transform the position of the car using this. It will clamp these x values between -2 and 2 to keep the car on the screen and on the track. It will then transform to this new position.
- Using the BoxCollider2D if there is a collision and it collides with a GameObject with the tag “Enemy” then the Player’s car will be destroyed and the game will switch to the Menu scene as the game is over and there has been an accident.

UI.cs -

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5
6   public class UIManager: Monobehaviour {
7
8       public void StartGame() {
9           SceneManager.LoadScene("Main");
10      }
11  }
12
```

- All unity libraries are referenced here including the SceneManagement. This is used to change between scenes for example.
- Here we create a public class known as UI.
- When the StartGame() function is called upon then the SceneManager loads the scene known as “Main”.

This script is applied to a button on a canvas in the menu state. Once the button is pressed it calls upon the StartGame() function. This then tells the SceneManager to load the main (game) scene. This means that when the button is pressed it will start the game. This provides a link between the two scenes and having two scenes is beneficial as it means assets that aren’t part of the current scene don’t have to be loaded in the background.

Scoring.cs -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Scoring : MonoBehaviour
{
    public Text ScoreText;
    public Text HighScoreText;
    public int Score;
    public int HighScore;
    public int ScoreChange = 4;

    void Start()
    {
        if (PlayerPrefs.HasKey("HighScore"))
        {
            HighScore = PlayerPrefs.GetInt("HighScore");
        }
        Score = 0;
        InvokeRepeating("ScoreUpdate", 0.5f, 0.25f);
    }

    void Update()
    {
        if (Score > HighScore)
        {
            HighScore = Score;
            PlayerPrefs.SetInt("HighScore", HighScore);
        }
        ScoreText.text = "Score: " + Score;
        HighScoreText.text = "Highscore: " + HighScore;
    }
    void ScoreUpdate()
    {

        Score += ScoreChange;
    }
}
```

- Here I create two public Text variables called ScoreText and HighScoreText which allows me to then specify which text will be allocated to which through unity. Then I create two private integer variables for the Score and HighScore values.

- PlayerPrefs is a built-in Unity way of storing information between scenes and when the program isn't running. This means it can be used to store the HighScore value.
- In the start function we first check if a HighScore has already been saved. .HasKey is a method that checks whether data with the name HighScore has already been created. If so then it will fetch that data to enable me to use the current HighScore. If not (In the case the game is being played for the first time) Then the HighScore will not be fetched and will be 0.
- At the beginning of the game the score will be reset to 0 and after 0.5 seconds every 0.25 seconds the ScoreUpdate() Function will run.
- Every frame it will check if the Score value is higher than the HighScore value. If it is then the HighScore value will be changed and the new HighScore will be saved using PlayerPrefs.SetInt meaning Store an Integer Value and name it HighScore. After this is done then the ScoreText.text and HighScoreText.text just make sure that this will be shown to the screen.
- The ScoreUpdate() Function is used to make the Score increase by adding the ScoreChange value every time it is called.

Enemy.cs -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Enemy : MonoBehaviour
{
    public float Speed;
    private Rigidbody2D rb;
    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    void FixedUpdate()
    {
        rb.velocity = Vector2.zero;
        rb.velocity = new Vector2(0, Speed * Time.fixedDeltaTime * -1);
    }
}
```

- A public float Speed variable is created
- A Rigidbody2D is imported to give us access to kinematics and velocity.
- Every Physics Update (FixedUpdate) we set the velocity to zero and then set the new velocity to the vector (0, speed * Time.fixedDeltaTime * -1) This means it transforms the velocity to travel downwards at the Distance that it should have travelled each frame.

CarSpawn.cs -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CarSpawn : MonoBehaviour
{
    public GameObject car;
    public float DelayTimer;
    float timer;

    void Start() {
        timer = DelayTimer;
    }

    void Update() {
        timer -= Time.deltaTime;
        if (timer <= 0)
        {
            Vector3 CarPos = new Vector3(Random.Range(-2.0f, 2.0f), transform.position.y, transform.position.z);
            Instantiate(car, CarPos, transform.rotation);
            timer = DelayTimer;
        }
    }
}
```

- In CarSpawn.cs I set a public GameObject as the car which is important as it means that you can drag in the prefab that will be spawned in as you can see above from the CarSpawn gameObject. I then made a public float variable named DelayTimer that will act as the Delay between the spawn of each car. This means it can be used to change the difficulty of the game. I then made an empty float variable called the timer
- When the game starts I set the timer equal to the DelayTimer meaning a car will not spawn initially. In the update function each frame then the timer value will have the time that has passed between each frame taken away from it. The purpose of this is to count down until the DelayTimer reaches 0.
- Then there is an if statement where if the timer reaches 0 (The DelayTimer has counted down to 0.) then the position of the car is defined as a Vector 3 known as CarPos which spawns the car between x values -2 and 2. This keeps it within the boundaries. The rest then makes sure it spawns at the CarSpawn gameObject location. The timer is then reset to the DelayTimer and will count down again. This means cars randomly and regularly spawn every DelayTimer between -2 and 2x.

Destroy.cs -

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Destroy : MonoBehaviour
6   {
7
8       void OnCollisionEnter2D(Collision2D col) {
9           if(col.gameObject.tag == "Enemy")
10          {
11              Destroy(col.gameObject);
12          }
13      }
14  }
```

- When there is a collision between the BoxCollider2D's then if the collision with a GameObject has the tag "Enemy" then this GameObject is destroyed.

Referencing Scripts:

ScrollingBackground.cs References -

1) This is referenced on the Background Quad with the aim of making the quad scroll vertically infinitely. This is then overlaid with the Road Background Sprite.

In both of these references it is important to note that because this is done to a 3d quad and not a sprite then the sprites that are assigned as the background of the quad are interchangeable. This is important in further development for example because it means that the gameplay graphics don't affect the logic of the game and so they can be easily swapped out. If new graphics are made it will not affect the game and it is unlikely that it will be needed to be tested again keeping maintenance down.

Player.cs References -

1) This is referenced to the Player sprite and allows the player to move their car horizontally to dodge oncoming obstacles. This movement speed is a public variable and therefore means that in the future for maintenance all kinematics have public speed variables making it easy to modify or balance game mechanics. For example if the user can respond too quickly to oncoming vehicles then the difficulty can be raised by making their movement speed decrease.

UI.cs References -

1) This is referenced to the start game canvas in the menu scene. When this is pressed then the main scene will be entered from the beginning.

Enemy.cs -

1) Referenced to the Mini_Truck prefab which was used as the enemy car in the Main scene. This was also given a Rigidbody2D to allow the Enemy.cs script to work using kinematics.

CarSpawn.cs -

1) Referenced to an empty GameObject above the screen in the Main scene. From these y and z coordinates the enemy cars would be spawned and the x value randomised between -2 and 2.

Destroy.cs -

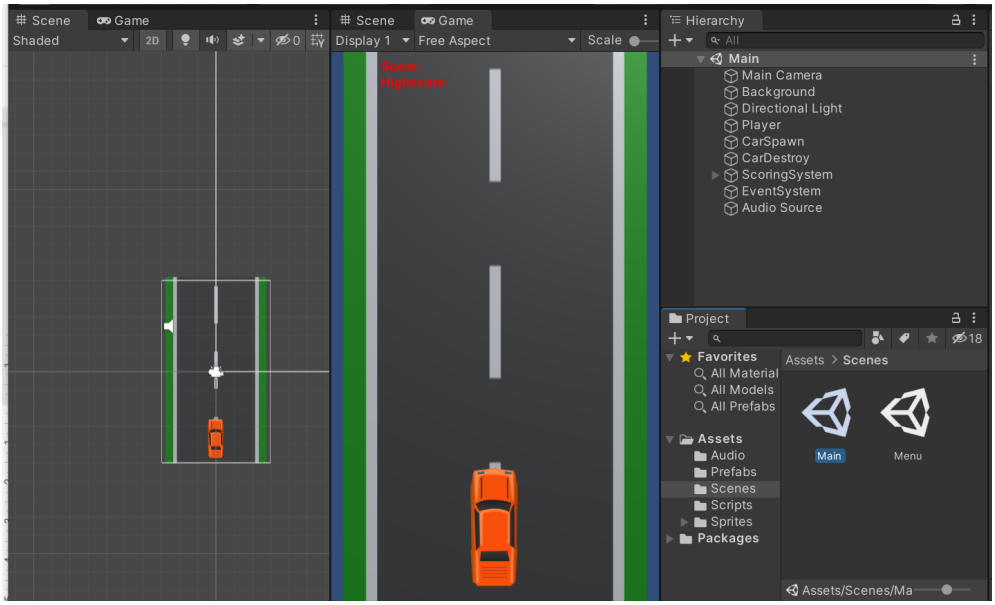
1) Referenced to an empty GameObject below the screen in the Main scene. Had a BoxCollider2D to be able to destroy any colliders

Scoring.cs -

1) Applied to the ScoringSystem Canvas in the main screen to make the text UI appear and show the Score values

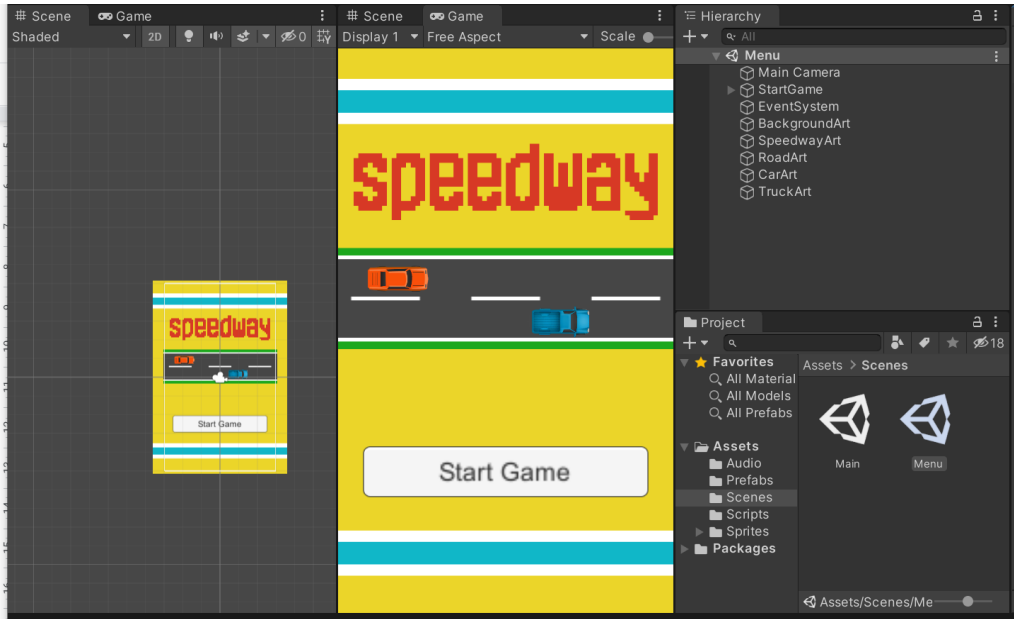
Scenes and States:

Main Scene -



Responsible for the Gameplay and the ScoreSystem this is where most of the development occurred as this is where most of the game requirements are and where the user would spend the most time.

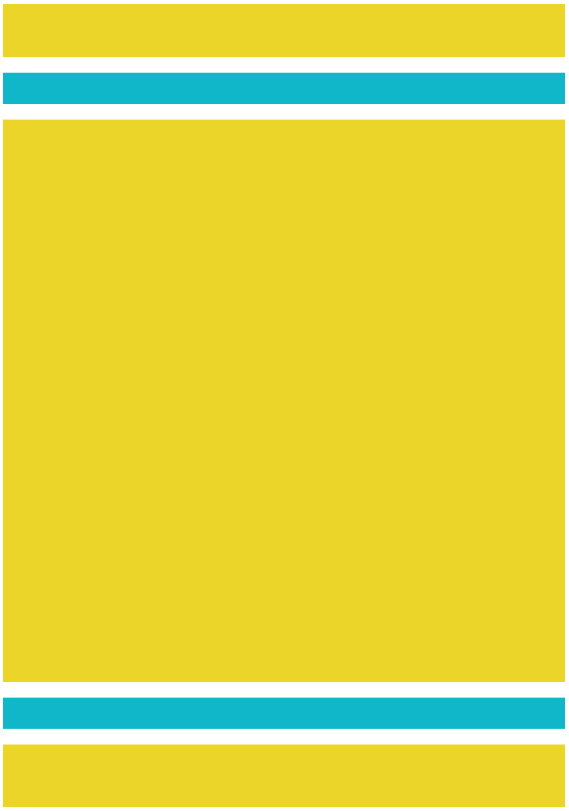
Menu Scene -



Responsible for acting as a pause between games and allowing for the user to start another game. When this is done it then transfers back to the Main Scene.

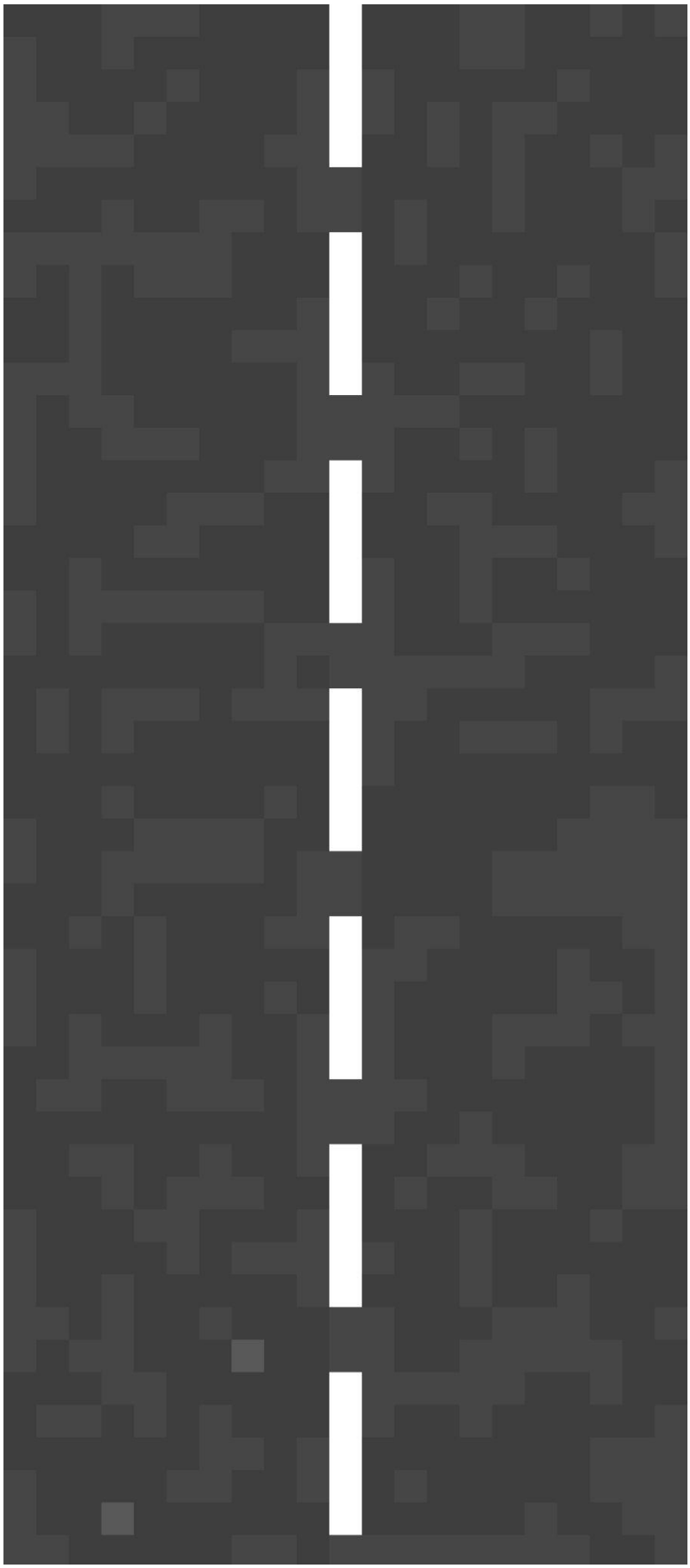
All Objects:

Background Menu Sprite -



Used in the Menu Scene and acts as the menu without the overlaid text. Has spaces for text and for providing high score information as well as a button that changes the menu state into the main state. In the original art style of the poster that was produced with the game in 1969. The overall layout is changed to fit a mobile 16:9 Device.

Speedway Text - Placed over the Background Menu Sprite when on a canvas and the menu state. Is the main logo of the game. In the original art style of the game poster font.



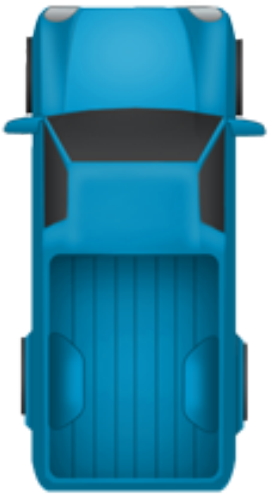
Road Background Sprite -

Produced for a 16:9 Portrait orientation mobile device (most mobile phones) This is the basis of the background when in the game state. This is used to create the scrolling background.

User's Car Sprite -

- Rectangular shaped for more logical collision detection. This sprite is royalty free and not produced by me. It Provides contrast both to the road and to the opponent's car. Hidden wheels mean that wheels don't have to spin and just an image can be used.

Opponent's Car Sprite -



- Rectangular shaped for more logical collision detection. This sprite is royalty free and not produced by me. It Provides contrast both to the road and to the player's car. Hidden wheels mean that wheels don't have to spin and just an image can be used.

RECORDED GAMEPLAY:

Video File

POST-DEVELOPMENT TESTING

Start Up -

Test No	Test Description	Expected Outcome	Outcome	Test Success
1	Testing that the Menu screen loads properly on game startup.	The Menu Screen is displayed and loads in with the Background, Text and Art sprites. It will also have a Button with Text “Start Game”.	Successful outcome. The Menu loads with correct art and with a Start Game button.	

Menu -

Test No	Test Description	Expected Outcome	Outcome	Test Success
1	Testing that the Menu button loads the Main Scene.	Clicking the “StartGame” button runs the StartGame() function from UI.cs script. This then switches the scene to the Main scene	Successful outcome. The StartGame button loads the Main Scene.	

Player -

Test No	Test Description	Expected Outcome	Outcome	Test Success
1	Testing that the Player can only move Horizontally	The Up and Down Arrow Keys have no effect	The Up and Down Arrow Keys have no effect	
2	Testing that the Player can move Horizontally	The Right arrow key moves the Player to the right. The Left arrow key moves the Player to the left.	The Right and Left Arrow Keys work as expected.	
3	Testing that the Player can’t exit the road	There are barriers blocking movement between x coordinates -2 and 2. This means	The player can only move horizontally between -2 and 2	

		the player should only be able to move within the range of the road.		

Enemy Cars -

Test No	Test Description	Expected Outcome	Outcome	Test Success
1	Enemy cars spawn every DelayTimer variable length of time.	The enemy cars will only spawn every 1.2 seconds as specified within the public float variable DelayTimer	The enemy cars only spawn every 1.2 seconds	
2	Enemy cars x coordinates are randomised from -2 and 2 x coordinates.	The enemy cars will spawn between -2 and 2 x coordinates meaning that they will stay on the road.	The enemy cars only spawn between -2 and 2	
3	Enemy cars should spawn at the CarSpawn GameObject location.	The y and z coordinates of the CarSpawn empty GameObject will be used when instantiating the Enemy cars.	The y and z coordinates are used from the CarSpawn object	
4	Enemy cars will travel downwards at the moveSpeed public variable set.	Enemy cars scroll down at the moveSpeed public variable	It has the speed of moveSpeed using a Vector2 displacement	
5	When cars collide with the CarDestroy empty GameObject they should be destroyed	Enemy cars are destroyed when the BoxCollider2D's collide	Enemy cars are destroyed when the BoxCollider2D's collide	

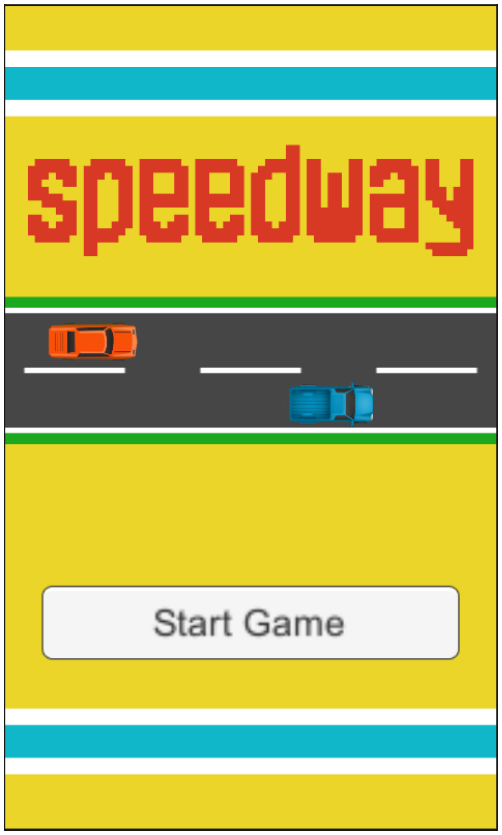
Scrolling Background -

Test No	Test Description	Expected Outcome	Outcome	Test Success
1	The Scrolling Background travels downwards at the scrollSpeed creating an illusion of scrolling.	The Background Quad will scroll down at the ScrollSpeed variable	The Background Quad scrolls down at the ScrollSpeed variable	

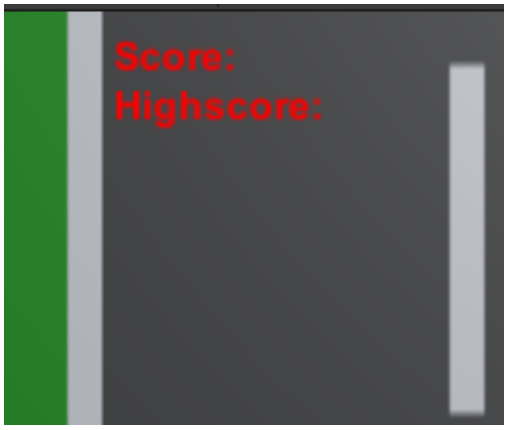
Scoring -

Test No	Test Description	Expected Outcome	Outcome	Test Success
1	Score starts at 0 every time the game is rerun and the “Main” scene loads.	Score always resets to 0	The Score is always reset every time the program is ran and never saved	
2	Score increases by the ScoreChange variable over time.	The Score increases by ScoreChange value every 0.25 seconds	The Score increases by 16 a second	
3	The initial HighScore value is zero	HighScore is 0	HighScore is 0	
4	Whenever the Score exceeds the HighScore a new HighScore is set	When the Score exceeds the HighScore the HighScore variable is then changed and this value is shown on the screen. This value is then saved using PlayerPrefs to keep this as local saved data	HighScore changes when Score changes and saves whenever it is exceeded	
5	The HighScore loads in at the beginning of the Scene meaning that the old HighScore will still be there.	The old highscore is there and will only increase if the score exceeds it	The old highscore is there and will only increase if the score exceeds it	

Usability Features



Start Game button - Allows the user to play again by taking them to the “Main” scene. When the user dies this screen will automatically be shown. Only one button means the game is easy to navigate without having a long menu and settings navigation bar and drop downs.



The Score and HighScore are visible in Bright Red text to show the user their current score and a target to aim for. When the Main scene is run there will be values appearing next to these two Text UI elements. This shows the user more information about their gameplay.

Controls -

The game uses the A and D keys whilst in development. If I didn't have extra time requirements I would have used mobile touchscreen controls to Build and Run the game on a mobile device. Whilst in my testing I imported Android JDK and SDK and installed the Unity Remote 5 to test the game on a mobile device. To do this I created empty UI buttons that took up half of the screen each and in my Player.cs script created public functions, Left(), Right() and Zero(). The Left button would move the car left and the Right button moved the car right. When the buttons were let go the Zero() function would set the Rigidbody2D velocity back to zero. Unfortunately I attempted to build and run this for a mobile device and from there whilst in development my code became unresponsive and I went back to the last save file without mobile touchscreen controls.

- 17/05/2021

Comparing to Requirements

No.	Requirement	Description and Explanation
1	Odometer	This provides the information for how long the car has driven for. This will be the score that allows the user to know when they are close to beating a high score.
The scoring system includes a volatile score that resets to zero whenever the program is run. The highscore is initially zero unless there is a previous save. When the score exceeds the highscore this is set as a new highscore value and saved as local data. These saves are done using the PlayerPrefs methods. I believe this fits the requirements as having a volatile Score and a HighScore that is saved and reused keeping it's value whilst the program isn't running meets the Description set within the requirements. - 17/05/2021		
2	Player Controller	Give the user movement on screen to enable them to dodge enemy cars for as long as possible
The Player has controls using arrow keys and for a limited amount of time during development had mobile touchscreen controls. They can move horizontally to dodge the enemy cars that scroll from the top of the screen. They have the movement needed to enable them to dodge enemy cars for as long as possible. The main requirement was achieved whilst in development however I would advise that if there was to be further maintenance or development this would need to be focused on converting this Unity Android development into making sure it worked on touchscreen controls and functioned on the originally suggested platform. - 17/05/2021		
3	Accident	When there is a collision the game will go back to the menu scene in which the user can restart the game.
When the user collides with a GameObject with the tag "Enemy" (Only the Enemy car prefabs") Then the car will be destroyed and the game will load the "Menu" scene. This means when there is a collision the game will go back to the menu screen. The menu screen then includes a functioning StartGame() button which then loads the "Main'' scene meaning the user can restart the game. This requirement was definitely met and successful. There is room for improvement however. In the menu screen it is possible to show further information in the future or have a Settings section allowing for the disabling of the audio for example. - 17/05/2021		
4	Moving Background	Provides the illusion of movement and distance travelled. Also means with a moving background the game will seem more dynamic and interesting.
The Background scrolls down at the scrollSpeed variable value. This value is public, making it changeable for the developer. It creates an illusion of movement and the car travelling when oftentimes the user is stationary.		

This is further helped by the use of the enemy cars which also scroll downwards increasing this illusion. The effective graphics that are suitable for all also make the game seem dynamic and interesting. This requirement was met successfully.

- 17/05/2021

5	Opponent Cars	Opponent Cars will be deployed to provide obstacles to the player and difficulty in overtaking. If you collide with an enemy car then the game ends.

Enemy cars are repeatedly spawned at the CarSpawn empty object. These cars are randomly spawned with an x coordinate between -2 and 2 meaning that the vehicles are only spawned within the correct area wanted by the developer. These vehicles then have an Enemy.cs script which uses a Rigidbody2D to move them downwards each frame. The result is an illusion of cars appearing from the top of the screen and scrolling down below it. They provide obstacles to the player and mean that the player will have to dodge these cars. The use of a BoxCollider2D on both the player and the enemy cars means that when the user collides with an opponent car then the game will end, the score will reset to 0 and the high score will change if it has been increased. The game will then enter the “Menu” state. I then further extended beyond the requirements by creating an empty CarDestroy GameObject with a BoxCollider2D so that any enemy cars that passed below the bottom of the screen were destroyed so that they wouldn’t keep increasing the asset numbers and therefore increasing memory consumption. This requirement was met and exceeded.

- 17/05/2021

6	Menu	This provides an interface for the user to enter the game.

The Menu provides an interface for the user to start the game by using the Start Game button with the StartGame() function. This means the user can continually replay the game spending as much time in the “Main” scene as possible. The “Menu” scene is appropriately decorative to provide a description of the game, mimic the original Chicago Coin Speedway poster and to provide the functionality of entering the game.

- 17/05/2021

7	Graphics	Retro Graphics will make the game look similar to how it looked in 1969. With car shapes and a moving background in a retro art style it can keep a large target audience and be suitable for all.

The Graphics are a combination of retro and modern. The “Menu” scene graphics uses retro pixel art text for the Speedway logo whereas in the “Main” scene the art is modern and effective. All art that has been created or used is royalty free and suitable for all ages. This makes the game very likely to have a PEGI 2 rating as there are no human likenesses, characters, voices, explicit content or any other reasons for having a higher rating such as violence and sexual themes. This game is suitable for all. All graphics are easily interchangeable as all sprites don’t have an affect on the code and they can be removed and replaced at will. There would not be significant maintenance needed for this. This requirement was met. If I had further time for development I would have solely created my own sprites for all that was needed. If this game was to be professionally produced it is likely graphics artists and animators would be used.

- 17/05/2021

8	Sound	Ambient Music
An audio source was implemented within the “Main” scene that plays Skateboarding Trance music that furthers the environment of the game feeling casual and replayable. It is catchy and contains no profanity or voices meaning it will also be suitable for all ages. The audio used is royalty free and allowed for fair use. - 17/05/2021		

Current and Future Maintenance

I have not implemented many maintenance features into the game for the user however I recognise that this would be necessary if it were to be distributed. I have made the game as transparent as possible to understand. This is effective as it means many of the variables are public. This makes it possible for the developer to change the values of the moveSpeed of the Player, scrollSpeed of the Background, DelayTimer of the CarSpawner, moveSpeed of the Enemy cars etc. This means the difficulty can be changed. Furthermore, all sprites can easily be swapped out as the scripts and components can be added relatively easily. This means that overall the Graphics and Audio are able to be swapped out quite painlessly.

My solution meets the requirements effectively however it also has limitations that if I were to develop further I would aim to meet.

Limitations and Theoretical Further Development

If I was to improve upon the game I would add further requirements. The great thing about Unity for development is that it allows easy portability to the Google PlayStore so I would release my game on that AppStore. From there it would then be possible to create an app icon and further app graphics that would be needed.

Extra Requirements -

Requirement Name	Requirement Description	Justification
Touchscreen	I would make the game available to work on touch controls. Initially this could be done using transparent buttons however for a professional release I would learn to code using touchscreen controls.	The game was initially designed to be released on a mobile system and so this should be a major aim.
I could create this using transparent buttons or finding out more about mobile development and importing libraries and methods that allow me to find the location of screen presses on a touchscreen device.		

Resolution and Aspect Ratio	I would then provide extra code in the terms of my UI development to make sure that the game would look the same when on different aspect ratios and screen sizes.	This would be important to make sure that players can play both on tablets and mobile devices without the use of black bars and not being able to use the entirety of the screen.
It would be possible here to create multiple programs by using an if statement that calculates the aspect ratio of the device. If it was a tablet device then a different background sprite would be used that could fit the screen. The Ui elements could then be appropriately resized to fit the screen size.		
Professional Graphics	I would commision somebody to create my own sprites and would be in regular contact to make sure that the game looked as professional as possible. This would mean creating multiple car sprites.	This would make the game look less like a personal project and something more studio produced.
Car Shop and Coin System	It would be possible to then create a Car Shop in the menu in which the user could purchase different cars for varying amounts of coins. These coins would be rewarded based on how far the user travelled. It would then be possible to provide gameplay differences between the cars in terms of handling, speed and acceleration for example.	This creates more variety within the game in driving different cars and having a shop then increases the chance of replayability in players wanting to try and unlock as many as possible to further their progression.
It would be possible to create a coin system by rewarding the user their score in terms of coins. This coin score can then be saved using PlayerPrefs similar to the HighScore. every time the score value would be added on meaning that the coins would increase. Another scene called the shop scene could then be created and linked to using a button in the “Menu scene” from here it would then be possible to create further buttons. These buttons would be shown below cars. When a button was clicked it would load the CoinValue and decrease the value by the cost of the car. It would then allow this car to be used in a function called CarAllow() from here it would then change which car was spawned in when the “Main”scene was loaded.		
Varying Enemy Cars	Using the professional car models it would then be possible to generate multiple types of enemy car. Possibly these could have different mechanics for example one could be a large wide lorry that decreases the room for driving whilst another is an extremely fast thing car.	This increases the skill needed to increase your score as it would require knowledge to tackle multiple enemy types. The mixture of this mixed in with the different user cars then creates strategy.
It is possible to spawn multiple car sprites by adding the Enemy tag to multiple prefabs and randomising which prefab would be spawned in. From here it would then be possible to create different scripts for each car type similar to Enemy.cs but instead with different speeds and mechanics.		

Pause screen	This would mean that the player can pause their game without worrying about losing their score.	This then increases the chance of a player playing the game in short breaks as they dont have to worry about losing an increasing score.
This can be created by using a UI button in the game that when clicked would run a function. Inside the function it could then say time.Timescale = 0. This means that the internal clock of Unity then stops and the game would be paused. Once paused the button could then be changed an an unpause button would then do the opposite and set time.Timescale equal to 1 so the clock would then begin again.		

If this was financially motivated it would then be possible to use Google AdsService to allow the user to watch ads to stay alive therefore increasing the coins that they would receive. This means they would progress further whilst also creating me money. However because this isn't financially motivated that isn't something that I would do in the future. InMany mobile games are oversaturated by adverts and in-app purchases. Instead I would be more likely to contain a link somewhere in the menu that would be voluntary and allow people who enjoyed the game to show their appreciation.

Project Appendixes

Complete Assets Listing:

Assets folder

Audio

- Skateboarding Trance Music

Prefabs -

- Min_Truck Prefab ("Enemy")

Scenes -

- "Main" Scene (Gameplay)
- "Menu" Scene (Menu)

Scripts -

- ScrollBackground.cs (Scrolling Background)
- UI.cs (StartGame Button)
- Player.cs (Player Movement)
- Enemy.cs (Enemy Movement)
- CarSpawn.cs (Enemy Car Spawning)
- Destroy.cs (Enemy Car Destroying)
- Scoring.cs (Score and HighScore)

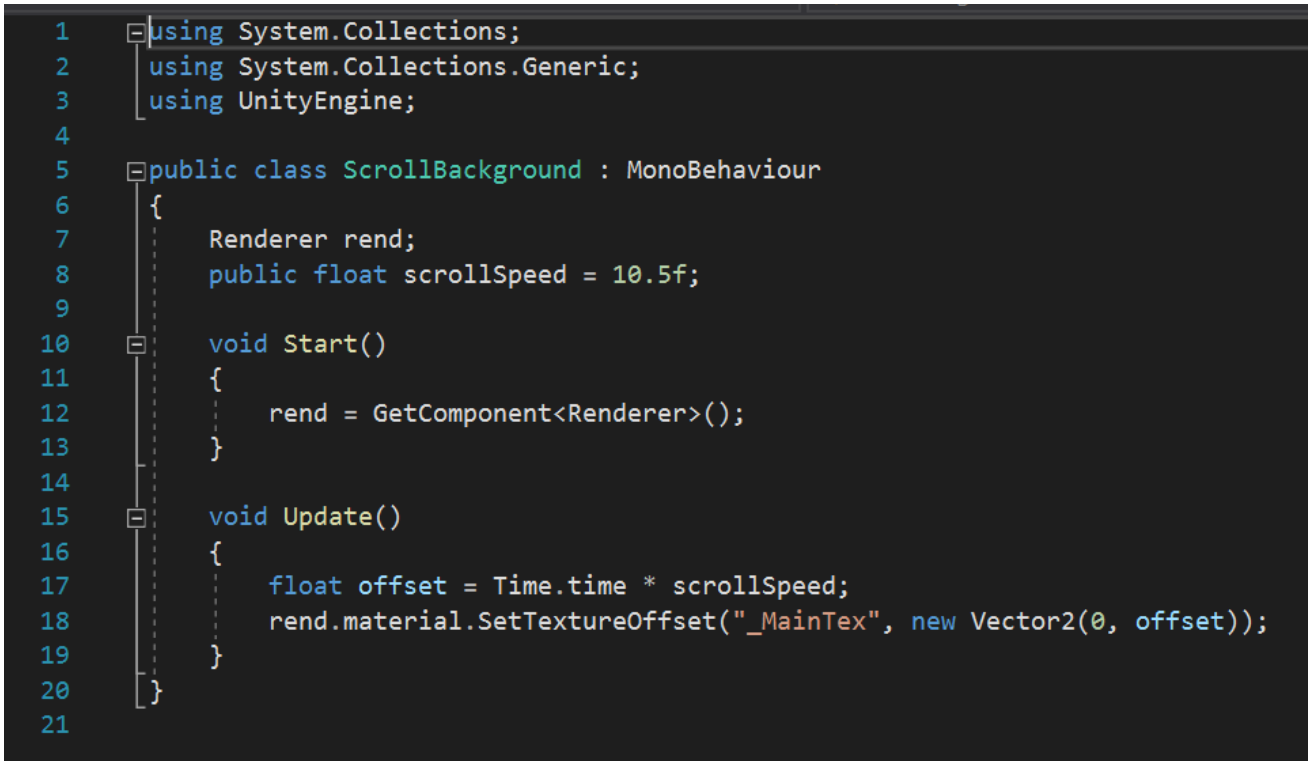
Sprites -

- BackgroundSprite (Road)
- MenuSprite (Menu Background)
- SpeedwayText
- Car (Player's Car)
- Mini_Truck (Enemy Car)

Complete Code Listing:

Scripts

ScrollBackground.cs -



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ScrollBackground : MonoBehaviour
{
    Renderer rend;
    public float scrollSpeed = 10.5f;

    void Start()
    {
        rend = GetComponent<Renderer>();
    }

    void Update()
    {
        float offset = Time.time * scrollSpeed;
        rend.material.SetTextureOffset("_MainTex", new Vector2(0, offset));
    }
}

```

UI.cs -

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using UnityEngine.SceneManagement;
5   using UnityEngine.UI;
6
7
8   public class UI : MonoBehaviour
9   {
10
11      public void StartGame()
12      {
13          SceneManager.LoadScene("Main");
14      }
15  }
```

Player.cs -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class Player : MonoBehaviour
{
    public float CarSpeed;
    Vector2 position;

    void Start()
    {
        position = transform.position;
    }

    // Update is called once per frame
    void Update()
    {
        position.x += Input.GetAxis("Horizontal") * CarSpeed * Time.deltaTime;
        position.x = Mathf.Clamp(position.x, -2f, 2f);
        transform.position = position;
    }
    void OnCollisionEnter2D(Collision2D col)
    {
        if (col.gameObject.tag == "Enemy")
        {
            Destroy(gameObject);
            SceneManager.LoadScene("Menu");
        }
    }
```

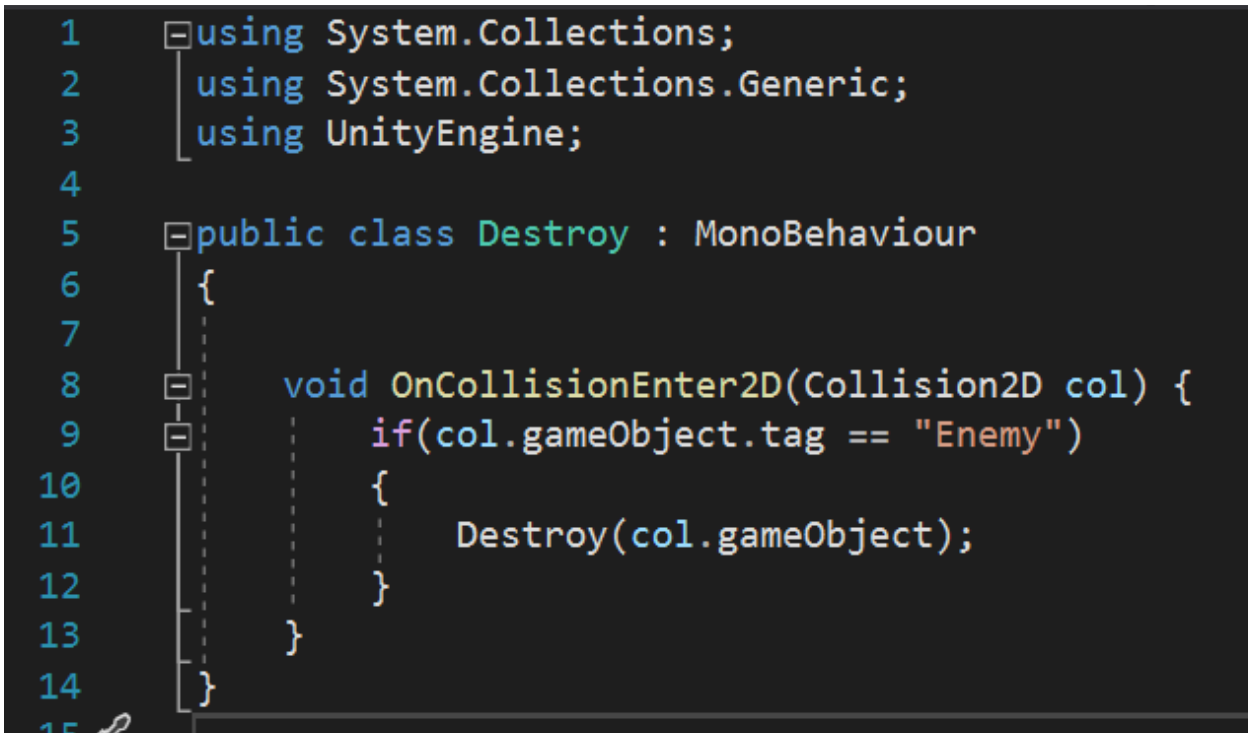
Enemy.cs -

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class Enemy : MonoBehaviour
6   {
7       public float Speed;
8       private Rigidbody2D rb;
9       // Start is called before the first frame update
10      void Start()
11      {
12          rb = GetComponent<Rigidbody2D>();
13      }
14
15      void FixedUpdate()
16      {
17          rb.velocity = Vector2.zero;
18          rb.velocity = new Vector2(0, Speed * Time.fixedDeltaTime * -1);
19      }
20  }
```

CarSpawn.cs -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CarSpawn : MonoBehaviour
{
    public GameObject car;
    public float DelayTimer;
    float timer;

    void Start() {
        timer = DelayTimer;
    }

    void Update() {
        timer -= Time.deltaTime;
        if (timer <= 0)
        {
            Vector3 CarPos = new Vector3(Random.Range(-2.0f, 2.0f), transform.position.y, transform.position.z);
            Instantiate(car, CarPos, transform.rotation);
            timer = DelayTimer;
        }
    }
}
```

Destroy.cs -

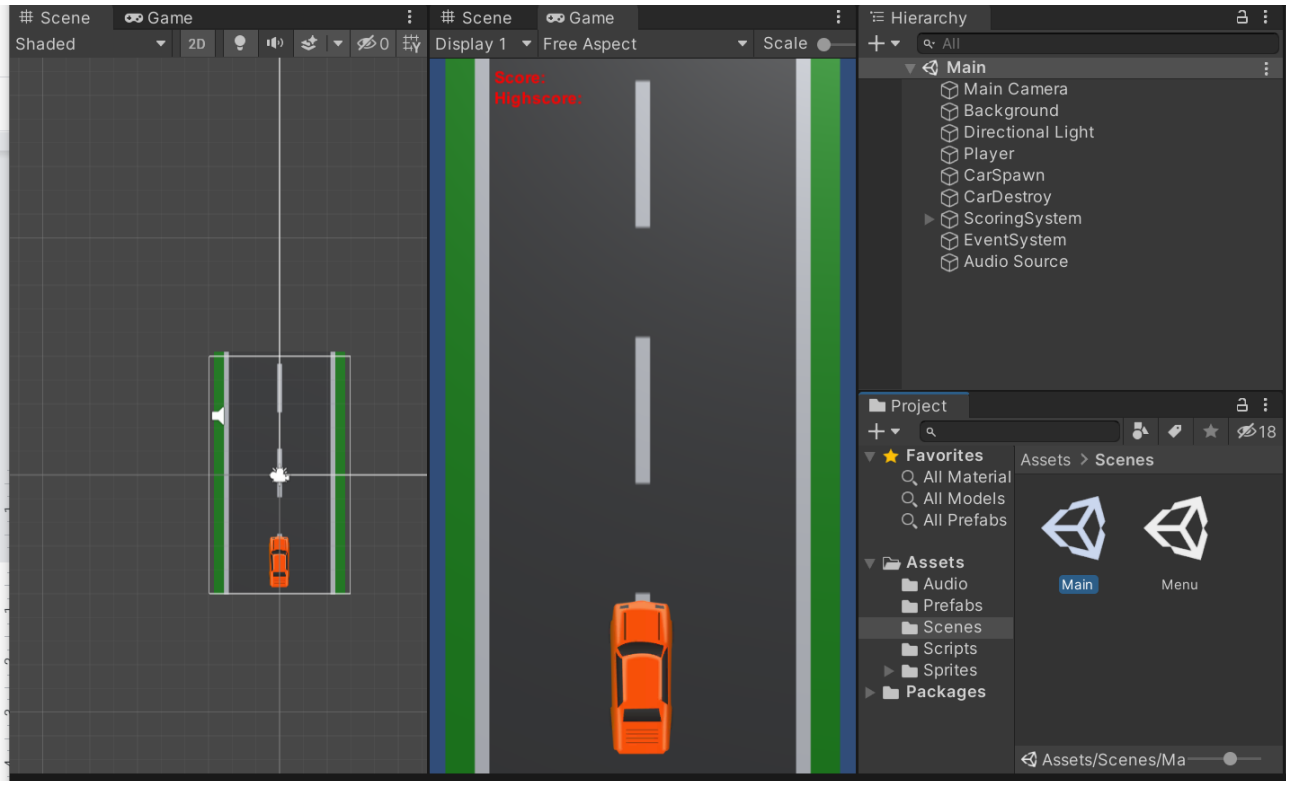


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Destroy : MonoBehaviour
{

    void OnCollisionEnter2D(Collision2D col) {
        if(col.gameObject.tag == "Enemy")
        {
            Destroy(col.gameObject);
        }
    }
}
```

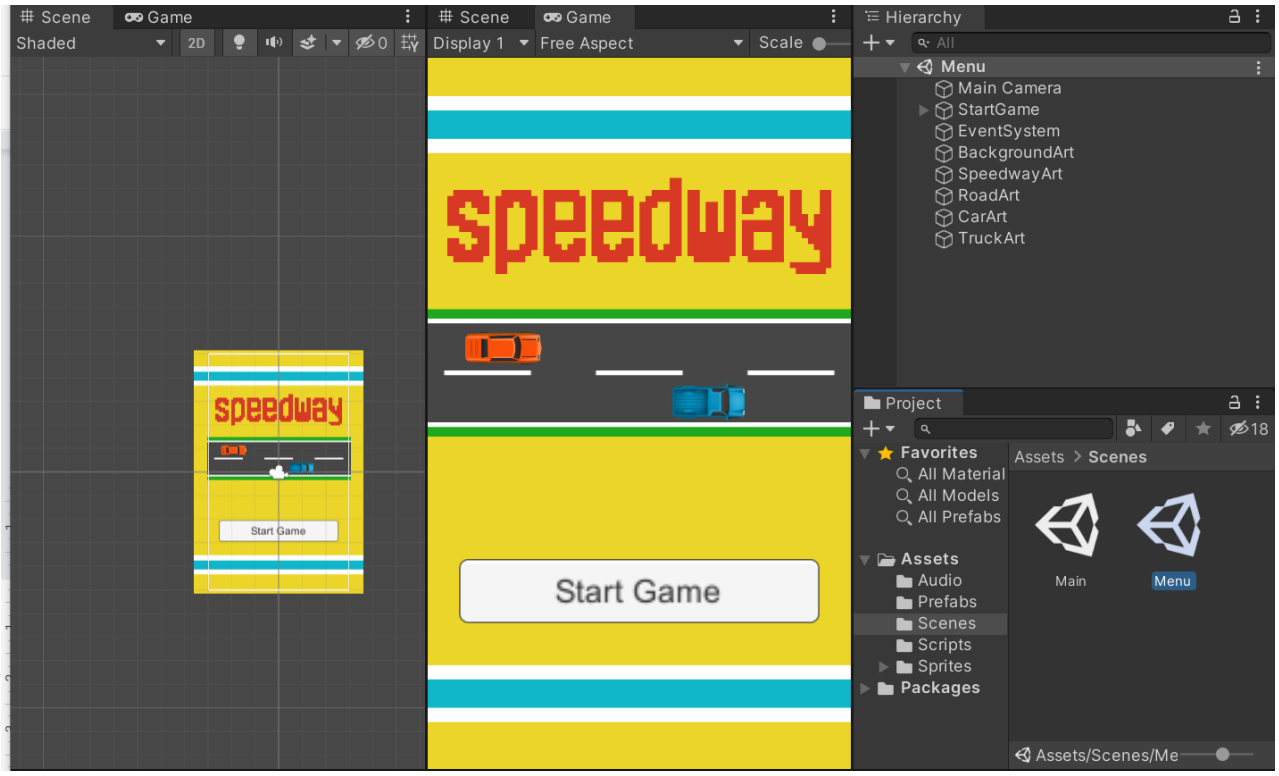
Scoring.cs -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Scoring : MonoBehaviour
{
    public Text ScoreText;
    public Text HighScoreText;
    public int Score;
    public int HighScore;
    public int ScoreChange = 4;

    void Start()
    {
        if (PlayerPrefs.HasKey("HighScore"))
        {
            HighScore = PlayerPrefs.GetInt("HighScore");
        }
        Score = 0;
        InvokeRepeating("ScoreUpdate", 0.5f, 0.25f);
    }

    void Update()
    {
        if (Score > HighScore)
        {
            HighScore = Score;
            PlayerPrefs.SetInt("HighScore", HighScore);
        }
        ScoreText.text = "Score: " + Score;
        HighScoreText.text = "Highscore: " + HighScore;
    }
    void ScoreUpdate()
    {

        Score += ScoreChange;
    }
}
```

“Main” Scene -


Scene
Game
Shaded
2D
Scene
Game
Display 1
Free Aspect
Scale
Score:
Highscore:
Hierarchy
All
Main
Main Camera
Background
Directional Light
Player
CarSpawn
CarDestroy
ScoringSystem
EventSystem
Audio Source
Project
Favorites
All Material
All Models
All Prefabs
Assets
Audio
Prefabs
Scenes
Scripts
Sprites
Packages
Assets > Scenes
Main
Menu
Assets/Scenes/Ma

“Menu” Scene -


Scene
Game
Shaded
2D
Display 1
Free Aspect
Scale
speedway
Start Game
Hierarchy
Menu
Main Camera
StartGame
EventSystem
BackgroundArt
SpeedwayArt
RoadArt
CarArt
TruckArt
Project
Favorites
All Material
All Models
All Prefabs
Assets
Audio
Prefabs
Scenes
Scripts
Sprites
Packages
Assets > Scenes
Main
Menu
Assets/Scenes/Me