Matt Gabelli, 26945, 58319

Motor Mayhem

ANALYSIS

Problem identification

For my project, I will be creating a vertical-scrolling computer game with a prominent theme of cars. I will mainly be taking inspiration from the game Highway Justice
The project will be made with the JavaScript language, in particular the P5.js library (due to this being my most comfortable environment) and the Visual Studio Code text editor. Production will take place on my computer and will be designed for use on a desktop.

A1 - Project stakeholders

Name	Description	Role	Stakeholder requirements
Tom	A year 12 computer science classmate, who spends considerable amounts of time playing mobile games. Tom is an avid gamer and so will be a good candidate to provide feedback.	Hardcore gamer	This user requires the option to select a higher level of difficulty* than the default to test gaming skills. Tom will appreciate a simplistic visual experience coupled with a challenging gaming experience. The game is appropriate for Tom because at the higher difficulties, he will be adamant to break new score records due to his love for competitive gaming
Alex	Alex does not play any games at all. She will make use of my game in effort to get a break from her long work hours	Less than casual user	This user requires an intuitive user interface and mechanics which are not hard to grasp to allow for the possibility of a quick and casual session of gameplay. Motor Mayhem will be appropriate for Alex's needs because of the very relaxed nature and

			negligible learning curve. She has requested a character customisation screen because she has creative hobbies and this will cause her to enjoy the game significantly more, as she is not a gamer.
Andrew	Andrew is a man who regularly plays video games at home on his PC. He mostly plays adventure games, but this can become stressful sometimes when he gets stuck on a puzzle. Motor Mayhem will be a relaxing break from this stress.	Regular user	This user requires a medium degree of challenge and a fun time. Mechanics which are too difficult to learn quickly, like Alex, may result in the user being discouraged from moving further through the game. Andrew has requested the game closely follow the arcade-like theme which he, as a Gen X, will be so familiar with. He feels it will make him feel more engaged

A2 - Research into existing solutions

Highway Justice

Highway Justice is a small indie game in which a user plays as a police officer in a car. They can move up, down, left and right. The user progresses through a road which is scrolled through vertically. A proximity overlay alerts the user of their distance to a ‘target’ and the targets are defeated by the player crashing into them repeatedly. After a set number of targets have been eliminated, the game ends and the user has ‘won’. After each crash, the player’s health decreases and if they run out of health, the game ends and the user has ‘lost’.

Pros	Cons
- Similar theme so this makes for efficient examination of features to be implemented/not implemented	- Graphics are advanced in the sense that the textures are quite detailed and may require more computing power than my stakeholders may have at their disposal
- The overlays used to display information like score is an efficient way to alert the user of their progression in the game	- Highway justice employs a health system for the character. This will not be implemented in Motor Mayhem because not only does it subtract from the simplicity of my project, but also I do not know how to program crash physics into my game, so this would take up a significant amount of time to successfully develop
- In highway justice, the way in which enemies spawn and move	- The way in which enemies are defeated is a difference between

will also be used in my game. This is because it works well with the scoring system.	my solution and Highway Justice. My game will feature a shooter system unlike the crash system in the game above. This is because a shooter system is easier to control for those users with less keyboard experience

Feature to be incorporated	Description of feature	Image	Justification for incorporation
Car theme	Highway justice employs a car theme. The user plays as a car which progresses on a road and defeats enemies which are also in the form of cars.		The theme of cars appeals to a very wide range of potential users. Many people like cars. Even those who don't particularly have cars as a hobby will probably have driven a fair bit in their life, so they can resonate with the theme. The choice of theme, along with the menu is the first thing a customer will see when playing Motor Mayhem, hence it is important to use a popular subject like cars.
GUI buttons	Highway Justice includes two main buttons, start and 'More games'		Motor Mayhem will employ a similar style of GUI button in that the buttons will be coloured brightly to attract maximum customers and they will be displayed clearly on the menu screen. This is so the user is not confused about the options available to them when playing the game

Hybris

Hybris is a vertical-scrolling shooter game developed by Cope-com, released in 1989. Players choose a character from one of two parties and play as a spaceship which can move up, down, left and right. Enemies spawn as the game scrolls upwards constantly, and the user builds score by shooting these enemies.
The shooting mechanics involved in this game will be the primary inspiration for the same system in my project.

Pros	Cons
- The shooting system in hybris will be what I take inspiration from. This is because it adds a level of uniquity to my solution, as it is a car game with a futuristic feel.	- The theme/setting is really the core difference between this game and mine. Hybris follows the theme of space/aeronautic battle atop a seemingly interplanetary environment. My game will feature the classic theme of cars.
- The overlay at the top of the screen is also inspiring for my own game. This is because the score, lives, and high score are presented simply and clearly to the user. But the information is also presented unobtrusively with a sleek appearance	- Hybris uses a different health system to Highway Justice in that a player is given a number of lives and respawns each time they lose a life, until all lives are depleted and the game is over. But still my game will not include a health system.
	- The way scores are calculated in Hybris differs from my game. In my game, score is incremented by 1 each time an enemy is defeated instead of the large numbers achieved in Hybris. This is not only because all of my

	enemies give the same score upon being defeated, but also so users can better understand the mechanics of my game, meaning they can focus on playing well.

Feature to be incorporated	Description of feature	Image	Justification for incorporation
Shooter system	Hybris allows players to fire projectiles vertically from their current position to hit enemies above in effort to eliminate them		The shooter system is a simple, yet unique selling point for my game. The combination of cars and shooting appeals to many and even those who do not enjoy violent games will appreciate my combat system because it is not violent and it presents itself in a very arcade-like style, to increase appeal. The method of firing projectiles is also easy to understand, compared to crash physics seen in Highway Justice
Score	The score overlay in Hybris is unobtrusively displayed in the top left corner of the player's screen. It also contrasts well with the background		Motor Mayhem will incorporate this design of score counter because the simple, cornered style is the best way to discreetly alert the user of their score so far. The contrasting property of the text colour on the background will be carried through in my game to maximise visual accessibility

A3 - Essential features

Essential feature	Explanation	Justification
Main menu	The main menu will feature the game's title and background, as well as all of the various options available to the user with music playing in the background.	The main menu is a crucial feature in my game because it ties together all the other features. It is the backbone from which all options are built and it is important that the user is able to navigate the menu however they please. The main menu will be the feature which most represents navigability in my project therefore it is important that is developed in a way that it is intuitive and flows seamlessly
Enemies (and combat)	Users will be able to move left and right and shoot bullets from their current position vertically up the screen in an attempt to hit an enemy with said bullets. Enemies will spawn randomly and descend vertically down the screen. Score counter will increment by 1 for each hit, and enemies will move faster at higher difficulties	The player-enemy combat system is arguably the most core component in Motor Mayhem. It is an essential feature because it defines my game as one that fits the genre of vertical-scrolling arcade shooter. This is an essential feature because it is what has enticed the clientbase to play the game, hence it is crucial that the combat works well and is satisfying to maintain user engagement
Character customisation	Accessed from the main menu, users can get to a customisation screen which gives them the option to alter the appearance of their character. Different levels of customisation will be available depending on the score obtained by the user most recently	This feature is an essential one in my game because it adds a whole new element to my game. Customisation in games gives the users a sense of inclusion and with the added score system, there will also be a sense of progression. This feature will immerse the user even more than before will significantly add to the enjoyment of the game as a whole

A4 - Limitations

Limiting feature	Explanation	Justification
Enemy movement pattern	During research, it became obvious that the behaviour of enemies was a significant factor for player immersion	My enemies will follow a very limited movement path by travelling down the screen vertically. This is because I want to ensure I have enough time to code the enemies, character, and combat system and ensure it works flawlessly instead of developing complex enemy behaviour which arguably will not notably add to the user enjoyment
Character customisation	It is certain that the degree to which a user can customise the components of a game, directly correlates to the enjoyment from the game because customisation gives players a sense of individuality	There are a number of ways I could include a level of customisation in my game. However I will keep it quite simple in the sense that a user will be able to change the colour of their character. The reason I am restricting customisation to the character and not the environment, enemies for example is that the increase in user satisfaction gained from adding higher levels of customisation is in my opinion not worth the time I would have to take from other important aspects of my game
High score	A high score is a popular way to display player progression in a game	As a user plays game after game, the array used to store the user's scores and therefore high score will become very large very quickly. For this reason, after 10 scores have been stored, the oldest one will be removed to make space for new ones. This means the scoreboard will be limited in the number of previous runs

		it can display
Multiplayer	Not many arcade scroller games are multiplayer, for this reason it would have been a unique selling point for Motor Mayhem	The budget for Motor Mayhem does not account for hosting an online game. It would also drastically prolong development as I have never coded multiplayer functionality before therefore I would have to learn which would mean debugging would probably take a long time as well

A5 - Computational approach

Abstraction

Abstraction is the process of removing unnecessary information from a scenario in order to focus on the important parts. By incorporating abstraction into my project, I will be able to better manage the complexity of the problem by focusing on the essential features.

Abstraction in my arcade shooter game simplifies things by hiding complicated concepts behind the scenes. It helps focus on making gameplay smooth and fun without getting lost in complex details. Using abstraction makes it easier to build and improve the game over time. It means I can swap out or upgrade parts without causing chaos. This simplicity means players can jump in without being confused. This also ensures a more intuitive user interface, empowering players to engage seamlessly. Ultimately, abstraction fosters a robust foundation, reducing cognitive load, fostering smoother gameplay, and enabling rapid adaptation to changing requirements.

An example of where abstraction will be used in my application is the scenery. The terrain and any background imagery will be extremely basic to ensure hardware requirements are not exceeded, therefore users with less powerful hardware environments are still able to enjoy Motor Mayhem. It also means that a user can more easily focus on progressing through the game, rather than being distracted by unnecessary objects on the screen such as shrubbery.

Decomposition

Decomposition breaks down the game creation process into manageable parts, simplifying development. It allows focusing on smaller elements like player controls, enemy behaviour, and visual effects separately. This methodical approach enhances debugging, pinpointing issues within specific components rather than worrying about making accidental changes elsewhere in my game. Decomposition promotes scalability, enabling easier expansion or modification of individual features without disrupting the entire game structure. It fosters efficient time management, as I can tackle isolated tasks concurrently. Ultimately, decomposition ensures a structured, organised game development process, fostering smoother workflows and a polished final product. This is an essential part of development and one I will be using from the start of my solution.

For example, different parts of the project include - enemies, movement, setting, menu. These will all be developed using a top-down approach to ensure complexity is reduced to a minimum and overall will make the development process more streamlined and straightforward. Decomposition will be manifested in my source code as subroutines and classes.

Reusable components

Reusability is a crucial component in the development of my solution. It means I can save lots of time by not needing to recreate algorithms if they have already been designed, developed, and tested.

Because I am developing my solution using the p5.js library, I am passively making use of reusable components when I call functions native to p5, because they are algorithms which I would otherwise need to have coded and debugged myself. Instead I can call these functions with confidence in knowing that they will work flawlessly when implemented into my code.
Certain components of my code will need to be reused throughout the solution. For example subroutines will be used to develop the shooting system so code does not need to be rewritten for every player-enemy encounter. For the enemies, bullets, and enemies I will be using object-oriented programming. This means I will be using methods to do things like draw and move my objects. This reusability in my project will significantly improve efficiency, as not only will it make debugging easier, but more time can be devoted to other components of the project.

A6 - Success criteria

Iteration 1
Iteration 2
Iteration 3

Criteria No.	Criteria	Expectation	Justification
1	There will be a main menu, from which all screens are accessed	The menu must clearly highlight the various options which are available for the user to access. However the menu must be kept simple to prevent overwhelming the user	The menu is the first thing the user is presented with after running the program. It is important that the menu is clearly labelled and concise to allow for an easy user experience
2	The game will include appropriate music	Music will commence upon running the program at a standard volume. This volume will then decrease as the user starts a run.	Music adds atmosphere to a game from which the user gains a subconscious feel of the game. This allows for a more immersive experience for the player.
3	Character movement	The user should be able to control the character's movement simply with the keyboard	This game is aimed for use by not only experienced gamers but also casual ones, for this reason the controls must be easy to grasp. Younger users may be discouraged if the controls are too challenging.
4	Character-enemy combat	Players should be able to fire projectiles at enemies and upon contact, the enemy is defeated and 'dies'.	The enemy-player attack process should be satisfying to more accurately reflect the arcade feel which the game is built on
5	Scoring system	Score should be clearly presented.The player should be able to increase their score by defeating various	The collected score is the main incentive for a user to play the game in order to gain a sense of progression

		enemies. These should be stored in a local data structure to be used in the scoreboard	
6	Scoreboard	An option in the menu will be available to visit a scoreboard screen which will showcase the user's scores on each of the past x amount of runs, including the difficulty preset at the time	This deepens the sense of progression in the game, as a user is able to track their past performance. The scoreboard is easily accessible from the menu, therefore less serious gamers may ignore and those with more experience may choose to utilise it.
7	Customisation of character	The user will be able to choose from a variety of colours to customise their car depending on high score	A customisation enables the user to earn rewards for performance in-game. It also allows for a more individualised experience overall
8	Option to change difficulty	There will be a range of difficulty options the user will be able to choose from	The ability to change difficulty is very important as the user will be able to increase the level of challenge should they feel the game is too easy. It also allows for overconfident gamers to reduce the difficulty in an effort to get higher scores.

A7 - Requirements

Hardware requirements

Requirement	Justification
1GB of RAM	Recommended to run Visual Studio Code
1.6Ghz or faster CPU	Recommended to run Visual Studio Code
Monitor connected to computer	A monitor needs to be connected to the PC to display the user interface, so the developer can see the IDE software and test the game.
Internet connection	An internet connection will be required to download the required development software (VS code, web browser, p5.js library) and communicate with the stakeholders via electronic mail.
Keyboard and mouse (with relevant drivers)	Both keyboard and mouse inputs are required for developer/stakeholders to play the game successfully. For example the keyboard is used to move the player character and to return to the menu. The mouse is used so the user can click on various menu options, including changing difficulty and customising the character.

Software requirements

Requirement	Justification
Visual Studio Code	VS Code is a lightweight and powerful source code editor developed by Microsoft. I am making use of VS Code due to its expansive range of developer-friendly features, coupled with its compatibility with JavaScript and p5. It offers an intuitive interface, syntax highlighting, and intelligent code suggestions. With a vast ecosystem of extensions, it can be customised and extended for various programming languages and tasks. VS Code integrates with Git for version control and collaboration. It provides robust debugging capabilities, task automation, and an integrated terminal. Its user-friendly interface, strong customization options, and efficient

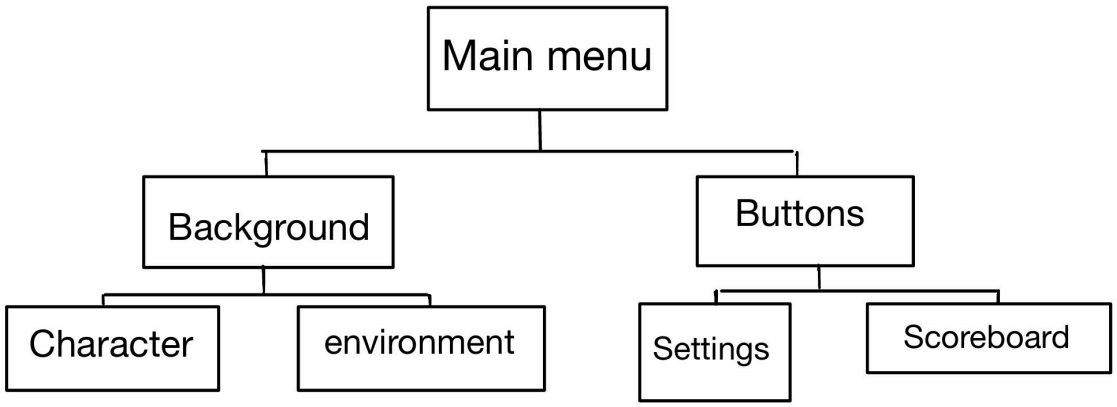
	coding features have made it a popular choice among developers.
P5.js	p5.js is a JavaScript library designed for creative coding and interactive graphics. With just a few lines of code, p5.js allows you to create animations, visual effects, and interactive experiences in web browsers. It provides a simple and intuitive API for handling graphics, user input, and audio. p5.js supports a wide range of features, including drawing shapes, manipulating pixels, handling mouse and keyboard events, playing sounds, and loading media files. It is beginner-friendly, making it accessible to those new to programming, while also offering advanced features for more experienced developers. Overall, p5.js empowers developers to unleash their creativity and build interactive web-based projects.
Live server VS Code extension	Live server allows the developer to execute the code and test the software on a local site. This means the code can be fully functional without being hosted on an actual server, which is more budget friendly and makes testing quicker and easier.
Operating system: Microsoft Windows 10 or newer MacOS X High Sierra or newer	Required to run Visual Studio Code

ITERATION 1

DESIGN

D1 - Problem decomposition

In order to be able to efficiently manage the series of tasks that my solution consists of, the computational principle of decomposition is crucial.
The first iteration of my proposed solution will involve the main menu.
The following diagram illustrates a top-down approach to these initial features:

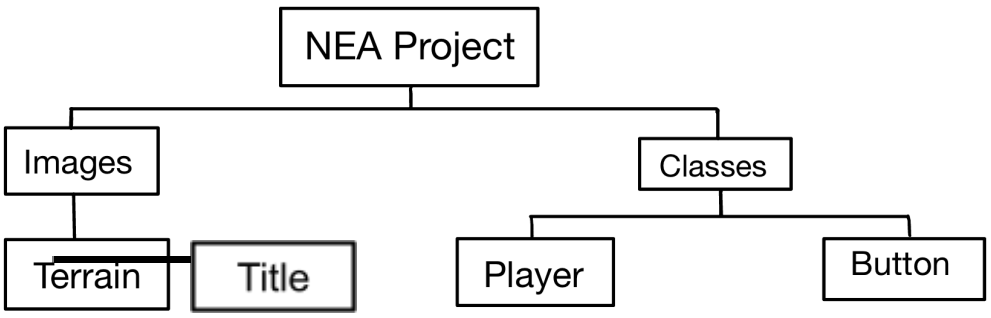


The above diagram illustrates the initial frame for my game. The reason for these initial features is that they provide a backbone from which the rest of the project will stem. By specifying the different regions of the game and options to the player, I am able to go further into each component incrementally, providing me a structured approach to development.

The background will feature two main components, a ghost object of the player character and the environment. The character object is part of the background because upon starting the game, control will be given to the user and gameplay will start properly.
The use of decomposition means that if the player does not load correctly or there is some graphical bug, I can debug the issue with confidence that I will not affect other areas of the game. The character can be further broken down into the body of the car, and the wheels. These will be drawn using the p5 functions and the drawing of the player will be done in a class method called player.draw().
The environment I am using in this project can be further split into two more sub-components. These are the background, and the road. The background will simply be a solid colour determined by the background-colour variable in p5. The road will be a png image which is loaded into the website and will be layered on top of the background.

The idea of decomposition can easily be applied to the construction of buttons in my game. I will use OOP to invoke a button class. Then from this, each of the buttons used in my game (including those in later iterations) will be built.
The use of class methods and attributes to code my buttons will make debugging so much easier as buttons are manipulated mostly by methods from within a class file, so I as the developer can narrow down the location of any issues.

D2 - Structure of solution



This iteration will be implemented using a top-down file structure. I am using object oriented programming in my solution, and so each class will be its own file in a folder called ‘Classes’ which will lie in the root folder.
I will be using few images in my project, however it is important to keep all files organised hence I am keeping these images in an ‘Images’ folder which will also lie in the root folder of my code.

The Player class will be used to create an instance of the player’s character for every new game or respawn:

- startPosX is the x-coordinate of the player’s start position
- startPosY is the y-coordinate of the player’s start position. This will not change as the player will spawn at the same height each time the game starts. The reason for this is to ensure they are not advantaged/disadvantaged in terms of game progression compared to previous attempts
- colour is the colour of the player’s character. The user will later be able to alter this to their liking in the character customisation section therefore a getter and setter has been implemented to allow for easy alterations. The attribute is a string because the colour selection menu will consist of discrete options, as opposed to a colour picker.

The button class is a very important component of my project. This is because throughout the entirety of development, several buttons will be implemented. For example, start, difficulty, customisation, and each of the difficulty options after the difficulty screen is reached

- text is the string input for the text that will appear on the button

- height is vertical measurement for the button dimensions
- width is the horizontal measurement for the button dimensions
- centreX and centreY are the coordinates for the centre point of the button. This allows for the button to be arranged naturally wherever is necessary
- getText() and setText() methods will be used to efficiently return and overwrite the text present on the button

D3 - Algorithms

In this iteration of the project, the required algorithms will be:

- Classes (Player and Button); attributes, constructors, getters and setters.
- Initialization of background and title

These algorithm plans will be appropriate to create a functional first iteration of Motor Mayhem

Player Class

This is the pseudocode for creating the class Player. It allows for objects to be created efficiently with the constructor method using spawn coordinates and colour as inputs.
The constructor method allows the character to spawn at given coordinates and because I will later be referring to these coordinates to program character movement, I have included getter and setter methods to better prepare myself. These methods form the basis for creating an instance of the character therefore I can confidently refer to them in coding my first iteration of the character.

```
public Class Player
     private startX
     private startY
     private colour

     public procedure new(givenX, givenY, givenColour)
          startX = givenX
          startY = givenY
          colour = givenColour
     endprocedure

     public function getX()
          return startX
     endfunction

     public procedure setX(newX)
          startX = newX
     endprocedure
```

```
     public function getY()
          return startY
     endfunction

     public procedure setY(newY)
          startY = newY
     endprocedure

     public function getColour()
          return colour
     endfunction

     public procedure setColour(newColour)
          colour = newColour
     endprocedure
endclass
```

Button Class

This is the pseudocode for the class Button. The constructor method allows for simple and effective instances of the class, by taking the necessary attributes as parameters. Only one of the buttons being developed in iteration 1 will be functional by the end, however it is important to fully set up the Button class in preparation for other buttons being given functionality.

It is unlikely that I will need to access/overwrite the text of the buttons however it is good OOP practice to include getters and setters for key attributes in the event that I do need to use them. In that case I would have more time to figure out how to optimise the problem because I have already provided myself with the necessary tools to do so.

This code equips me to be able to meet the success criteria of iteration 1, in that a user will be able to click start and their character will be ready.

```
public Class Button
     private text
     private height
     private width
     private centreX
     private centreY

     public procedure new(givenText, givenHeight, givenWidth,
givenCentreX, givenCentreY)
          text = givenText
          height = givenHeight
          width = givenWidth
          centreX = givenCentreX
          centreY = givenCentreY
```

```
     endprocedure

     public function getText()
          return text
     endfunction

     public procedure setText(newText)
          text = newText
     end procedure
endclass
```

Background/Title

For the programming of the background and title, I will not be taking advantage of object oriented programming, this is because there will never be more than one instance of each of them, and they are unlikely to need changing.
Both of these components will use p5 methods to be drawn to the screen.

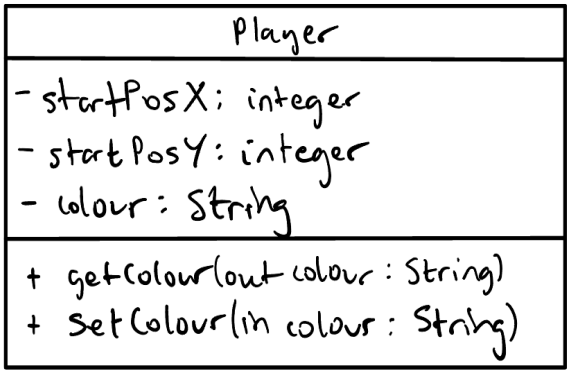
When written with correct JavaScript syntax, the following code should correctly display the relevant menu components required for iteration 1:

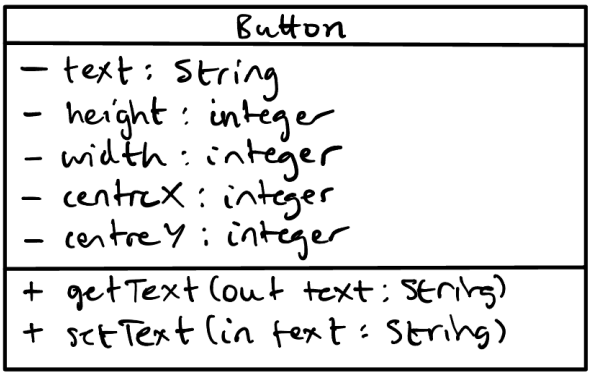
```
function setup()
     fill background(colour)
     load image(road)
     load image(title)
endfunction

function draw()
     draw road, title
     draw title
endfunction
```

D4 - Variables and validation

Having written the algorithms I intend to use for the first iteration of the project, I must identify what the key variables and data structures are which I am planning to use within the code. This is important so that I can keep track of all the most important variables and what they do and how they are going to work, and also so I can avoid implementing unnecessary variables which could easily introduce bugs into the code.

First I will look at the variables involved in the player class, and justify their use. Then I will do the same for the button class.





Variable name	Description/justification	Validation
startX, startY	the coordinates at which an object of class Player will spawn on the screen in order for the user to be able to start playing. These will obviously be integer values because the javascript canvas uses a numeric coordinate system	Range check - Must be positive to appear on the javascript canvas. startPosY must be small enough so that entire car is visible high up enough on the screen
colour	This will store the colour of the player's car to enable an element of customisation for the user. It will be of the string data type because that allows for easier and readable inputs.	Presence check - Cannot be null, as the car must be a colour to be visible to the user. Format check - String for easy discrete customisation, allows input to either be in hex code format or the name of the colour. Could also be passed as R, G, B values because p5 allows it, however my

		code will only use hex codes/names
text	the text which is to be displayed on a button in various parts of the game. This is obviously a string because the text displayed will be in English. This is required because otherwise the buttons would be unlabelled and the user would have no sense of navigation in the menu	type check - must be a string to read button labels
centreX, centreY	the coordinates of the centre point of the button so it is easy to plan where it will be placed. These coordinates are required to actually draw the button in the first place. I chose to use centre coordinates instead of corner coordinates to make it easier to calculate the appearance of the buttons on the screen.	Range check - Must be in the interval {0, 800} because the dimensions of the canvas are 800x800 pixels
Height, width	The size of a button object is defined by its horizontal and vertical measurements. These variables are required by the rect() function. Without them, the buttons cannot be drawn. Each of the menu buttons will have the same height/width	Range check - Cannot be null or small, because the button needs to be visible. Should definitely not exceed 800 to ensure entire button is visible on the canvas

Extra variables

Variable name	Description/justification	Validation
background colour	This is the string/RGB value of the colour of the p5 backdrop. Without this, the background defaults to white, which is not what I want for the theme of Motor Mayhem	Type check - must be a string or RGB value for p5 to recognise it as a usable value

D5 - Usability features

Feature	Justification/Implementation
1 - Accessibility	My game will accommodate the needs of those with impaired vision. Text colours contrast with background to maximise visual compatibility, along with a bright colour palette. Visual accessibility matters greatly in web arcade games for inclusive play. Clear visuals cater to diverse players, ensuring everyone can enjoy the game regardless of visual abilities or preferences.
2 - Navigability	Smooth navigation in arcade game menus enhances user experience. Easy-to-use menus ensure players swiftly access game features, maintaining engagement and reducing frustration during gameplay. The menu will be easy to navigate with clear routes for different menu options. Buttons are designed simply yet enticingly to engage the user.
3 - Compatibility	The game will be playable across a range of browsers on different computers. The majority of modern browsers will be sufficient to run the game. Hardware/software environments which meet the requirements stated earlier will have no trouble executing the program. I'm deciding to only use p5.js 2D functions because adding a third dimension requires webGL support in browsers. But not all updated browsers have this, so I'm keeping things in 2D to make sure my solution works for everyone. I could have used an older library that works with more browsers, but those have fewer features and info than p5.js. I chose p5.js because it's got lots of features and helpful info, even though it might not work on every single browser out there.
4 - Load times	My program will feature very fast load times to ensure that those with slower internet connections are still able to open the game and play it comfortably. This means that the file sizes for the game must be kept low so that less data must be downloaded in order to play the game. If file sizes are too large,

	players with slower internet connections will take too long to load the game and therefore will limit the accessibility of the solution.

D6 - Iterative test plan

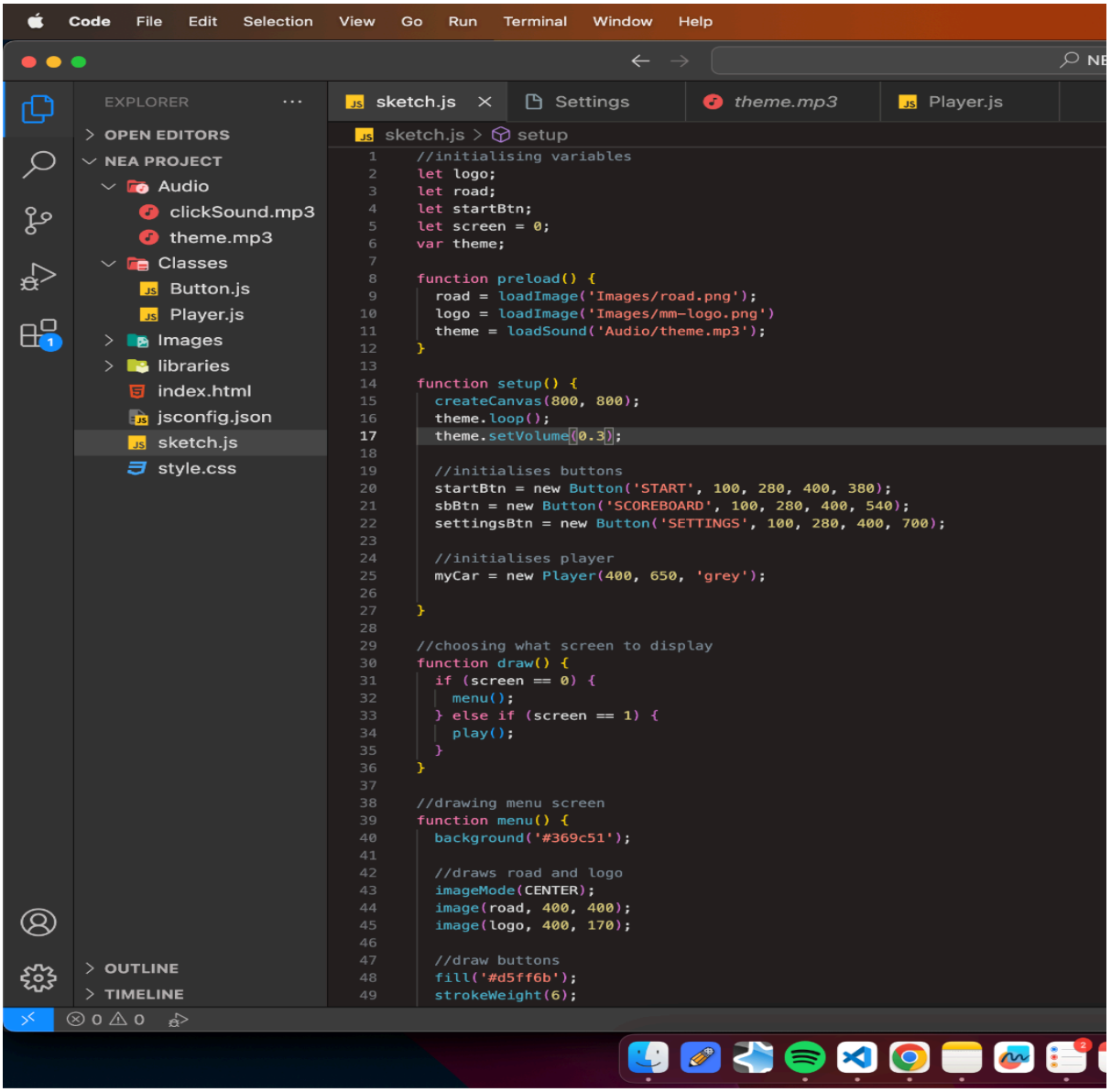
Iteration.test	Success criteria	Test details	Expected outcome
1.1	(1)	-Run index.html in 'NEA project' folder	All components of the main menu appear on screen and in a central layout; title, background, buttons.
1.2	(1)	-Run index.html in 'NEA project' folder	Instance of player is created and is presented in correct starting position
1.3	(2)	-Run index.html in 'NEA project' folder	Music will start to play when the program starts. Music volume decreases appropriately when the player starts the game.

D7 - Beta test plan

Test reference
Functional test 1 Tester: Stakeholder Success criteria: 1 - The menu must present the title and start, customise, and difficulty buttons on a background
Usability test 1 Tester: Developer Usability criteria: 1 - Accessibility Expected outcome: Menu buttons are clearly labelled and are highly visible by contrasting text on its background

For this iteration, there is no need for robustness testing as the only input allowed so far is clicking with the mouse on the area of the screen that is the start button, hence there is no way a user could input something harmful.

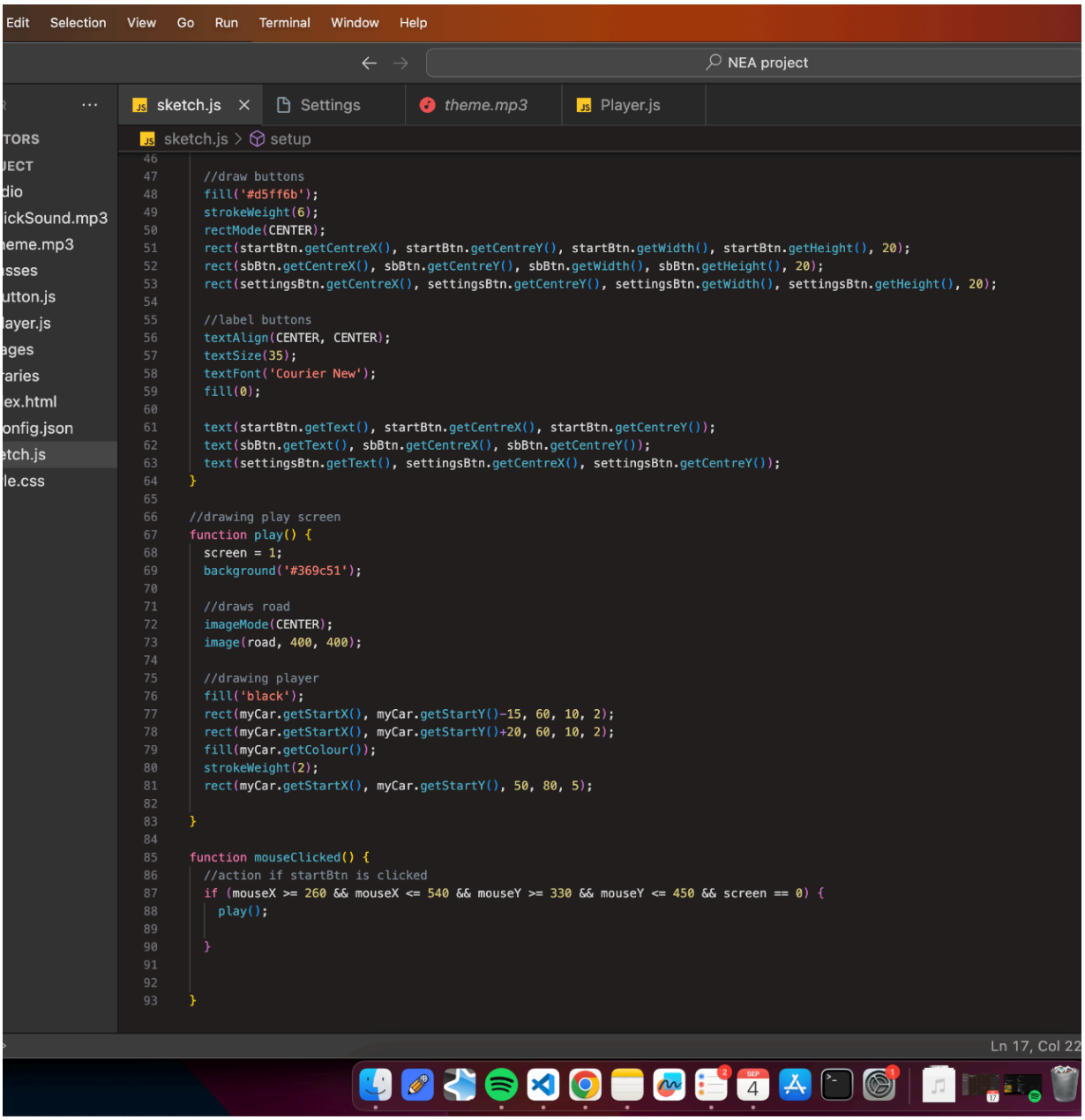
D8 - Iterative Development



I set up the project with a simple file layout. I had folders for audio, images, and classes. I also needed a sketch.js file for the main game program and an index.html file so I was able to execute the code in a local web browser, for debugging purposes later on.

I started by setting up the setup() and draw() functions. I decided to take advantage of a screen system for my game, because I knew there would be several screens by the end of development. This means creating a different drawing function for each game screen and entering each function depending on a variable called ‘screen’, which would change when buttons were clicked. I felt this decomposed my program very well and

The first thing to program was the background. After researching how to add images to p5.js, it was clear that I needed to include a preload() function to initialise these images. I used a free logo maker to create a title-logo for Motor Mayhem, as well as finding a fitting 2d road png to use as the game's terrain.
After positioning the images to make the background look flawless, I began instantiating the buttons.

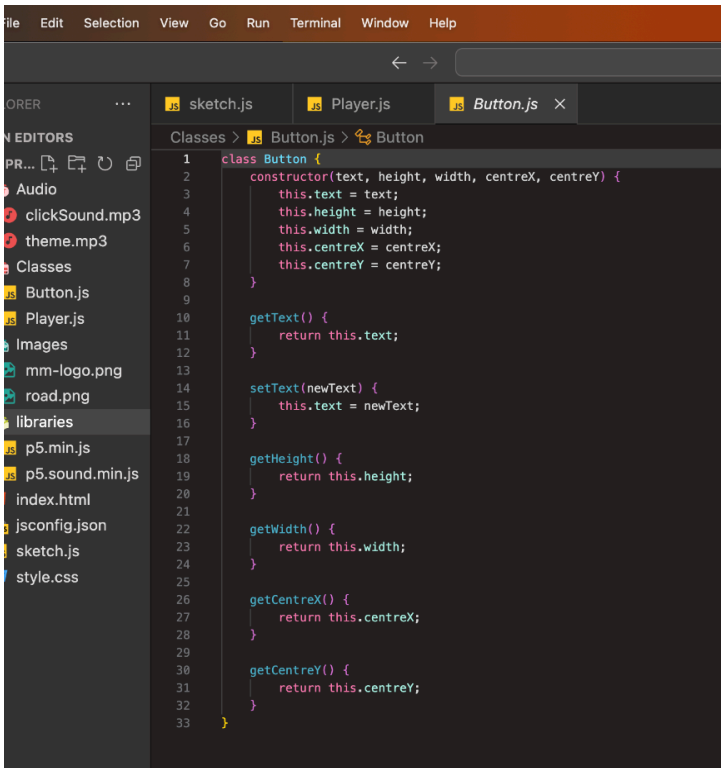


I began by initialising each menu button in the setup() function. This was made easy by my implementation of object oriented programming, because all it took was calling the constructor function for each button and passing in suitable values. I made sure to position the buttons quite centrally on the screen, but made sure to leave room for the title image. After initialisation, I drew the buttons using the p5 rect() function and passing the appropriate button attributes. The next logical step was to write the labels for each button to maintain navigability in my project, using the text() in p5 and again passing suitable attributes as parameters. The penultimate task for iteration 1

was to include theme music. This was simply completed by loading the theme music file in the preload() function and then using the loop() method to play it constantly. However, when I initially added the music it was far too loud, to the point where it would be distracting for players. For this reason, I used the setVolume() method to decrease the volume substantially to 30%.
At this point, the menu screen was in a good place as I had successfully implemented success criteria 1 and 2, so I decided to move on to coding the functionality of the start button, the last thing to do in the first iteration of Motor Mayhem.

I had two options when creating the button functionality. I could either create functional javascript button types or use the mouseClicked() function with coordinate conditions. I decided to use the second approach, because this meant I would not have to concern myself with CSS styling rules to make the button look the same as it did at this point. The functionality was programmed simply, using an if statement with mouseX and mouseY checks to determine if the user was actually clicking on the button. An additional screen == 0 check was included to make sure this button click only took effect on the menu screen, where the buttons were visible.
Refer to Iteration 1: D16-17 for iterative testing documentation.
This marks the completion of iteration 1.

Button class:



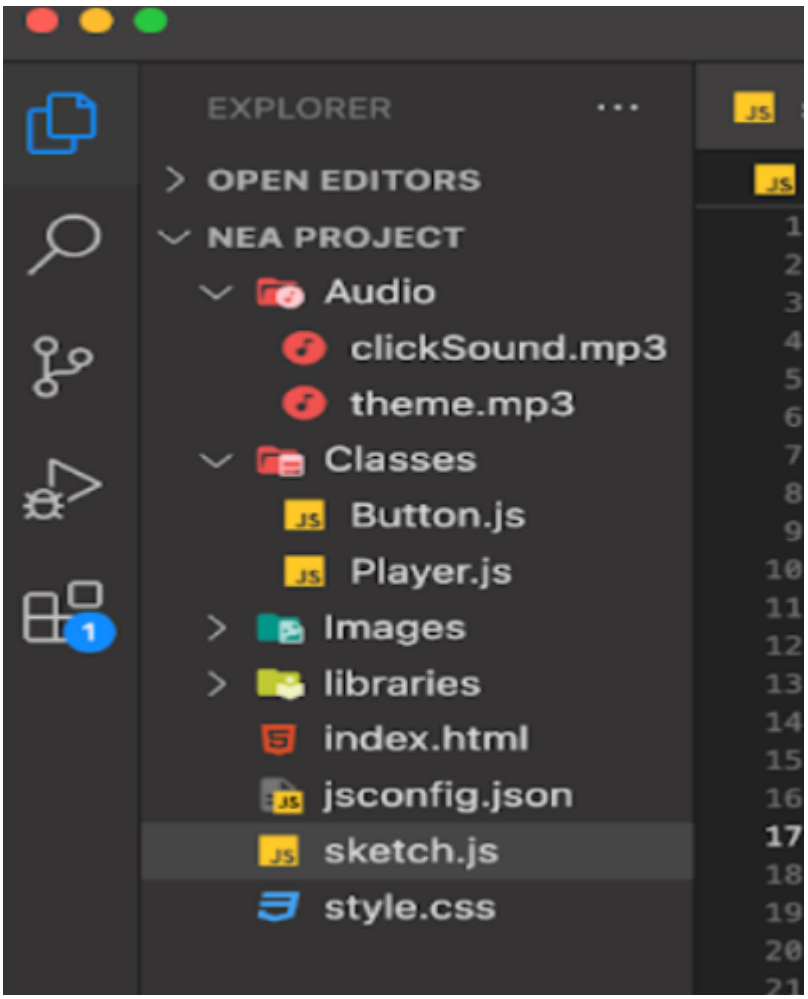
This is a screenshot of the button class as it stands after the completion of iteration 1.
I added getters and setter methods for all of the attributes, despite only planning for the text attribute in the design stage. This is because I needed to refer to the dimensions of the button objects to draw them with the rect() function.

Player class:

The player class was created obviously so I could easily create instances of the player in the game. Despite there only being one player on the screen at any given time, object oriented programming helps to decompose the code and therefore makes the code easier to understand, test, and debug.
The methods in the Player class were programmed exactly as described in the design stage and this allowed me to program the first iteration of my solution with minimal debugging.

D9 - Structure and modularity

This is a final look at the file structure of my solution upon the completion of iteration 1.
It is exactly how I planned it back in D2.



D13 - Prototype 1

https://drive.google.com/file/d/19tZAvsE9pl4PjAwFbpMat4CsWogJMzCA/view?usp=drive_link

D14 - Review of Iteration 1

Accomplishments

Criteria No.	Criteria description	Status
1	Navigable interface	✓
2	Background music	✓

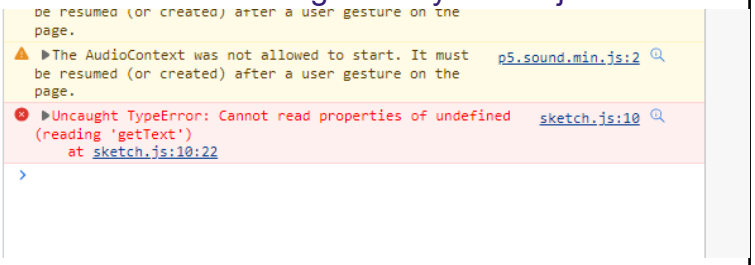
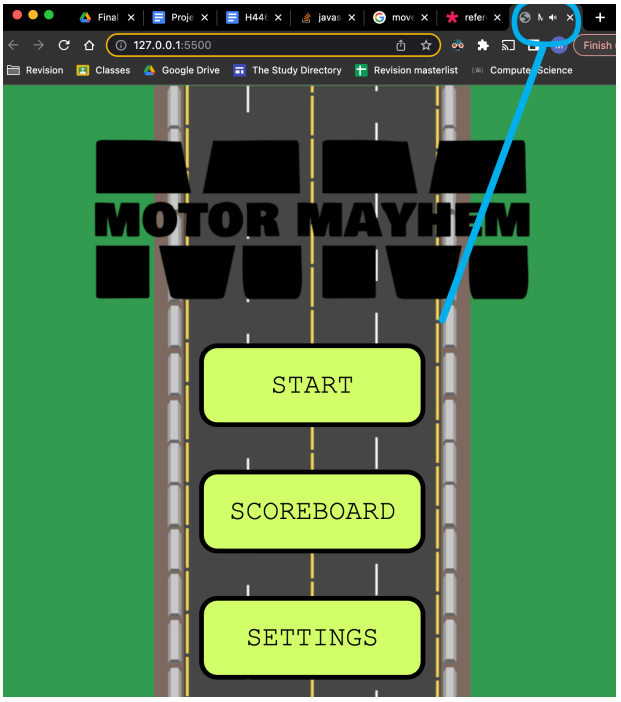
Modifications

Initially, I intended to include an instance of the character on the menu screen which would then be controlled once title and menu buttons disappeared.
However I realised that fitting the character onto the screen would take away from the appearance of the menu entirely, so I decided against it. I added a start button to more closely follow the old-school theme.

Stakeholder Feedback

Stakeholder	Feedback
Tom	“Your game's first look is pretty good. The menu is super slick, the theme is nice, and character spawns smoothly. I wonder what's next.”
Alex	“The menu screen looks very enticing to me. I would definitely be interested in playing this game. Your first version is off to a solid start!”
Andrew	“Seems to nicely sit within the theme of old-school arcade games, which I especially like. Looks promising, can’t wait to play!”

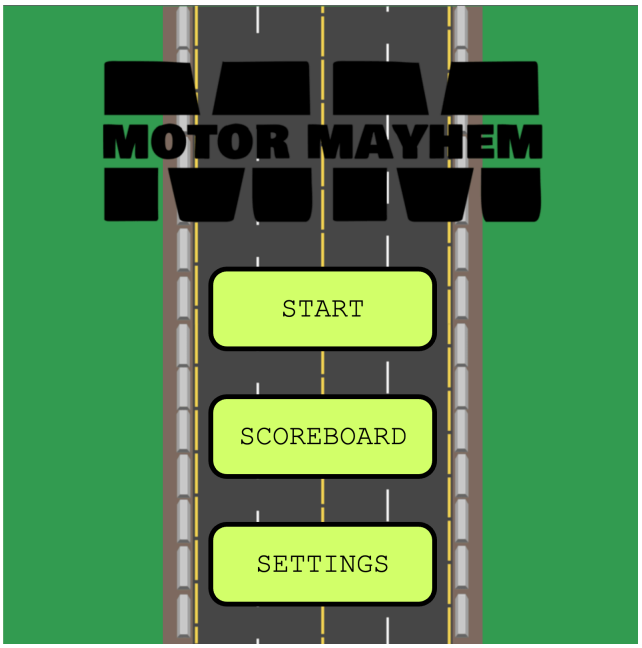
D16 - Iterative testing

Iteration.test	Outcome	Evidence
1.1	Fail	Class methods are not recognised by sketch.js file 
1.2	Fail	Absence of car on screen
1.3	Pass	

D17 - Failed tests/remedies

I ran test 1.1 and it failed.
The error shown above presented itself.
Before invoking any instances of class ‘Button’ the background and title would appear perfectly, however once I tried to include the menu buttons, the program crashes every time.

The issue was resolved by uninstalling the ‘Live Preview’ VS code extension and using instead the in-built ‘Live Server’ extension:



Upon re-running test 1.1, it passed

Test 1.2 failed at first.
The car was created as an object of class Player and instantiated with correct syntax.

The issue was in the class code, specifically the constructor method.
The colour attribute was initialised as:

```
    globalThis.colour = colour;
}
```

Upon changing to the following, the issue was fixed and test 1.2 passed

```
    this.colour = colour;
}
```

ITERATION 2

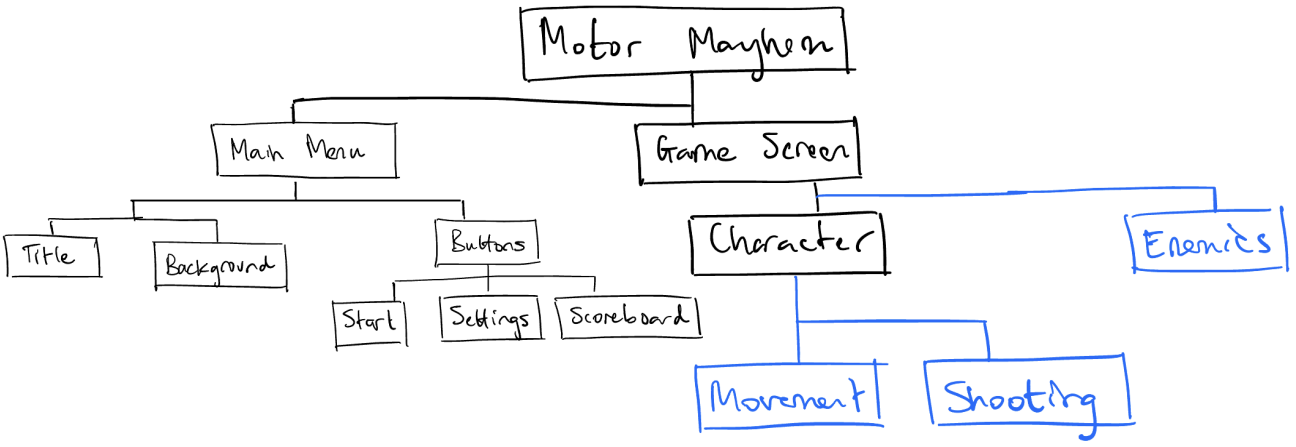
DESIGN

D1 - Problem decomposition

The second iteration of my solution will include the following:

- character movement
- character shooting
- enemy spawn/movement
- enemy death

The second iteration is more complex than the previous one therefore it is especially important to take full advantage of the decomposition technique. There are not necessarily more sub-problems, however each of them are more complicated. Here is a top-down structure chart of the components of my game so far.
I have added in blue the features which are new to iteration 2.



As seen in the figure above, this iteration covers the combative interaction between player and enemy, which does not include the aftermath of said interaction (score, lives, etc).

The movement of the character can be broken down into moving left and right, using the ‘a’ and ‘d’ keys respectively. Moving the character is the most straightforward aspect of Motor Mayhem’s second iteration, hence only light use of the concept of decomposition.

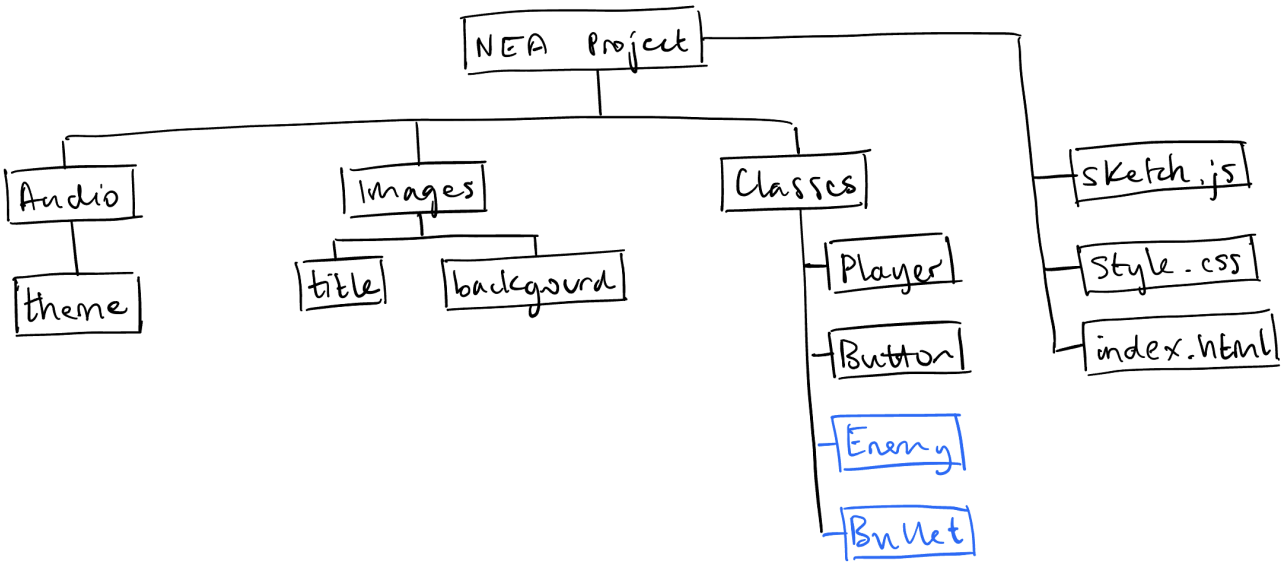
The shooting system can also be separated into two main subproblems: creating the bullet projectiles, and moving them up the screen. I have decided to create a new class Bullet for the bullets because it is most likely that there will be a large number of bullets being fired and therefore many bullets will be visible on the screen at one time. To aid in manipulating a large number of objects in a short amount of time, I have chosen to use a ‘bullets’ array to store the bullets currently on the screen.

In order to move the bullets up the screen, after drawing them with the rect() function upon pressing SPACE, I will simply use similar mechanics as the player movement, where the vertical position of the bullets will be decremented (y position 0 is the top of the canvas). When a bullet reaches the top of the screen (and later when it hits an enemy), it will be removed from the array to ensure the array size does not exceed hardware capabilities, therefore maintaining robustness.

The last component to decompose in iteration 2 is the enemies. These need to be successfully created, moved and there needs to be a collision system.
Due to the numerous nature of the enemies in Motor Mayhem, I will be creating another array ‘enemies’ to store them. Every 2 seconds I will be adding a new enemy to the array to ensure that enemies spawn at a good frequency.
After the object is initialised and in the array, I can draw them using the rect() function once again.
Their horizontal spawn locations will be random, and, oppositely to the bullets, move down the screen by continuously incrementing their y values.
To solve the problem of enemy-bullet collisions, I need to make a collisions() function which checks if a collision has happened i.e the coordinates of a bullet are within the enemy hitbox.
Finally, the bullet and enemy involved need to be removed from their respective arrays.

D2 - Structure of solution

The addition of enemies in my solution calls for the addition of an enemy class. This is because every enemy will act the same, differing only in location on the screen. The player also needs to fire projectiles at enemies. These ‘bullets’ will again be instances of a class ‘Bullet’, to increase efficiency in programming them
The following figure depicts the revised file structure following these changes:



The enemy class will simply consist of coordinates and a colour. PosY and colour are likely to be static, however it is good practice to make them attributes in case they do need referring to/changing

- enemyX is the x-coordinate of the enemy. This will not change from the moment they spawn as they will move vertically down the screen towards the player character. this will be randomly generated to ensure enemies cover the screen
- enemyY is the y-coordinate of the enemy. This will be constantly incremented to give a steady speed of descent down the screen
- colour will be the colour of the enemy car. The enemy will follow an identical design to the player character itself, differing only in body colour, hence the attribute

The bullet class will likely be the simplest class design in my solution. Every bullet will be identical, differing only in position from which they are shot, which will be that of the current player x-coordinate

- bulletX is the x-coordinate from which the bullet is shot. This will be the same as the x-coordinate which the player is currently at as they are shooting the bullets. It will remain constant as it travels vertically up the screen
- bulletY is the y-coordinate of the bullet. This (similar to enemyY) will be constantly decremented to allow for smooth motion up the screen
- bulletColour is the colour of the bullet which will be initialised as ‘black’ and will not change at any point, hence no getter/setter method

D3 - Algorithms

For iteration 2, I will need to implement the following algorithms:

- Enemy class (including getters/setters)
- Bullet class (including getters/setters
- movement of bullet
- movement of enemy

Enemy class

The enemy class will be the template from which all enemies will be created. It is important to make use of encapsulation by declaring the attributes as private, in order to prevent accidental changes to their values. This is more likely to happen than in the first iteration, because we are introducing movement and collisions.
I will be coding getter and setter methods for each attribute, apart from enemy colour, because they will be accessed at some point during the success criteria for iteration 2.I will also be creating a move() and draw() method in the class rather than the main program to maintain good OOP practices and to break up the code.

```
public class Enemy
     private enemyX
     private enemyY
     private enemyColour

     public procedure new(givenX, givenY, givenColour)
          this.enemyX = givenX
          this.enemyY = givenY
          this.enemyColour = givenColour
     endProcedure

     public function getX()
          return this.enemyX
     endfunction

     public function getY()
          return this.enemyY
     endfunction

     public procedure setY(newY)
```

```
            this.enemyY = newY
        endprocedure

        public function getColour()
            return this.enemyColour
        endfunction

        public procedure draw()
            draw enemy
        endprocedure

        public procedure move()
            increment enemy y-position by 2
        endprocedure
endclass
```

Bullet class

Like the enemy class, it is absolutely necessary to code the bullets as objects in a class, because as previously mentioned there will be a lot of bullets being processed very quickly.

It goes without saying that I have planned the getter and setter methods for the bullet class. I did not include a set() method for the x-coordinate of the bullet as this will never be written to. This is because it travels vertically up the screen.

The bullet class, like the enemy class, will also feature a draw and move function within the class file itself to make the most of OOP.

```
public class Bullet
        private bulletX
        private bulletY
        private colour

        public procedure new(givenX, givenY)
            this.bulletX = givenX
            this.bulletY = givenY
            this.colour = 'Black'
        endprocedure

        public function getX()
            return this.bulletX
        endfunction

        public function getY()
            return this.bulletY
        endfunction
```

```
     public procedure setY(newY)
          this.bulletY = newY
     endprocedure
     public procedure draw()
          draw bullet
     endprocedure

     public procedure move()
          decrease bullet bullet y-position by 2
endclass
```

Additional code

In the main program, there is some code which needs to be written in order to tie together the functionality presented in iteration 2.

Firstly there needs to be a keyPressed function to deal with the space key being used to fire bullets, and the ‘a’ and ‘d’ keys being used to move the player character.

I am including a newEnemy() function to set up the spawning of enemies. The subroutine will, every 2 seconds, create a new object of class Enemy with a random x coordinate and append it to enemies array for accessing.

Lastly, I need to continuously be updating the positions of new bullets and enemies by iterating through each array and calling their move and draw functions. Lastly, a collision function will be used to check that a bullet has hit an enemy

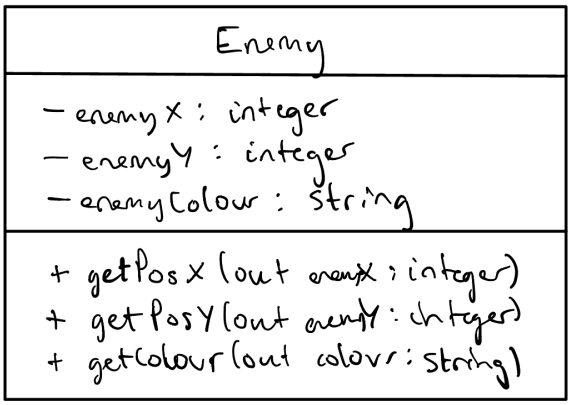
```
function keyPressed()
     if (keyCode == SPACE) then
          draw bullet from player.getX(), player.getY()
          move bullet up
     else if (keyCode == a) then
          decrease player x coordinate by 2
     else if (keyCode == d) then
          increase player x coordinate by 2
     endif
endfunction
function newEnemy()
     every 2 seconds: instantiate new enemy
endfunction

for i = 0 to enemies.length - 1
     enemy[i].draw
     enemy[i].move
next i
for i = 0 to bullets.length - 1
     bullet[i].draw
     bullet[i].move
next i
```

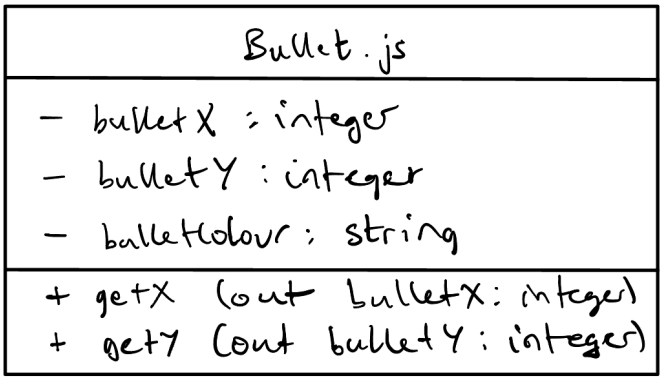
```
function collision(bullet, enemy)
     if bullet x is within enemy x AND bullet y is within
enemy y then
          return true
     endif
endfunction
```

D4 - Variables and validation: 2

Having written the algorithms I intend to use for the first iteration of the project, I must identify what the key variables and data structures are which I am planning to use within the code. This is important so that I can keep track of all the most important variables and what they do and how they are going to work, and also so I can avoid implementing unnecessary variables which could easily introduce bugs into the code.

First I will look at the variables involved in the enemy class, and justify their use. Then I will do the same for the bullet class.





Variable name	Description/justification	Validation
enemyX, enemyY	These variables are the integer values which will be used as enemy coordinates. These are required because without them, enemies would not appear and this would result in not meeting success criterion 4	range check - must be in interval {0, 800} to appear on canvas without issue

enemyColour	This will store the colour of the enemy cars, which will be a constant solid colour because there is no element of customisation for the enemies. It will be of the string data type because that allows for easier and more readable inputs in my code.	presence check - cannot be null as the rect() function requires an argument for colour parameter
bulletX, bulletY	These variables are the integer values which will be used as bullet coordinates. These are required because without them, bullets would not appear and this would result in not meeting success criterion 4	range check - must be in interval {0, 800} to appear on canvas without issue
bulletColour	This will store the colour of the bullets which are fired by the player, which will be a constant solid colour because there is no element of customisation for the bullets. It will be of the string data type because that allows for easier and more. It is important that they contrast strongly with the environment because the user needs to be able to see them while they are moving up the screen	presence check - cannot be null as the rect() function requires an argument for colour parameter

D5 - Usability features

New features

Feature	Justification/implementation
Accessibility	My game will accommodate the needs of those with impaired vision. Text colours contrast with background to maximise visual compatibility, along with a bright colour palette. Visual accessibility matters greatly in web arcade games for inclusive play. Clear visuals cater to diverse players, ensuring everyone can enjoy the game regardless of visual abilities or preferences. Player character and enemies will be coloured using a bright palette to maximise the visibility for users. This is important as without proper visibility of the character, bullets, or enemies, a user may struggle with playing the game entirely which would make for a less than desirable user experience Music will commence when the program runs. The volume will decrease when the user starts a game. This is a crucial addition for my game as it greatly helps those who are hard of hearing to confirm whether or not they have started the game.
Navigability	Smooth navigation in arcade game menus enhances user experience. Easy-to-use menus ensure players swiftly access game features, maintaining engagement and reducing frustration during gameplay. The menu will be easy to navigate with clear routes for different menu options. The back button at each stage allows for an easy return to the previous stage. Buttons are designed simply yet enticingly to engage the user. The way in which a user progresses through the game is clear, by ensuring it flows seamlessly with a vertical line of motion. This is achieved by implementing a clear, vertical scrolling approach to my game, allowing a user to know how to progress.
Compatibility	The game will be playable across a range of browsers on different computers. The majority of modern browsers will be sufficient to run the game. Hardware/software environments which meet

	the requirements stated earlier will have no trouble executing the program. I'm deciding to only use p5.js 2D functions because adding a third dimension requires webGL support in browsers. But not all updated browsers have this, so I'm keeping things in 2D to make sure my solution works for everyone. I could have used an older library that works with more browsers, but those have fewer features and info than p5.js. I chose p5.js because it's got lots of features and helpful info, even though it might not work on every single browser out there. It is even more important to render the on-screen objects resource-efficiently as possible because with the addition of bullets and enemies, there will be far more objects on a screen at once and I need to ensure that the game is still playable for as many users as possible by making the most of hardware resources available
Load time	My program will feature very fast load times to ensure that those with slower internet connections are still able to open the game and play it comfortably. This means that the file sizes for the game must be kept low so that less data must be downloaded in order to play the game. If file sizes are too large, players with slower internet connections will take too long to load the game and therefore will limit the accessibility of the solution. In iteration 2, the large number of values being added to arrays could be a significant cause of increasing the file size. For this reason I made sure to remove redundant objects from their arrays in order to keep file size down to minimise load times and therefore maximise the range of internet speeds with which my project can be used.

D6 - Test plan

Iteration.test	Success criteria	Test details	Expected outcome
2.1	(3)	-Run index.html in 'NEA project' folder	Player movement works without issue; user is able to move character left/right, within usable 'road'
2.2	(3)	-Run index.html in 'NEA project' folder	Upon a user pressing space, a bullet is fired from character position and travels up the screen
2.3	(4)	-Run index.html in 'NEA project' folder	Enemies spawn at appropriate positions and frequency and move down the screen at correct speed
2.4	(4)	-Run index.html in 'NEA project' folder	Upon a bullet hitting an enemy, enemy is removed from screen

D7 - Beta test plan

Test reference
Functional test 2 Tester: Stakeholder Success criteria: 3 - The character can be moved left and right using the keys 'A' and 'D' respectively Test details: ●
Functional test 3 Tester: Stakeholder Success criteria: 4 - Upon pressing the space key, a bullet is fired from the current player position. If it hits an enemy, bullet and enemy disappear Test details: ●
Usability test 2 Tester: Developer Usability criteria: 1 - Accessibility

Test details:

-

Expected outcome:
Character, bullets and enemies all present themselves clearly with bright colours on the screen to aid those with visual impairments

Robustness test 1
Tester: Stakeholder
Criteria: Array overload is prevented
Justification: If an array is added to infinitely, eventually the hardware/software will start to struggle to run efficiently when handling such a large number of values
Test details:

-

Expected outcome:
Upon a user firing as many bullets as they can, the program will stay running efficiently, as bullets are removed from array upon enemy contact or when they reach the edge of the screen

D8 - Iterative development

I began developing iteration 2 by programming the movement for the player's character. I researched the appropriate keycodes for the 'a' and 'd' and created a simple if statement using the keyIsDown() function from p5. This functionality was coded quickly and successfully, however I realised quickly that I needed to set some coordinate boundaries so that the character could only move while on the 'road' as opposed to the 'grass'.
This was more difficult than I anticipated and the details of this setback can be found in iterative test 2.1.
Ultimately, the character movement worked as intended.

```
34    //player movement
35    if (keyIsDown(68) && myCar.getStartX() >= 225 && myCar.getStartX() <= 570) {
36      //moves right while not at right boundary
37      myCar.setStartX(myCar.getStartX()+6);
38    } else if (keyIsDown(65) && myCar.getStartX() >= 230 && myCar.getStartX() <= 575) {
39      //moves left while not at left boundary
40      myCar.setStartX(myCar.getStartX()-6);
41    }
42
```

The next part of development was programming the bullets. This started off as quite straightforward because I was merely translating the bullet class code seen in D3 into JavaScript in a Bullet.js file:

```
1   class Bullet {
2       constructor(x, y) {
3           this.x = x;
4           this.y = y;
5       }
6
7       draw() {
8           fill(0, 230, 222);
9           rect(this.getBX(), this.getBY()-15, 10, 10, 2);
10      }
11
12      move() {
13          this.setBY(this.getBY()-3);
14      }
15
16      getBX() {
17          return this.x;
18      }
19
20      getBY() {
21          return this.y;
22      }
23
24      setBY(newY) {
25          this.y = newY;
26      }
27  }
```

After this was done, it was time to code the keypress to fire bullets.
I called the p5 keyPressed() function using the keyCode of the 'space' key as a condition.

Inside the if statement, I created an object of the Bullet class. I then called the bullet.move() function in the main program, but it did not work as expected because I could not fire more than 1 bullet.
See iterative test 2.2 for error info and the debugging strategy.
After fixing the issue, this is how the subroutine looked, along with the function calls to draw and move the bullets:

```
//draw bullet
for (let i = bullets.length - 1; i >= 0; i--) {
  bullets[i].draw();
}

//update bullet position
for (let i = bullets.length - 1; i >= 0; i--) {
  bullets[i].move();
}
```

```
//adds new bullet to bullets[] array
function keyPressed() {
  if (keyCode == 32) {
    bullets.push(new Bullet(myCar.getStartX(), myCar.getStartY()-30))
```

The next step was to program the enemies. I started by once again translating the class code from pseudocode in D3 into JavaScript.

```
class Enemy {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    draw() {
        fill('black');
        rect(this.getX(), this.getY()-15, 50, 10, 2); //wheels
        rect(this.getX(), this.getY()+10, 50, 10, 2);
        fill('#6b0500');
        strokeWeight(2);
        rect(this.getX(), this.getY(), 40, 70, 12);
    }

    move() {
        this.setY(this.getY()+2);
    }

    getX() {
        return this.x;
    }

    getY() {
        return this.y;
    }

    setY(newY) {
        this.y = newY;
    }
}
```

Once this was coded, I had to use the newEnemy() function planned in D3 to create a new object of class Enemy every so often. I decided to use a variable ‘interval’ to determine the frequency at which enemies spawn because I knew this would be used later on in development when programming higher levels of difficulty

```
//pushes new enemy to array every 2 seconds
function newEnemy() {
  if ((frameCount % interval) == 0) {
    enemies.push(new Enemy(random(230, 570), -25));
  }
}
```

After the objects are added to the enemies array, I need to draw them to the JavaScript canvas and then also move them down the screen through time.
I did this by calling the draw and move functions from the class in the main program, exactly like I just did for the bullets class.

```
//draw enemy
newEnemy();
for (let i = enemies.length - 1; i >= 0; i--) {
  enemies[i].draw();
}

//update enemy position
for (let i = enemies.length - 1; i >= 0; i--) {
  enemies[i].move();
}
```

Finally, I needed to program the collisions between bullets and enemies.
I started by adding the collision() function to determine whether or not a bullet has indeed hit an enemy.
I used ChatGPT to figure out the coordinate hitbox and how to incorporate that into a JavaScript function and ended up with the following function which returns a boolean value, true for collision and false for no collision:

```
//determines whether or not there was collision and returns boolean
function bulletCollision(b, e) {
  return (abs(b.getBX() - e.getX()) < 40 &&
  abs(b.getBY() - e.getY()) < 40);
}
```

However, I ran into some problems when trying to call this function in the main program.
For the full breakdown of the issue, see iterative test 2.4. After fixing the problem, this was the block of code used to act on said collisions detected in the function above:

```
//detect collision
for (let i = bullets.length - 1; i >= 0; i--) {
  for (let j = enemies.length - 1; j >= 0; j--) {
    if (bullets[i] && enemies[j] && bulletCollision(bullets[i], enemies[j])) { // null checks for both arrays
    //handle collision
    bullets.splice(i, 1);
    enemies.splice(j, 1);
    }
  }
}
```

I was very nearly finished with iteration 2, however some adjustments needed to be made in order for the enemy elimination system in my game to work flawlessly.
The first of these changes was to create a ‘game over’ effect when a player missed an enemy and it got past the player.
Using knowledge from previous issues and their solutions, I was able to solve this problem for the first time using another nested iteration through both bullets[] and enemies[]:

```
//returns to menu if enemy passes the player
for (let i = bullets.length - 1; i >= -1; i--) {
  for (let j = enemies.length - 1; j >= 0; j--) {
    if (enemies[j].getY() >= myCar.getStartY()) {
      bullets.splice(0, bullets.length);
      enemies.splice(j, enemies.length - j);
      myCar.setStartX(400);
      menu();
    }
  }
}
```

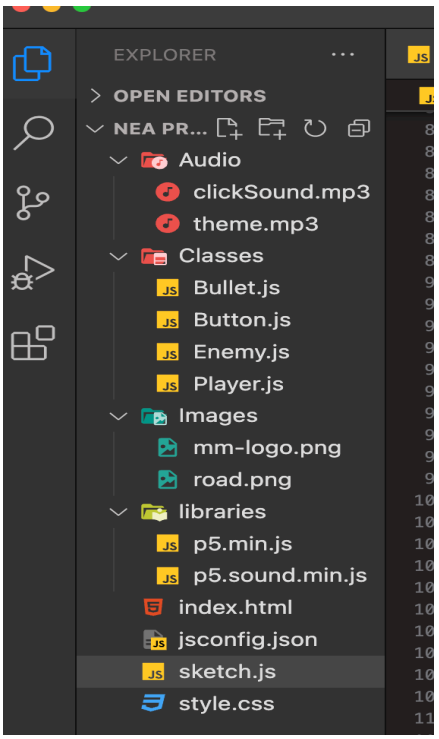
The enemy and bullet arrays needed to be completely wiped to ensure a smooth start should a user play another round of the game, as well as the character’s position needing to be reset.

The last change made while developing the second iteration of Motor Mayhem, was the return to menu functionality which was outlined in D5.
This was done simply by adding a second if statement to the keyPressed() but instead using the keyCode for ‘q’

```
function keyPressed() {
  if (keyCode == 32) {
    bullets.push(new Bullet(myCar.getStartX(), myCar.getStartY()-30))
  }
  for (let i = bullets.length - 1; i >= -1; i--) {
    for (let j = enemies.length - 1; j >= 0; j--) {
      if (keyCode == 81) {
        bullets.splice(0, bullets.length);
        enemies.splice(j, enemies.length);
        myCar.setStartX(400);
        screen = 1;
        menu();
      }
    }
  }
```

D9 - Structure and modularity

This is how my files and folders look upon completing the second iteration of Motor Mayhem. I exactly followed the structure chart which was planned out in D2



Subroutines:

```
//adds new bullet to bullets[] array
function keyPressed() {
  if (keyCode == 32) {
    bullets.push(new Bullet(myCar.getStartX(), myCar.getStartY()-30))
  }
}

//pushes new enemy to array every 2 seconds
function newEnemy() {
  if ((frameCount % 120) == 0) {
    enemies.push(new Enemy(random(230, 570), -25));
  }
}

function collision(b, e) {
  return (abs(b.getBX() - e.getX()) < 40 &&
  abs(b.getBY() - e.getY()) < 40);
}

function mouseClicked() {
  //action if startBtn is clicked
  if (mouseX >= 260 && mouseX <= 540 && mouseY >= 330 && mouseY <= 450 && screen == 0) {
    play();
  }
}
```

D13 - Prototype 2

https://drive.google.com/file/d/1ry8nO1L8SjVLwihOB6EMABJaxUS4UHOL/view?usp=drive_link

D14 - Review of iteration 2

Accomplishments

Criteria No.	Criteria description	Status
3	Character control	✓
4	Combat system	✓

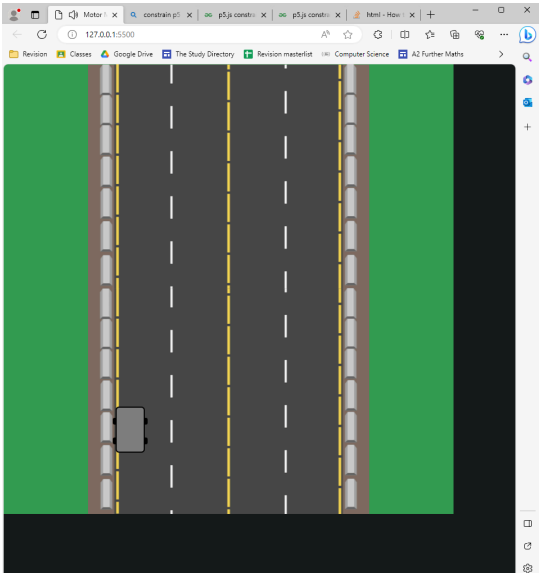
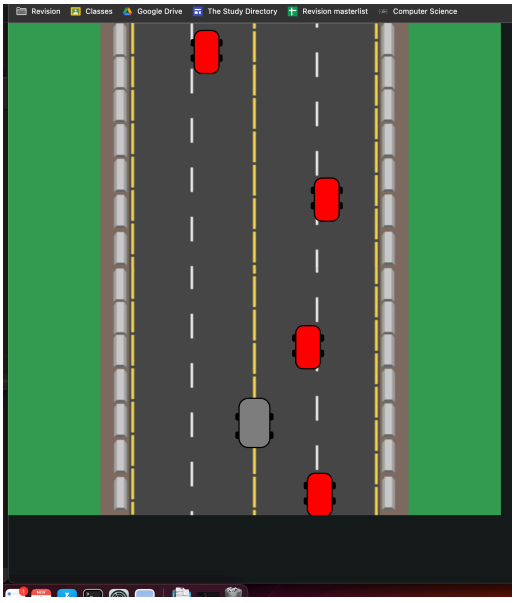
Modifications

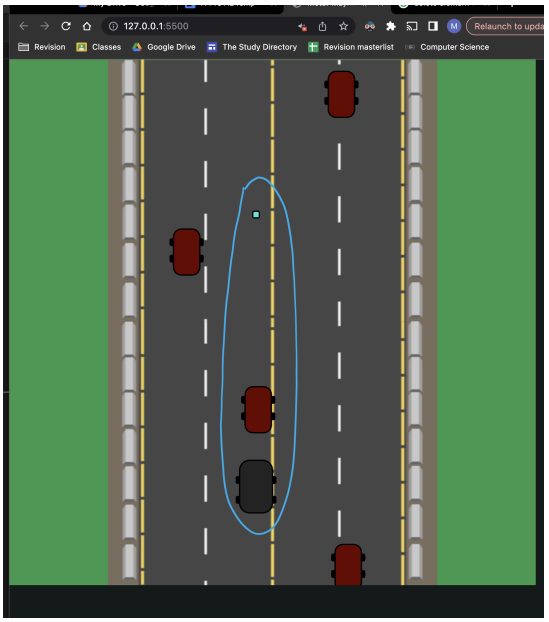
Due to the in depth nature of my iteration 2 design, only minor adjustments such as removing certain getters/setters that would never be used, such as setX() for the bullet class.

Stakeholder Feedback

Stakeholder	Feedback
Tom	Tom was very pleased with the momentum he felt the game possessed. This is because with the extensive gaming experience that Tom has, it is important that Motor Mayhem has an attractive energy so it appeals to even the most experienced gamers. Tom has made it clear that he is anticipating the difficulty adjustment because he still craves the challenge
Alex	The thing that impressed Alex the most is the seamless flow of the game, including the navigability. Because Alex does not play computer games the mechanics of Motor Mayhem will naturally be less exciting for her. This is why it's important for features like navigation and user interface to be working flawlessly. This idea will allow my game to cater for a wider audience because there are many people my game appeals to who otherwise have little to no gaming experience
Andrew	As a middle aged user who has a decent knowledge of games, what really appealed to Andrew about my game was the retro arcade feel. The combination of 2d shapes, simple controls, a theme tune, and a classic shooter system really resonated with this stakeholder. This is a good preview of the wide range of Gen X customers who should be interested in my project.

D16 - Iterative testing

Iteration.test	Outcome	Evidence
2.1	Fail	Player character gets stuck on movement boundary: 
2.2	Fail	Multiple bullets could not be drawn as there was only ever one instance of a class
2.3	Pass	

2.4	Fail	

D17 - Failed tests/remedies

Test 2.1 failed

The test failed because: I only let the car move when it was between certain x-coordinates, however when the loop repeated the previous movement meant the car was now out of bounds, so the instruction to move could not run:

```
//player movement
//if (myCar.getStartX() >= 230 && myCar.getStartX() <= 570) {
  if (keyIsDown(68) && myCar.getStartX() >= 225 && myCar.getStartX() <= 575) {
    myCar.setStartX(myCar.getStartX()+5);
    console.log(myCar.getStartX());
  } else if (keyIsDown(65) && myCar.getStartX() >= 225 && myCar.getStartX() <= 575) {
    myCar.setStartX(myCar.getStartX()-5);
    console.log(myCar.getStartX());
  }
```

I fixed it by having different x bounds for moving in different directions, as once the car is too far left, you can only move right and vice versa:

```
//player movement
//if (myCar.getStartX() >= 230 && myCar.getStartX() <= 570) {
  if (keyIsDown(68) && myCar.getStartX() >= 225 && myCar.getStartX() <= 570) {
    myCar.setStartX(myCar.getStartX()+5);
    console.log(myCar.getStartX());
  } else if (keyIsDown(65) && myCar.getStartX() >= 230 && myCar.getStartX() <= 575) {
    myCar.setStartX(myCar.getStartX()-5);
    console.log(myCar.getStartX());
  }
}
```

Test 2.2 failed

The test failed as I had not remembered to invoke a suitable system to account for multiple bullet objects being drawn on the screen.

I referred back to my algorithm plan from D3, and I created an array of bullets which I could append each object to. I drew and continuously moved these bullets by calling their functions in the main program

```
function keyPressed() {
  if (keyCode == 32) {
    //adds new bullet to bullets[] array
    bullets.push(new Bullet(myCar.getStartX(), myCar.getStartY()-30))
  }
}
```

```
//moves bullet most recently drawn
for ( i = 0; i < bullets.length; i++) {
  bullets[i].draw();
  bullets[i].move();
}
```

Test 2.2 passes this time.

Test 2.4 was difficult to resolve

Initially I had two separate for loops which iterated through the bullet and enemy arrays. A nested for loop seemed to mostly fix the problem however if I missed an enemy, then the next time I hit an enemy the game crashed.

I worked out that because the error message said that bullet x coordinate was undefined, this meant that after a bullet was removed from the array bullets, it was still being accessed.
I added null checks in the if statement before removing the bullet from the array:

```
//draw bullet
for (let i = bullets.length - 1; i >= 0; i--) {
  bullets[i].draw();
}

//draw enemy
newEnemy();
for (let i = enemies.length - 1; i >= 0; i--) {
  enemies[i].draw();
}

//update bullet position
for (let i = bullets.length - 1; i >= 0; i--) {
  bullets[i].move();
}

//update enemy position
for (let i = enemies.length - 1; i >= 0; i--) {
  enemies[i].move();
}

//detect collision
for (let i = bullets.length - 1; i >= 0; i--) {
  for (let j = enemies.length - 1; j >= 0; j--) {
    if (bullets[i] && enemies[j] && collision(bullets[i], enemies[j])) {
      //handle collision
      bullets.splice(i, 1);
      enemies.splice(j, 1);
    }
```

This fixed the issue

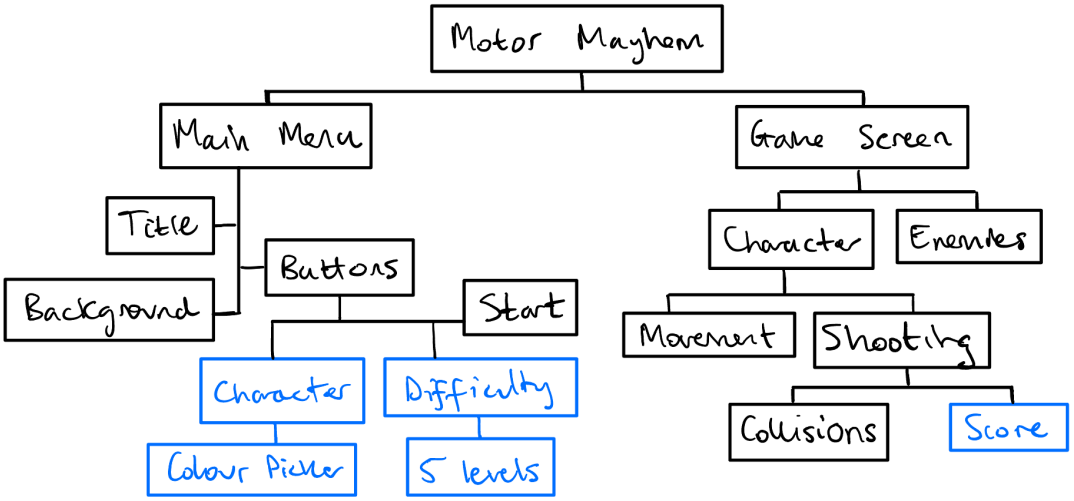
ITERATION 3

DESIGN

D1 - Problem decomposition

The third and final iteration of Motor Mayhem shifts the focus from gameplay to menu features/customization options, more specifically success criteria 5, ~~6,~~ 7, and 8:

- Scoring system
- Character customisation
- Difficulty adjustment



I have started iteration 3 by deciding not to go ahead with the sixth success criteria; scoreboard. The scoreboard feature would take too much time to implement for how important it is. A game of this simplicity does not crucially need to track player progression.
I also removed the settings option as this would just add unnecessary navigation to reach character and difficulty customisation. I will instead be moving those options to the main menu as the removal of scoreboard and settings has made space for

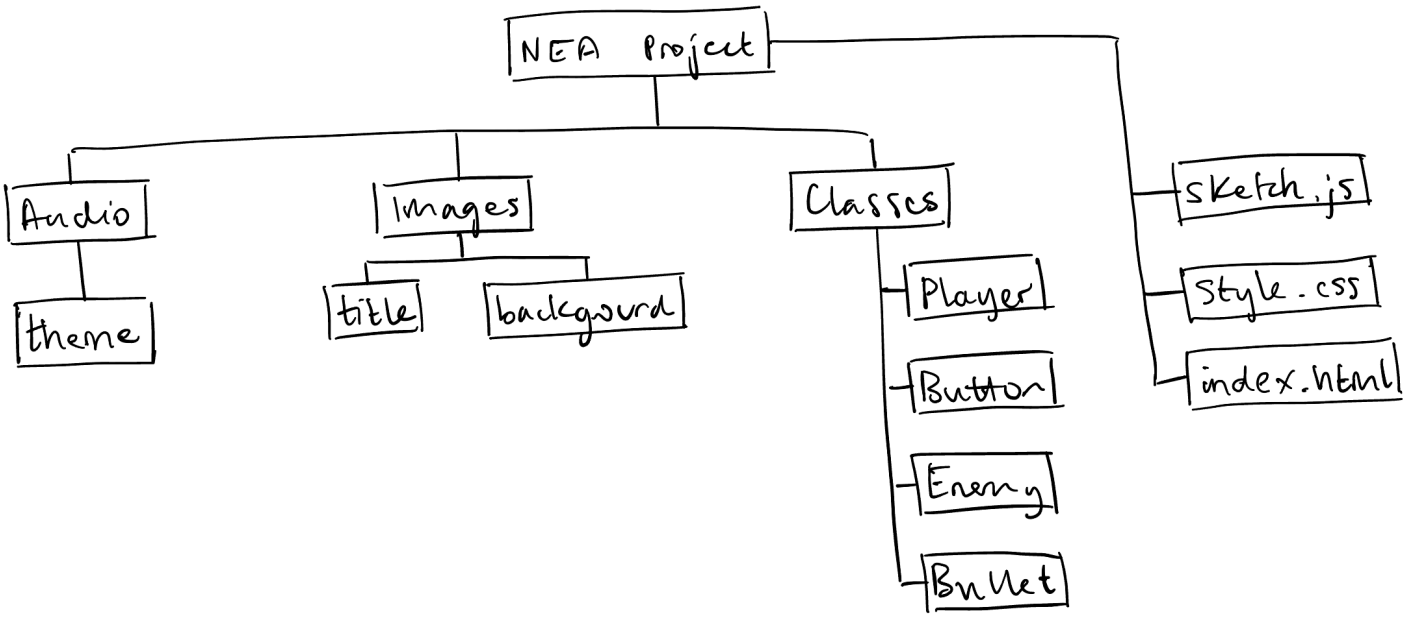
Success criterion 5 covers everything to do with score. This includes adding a score counter on the screen, giving it functionality to increment when an enemy is defeated, and also resetting the counter when a user restarts the game.

The current difficulty level (enemy speed) is quite low, and while I could just increase this and use it throughout the whole game, this would remove the level of accessibility that comes from a user being able to set their own difficulty level.
The difficulty aspect is decomposed into 2 main areas; the menu navigation, meaning the difficulty screen must be reached by clicking on the ‘difficulty’ button.

The second being the actual adjustments. There will be five options to choose from, which will change the speed of an enemy and/or the frequency at which they spawn.

Success criterion 8 covers the character customisation feature being added to my solution.
There are 3 sub-problems which this can be decomposed into. The first is like the difficulty system, there needs to be a customisation screen which the user can access from the main menu through clicking on the ‘customise’ button. There must obviously then be an interaction which causes the character’s colour to actually change, in this case a colour picker, and save the effects. Lastly there needs to be a preview of the character’s changed appearance on the customisation screen, so a user can see what adjustments they are about to make.

D2 - Structure of solution



Due to the nature of iteration 3, the file structure for the project will be the same as that in iteration 2. No new files or classes will be created.

D3 - Algorithms

The algorithms required for the third iteration are as follows:

- Getter and setter for enemy speed attribute
- 5 new button objects for each level of difficulty
- Creating a colour picker and showing an example of new character

There needs to be new class methods for Enemy to account for the new enemySpeed attribute. The getSpeed() method is unlikely to be used as I cannot think of where I would need to read the value, however it could prove useful so I am including it nevertheless.
The setSpeed() method will be used in the main if statement to change the difficulty level. Encapsulation is necessary to prevent unauthorised changes being made to the speed

```
function getSpeed()
     return this.speed;
endfunction

procedure setSpeed(newSpeed)
     this.speed = newSpeed
endprocedure
```

The code for changing the level of difficulty will follow a similar structure as below. I am using a screen system in my game, meaning if the difficulty button is clicked then the difficulty function is run.
In the difficulty function, this is where the buttons will be initialised and drawn. The separation of this code helps to maintain the decomposition methods which I have been using in development up to this point.

```
if difficultyButton is pressed then
     difficulty()
endif

function difficulty()
     draw button Beginner
     draw button Novice
     draw button Intermediate
     draw button Advanced
     draw button Expert
     if (button) is pressed then
          enemy.setSpeed(corresponding value)
          interval = (corresponding value)
     endif
endfunction
```

The customisation functionality will be implemented in a similar way to the difficulty. I will be calling a new function customise() when the customise button is clicked. Inside the actual customise() function, the main functionality will be programmed. This includes initialisation and drawing of the colour picker, along with assigning the selected colour to the player character, and lastly drawing this instance of the player.

```
if characterButton is pressed then
     customise()
endif

function customise()
     create colourPicker
     position colourPicker = centre
     myCar.setColour(colourPicker.colour())
     myCar.draw()
endfunction
```

D4 - Variables and validation

The only new variables being added are enemy speed, and score (and a colour picker, but all validation is done by p5 as it is a p5 datatype).

Aside from that, new buttons will simply be instances of the button class.

Variable name	Description/justification	Validation
speed	This is the float value which the enemy Y position will increment by every time the draw function is executed, changing this value will therefore change the distance covered by the enemy per frame therefore producing the effect of the enemy speed being adjusted	Format check - this must be an float/integer because the value must not only be operated on mathematically, but also passed to p5 functions which use the coordinates of the canvas Range check - the value must be a positive in
scores	This is the integer value which will be used to store the number of enemies eliminated in the current run of the game. This is the variable solely responsible for the successful completion of success criteria 5	Range check - must be at least 0 because the score is added to each time an enemy is eliminated, and the user starts the game on score = 0 Format check - the scores variable must be an integer because the increment when an enemy is killed is 1 and there is no way to kill a fraction of an enemy

Extra variables

The colour picker is a variable of an abstract data type. There will be no need to validate the input as this is all done server side by p5 because the colour picker is available from the p5 libraries.

D5 - Usability features

For iteration 3, the priority for maximising usability is ensuring the menu is easily navigable, and labelled clearly:

I removed the sound effects (apart from the main theme) because depending on the computer system I tested on, the sounds would be delayed and sometimes not play at all.

Unfortunately this means my game will be less catering to those with severe visual impairment, but there are still some measures in place.

New features

Feature	Justification/Implementation
Accessibility	My game will accommodate the needs of those with impaired vision. The game will load quickly, provided an internet connection. Bright colours are used, as well as contrasting colours for text to ensure all components on screen at a given time are most visible to those with visual impairments. Provided a user has access to a stable internet connection, Motor Mayhem will load quickly for them, due to carefully curated use of both software and hardware resources. Running the game locally (like I am in development) means the game loads instantly. Player character and enemies will be coloured using a bright palette to maximise the visibility for users. This is important as without proper visibility of the character, bullets, or enemies, a user may struggle with playing the game entirely which would make for a less than desirable user experience Music will commence when the program runs. The volume will decrease when the user starts a game. This is a crucial addition for my game as it greatly helps those who are hard of hearing to confirm whether or not they have started the game. The instructions and score overlay will be white text against dark green background to maximise contrast. A user will be able to select a different level of difficulty, with a choice of 5 in total which significantly increases the level of accessibility

Navigability	Smooth navigation in arcade game menus enhances user experience. Easy-to-use menus ensure players swiftly access game features, maintaining engagement and reducing frustration during gameplay. The menu will be easy to navigate with clear routes for different menu options. The back button at each stage allows for an easy return to the previous stage. Buttons are designed simply yet enticingly to engage the user. The way in which a user progresses through the game is clear, by ensuring it flows seamlessly with a vertical line of motion. This is achieved by implementing a clear, vertical scrolling approach to my game, allowing a user to know how to progress. The inclusion of a score counter means that it is even more obvious to the user how to progress through the game. Because it hints that the player must gain score to advance, the visible incrementation upon eliminating an enemy further highlights this level of navigability
Compatibility	My program will feature very fast load times to ensure that those with slower internet connections are still able to open the game and play it comfortably. This means that the file sizes for the game must be kept low so that less data must be downloaded in order to play the game. If file sizes are too large, players with slower internet connections will take too long to load the game and therefore will limit the accessibility of the solution. In iteration 2, the large number of values being added to arrays could be a significant cause of increasing the file size. For this reason I made sure to remove redundant objects from their arrays in order to keep file size down to minimise load times and therefore maximise the range of internet speeds with which my project can be used.

D6 - Test plan

Iteration.test	Success criteria	Test details	Expected outcome
3.1	(5)	-Run index.html in ‘NEA project’ folder	Every time a user eliminates an enemy, the score counter is incremented and this is visible to the user in the bottom left of the screen
3.2	(6)	-Run index.html in ‘NEA project’ folder	The ‘customise’ button will lead to a screen which features an interactive colour picker, and an instance of the character to preview any new selections
3.3	(6)	-Run index.html in ‘NEA project’ folder	Upon returning to the main menu and starting a new game, the new colour selected remains on the character
3.4	(7)	-Run index.html in ‘NEA project’ folder	The ‘difficulty’ button will lead to a screen with 5 new buttons, each labelled with a different level of difficulty
3.5	(7)	-Run index.html in the ‘NEA project’ folder	Each time a user selects a new difficulty level, the effects are carried through to when the user starts a new game

D7 - Beta test plan

Test reference
Functional test 4 Tester: Stakeholder Success criteria: 5 - When a bullet successfully hits an enemy, score counter is incremented. Upon starting a new game, score resets
Functional test 5 Tester: Stakeholder Success criteria: 7 - When a user enters the customisation screen, they can choose a different colour, see an example on their character and selection will be saved.
Functional test 6 Tester: Stakeholder Success criteria: 8 - A user can enter a difficulty screen and select any of the five options. Each option varies enemy speed and/or spawn frequency. Current difficulty level is written in the menu screen.
Usability test 3 Tester: Developer Usability criteria: 1 - Navigability Expected outcome: At any given time, a user will be able to return to the menu screen by pressing ‘Q’, no matter what screen they are in. Menu options are navigated through easily
Robustness test 2 Tester: Stakeholder Criteria: Enemy speed can only be set to preset values Justification: If a user was somehow able to override enemy speed to a value other than specified, this could harm the experience as the game would either become impossible or enemies could remain stationary Expected outcome; The only way enemy speed can be updated is through clicking on one of the buttons, and therefore only changing the value to one specified by the developer
Robustness test 3 Tester: Stakeholder Criteria: Character colour can only be changed through the presented colour picker Justification: If a typed input which was not valid was entered by a user, this could result in the program not being able to process it as a colour value, hence crashing the program Expected outcome; The only user interaction in the customisation screen is via the colour picker

D8 - Iterative development

The first thing I will be developing in iteration 3 is the scoring system. The core functionality was straightforward to program and I did this by initialising the variable score as zero at the top of sketch.js, then inside the collision() function I increment the score variable. This means the player's score goes up each time they eliminate an enemy:

```
3    let score = 0;
```

```
//detect collision
for (let i = bullets.length - 1; i >= 0; i--) {
  for (let j = enemies.length - 1; j >= 0; j--) {
    if (bullets[i] && enemies[j] && bulletCollision(bullets[i], enemies[j])) { // null checks for both arrays
    //handle collision
    score++;
    bullets.splice(i, 1);
    enemies.splice(j, 1);
    }
```

To check that this worked as expected, I then output the value of 'score' to the console in the collision function so that I could see it was incrementing as appropriate. Next, I ensure the score is reset to 0 each time the user clicks start again.

```
function mouseClicked() {
  //action if startBtn is clicked
  if (mouseX >= 260 && mouseX <= 540 && mouseY >= 330 && mouseY <= 450 && screen == 0) {
    score = 0;
    play();
```

Lastly for success criterion 5 I needed to program the score overlay that will allow the user to visibly watch as their score increases.
This was some simple text manipulation in bottom left corner of the screen:

```
1    //score
2    textSize(30);
3    text('Score: ' + score, 11, 776);
4
```

The next success criterion to develop would be the character customisation feature. The first step is to code the functionality of the customise button, by first adding a condition to the draw() function to run the new customise() function when screen = 2, and then adding to the mouseClicked function and also editing the background to remove the road in order to focus on the character customisation and nothing else:

```
function customise() {
  screen = 2;
  background('#00472E');
  imageMode(CENTER);
  image(logo, 400, 170);

  //instructions
  textAlign(LEFT);
  textSize(18);
  fill('white');
  text('Quit - Q', 11, 17);
```

```
//instructions
textAlign(LEFT);
textSize(18);
fill('white');
text('Movement - A, D', 11, 17);
text('Shoot - SPACE', 11, 37);
text('Quit - Q', 11, 57);
```

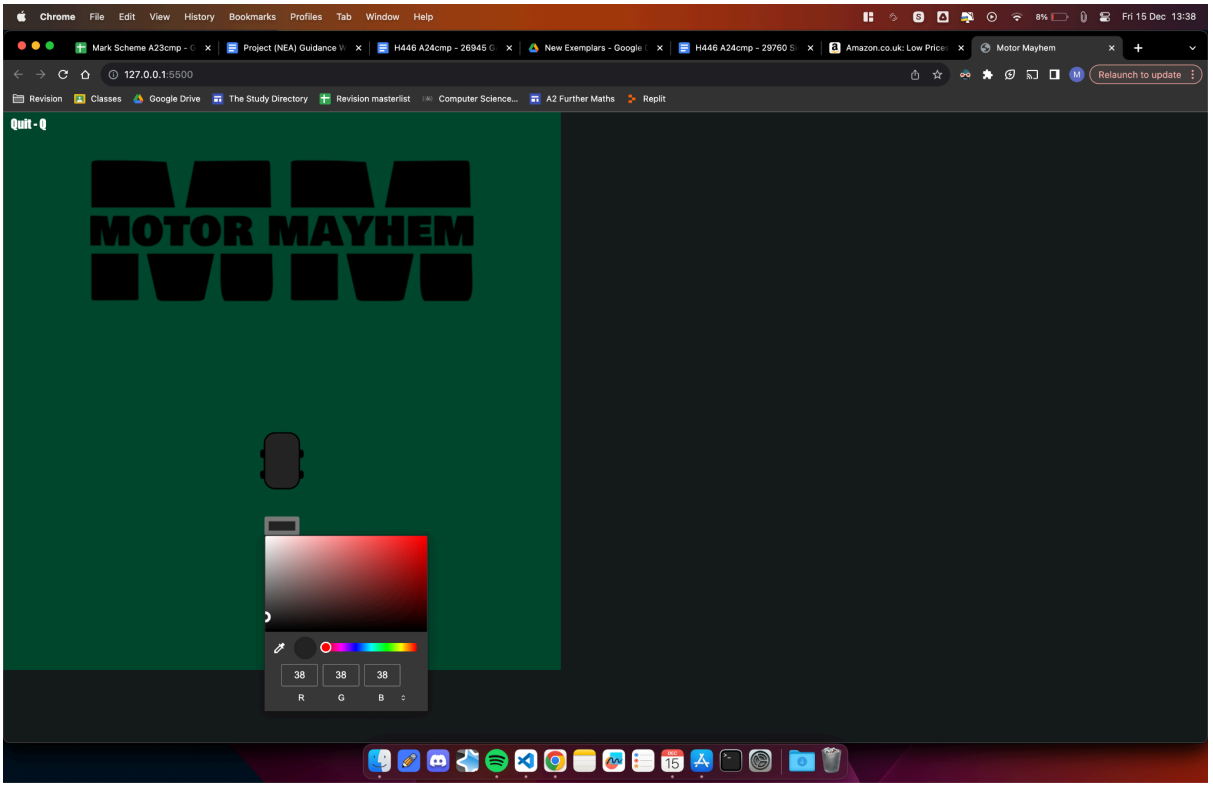
While I was here, I decided it would be a good idea to add an instruction overlay to both the customise screen and the play screen. In an effort to increase the comprehensibility of Motor Mayhem, I will be sure to also include instructions in the difficulty screen later on as well.
After this, it was time to code the colour picker and the character example in the customise screen. After reading up on the p5 reference on how to invoke a colour picker, I ran into some difficulties after coding it into the customise() function. For full details see test 3.2 in D17.
After successful debugging, the colour picker code looked as follows:

```
//creates colour picker
if (!colourPicker) {
  colourPicker = createColorPicker(myCar.getColour());
  colourPicker.position(375, 580);
}
colourPicker.show();
```

Lastly for the character customisation functionality, I needed to add the preview character. This was very straightforward and I did not encounter any issues when coding this:

```
//draws example character
myCar.draw();

//change colour of character to selection
textAlign(CENTER);
myCar.setColour(colourPicker.color());

myCar.setStartY(500);
myCar.draw();
```

This is how the customisation screen looks at this point:



The final task for the development of Motor Mayhem is the difficulty adjustment system.
This starts off like most features, by adding the new function difficulty() to the main draw function depending on the value of 'screen', in order to maintain readability of code and to decompose it to a more manageable state which is also easier to debug.
I then coded the difficulty button and its functionality in the mouseClicked() function, as well as making sure to add an instructions overlay in the difficulty screen.

```
//initialises buttons
startBtn = new Button('START', 100, 280, 400, 380);
customiseBtn = new Button('CUSTOMISE', 100, 280, 400, 540);
difficultyBtn = new Button('DIFFICULTY', 100, 280, 400, 700);
```

```
function difficulty() {
  screen = 3;
  background('#00472E');
  imageMode(CENTER);
  image(logo, 400, 170);

  if (colourPicker) {
    colourPicker.hide();
    colourPicker = null;
  }

  //instructions
  textAlign(LEFT);
  textSize(18);
  fill('white');
  text('Quit - Q', 11, 17);
```

The next step was to create all five button options in the difficulty screen, assigning the appropriate labels. I decided to make a colour gradient with the buttons (each one being slightly differently coloured) to increase the comprehensibility because the easier levels were given greener buttons, whereas they got more red as the difficulty increased.
I researched common naming conventions for difficulty levels and decided to use the following 5 in levels of increasing difficulty: beginner, novice, intermediate, advanced, and expert.

Creating the buttons and giving them text resulted in the following addition to the difficulty() function:

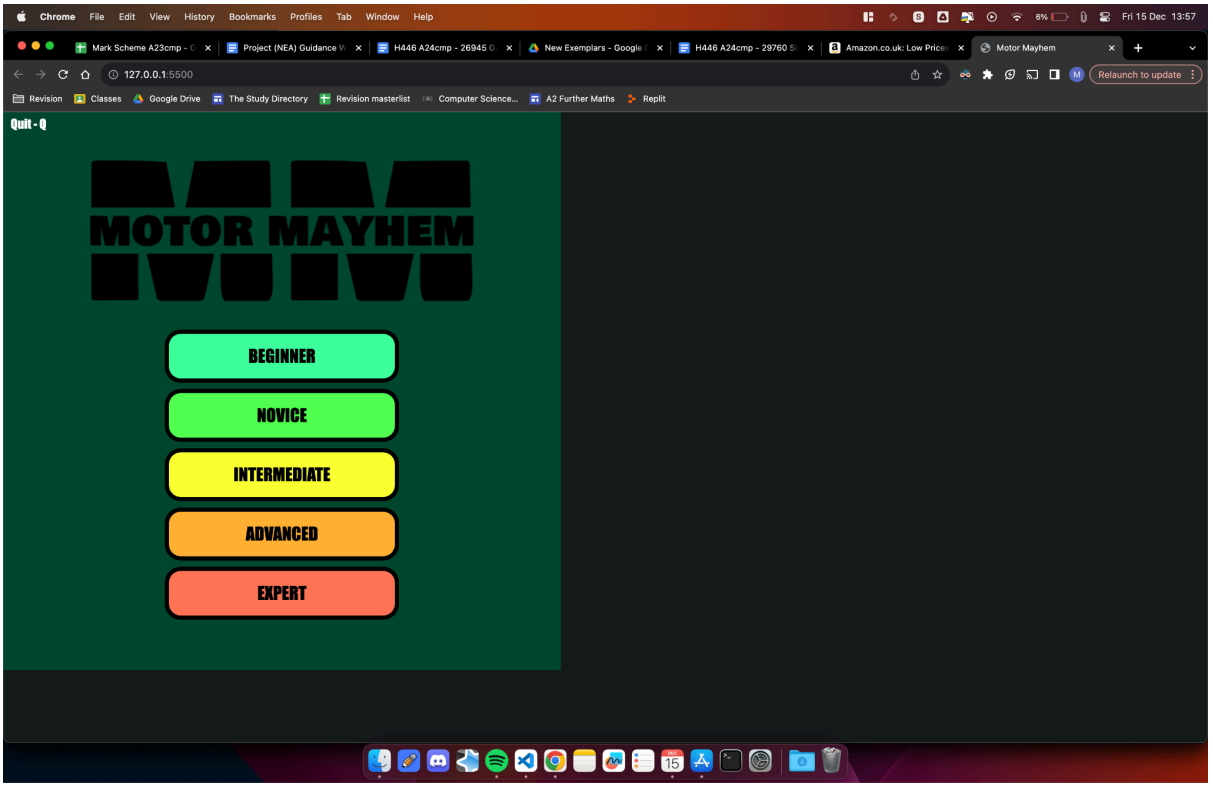
```
fill('#3EFF9F');
rect(beginnerBtn.getCentreX(), beginnerBtn.getCentreY(), beginnerBtn.getWidth(), beginnerBtn.getHeight(), 20);

fill('#51FF53');
rect(noviceBtn.getCentreX(), noviceBtn.getCentreY(), noviceBtn.getWidth(), noviceBtn.getHeight(), 20);

fill('#FCFF34');
rect(intermediateBtn.getCentreX(), intermediateBtn.getCentreY(), intermediateBtn.getWidth(), intermediateBtn.getHeight(), 20);

fill('#FFB232');
rect(advancedBtn.getCentreX(), advancedBtn.getCentreY(), advancedBtn.getWidth(), advancedBtn.getHeight(), 20);

fill('#FF7556');
rect(expertBtn.getCentreX(), expertBtn.getCentreY(), expertBtn.getWidth(), expertBtn.getHeight(), 20);

textAlign(CENTER);
textSize(25);
textFont('Impact');
fill(0);
text(beginnerBtn.getText(), beginnerBtn.getCentreX(), beginnerBtn.getCentreY());
text(noviceBtn.getText(), noviceBtn.getCentreX(), noviceBtn.getCentreY());
text(intermediateBtn.getText(), intermediateBtn.getCentreX(), intermediateBtn.getCentreY());
text(advancedBtn.getText(), advancedBtn.getCentreX(), advancedBtn.getCentreY());
text(expertBtn.getText(), expertBtn.getCentreX(), expertBtn.getCentreY());
```

The final task was to then give each of these five new buttons its own functionality in changing the enemy speed and/or spawn rate of enemies.
I strayed too far from the algorithm plan in D3 and therefore experienced an issue which really confused me at first. See test 3.5 in D17 for the complete breakdown of this setback.
By the end of debugging, my mouseClicked function appeared as follows:

```
function mouseClicked() {
  //action if startBtn is clicked
  if (mouseX >= 260 && mouseX <= 540 && mouseY >= 330 && mouseY <= 450 && screen == 0) {
    score = 0;
    play();
  } else if (mouseX >= 260 && mouseX <= 540 && mouseY >= 490 && mouseY <= 610 && screen == 0) { //first 3 conditions for main menu buttons
    customise();
  } else if (mouseX >= 260 && mouseX <= 540 && mouseY >= 650 && mouseY <= 770 && screen == 0) {
    difficulty();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 315 && mouseY <= 385 && screen == 3) { //last 5 conditions for difficulty buttons
    enemySpeed = 2;
    interval = 60;
    diffMode = 'Beginner';
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 400 && mouseY <= 470 && screen == 3) {
    enemySpeed = 5;
    interval = 45;
    diffMode = 'Novice';
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 485 && mouseY <= 555 && screen == 3) {
    enemySpeed = 7;
    interval = 40;
    diffMode = 'Intermediate';
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 570 && mouseY <= 640 && screen == 3) {
    enemySpeed = 9.5;
    interval = 40;
    diffMode = 'Advanced';
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 655 && mouseY <= 725 && screen == 3) {
    enemySpeed = 11;
    interval = 30;
    diffMode = 'Expert';
    menu();
  }
}
```

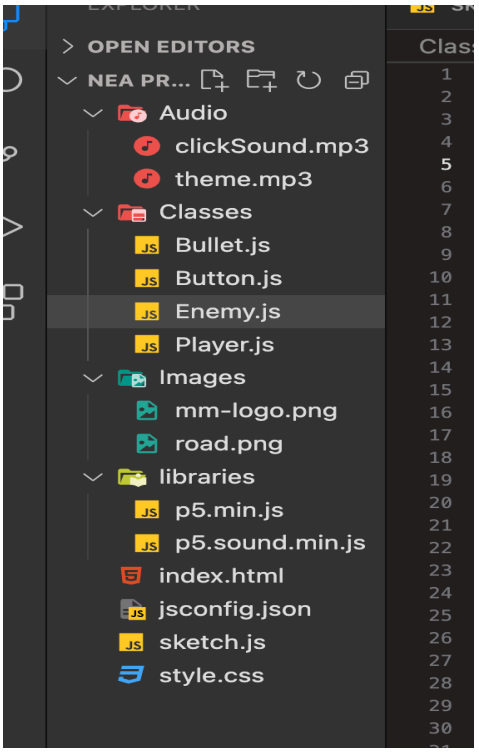
I simply used a trial & error approach in order to determine the appropriate values for enemySpeed and interval. I decided that at the higher levels of difficulty, not only

would the enemies descend down the screen faster, but they would also spawn at a slightly higher frequency.
By the end of developing iteration 3, here is how the difficulty screen looked:



At this point I felt that all success criteria outlined for development in iteration 3 had been successfully implemented.

D9 - Structure and modularity



This is how my files and folders look upon completing the second iteration of Motor Mayhem, identical to that in iteration 2. I exactly followed the structure chart which was planned out in D2

mouseClicked()

the majority of this subroutine was added in iteration 3 as I was coding the functionality to the five new difficulty buttons as well as the difficulty option itself

```
function mouseClicked() {
  //action if startBtn is clicked
  if (mouseX >= 260 && mouseX <= 540 && mouseY >= 330 && mouseY <= 450 && screen == 0) {
    score = 0;
    play();
  } else if (mouseX >= 260 && mouseX <= 540 && mouseY >= 490 && mouseY <= 610 && screen == 0) { //first 3 conditions for main menu buttons
    customise();
  } else if (mouseX >= 260 && mouseX <= 540 && mouseY >= 650 && mouseY <= 770 && screen == 0) {
    difficulty();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 315 && mouseY <= 385 && screen == 3) { //last 5 conditions for difficulty buttons
    enemySpeed = 2;
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 400 && mouseY <= 470 && screen == 3) {
    enemySpeed = 5;
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 485 && mouseY <= 555 && screen == 3) {
    enemySpeed = 6.5;
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 570 && mouseY <= 640 && screen == 3) {
    enemySpeed = 11;
    interval = 40;
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 655 && mouseY <= 725 && screen == 3) {
    enemySpeed = 11;
    interval = 30;
    menu();
  }
}
```

difficulty()

This is the final game screen function I developed. Towards the top of the function is drawing the background and logo, whereas the last part of the subroutine is dedicated to drawing the five new difficulty setting buttons.

```
function difficulty() {
  screen = 3;
  background('#00472E');
  imageMode(CENTER);
  image(logo, 400, 170);

  if (colourPicker) {
    colourPicker.hide();
    colourPicker = null;
  }

  //instructions
  textAlign(LEFT);
  textSize(18);
  fill('white');
  text('Quit - Q', 11, 17);

  //creating buttons
  strokeWeight(6);
  rectMode(CENTER);

  fill('#3EFF9F');
  rect(beginnerBtn.getCentreX(), beginnerBtn.getCentreY(), beginnerBtn.getWidth(), beginnerBtn.getHeight(), 20);

  fill('#51FF53');
  rect(noviceBtn.getCentreX(), noviceBtn.getCentreY(), noviceBtn.getWidth(), noviceBtn.getHeight(), 20);

  fill('#FCFF34');
  rect(intermediateBtn.getCentreX(), intermediateBtn.getCentreY(), intermediateBtn.getWidth(), intermediateBtn.getHeight(), 20);

  fill('#FFB232');
  rect(advancedBtn.getCentreX(), advancedBtn.getCentreY(), advancedBtn.getWidth(), advancedBtn.getHeight(), 20);

  fill('#FF7556');
  rect(expertBtn.getCentreX(), expertBtn.getCentreY(), expertBtn.getWidth(), expertBtn.getHeight(), 20);

  textAlign(CENTER);
  textSize(25);
  textFont('Impact');
  fill(0);
  text(beginnerBtn.getText(), beginnerBtn.getCentreX(), beginnerBtn.getCentreY());
  text(noviceBtn.getText(), noviceBtn.getCentreX(), noviceBtn.getCentreY());
  text(intermediateBtn.getText(), intermediateBtn.getCentreX(), intermediateBtn.getCentreY());
  text(advancedBtn.getText(), advancedBtn.getCentreX(), advancedBtn.getCentreY());
  text(expertBtn.getText(), expertBtn.getCentreX(), expertBtn.getCentreY());
}
```

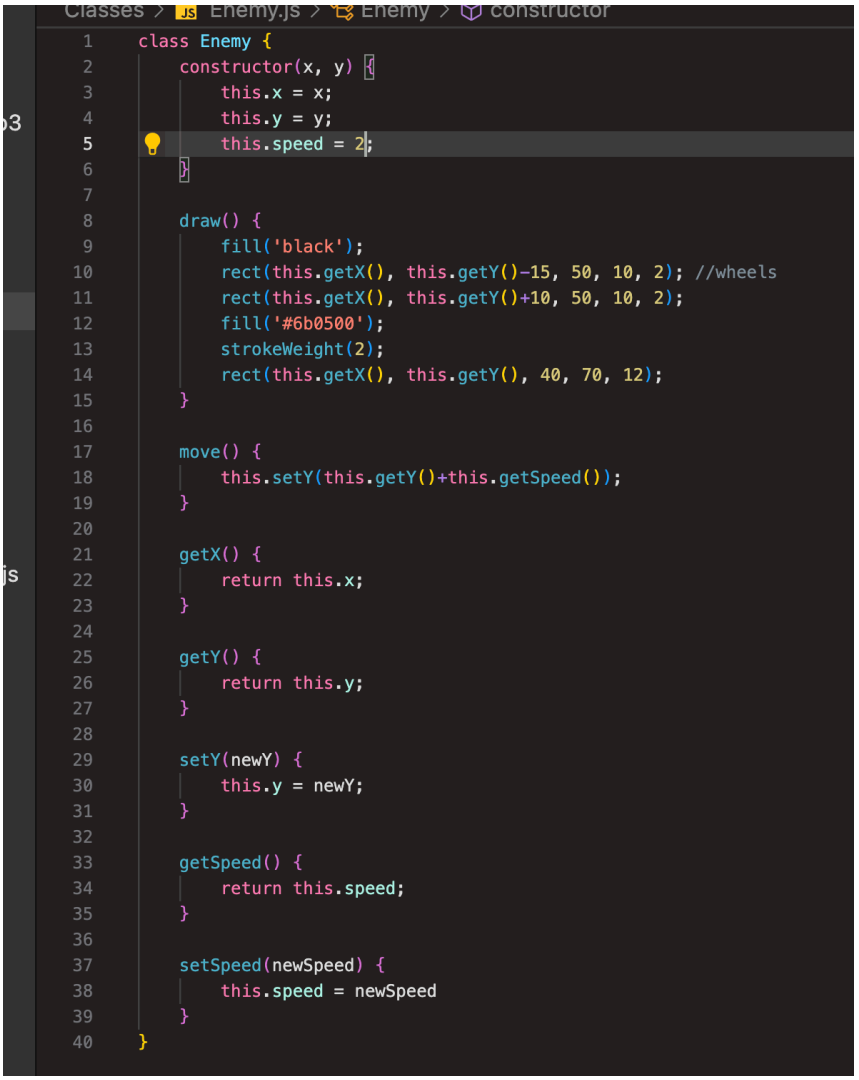
customise()

```
function customise() {
  screen = 2;
  background('#00472E');
  imageMode(CENTER);
  image(logo, 400, 170);

  //instructions
  textAlign(LEFT);
  textSize(18);
  fill('white');
  text('Quit - Q', 11, 17);

  //creates colour picker
  if (!colourPicker) {
    colourPicker = createColorPicker(myCar.getColour());
    colourPicker.position(375, 580);
  }
  colourPicker.show();

  //draws example character
  myCar.draw();

  //make colour picker
  textAlign(CENTER);
  myCar.setColour(colourPicker.color());

  myCar.setStartY(500);
  myCar.draw();
}
```

This is the penultimate game screen function.
It includes creating the colour picker, creating an example instance of character, and assigning the new colour to that character instance.

enemy class

The enemy class was updated to account for changing enemy speed



D13 - Prototype 3

https://drive.google.com/file/d/11Sma0R3IOZIeI37Uu4sc-bnPIBQ9IX2I/view?usp=drive_link

D14 - Review of iteration

Accomplishments

Criteria No.	Criteria description	Status
5	Scoring system	✓
6	Scoreboard	✘
7	Character customisation	✓
8	Difficulty adjustment	✓

Modifications

I implemented a few features which were not planned during the design stage of this iteration. These include, an instructions overlay and a current difficulty reminder.
I also slightly modified the colour scheme of the background and GUI buttons. This is because I feel it is easier on the eyes and more aesthetically pleasing, while still maintaining good visibility for customers with difficulty seeing.

Along with this, I slightly modified the updates made to enemies when changing difficulty, if a higher difficulty is selected, not only is enemy speed increased but they also spawn more frequently to create a more challenging atmosphere.

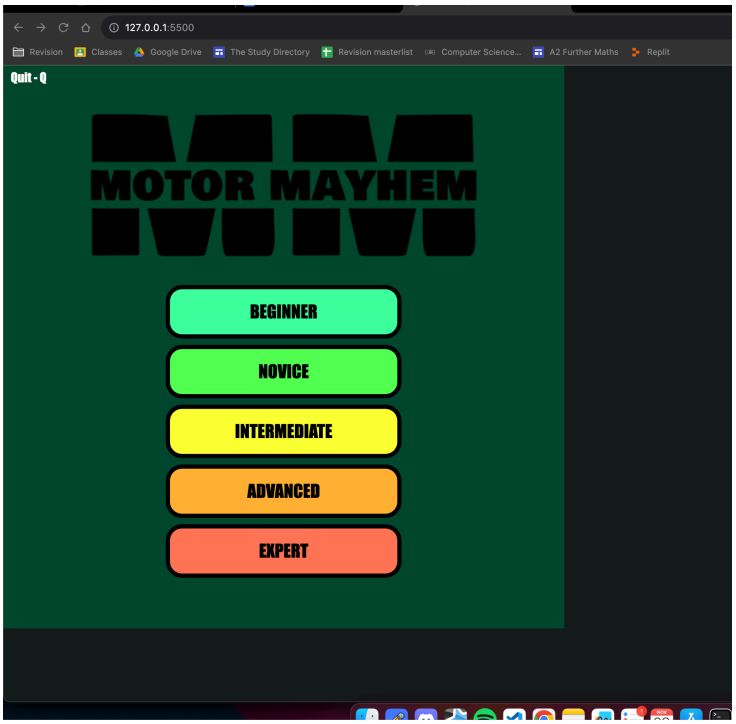
The major modification in iteration 3 was the dismissal of a scoreboard system. This is because it not only causes troubles and deadline issues, but also it subtracts from the simplistic arcade feel of Motor Mayhem - not massively but enough. For the complete breakdown on the removal of this feature, see E6.

Stakeholder Feedback

Stakeholder	Feedback
Tom	By the completion of iteration, Tom was really happy with how my game had turned out. Unsurprisingly, he was most impressed with the ability to set his own level of difficulty, because as an experienced gamer he craved a challenge. I am happy he felt his craving was catered to, because that means that upon rolling out

	my game to a larger audience, the portion of which are avid gamers will still be able to enjoy Motor Mayhem. As it is still a straightforward arcade game, I am glad that success criteria 8 was able to be implemented to a great standard.
Alex	On the opposite end of the gaming spectrum is Alex. She is equally joyous about the inclusion of a difficulty adjustment system, as she found the default level of play a little too much for her liking. For similar reasons as mentioned above, I am amazed to hear that I have been able to provide a fun gaming experience for players of varying skill levels. Alex also made it clear how satisfied with the customisation feature she was, although she has commented on how it is unfortunate that there were not more levels of customisation, nor an unlocking system.
Andrew	Andrew is very pleased with my final iteration of Motor Mayhem, although has some constructive criticism. As intended and previously mentioned, he is very happy with the overall tone of the solution in its inspiration from old-school arcade games. He also believes that customisation of character and difficulty were both great additions which were also implemented well. However he has made it apparent that he would have really liked to see the scoreboard feature being included in the final version of Motor Mayhem, which is why it would be the first task to consider given the chance to further develop the solution

D16 - Iterative testing

Iteration.test	Outcome	Evidence
3.1	Pass	https://drive.google.com/file/d/10gQZ6DHD7vBQWumWErWvxyRXHrQg6spO/view?usp=drive_link
3.2	Fail	
3.3	Pass	https://drive.google.com/file/d/1vDWFtz993DBQ2x1MnsNmRII79kLR8GmB/view?usp=drive_link
3.4		
3.5	Fail	

D17 - Failed tests/remedies

Test 3.2 failed

To begin with, the colour picker would appear correctly, along with the example character however I could not interact with the colour picker.

I fixed the issue by checking the colourPicker held a 'falsy' value.
I believe the issue came somewhere from the colour picker being drawn every second so the check made sure it was only initialised once.

```
//creates colour picker
if (!colourPicker) {
  colourPicker = createColorPicker(myCar.getColour());
  colourPicker.position(375, 580);
}
colourPicker.show();
```

I also added the following check to every function other than customise(), to ensure the colour picker could only be interacted with in the customise screen:

```
//hides colour picker
if (colourPicker) {
  colourPicker.hide();
  colourPicker = null;
}
```

Test 3.5 failed

The problem was in updating the enemySpeed value. Initially, I used an additional function updateEnemySpeed(value) which took in a parameter set by the relevant button from the mouseClicked function.
However I was running into issues with variable scope, specifically with the enemies array.

I eventually figured out the solution was to remove the updateEnemySpeed function and just use a global variable of enemySpeed which would be assigned a value depending on the button pressed. Finally, I included a for loop in the main play() function to continuously update enemySpeed to the relevant value.
See below:

```
//updates enemy speed depending on setting
for (let i = enemies.length - 1; i >= 0; i--) {
  enemies[i].setSpeed(enemySpeed);
}
```

```
function mouseClicked() {
  //action if startBtn is clicked
  if (mouseX >= 260 && mouseX <= 540 && mouseY >= 330 && mouseY <= 450 && screen == 0) {
    score = 0;
    play();
  } else if (mouseX >= 260 && mouseX <= 540 && mouseY >= 490 && mouseY <= 610 && screen == 0) { //first 3 conditions for main menu buttons
    customise();
  } else if (mouseX >= 260 && mouseX <= 540 && mouseY >= 650 && mouseY <= 770 && screen == 0) {
    difficulty();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 315 && mouseY <= 385 && screen == 3) { //last 5 conditions for difficulty buttons
    enemySpeed = 2;
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 400 && mouseY <= 470 && screen == 3) {
    enemySpeed = 5;
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 485 && mouseY <= 555 && screen == 3) {
    enemySpeed = 6.5;
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 570 && mouseY <= 640 && screen == 3) {
    enemySpeed = 11;
    interval = 40;
    menu();
  } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 655 && mouseY <= 725 && screen == 3) {
    enemySpeed = 11;
    interval = 30;
    menu();
  }
}
```

POST DEVELOPMENT TESTING

Test reference	Test evidence	Tester's comment on outcome
Functional test 1 Tester: Developer Success criteria: 1 - The menu must present the title and start, customise, and difficulty buttons on a background Test details: • Run the program in VSCode	https://drive.google.com/file/d/1KL_FAhJt_j5oa82e4uo7TUmLh0DqPG2z/view?usp=drive_link	All menu components loaded as intended
Functional test 2 Tester: Developer Success criteria: 3 - The character can be moved left and right using the keys 'A' and 'D' respectively Test details: • Run the program in VSCode • Press/hold 'a' to move left • Press/hold 'd' to move right	https://drive.google.com/file/d/1vWJMunO1hN2jpeBeL70z-SVHxznY1SIR/view?usp=drive_link	Pressing 'a' moves the character to the left and 'd' moves them right
Functional test 3 Tester: Developer Success criteria: 4 - Upon pressing the space key, a bullet is fired from the current player position. If it hits an enemy, bullet and enemy disappear Test details: • Run the program in VSCode • Press the space key to fire a bullet • Hit an enemy with a bullet	https://drive.google.com/file/d/1ZQ8LQdLYKWUWoP8a6lf8lvm5yAmmLafK/view?usp=drive_link	Bullets travel up the screen from player's current position, when an enemy is hit, the bullet and enemy disappear
Functional test 4 Tester: Developer Success criteria: 5 - When a bullet successfully hits an	https://drive.google.com/file/d/1LlxTnGxk3GAwkJCI_nxKwGHrNggzBfHk/view?usp=drive_link	Score goes up when an enemy is hit. Score resets when user restarts

enemy, score counter is incremented. Upon starting a new game, score resets		game
Functional test 5 Tester: Developer Success criteria: 7 - When a user enters the customisation screen, they can choose a different colour, see an example on their character and selection will be saved.	https://drive.google.com/file/d/12nq8aGc5BrAKfXpUOPgyl3zyKPghUn1p/view?usp=drive_link	Users can enter the customisation screen and change the colour of the character. The example character updates with selection. Selection is saved when character starts game
Functionality test 6 (Usability test 1) Tester: Stakeholder Success criteria: 8 - A user can enter a difficulty screen and select any of the five options. Each option varies enemy speed and/or spawn frequency. Current difficulty level is written in the menu screen.	https://drive.google.com/file/d/1COMf0oXT8ltXJKkrUJUnHg3zwaVh_Qna/view?usp=drive_link	For each difficulty selection, changes are saved for when the user starts a game and the reminder updates on the menu screen
Usability test 2 Tester: Stakeholder Usability criteria: 1 - Accessibility Test details: • Run the program • Open chrome dev tools • Run the lighthouse test for accessibility Expected outcome: Menu buttons are clearly labelled and are highly visible by contrasting text on its background	https://drive.google.com/file/d/1-OHzi96YtXyY88Aai897JCW71vQCsIn2/view?usp=drive_link	Menu screen achieves maximum score in accessibility test
Usability test 3 Tester: Stakeholder Usability criteria: 1 - Navigability Test details: • Run the program • Click start	https://drive.google.com/file/d/1GvD7NPKXXjrvnoaSiRCAqEppef3tWbCn/view?usp=drive_link	The menu is straightforward to navigate and the main menu can always be accessed using the ‘q’ key

• At any screen in the game, a user can press ‘q’ to quit to the main menu Expected outcome: At any given time, a user will be able to return to the menu screen by pressing ‘Q’, no matter what screen they are in. Menu options are navigated through easily		
Usability test 4 Tester: Stakeholder Usability criteria: 1 - Accessibility Test details: • Run the program • After a few seconds, click start Expected outcome: When the program first starts, the theme music plays at a moderate volume, then once the user clicks ‘start’, the volume decreases to help the user focus on the game	https://drive.google.com/file/d/1SS6jdBJxSSkBlsjpUhZE0i9aaHsBLtDb/view?usp=drive_link	The music starts successfully and volume decreases appropriately during the play screen. Volume returns to original level upon returning to the menu
Robustness test 1 Tester: Stakeholder Criteria: Array overload is prevented Justification: If an array is added to infinitely, eventually the hardware/software will start to struggle to run efficiently when handling such a large number of values Test details: • Run the program • Click start • Fire as many bullets as possible for at least 5 seconds Expected outcome: Upon a user firing as many bullets as they can, the program will stay running	https://drive.google.com/file/d/1JPumZVfl_DFWZAxpGu6Vo1gAUmIpkVZm/view?usp=drive_link	Program does not crash and there is no reduction in performance when the bullets array is being written to very quickly

efficiently, as bullets are removed from array upon enemy contact or when they reach the edge of the screen		
Robustness test 2 Tester: Stakeholder Criteria: Enemy speed can only be set to preset values Justification: If a user was somehow able to override enemy speed to a value other than specified, this could harm the experience as the game would either become impossible or enemies could remain stationary Expected outcome: The only way enemy speed can be updated is through clicking on one of the buttons, and therefore only changing the value to one specified by the developer	https://drive.google.com/file/d/1iLgjVySypWFt1hXeAEaBXGa41v7jeRHa/view?usp=sharing	There is no way to possibly input a different enemy speed/interval except for selecting one of the presets

EVALUATION

E3 & E4 - Assessment of success criteria

1. The menu must clearly highlight the various options which are available for the user to access. However the menu must be kept simple to prevent overwhelming the user

This criterion has been fully met because the main menu is fully functional; in both the ability to navigate to any of the 3 screens (start, customise, difficulty) and at any time a user is able to return back to the main menu. The simplicity of the main menu has also been maintained because there is a clear button layout at each stage and not an overwhelming number of options on any given screen. This is evident in functional tests 1 and 6, which clearly identify the menu layout. The return to menu feature is shown in usability test 3, where the user plays a game, and uses the ‘q’ button to return to menu, hence this criterion has been fully met.
I successfully created the menu in an accessible way using contrasting colours and big shapes/words. The level of usability is highlighted by usability test 2, where the program scored maximum works in an accessibility benchmark.

2. Music will commence upon running the program at a standard volume. This volume will then decrease as the user starts a run.

This criterion has been fully met as the game audio acts exactly as described. This is evident in usability test 4, because the theme music starts and changes volume as described. The audio feature increases the accessibility of my project because the decrease in volume upon starting a game assists who are visually impaired to confirm they have actually started the game

3. The user should be able to easily move their character left and right using the ‘a’ and ‘d’ keys respectively.

This criterion has been fully met. As long as the user’s character is within the boundaries of the road, they are able to move left and right as they please using the keys named above.
This is shown in functional test 2, where the user can be seen moving left and right following the instructions in the top right of the screen (‘a’ and ‘d’ keys)

4. Players should be able to fire projectiles at enemies and upon contact, the enemy is defeated and ‘dies’.

This criterion has been fully met as the enemy elimination system works exactly as described. When a user presses the space key, they can fire a bullet which travels directly upwards at a constant speed until either it hits an enemy, at which point the bullet and enemy are removed from the screen, or if it reaches the top boundary of the screen, where it will be deleted from the bullets array to maintain robustness.
The evidence for this is in functional test 3, where the combat system can be seen working just as described, hence this criterion has been fully met.

5. Score should be clearly presented.The player should be able to increase their score by defeating various enemies. These should be stored in a local data structure to be used in the scoreboard

This criterion has been partially met. Functionally, it works flawlessly. When an enemy is killed the score is increased and the score resets each run of the game. However the scores are not stored in the local array as mentioned above because the purpose of that was to form the basis of the scoreboard, which was not developed in the end.
Through further development of the scoreboard system, the 10 most recent scores would have been appended to a scoreboard array to be used accordingly with the following success criteria.

6. An option in the menu will be available to visit a scoreboard screen which will showcase the user's scores on each of the past x amount of runs, including the difficulty preset at the time

This criterion was unfortunately not met. During the design stage of iteration 3, I decided that I would not be developing the scoreboard feature in order to make sure deadlines were met and development was fully documented. I felt that the removal of the scoreboard feature did not significantly subtract from the overall experience of Motor Mayhem, however I would be keen to have another go provided more time because my stakeholder, Andrew, commented on how he would have liked the feature to be implemented.
Through further development, I would have used the scoreboard data structure mentioned above and implemented a graphical representation of a user's 10 most recent runs of Motor Mayhem. This would be shown in a table form in a screen reached by a 4th button on the main menu, labelled 'scoreboard'. The scoreboard would be a 3 column table made up of: time of run, score, and level difficulty at the time. The scoreboard would be a static feature with no interaction and would be drawn in white to contrast most with the forest green background that my project already has.
To maintain maximum navigability, the 'q' key would again give the user the option to return to the main menu.

7. The user will be able to choose from a variety of colours to customise their car depending on high score

This criterion has been partially met. I have successfully added a character customisation tool into my game. The user is able to choose any colour from a colour picker, and an example character is shown on the same screen to show how a given colour will look on the car.
The customisation screen is reached from clicking on the 'customisation' button in the main menu. The feature is shown in action working as described above in t
However this success criterion was only partially met because the customisation options do not depend on score. With the time provided, it was clear to me I would be unable to design, program, and document this feature.
Given the chance to develop further, I would have changed the customisation option to feature 3 stages of customisation. Stage 1 would be single colours, stage 2 would be colour gradients, and stage 3 would be decals implemented as images. There would be 3 options at each stage and each option would be a button on the

customisation screen, replacing the colour picker. The options available to a user would depend on the score achieved in the most recent run, therefore if a stage 3 colour was selected, the player would reset to the default grey colour if a stage 3 score was not achieved in the next run.
It is unfortunate that this feature was not implemented, as it would have given the game a fun sense of progression for a user.

8. There will be a range of difficulty options the user will be able to choose from

This criterion has been fully met. There is a button in the main menu labelled 'difficulty', which when clicked takes the user to a difficulty selection screen. There are 5 options to choose from, each changing the speed of the enemy and/or the spawn frequency. Functional test 6 is evidence of the criterion being met fully as it shows a user selecting each of the 5 difficulty levels in turn, and testing the effects that are carried out into the next game. The current difficulty level is always displayed in the bottom left corner of the menu screen. Alex and Tom's stakeholder feedback from D14 are great examples of evidence which supports this criterion being met fully.

E5 & E6 - Assessment of usability criteria

Accessibility

I believe this usability criterion has been partially met. There are several features of my game which highlight a level of accessibility.
Firstly, the colours I used in my game are bright because I want objects on the screen to be as obvious to the user as possible. I also made sure that any time there was text on the screen, it showed in a way that meant it contrasted well with its background colour. These graphical features are shown as fully met through usability test 2.
My game features a difficulty selection menu which allows a user to choose 1 of 5 difficulty settings in order to tailor the game to fit their skill level. This feature has hugely increased the accessibility of my game. The game now appeals to a wider clientbase as it will be playable for different denominations of players, which is shown in my final stakeholder feedback. The difficulty system works flawlessly and this is highlighted through usability test 1.
Accessibility matters a lot in arcade games. It's about making games easy to play for everyone. By letting people change controls, adjust visuals or sounds, and use subtitles, games become inclusive. This helps players with different abilities to join in and have fun. It allows for organisations/developers like myself to expand their clientbase significantly.
However I think more could have been done in development to maximise user accessibility. Through further development, I would have focused more on audio queues through Motor Mayhem. There would be sound effects for both firing bullets and eliminating enemies, as well as for when the game ends. The reason for wanting to add these extra sound queues is because it would increase the appeal of my

game to those with more severe visual impairments; the sound effects would confirm key points in a user's playing experience when it may be otherwise quite difficult. There was however a reason for why I did not end up including these initially. The process of programming sound effects is very straightforward, and I had already done it for the theme music. However I found that depending on the device I was developing on (Macbook pro/windows desktop), the audio queues would not play at the right times, and sometimes not at all. I assumed it was a hardware issue but it made both documentation and testing inconsistent so it was not taken any further.

Navigability

This usability criterion has been fully met. Making it easy to move around in an arcade game is super important. It's not just about getting from one place to another; it's about making sure players can easily figure out where to go and how to get there. Right from the start, game designers focus on this. They create menus with buttons you can click to go to different parts of the game. These buttons act like maps, guiding players smoothly through the game without them getting confused.
There are also cool shortcuts, like pressing 'q' to quickly go back to the main menu. This makes it easy for players to control where they want to be in the game without any hassle.
But the real test happens when people actually try the game. Watching how players move around and if they have any trouble tells me as the developer whether I have done well. When players can easily find their way, it shows that the designers made the game in a way that puts players first.
When moving around the game feels easy and natural, it's like having a helpful friend showing you where to go without you even noticing. It makes the game more fun because players can focus on playing, not on figuring out how to get around.
From stakeholder remarks, it is fair to say that this usability criterion has been fully met

Compatibility

This criterion has been partially met. My game is playable on the majority of modern browsers. Evidence is shown throughout my post-development testing, where Chrome and Edge are seen running my program with no trouble.
Compatibility is key in web-based arcade games. Ensuring games run smoothly across different browsers and devices widens accessibility. Universal compatibility guarantees that players can enjoy the game on various platforms without glitches, enhancing user experience and attracting a broader audience to engage effortlessly with the gaming content.

However, through further development, I would have added mobile device functionality, because in the current world of technology, an increasing number of people have access to mobile devices rather than desktop computers or laptops. This would mean I reconstruct the majority of my code because all uses of canvas coordinates would need to be revisited and altered to refer to the coordinates appropriate to the current device - as opposed to fixed numbers. The character

movement would need to account for those on mobile devices and who therefore lack a keyboard, for example on screen buttons or tilt functionality. The ‘q’ key to return to the main menu would hence also be reworked due to the lack of a physical ‘q’ key.
Enemy spawning would need to be changed because less enemies will be able to fit on a phone screen compared to a 16:9 computer screen.
These features as a whole would certainly take up a lot of time, and possibly even more software, which I unfortunately did not have at my disposal, on my budget it was not appealing to pay to publish my code to the mobile app stores. Leaving this criterion as partially met was the right decision to focus on other parts of my project, despite the limitations it presents to my audience, as shown in Alex’s feedback.

E7 - Maintenance/Limitations

Maintenance

Possible maintenance issue	Solution
Changes to popular browsers (Chrome, Safari, Edge, Firefox) could stop some features in the web app from displaying as expected or lead to certain features being unsupported completely due to security concerns from the browser companies.	I can make use of an adaptive maintenance solution. Once a month I will check in and make sure there have not been updates to any of the most popular browsers which could lead to issues with running Motor Mayhem. If there are any features which have been unsupported, I can make appropriate changes to the code so that the solution works the same or similar but with different technology.
Through publishing my project as a game, the user base will inevitably grow and therefore they will pitch ideas for changes/new features	Perfective maintenance can be used so that the game stays relevant to its audience and fulfils its purpose as an arcade game when it is rolled out to a larger market. To account for the changes that my audience will undoubtedly request, I will give everybody access to a form which will let them input any ideas they may have for possible updates for Motor Mayhem.
As people play the game more, they may encounter bugs which my stakeholders and I were not able to identify during development/testing. These could bugs could range in severity from graphical glitches, to game crashing issues	I will utilise corrective maintenance to accommodate this. Another form will be made available to all players which will let them remark on bugs they have found with details on how to find it for myself and about their own system, to ensure I can debug it most efficiently

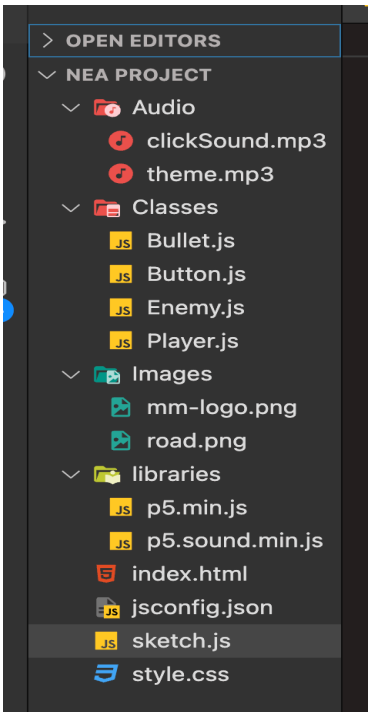
Certain bugs may present themselves which are very difficult to fix, and could potentially require large portions of my code to be reworked or even rewritten. The bugs will prove even harder to fix should I pass my code onto another developer as there is a high chance they do not understand the code well enough to solve certain errors	To future proof my project to the highest possible degree, I will use preventative maintenance. I will ask several other members of the computer science department to look over my code and alert me of any parts which they do not understand. I will then use this feedback to add new documentation which makes the code easier to follow. I will also work on optimising existing algorithms in my project so they work as well as they possibly can. This will ensure that, should a serious bug make itself known, I have the best possible environment to solve the issue, and hence debugging will be significantly easier.

Final limitations

Limitation	Potential improvement
The enemies follow a very simple trajectory, as it is merely a vertical path from their initial spawn location. Horizontal movement would make the enemies less predictable	I would probably add an element of horizontal movement to the enemies to increase difficulty and user immersion. AI models would not be an ideal method of improvement, as I have never used one before and it would most likely require additional software technologies to develop/test. As well as descending down the screen, enemies found at higher difficulty levels (advanced, expert) would be given horizontal movement where they oscillate from left right across a random horizontal coordinate interval. This interval would be random and smaller than the width of the road so the user is not able to just anticipate the oscillations. New attributes in the enemy class for the left and right bounds for the random x interval would be required. The enemy draw method would also need to be changed so that the enemy x coordinate is updated to move it left and right continuously. I would need to check that the enemy is in the x interval to determine when to switch its direction.
Customisation is fairly limited in that the options are just solid colours. More character options and a customisable environment would make for a more memorable user experience	As discussed in E3/E4, I would have liked to add a system which allows for a selection of customisation options at 3 different levels dependent on score. The options available would depend on the score obtained in a user's most recent game attempt. This would then be displayed in a corner of the customisation screen to alert the user of the options currently available to them. I would add new objects to the button class for different levels of customisation, for example colour gradients and patterns. The unavailable options would be greyed out using a simple for score check in the function which they are drawn in. On top of this, another portion of the customisation screen could be dedicated to altering the terrain of the game. For example 4 different options each for a different season. The background colour and the road would be changed to fit the theme. This would require the draw function to check for what 'season' the

	game was at in order to load the correct background combination. The background colour variable would now be assigned one of four colours, and the road would now be one of four different images.
There is no multiplayer functionality, which would have allowed for players to play cooperatively to obtain higher scores and progress at higher difficulty levels	In order to implement multiplayer functionality into Motor Mayhem, I would need to take advantage of additional software such as WebSocket to then include a room system which allows players to join into another player's game like the popular game Among Us

Appendix A



Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Motor Mayhem</title>

    <link rel="stylesheet" type="text/css" href="style.css">

    <script src="libraries/p5.min.js"></script>
    <script src="libraries/p5.sound.min.js"></script>
  </head>

  <body>
    <script src="sketch.js"></script>
    <script src="Classes/Button.js"></script>
    <script src="Classes/Player.js"></script>
    <script src="Classes/Bullet.js"></script>
    <script src="Classes/Enemy.js"></script>
  </body>
</html>
```

sketch.js

JavaScript

```
/* This is the file where objects are initialised
the methods are run every frame
this is where all functionality across different files is centralised
*/
let logo, road;
let screen = 0;
let score = 0;
var theme, colourPicker;
var bullets = [];
var enemies = [];
let x1, y1, x2, y2, enemySpeed, interval;
let diffMode = 'Novice';

function preload() {
 road = loadImage('Images/road.png');
 logo = loadImage('Images/mm-logo.png')
 theme = loadSound('Audio/theme.mp3');
}

function setup() {
 frameRate(60);
 createCanvas(800, 800);
 theme.loop();
 theme.setVolume(0.3);

 //initialises buttons
 startBtn = new Button('START', 100, 280, 400, 380);
 customiseBtn = new Button('CUSTOMISE', 100, 280, 400, 540);
 difficultyBtn = new Button('DIFFICULTY', 100, 280, 400, 700);

 //initialise difficulty buttons
 beginnerBtn = new Button('BEGINNER', 70, 330, 400, 350);
 noviceBtn = new Button('NOVICE', 70, 330, 400, 435);
 intermediateBtn = new Button('INTERMEDIATE', 70, 330, 400, 520);
 advancedBtn = new Button('ADVANCED', 70, 330, 400, 605);
 expertBtn = new Button('EXPERT', 70, 330, 400, 690);

 //initialises player
 myCar = new Player(400, 650, '#262626');

 interval = 45;
 enemySpeed = 4;
}

//choosing what screen to display
function draw() {
 road.resize(500, 800);
```

```
 if (screen == 0) {
  menu();
 } else if (screen == 1) {
  play();
 } else if (screen == 2) {
  customise();
 } else if (screen == 3) {
  difficulty();
 }

}

//drawing menu screen
function menu() {
 screen = 0;
 background('#00472E');

 //hides colour picker
 if (colourPicker) {
  colourPicker.hide();
  colourPicker = null;
 }

 //draws road and logo
 imageMode(CENTER);
 image(road, 400, 400);
 image(logo, 400, 170);

 //instructions
 textAlign(LEFT);
 textSize(18);
 fill('white');
 text('Movement - A, D', 11, 17);
 text('Shoot - SPACE', 11, 37);

 //current difficulty notice
 fill('grey');
 textSize(22);
 text(diffMode, 11, 776);

 //draw buttons
 fill('#83C785');
 strokeWeight(6);
 rectMode(CENTER);
 rect(startBtn.getCentreX(), startBtn.getCentreY(), startBtn.getWidth(),
startBtn.getHeight(), 20);
 rect(customiseBtn.getCentreX(), customiseBtn.getCentreY(),
customiseBtn.getWidth(), customiseBtn.getHeight(), 20);
```

```
 rect(difficultyBtn.getCentreX(), difficultyBtn.getCentreY(),
difficultyBtn.getWidth(), difficultyBtn.getHeight(), 20);

 //label buttons
 textSize(35);
 textFont('Impact');
 fill(0);
 textAlign(CENTER, CENTER);
 text(startBtn.getText(), startBtn.getCentreX(), startBtn.getCentreY());
 text(customiseBtn.getText(), customiseBtn.getCentreX(),
customiseBtn.getCentreY());
 text(difficultyBtn.getText(), difficultyBtn.getCentreX(),
difficultyBtn.getCentreY());
}

//drawing play screen
function play() {
 screen = 1;
 background('#00472E');

 if (colourPicker) {
  colourPicker.hide();
  colourPicker = null;
 }

 //instructions
 textAlign(LEFT);
 textSize(18);
 fill('white');
 text('Movement - A, D', 11, 17);
 text('Shoot - SPACE', 11, 37);
 text('Quit - Q', 11, 57);

 //score
 textSize(30);
 text('Score: ' + score, 11, 776);

 //draws road
 imageMode(CENTER);
 image(road, 400, 400);

 //drawing player
 myCar.setStartY(650);
 myCar.draw();

 //player movement
 if (keyIsDown(68) && myCar.getStartX() >= 225 && myCar.getStartX() <= 570) {
  //moves right while not at right boundary
  myCar.setStartX(myCar.getStartX()+6);
```

```
 } else if (keyIsDown(65) && myCar.getStartX() >= 230 && myCar.getStartX() <= 575)
{
  //moves left while not at left boundary
  myCar.setStartX(myCar.getStartX()-6);
 }

 //draw bullet
 for (let i = bullets.length - 1; i >= 0; i--) {
  bullets[i].draw();
 }

 //update bullet position
 for (let i = bullets.length - 1; i >= 0; i--) {
  bullets[i].move();
 }

 //draw enemy
 newEnemy();
 for (let i = enemies.length - 1; i >= 0; i--) {
  enemies[i].draw();
 }

 //update enemy position
 for (let i = enemies.length - 1; i >= 0; i--) {
  enemies[i].move();
 }

 //updates enemy speed depending on setting
 for (let i = enemies.length - 1; i >= 0; i--) {
  enemies[i].setSpeed(enemySpeed);
 }

 //returns to menu if enemy passes the player
 for (let i = bullets.length - 1; i >= -1; i--) {
  for (let j = enemies.length - 1; j >= 0; j--) {
   if (enemies[j].getY() >= myCar.getStartY()) {
    bullets.splice(0, bullets.length);
    enemies.splice(j, enemies.length - j);
    myCar.setStartX(400);
    menu();
   }
  }
 }

 //detect collision
 for (let i = bullets.length - 1; i >= 0; i--) {
  for (let j = enemies.length - 1; j >= 0; j--) {
   if (bullets[i] && enemies[j] && bulletCollision(bullets[i], enemies[j])) { //
null checks for both arrays
   //handle collision
```

```
    score++;
    bullets.splice(i, 1);
    enemies.splice(j, 1);
    }
   }
  }
 }

function customise() {
 screen = 2;
 background('#00472E');
 imageMode(CENTER);
 image(logo, 400, 170);

 //instructions
 textAlign(LEFT);
 textSize(18);
 fill('white');
 text('Quit - Q', 11, 17);

 //creates colour picker
 if (!colourPicker) {
  colourPicker = createColorPicker(myCar.getColour());
  colourPicker.position(375, 580);
 }
 colourPicker.show();

 //draws example character
 myCar.draw();

 //change colour of character to selection
 textAlign(CENTER);
 myCar.setColour(colourPicker.color());

 myCar.setStartY(500);
 myCar.draw();
}

function difficulty() {
 screen = 3;
 background('#00472E');
 imageMode(CENTER);
 image(logo, 400, 170);

 if (colourPicker) {
  colourPicker.hide();
  colourPicker = null;
 }

 //instructions
```

```
 textAlign(LEFT);
 textSize(18);
 fill('white');
 text('Quit - Q', 11, 17);

 //creating buttons
 strokeWeight(6);
 rectMode(CENTER);

 fill('#3EFF9F');
 rect(beginnerBtn.getCentreX(), beginnerBtn.getCentreY(),
beginnerBtn.getWidth(), beginnerBtn.getHeight(), 20);

 fill('#51FF53');
 rect(noviceBtn.getCentreX(), noviceBtn.getCentreY(), noviceBtn.getWidth(),
noviceBtn.getHeight(), 20);

 fill('#FCFF34');
 rect(intermediateBtn.getCentreX(), intermediateBtn.getCentreY(),
intermediateBtn.getWidth(), intermediateBtn.getHeight(), 20);

 fill('#FFB232');
 rect(advancedBtn.getCentreX(), advancedBtn.getCentreY(),
advancedBtn.getWidth(), advancedBtn.getHeight(), 20);

 fill('#FF7556');
 rect(expertBtn.getCentreX(), expertBtn.getCentreY(), expertBtn.getWidth(),
expertBtn.getHeight(), 20);

 textAlign(CENTER);
 textSize(25);
 textFont('Impact');
 fill(0);
 text(beginnerBtn.getText(), beginnerBtn.getCentreX(),
beginnerBtn.getCentreY());
 text(noviceBtn.getText(), noviceBtn.getCentreX(), noviceBtn.getCentreY());
 text(intermediateBtn.getText(), intermediateBtn.getCentreX(),
intermediateBtn.getCentreY());
 text(advancedBtn.getText(), advancedBtn.getCentreX(),
advancedBtn.getCentreY());
 text(expertBtn.getText(), expertBtn.getCentreX(), expertBtn.getCentreY());
}

//adds new bullet to bullets[] array
function keyPressed() {
 if (keyCode == 32) {
  bullets.push(new Bullet(myCar.getStartX(), myCar.getStartY()-30))
 }
 //return to menu option which wipes both arrays
 else if (keyCode == 81) {
```

```
  menu();
 }
 for (let i = bullets.length - 1; i >= -1; i--) {
  for (let j = enemies.length - 1; j >= 0; j--) {
   if (keyCode == 81) {
    bullets.splice(0, bullets.length);
    enemies.splice(j, enemies.length);
    myCar.setStartX(400);
    screen = 1;
    menu();
   }
  }
 }
}

//pushes new enemy to array every 2 seconds
function newEnemy() {
 if ((frameCount % interval) == 0) {
  enemies.push(new Enemy(random(230, 570), -25));
 }
}

//determines whether or not there was collision and returns boolean
function bulletCollision(b, e) {
 return (abs(b.getBX() - e.getX()) < 40 &&
 abs(b.getBY() - e.getY()) < 40);
}

function mouseClicked() {
 //action if startBtn is clicked
 if (mouseX >= 260 && mouseX <= 540 && mouseY >= 330 && mouseY <= 450 && screen == 0) {
  score = 0;
  play();
 } else if (mouseX >= 260 && mouseX <= 540 && mouseY >= 490 && mouseY <= 610 && screen
== 0) { //first 3 conditions for main menu buttons
  customise();
 } else if (mouseX >= 260 && mouseX <= 540 && mouseY >= 650 && mouseY <= 770 && screen
== 0) {
  difficulty();
 } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 315 && mouseY <= 385 && screen
== 3) { //last 5 conditions for difficulty buttons
  enemySpeed = 2;
  interval = 60;
  diffMode = 'Beginner';
  menu();
 } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 400 && mouseY <= 470 && screen
== 3) {
  enemySpeed = 5;
  interval = 45;
```

```
  diffMode = 'Novice';
  menu();
 } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 485 && mouseY <= 555 && screen
== 3) {
  enemySpeed = 7;
  interval = 40;
  diffMode = 'Intermediate';
  menu();
 } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 570 && mouseY <= 640 && screen
== 3) {
  enemySpeed = 9.5;
  interval = 40;
  diffMode = 'Advanced';
  menu();
 } else if (mouseX >= 235 && mouseX <= 565 && mouseY >= 655 && mouseY <= 725 && screen
== 3) {
  enemySpeed = 11;
  interval = 30;
  diffMode = 'Expert';
  menu();
 }
}
```

Player.js

JavaScript

```
/* Class file from which player is created
this forms the backbone for the gameplay
*/
class Player {
  constructor(startX, startY, colour) {
    this.startX = startX;
    this.startY = startY;
    this.colour = colour;
  }

  draw() {
    fill('black');
    rect(myCar.getStartX(), myCar.getStartY()-10, 60, 10, 2); //wheels
    rect(myCar.getStartX(), myCar.getStartY()+20, 60, 10, 2);
    fill(myCar.getColour());
    strokeWeight(2);
    rect(myCar.getStartX(), myCar.getStartY(), 50, 80, 15);
  }

  getStartX() {
```

```
    return this.startX;
  }

  setStartX(newX) {
    this.startX = newX;
  }

  getStartY() {
    return this.startY;
  }

  setStartY(newY) {
    this.startY = newY;
  }

  getColour() {
    return this.colour;
  }

  setColour(newColour) {
    this.colour = newColour;
  }
}
```

Button.js

JavaScript

```
/* Class from which buttons are created
these buttons are the primary form of navigation in my game
*/
class Button {
  constructor(text, height, width, centreX, centreY) {
    this.text = text;
    this.height = height;
    this.width = width;
    this.centreX = centreX;
    this.centreY = centreY;
    this.colour = '#d5ff6b';
  }

  getText() {
    return this.text;
  }

  setText(newText) {
    this.text = newText;
```

```
  }

  getHeight() {
    return this.height;
  }

  getWidth() {
    return this.width;
  }

  getCentreX() {
    return this.centreX;
  }

  getCentreY() {
    return this.centreY;
  }
}
```

Bullet.js

JavaScript

```
/* Class from which bullets are created
bullets fired from player hit enemy and increase score
*/
class Bullet {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  draw() {
    fill(0, 230, 222);
    rect(this.getBX(), this.getBY()-15, 10, 10, 2);
  }

  move() {
    this.setBY(this.getBY()-3);
  }

  getBX() {
    return this.x;
  }

  getBY() {
    return this.y;
```

```
  }

  setBY(newY) {
    this.y = newY;
  }
}
```

Enemy.js

JavaScript

```
/* Class file from which enemies are created which
form the main challenge in the game
*/
class Enemy {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.speed = 2;
  }

  draw() {
    fill('black');
    rect(this.getX(), this.getY()-15, 50, 10, 2); //wheels
    rect(this.getX(), this.getY()+10, 50, 10, 2);
    fill('#6b0500');
    strokeWeight(2);
    rect(this.getX(), this.getY(), 40, 70, 12); //body
  }

  move() {
    this.setY(this.getY()+this.getSpeed());
  }

  getX() {
    return this.x;
  }

  getY() {
    return this.y;
  }

  setY(newY) {
    this.y = newY;
  }

  getSpeed() {
```

```
    return this.speed;
  }

  setSpeed(newSpeed) {
    this.speed = newSpeed
  }
}
```