OCR A Level Computer Science

H446 Component 3
Programming Project

Name: Krishan Sritharar

Candidate number: 8422

Centre number: 12304

Centre name: Queens Park Community School

Contents Page

Analysis

Identifying the Problem

The advancement of technology in the recent decade has improved the lives of humans in many ways. It is the same case in schools, where the introduction of widely available internet access and interactive whiteboards in lessons has made the learning process more efficient. The aid of calculators in mathematics has allowed students to focus more on the problems they are solving in oppose to any simple calculations they need to perform to reach it. At first, this last statement may seem very productive however, as calculators are becoming increasingly powerful and more and more young children are using them, students are becoming more reliant on their calculators for even the simplest calculations. Students don't have an initiative to want to improve their mental maths from a young age as they believe that they could just use a calculator instead. This is a problem as in later life these people would not have developed their cognitive thinking skills, leaving them in search for a calculator in everyday tasks, such as tallying up a shopping bill. This is the problem that I have chosen to investigate and attempt to provide a solution for.

For my project, I will be creating a multi-level, 2D game with the intention of improving the mental mathematical calculation speeds of the player. I will be achieving this by asking the player questions that can be solved mentally, and providing an in-game benefit if they answer correctly. In order to avoid decreasing the flow of the game by a large amount, the questions will be multiple-choice and there will be a time limit to answer each question. The game itself will consist of the player controlling a car to manoeuvre it across a level. To complete each level, the car will need to complete a big jump, during which the player will be asked the most difficult question asked so far in the level. They will need to correctly answer this last question, and the time that they have to do so is determined by how quickly they answered the other questions on the level. As the levels increases, the player will need to control the car over longer distance and through obstacles, adding an element of skill and thus making the game more enjoyable.

Stakeholders

I am making this game in particular for my Maths teacher (Mr Lemma), and the Maths faculty at my school. The finished game will be aimed at students studying in secondary school and in sixth form. Despite this, anyone who wants to improve their mental mathematical calculation speed can play this game and progressively get better. This will be possible as there will be a difficulty selector, meaning that the questions asked will be different depending on what stage the player is through their education. This means that if a player is studying A-Level Mathematics, they will be asked completely different questions to a player who is currently in Year 7. This feature is essential as it makes the game playable by players *with all levels of maths knowledge*. Otherwise, if a player who is in Year 9 is asked an A-Level maths question there is high chance that they will guess the answer thus reducing the fun of the game.

In my school, from Year 7 through Year 10, maths is taught through the form of SMILE (Secondary Mathematics Individualised Learning Experiment) cards. These are tasks that are assigned to each student depending on their ability, and they are expected to complete a fixed number of them per week. Teachers cannot constantly oversee each pupil's work and therefore they can't stop them, if necessary, from using their calculators for simple sums. This form of learning results in less class discussions, in which there could be competitions between students to calculate a sum faster. These situations will usually initiate a desire for one to improve their mental calculation speed in order to beat/continue beating somebody next time. Moreover, having asked a maths teacher at my school, there are very few SMILE tasks that have the aim of getting the student to do calculations without a calculator, and that there is no guarantee that a student will not use their calculator for a non-calculator task as they quote 'don't want to do it the long way'. Additionally, maths teachers use the website 'https://www.mymaths.co.uk/' to provide homework tasks to students however upon analysis, this website also has an online calculator even on very simple tasks, giving

students the choice to use it. Most of the lessons and tasks on this website are text-based and there are very few games to make the learning experience more enjoyable.

As a game where the player is constantly quizzed, a secondary end user of my game could be maths teachers. Allowing maths teachers to provide their own questions for students to answer will make the game into a more enjoyable homework/revision task. There will also be a competitive element to see who can get the highest score in the class, leading to the game being played more times and therefore improving the player's mental calculation speed overall.

I want my game to provide a more fun method of revising topics learnt in class whilst also making the player able to solve those types of questions faster. This hopefully improved cognitive thinking ability will help students approach more difficult questions they face in different ways, as they may have developed different methods of solving questions whilst thinking on the spot.

Why is it suited to a Computational Approach?

The solution that I have proposed to the problem I am intending to solve is a game and therefore it is completely reliant on a computational approach to solve. The fundamental and basic reasoning for this is because games are programmed and computers are required to write this program as well as provide a platform to run the game on. However, analysing the problem in more detail reveals that the output of a game is a series of frames played at a fast rate to make objects on the screen appear to move. To do this these frames need to be drawn and rendered at upwards of 60 frames per second, accounting for any inputs provided by the player as this will change the content shown on the screen. Computers are optimal to do these tasks as they are able to do millions of these calculations per second (to determine what needs to be drawn on the frame), and can then render these frames and output then to a screen at an incredibly fast rate.

During the process of making the game several problem will be encountered including: creating different screens for the user to navigate through, creating animations to enhance the experience of the game, determining collisions between game objects and maintaining a similar experience of the game regardless of one device having better hardware resources. These problems can be most efficiently approached using computational power. For example, when coding the game screens (such as the main menu, level select and pause menu), an overall class and common functions can be used so that code doesn't need to be unnecessarily repeated. The problem with this is not only that more space will be taken up, the program will become more difficult to maintain as any changes need to be made in multiple places.

When calculating the motion of objects in the game, multiplying by a constant usually called 'delta time' maintains the same motion on all devices. For example, on a powerful computer that is able to output 100 frames per second, a game object will move 100 times in one second, and therefore a longer distance that if it was run on a lower power computer at 40 frames per second. The constant 'delta time' is equivalent to '1/frames per second (FPS)' and adding this to the motion calculation makes the computer's FPS irrelevant in the equation. This is because 'FPS' x '1/FPS' = 1, therefore providing the same movement on all devices the game is run on, and maintaining the same experience for all users. Animations are a series of images played at a constant rate and so can be created with the help of definite ('for') loops. Changing the speed at which these images are played controls the length of the animation. Furthermore, collisions can be detected by keeping note of all the objects on the screen and their positions, and checking if any objects have the same position. All of these tasks are perfect examples of jobs that are efficiently solved by computers as they are very repetitive and need to happen at a fast rate.

Computational methods to solve the problem

The computational methods needed to solve this problem can be broken up into 4 sections:

Problem Decomposition – (Breaking down a complex problem into smaller more manageable parts)

The problem is to make a game that the user can play to improve their mental mathematical calculation speeds. This complex problem can be broken down into smaller, more manageable parts. This would include the following steps:

1. Asking the player questions while they are playing, and providing in-game benefits depending on how quickly they correctly answer each question. The questions that are asked are multiple-choice and will depend on the difficulty level selected.

2. The questions need to be stored with the possible answers and the correct answer needs to be correctly identified. However, the correct answer cannot always be the same option as the concept of answering questions will no longer be a challenge if the player works out the pattern. Therefore, the questions can be stored in a CSV (comma separated value) file, with the question, three incorrect answers, correct answer and difficulty scale of the question on each row.

3. After reading in the CSV file in the program, the order of the four answers can be randomised so that there is no consistent pattern in the options. This means that the correct answer can be stored in the same index in the CSV file, making it easier to maintain.

4. The 'fun' element of this game comes from the player having to control a car across multiple levels, which will progressively get more difficult. The physics behind the game can be implement using equations of motion and collisions between game objects, such as the car and the track. The player will have the control to accelerate and brake the car. This introduces skill into the game as the player needs to carefully control the car so that it doesn't flip and crash, which will result in them failing the level. The player will also need to provide an input when selecting their answer to a question. Additionally, the player will need to provide input when navigating through the menu screens.

For each level, there will be a fixed number of minor questions that the player will need to answer and this amount will increase with harder levels. In order to complete each level, there will be a major question that the player needs to answer correctly at the end. This question will be more difficult than the minor questions, and the amount of time that the player has to answer this is determined from how quickly they answered the minor questions. There will be a time limit for the player to answer each question, which will depend on the question's difficulty, and exceeding this time limit or getting too many of the minor questions incorrect will result in a failure of the level. As there is now a consequence for incorrect answers, the player is less likely to repeatedly guess to avoid answering the questions.

Pattern Recognition – (Looking for similarities among and within problems)

Fundamentally, each level has the same concept of getting the car from the start to the end of the track. This means that the same code for controlling, calculating the motion of and rendering the car can be used for each level. Moreover, the questions that will be asked will always follow the same format, i.e. the question and multiple-choice answers will be in the same places. Therefore, once again the same code can be used to ask the question to the user at several points in a level. Having these sections of code in functions or classes enables them to be efficiently used again. For example, if the position of multiple-choice answer needs to be moved then the code will only need to be changed in one place, reducing the chances of errors occurring.

Another similarity in the problem is in the menu screens. Once again, all of the screens will be essentially the same, just with different content. Therefore, having common functions or classes for displaying text, images and buttons on these screens is a more efficient method of solving this problem.

Algorithms – (Develop a step-by-step solution to the problem or rules to follow to solve the problem)

In order to solve this problem I will be using the divide and conquer approach and within each stage, use iterative development. This method of breaking down the problem into smaller, still challenging but more achievable tasks helps to solve the overall problem more efficiently. This is because it becomes certain aspects of how to solve a task may become apparent when solving the current task. The iterative development process ensures that there is always a working solution even if all the desirable functions are not included. This process allows functionality to be progressively added to the program until all of the set requirements are met.

Abstraction – (Focusing on only the important information and ignoring irrelevant detail to reduced computational resources and programming)

Abstraction is key concept when designing software programs as it reduces the complexities of the final program by removing any unnecessary details. This in turn reduces the time required to create a working solution, as less code will be needed, whilst also reducing the computational power/resources required in executing the program. For my project the following aspects of the game can be abstracted:

The friction between the car and the track needs to be abstracted. In a real-world situation, friction between the car's tyres and the track will cause the car to slow down and this resistive force will increase with the car's velocity. In order to maintain a constant experience for the user, i.e. the same acceleration of the car is felt each time the player presses the button, this concept needs to be abstracted from my game.

The concept of having a fuel and the fuel level decreasing over time in the levels needs to be abstracted. The reason for this is because the main aim in the fixed levels is to get the car to the end of the course and adding fuel to this problem will make it more complicated for the player. Furthermore, with the concept of changing fuel level comes the idea of the car's mass decreasing as fuel is used. This would mean that over the same duration of time in a later stage of the level, the car will accelerate faster because it will have a lower mass than it did at the start. This adds unnecessary complexity to the game, and so is another reason why the idea of fuel should be abstracted from the levels in the game.

When the car crashes the damage made to the car can be abstracted. This can be achieved by adding graphical effects to cover up the car as soon as it crashes. It would be unnecessary to calculate and draw the damaged car as when the player presses the 'try again' button, the car is in its original condition again. Moreover, the player could be startled by the fact that they just crashed a vehicle (albeit in a game) and decide not to continue playing the game. This is another reason why the damage made to the car when a level is failed needs to be abstracted.

When the user is playing the game and pressing the buttons there is no need to keep track of how long the 'accelerate' or 'brake' buttons are held down for. Checks only need to be made to see if the button is still pressed, and if so perform the appropriate action. When answering the questions, if the player attempts to press all the answers at the same time, the answer that was pressed first will be taken as their choice.

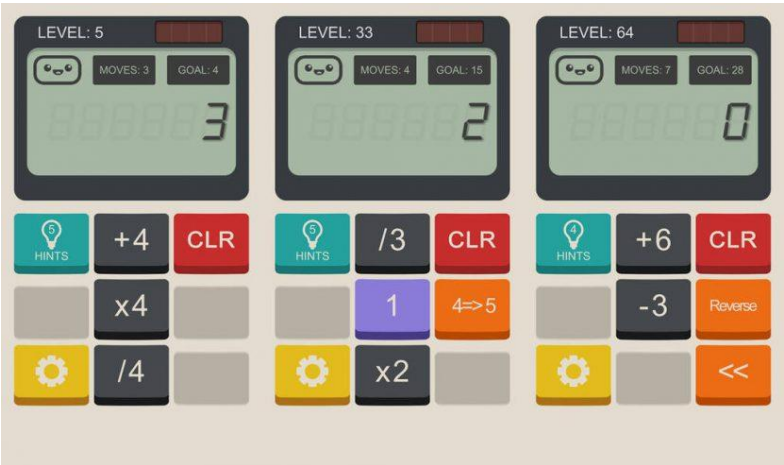
Researching the problem

Existing Solutions - Calculator: The Game

'Calculator: The Game' is a 2D, mobile, puzzle game that was published to the 'Google Play' and 'Apple App' stores in the middle of 2017. This games attempts to develop the player's ability to approach and solve complicated puzzles and problems.

Links to the game:
Google Play Store: https://play.google.com/store/apps/details?id=com.sm.calculateme&hl=en
Apple App Store: https://itunes.apple.com/gb/app/calculator-the-game/id1243055750?mt=8
Developer website: http://www.simplemachine.co/game/calculator-the-game/

The aim of this game is for the player to complete all 199 levels and as a result, develop their ability to spot patterns in and solve problems they later encounter. In this game the player has to reach the number shown beside 'GOAL:' in the specified number of moves using the numbers and functions on the calculator's keypad. The game begins with basic functions, such as addition, subtraction, multiplication and division, but quickly advances to levels which take more thinking. The advanced functions increase the difficulty of the game by making the player approach each problem in different ways. This however achieves the aim of the game whilst also giving a sense of achievement to the player when a level is complete. This emotion is what usually fuels the progression of the game by making the player want to continue playing. These difficult functions include: 'SUM', which gives the total of the addition of all the digits on the screen, '$a \rightarrow b$', where every instance of digit a is changed to digit b, 'REVERSE', which reverses the order of the digits, 'MIRROR', which changes the digits ab to $abba$, 'a^n', which multiplies a by a power of n, and 'SHIFT >', which changes the digits $abcd$ to $dabc$.



This game has a simple yet aesthetically pleasing design and user interface. From the start of the game, there is an assistant called 'Clicky' (shown in the top-left corner of the calculator screen) which welcomes the user to the game and walks them through a tutorial. This immediate interaction with the user grabs their attention whilst also making them familiar with the game and the controls. There is a realistic, button-pressing sound effect when a key is pressed in this game. Also, there is no way of muting the game noise, which may have been intentional by the developers so that the user is satisfied by the key pressing sound effect every time they tap on the screen. The only options in the settings screen are to change the current level, view the leader board and buy hints. Therefore there is very limited user customisation and no rewards for completing levels.

This game has been heavily reviewed at given 4.7 stars out of 5 in a total of 23,908 reviews on the Google Play Store, and 4.6 starts out of 5 in a total of 161 reviews on the Apple App Store. After analysing a considerable amount of the reviews marked as "most helpful", it is clear that this game is liked amongst the puzzle game community. Reviewers have said that "the rules are simple and allow depth and complexity" and that they "love the simple interface and cute visuals and sounds". However, there have been several consistent complaints such as "other than the mascot's reactions, there is no goal" and "the game bothers you too often with advertisements and requests for reviews early in the game".

Parts that I can apply to my solution

The game's simple and user-friendly interface and concept of an assistant to grab the user's attention whilst also helping them through early levels are ideas that I plan on adopting to my game.

In my opinion, this game has a lack of an achievable goal and therefore stops being fun after the player has completed around 20-30 levels. This is a problem that I will face in my game and I plan on solving it by introducing the idea of the player collecting coins during the level. This in-game currency can later be used to purchase different vehicles and themes. This would keep the player interested as there will consistently be a goal that could be achieved in the near future.

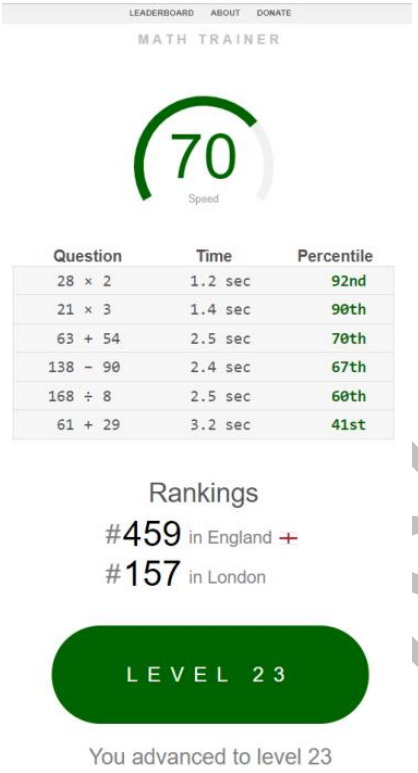
Existing Solutions - Math Trainer

Math trainer is a web-based application intended for improving the player's mental arithmetic.

Link to the application: https://www.mathtrainer.org/

"It is designed to adapt to a user's current ability, offering highly-tailored practice questions, tips for improving, and accurate performance assessments." The app is free to use and based in the browser so the user's progress is constantly saved. There are 100 levels, which range in difficulty, and in each level there will be up to ten questions that the player has to answer as fast as they can. The player cannot move onto the next question until answering the current question correctly. There is a target time associated with each question and this time depends on several factors, such as the number of digits in the answer, whether borrowing or carrying is needed and the number of mental computations needed to solve the problem. The time limit to complete each level is the sum of the target times of the question in the level. If the player completes the level in a quicker time than the limit then they can advance onto the next level. Otherwise, the player will need to redo the level and try to complete it faster. Every time a question is answered the time taken to do so is added to a database and this data is directly used to calculate the target time.

$$\begin{array}{r} 176 \\ \div \quad 8 \\ \hline 22 \end{array}$$



The website has a minimalistic design with contrasting green buttons and black text on a white background. After completing a few levels, the player starts to get ranked in London, and then in England. This feature gives the player a sense of accomplishment as they see that they are the n^{th} best person in London and England in this game. Moreover, in the level end screen the player is shown what percentile their speed of answering each question is. From this the player can analyse which types of questions and operations that they are slow at and choose to further practice them. The only other feature on this application is the leader board. This page shows the username and nationality of a list of people worldwide who have managed to complete a large majority of the levels.

After analysing this game, I believe that it achieves its aim to advance the learning of mental arithmetic for the player however, the only goal that the player is working towards is a place on the leader board. This will make most players stop playing this game after completing some levels, as they have no reason to continue. In the FAQ section, the developer stated that the reason that they questions are not multiple choice was because "there's no way to prevent a user from using the process of elimination, which is really a different skill". This statement is true however in my game, if the player had to manually type out the answer, the flow of the game will be massively affected. Additionally, the anxiety of getting the question wrong and failing the level may cause them avoid guessing.

Parts that I can apply to my solution

Something that I will however be adopting from this application is the concept of ranking the player's score against scores achieved by other local (played on the same device) players. The reason for this is because the ranking will create a competitive element to the game, making the player want to complete the level in a quicker time thus making the player answer the questions asked in a quicker time. As the player is answering the questions quicker, they are developing their mental calculation speeds and therefore my game will be achieving its aim.

Interview with Client

The following is a detailed transcript of a conversation I had with a maths teacher at my school regarding my project.

Client: Mr Lemma (Head of Mathematics Faculty)

Krishan: "Good Morning Mr Lemma, thank you for agreeing to take part in this interview"

Mr Lemma: "That's not a problem Krishan, I am happy to help"

Krishan: "The aim of my computer science project is to increase the mental calculation speeds of students in secondary school any higher. So you control your car and have to maintain full control of it throughout the level. If you flip your car over the game ends. However in my game, at points during the level, the game will pause and you will need to answer question maths questions to procced.

Mr Lemma: Ah ok

Krishan: So for level one, there could be a simple hill to climb and in the level you have to answer three questions. There will be one final question in the level and the amount of time you get to answer this question depends on how quickly you answered the first three questions.

Mr Lemma: Ah I see, so you get punished for taking a long time to answer the easier questions.

Krishan: Exactly, answering the easier questions quicker gives you more time to do the final, more difficult question. You can only complete the level by answering the final question correctly.

Mr Lemma: Ah ok, I know what you mean.

Krishan: I want to make my game for secondary school students so there are three age brackets, years 7 to 9, GCSE level which is years 10 to 11, and A Level and beyond. So I was wondering what topics you think will be suitable for these age groups.

Mr Lemma: Algebra is quite easy to code correct?.

Krishan: Yes, it I, but I am going to have the questions in spreadsheet alongside the answers.

Mr Lemma: Can it include diagrams for example.

Krishan: Yes, I'm could figure out a way to include images for the questions.

Mr Lemma: So we can ask about triangles and --

Krishan: but the question can't take too long to answer because then it will feel as though the game has stopped.

Mr Lemma: No, no, no. Say for example we show the user a triangle --

Krishan: and ask them what the area?

Mr Lemma: Yes, and what is a particular angle and the angles can be kind like 60, 30 and 90 so the numbers are easy to add mentally. Or you can make two of them equal and get them to find the remaining angle.

Krishan: Would that type of question be suitable for students in years 7, 8 and 9.

Mr Lemma: Yeah, and the algebra could be used for all ages by making the questions more challenging. So for example F(x) = 2x + 5, if x =5 what is f(x), questions like that. To make it slightly harder you can have questions like ½ x +6 = 8, what is x.

Krishan: Yes, I see. The aim of the game is to improve your mental arithmetic speed and in the process make you a better mathematician.

Mr Lemma: Yeah it does. How would you do the graphics for the triangle questions.

Krishan: I would have to draw the triangle beforehand and somehow link the question to the image so that it is loaded when the question is asked.

Mr Lemma: How many questions do you need?

Krishan: I am not sure yet.

Mr Lemma: What we can do is sit down together and make some questions up because you wouldn't know what mathematics a year 8 or 9 student knows. I mean that your standard is pretty high and so you don't want to give them questions which are too difficult. How many questions would you need?

Krishan: I thinking of creating around 10 to 12 levels but there will also be another mode in my game where you continuously play until you crash the car.

Mr Lemma: Ah ok, so you need a large number of questions but you can progressively make the questions trickier.

Krishan: and the car could travel faster, making it harder to control and so easier to crash and fail.

Mr Lemma: Oh does it

Krishan: Yes because game makers don't want you to finish the game.

Mr Lemma: That's typical, that's what frustrates my son (laughs)

Krishan: (laughs)

Mr Lemma: You could also ask them about chances

Krishan: Probability questions?

Mr Lemma: Yes, so what is the probability of rolling a three or a four on a dice? Are the questions multiple choice?

Krishan: Yes, as the game is testing how quickly you can do mental arithmetic and not how quickly you can type in the answer.

Mr Lemma: Precisely. You can even be a bit cruel and give wrong answers which are very close to the correct answer.

Krishan: Ah yes, so it's more difficult to guess or use the process of elimination to answer the question.

Mr Lemma: Yes, that's very good, I think the idea is very good.

Krishan: Do you think mental arithmetic is important?

Mr Lemma: Absolutely, at any stage in maths if you are not strong with your mental arithmetic, even if all of your other calculations or steps are correct, you will get caught out.

Krishan: If you don't know your mental arithmetic

Mr Lemma: Yes, it will cause you to struggle.

Krishan: So do you think that mental arithmetic skills should be taught and improved from a young age

Mr Lemma: Yes absolutely.

Krishan: So this is a questionnaire that I have created and am going to send out to students. (Shows Mr Lemma the questionnaire). So it asks you for your year group and then you have to give an answer of how long it took you to answer this question. This is to get an idea of how much time to give to answer the questions.

Mr Lemma: (talks about question number 7) Oh this is a hard question.

Krishan: Really, can't you just divide to simplify the left hand side

Mr Lemma: Yes but that using your advanced knowledge. Lots of people would try an answer so Does 3 work?, Does 2 work? to find the solution.

Krishan: There is a mode in the game where you just answer questions to earn some in game currency.

Mr Lemma: Oh ok

Krishan: So in this mode if you answer a question incorrectly, time gets deducted from the remaining time. So how long do you think this time limit should be?

Mr Lemma: The best way to find this out would be to try it on people.

Krishan: Yes, this is a question on my questionnaire (question number 9)

Mr Lemma: In that case I think that 90 seconds in a good amount of time.

Krishan: Ok

Mr Lemma: So for the question topics include algebra, numbers, approximation, angles, geometry and a bit of probability.

Krishan: Thank you sir for all your help and advice.

Mr Lemma: I look forward to playing your final game.

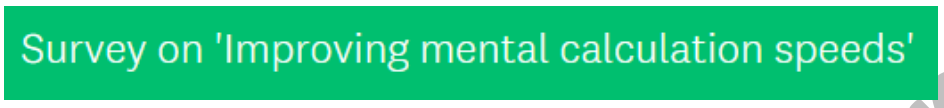
Krishan: I'm sure you would complete the game too quickly sir (laughs)

Mr Lemma: (laughs)

Krishan: Thank you sir, once again.

Survey

In order to determine values to use in the game, such as how much time the player should be given to answer different difficulty questions, I have decided to conduct a survey. The survey was created and conducted using a piece of software called 'Survey Monkey'. This software allows surveys to be efficiently created and distributed using web and mobile links, whilst also providing graphs and summaries of the data when analysing the results.

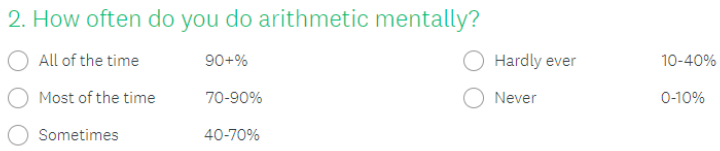


Here are the questions that I decided to ask and justifications for why I chose them.

1. What year are you in?/ How old are you?
   - Year 7 - 8 - 9 Ages 11-14
   - Year 10 -11 Ages 15-16
   - Year 12 - 13 Ages 17-18
   - Ages 19+

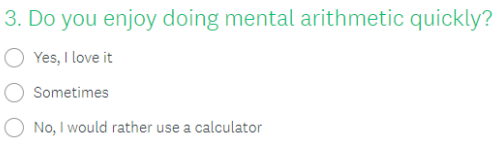
This question is used to estimate how the rest of the answers in this questionnaire are distributed by age. I will aim to distribute this survey to an equal number of students in each age range.

2. How often do you do arithmetic mentally?
   - All the time 90+%
   - Most of the time 70-90%
   - Sometimes 40-70%
   - Hardly ever 10-40%
   - Never 0-10%



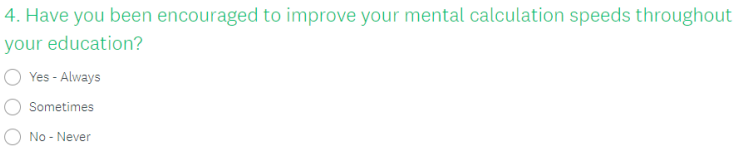
This question is used to discover student's opinions on mental arithmetic and get an idea of how much they use it in their daily lives.

3. Do you enjoy doing mental arithmetic quickly?
   - Yes, I love it
   - Sometimes
   - No, I would rather use a calculator
   - Depends on the question



The aim of this question is to discover how much people enjoy doing mental arithmetic. The project's goal is be to increase this enjoyment or make the process of improving their mental arithmetic more fun.

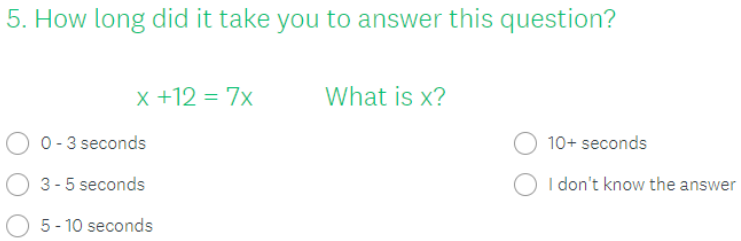
4. Have you been encouraged to improve your mental calculation speeds throughout your education?
   - Yes - Always
   - Sometimes
   - No – Never



This question has to motive of discovering if there are any existing methods of increasing mental arithmetic speeds and whether the students are taking those opportunities.

5. How long did it take you to answer this question in your head?

   - $x + 12 = 7x$ What is x?

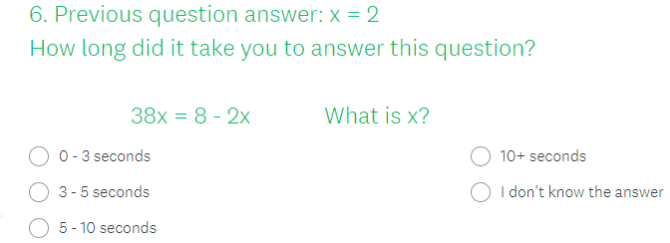


- 0 - 3 Seconds
- 3 - 5 Seconds
- 5 - 10 Seconds
- 10+ Seconds
- I don't know the answer

This is an easy algebra question and the answer provided will be used to get an idea of the time limit to assign easy algebra questions.

6. Previous question answer: $x = 2$
   How long did it take you to answer this question in your head?

   - $38x = 8 - 2x$ What is x?

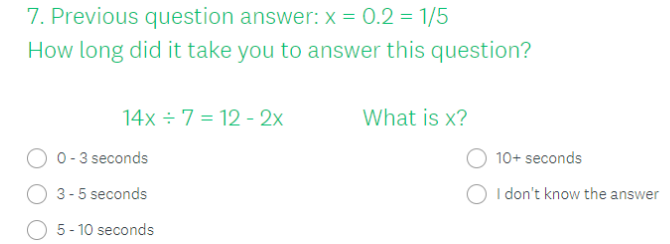


- 0 - 3 Seconds
- 3 - 5 Seconds
- 5 - 10 Seconds
- 10+ Seconds
- I don't know the answer

This is a more difficult algebra question as he answer is a fraction. The answer provided will help determining the time limit to allocate intermediate difficulty questions.

7. Previous question answer: $x = 0.2 = \frac{1}{5}$
   How long did it take you to answer this question in your head?

   - $\frac{14x}{7} = 12 - 2x$ What is x?



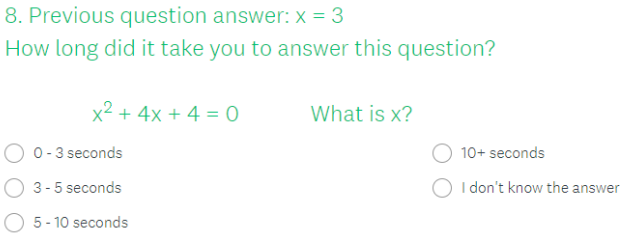
- 0 - 3 Seconds
- 3 - 5 Seconds
- 5 - 10 Seconds
- 10+ Seconds
- I don't know the answer

This is an even more difficult algebra question as it involves division. The trick that needs to be spotted is to first perform the division to get a simpler equation. The answer provided will give a representation of the time limit required for the questions for Years 10-11 and Years 12 -13.

8. Previous question answer: $x = 3$
   How long will you need to do this question in your head?

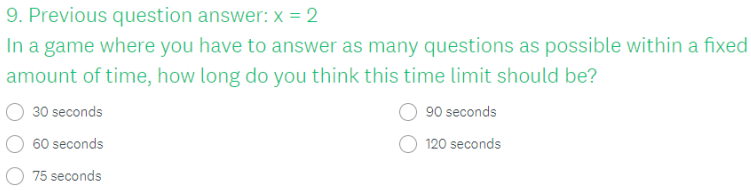
   - $x^2 + 4x + 4 = 0$ What is x?

   - 0 - 3 Seconds
   - 3 - 5 Seconds
   - 5 - 10 Seconds
   - 10+ Seconds
   - I don't know the answer



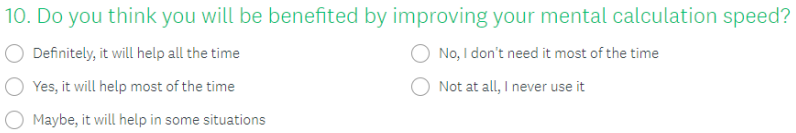
This is an algebra question involving a quadratic. The trick is to notice that it can be easily factorised. This question would fall into the difficult category and the answer will help determine the time limit for this.

9. Previous question answer: $x = 2$
   In a game where you have to answer as many questions as possible within a time limit, how long do you think the time limit should be?
   - 30 seconds
   - 60 seconds
   - 75 seconds
   - 90 seconds
   - 120 seconds



This question is for deciding on the time limit in the mini game. This time limit shouldn't been too long as then the mode would appear to be boring, however it also can't be too short and make the player feel that they have been treated unfairly.

10. Do you think it will benefit you to improve your mental calculations speeds?
    - Definitely, it will help all the time
    - Yes, it will help most of the time
    - Maybe, it will help in some situations
    - No, I don't need it most of the time
    - Not at all, I never use it



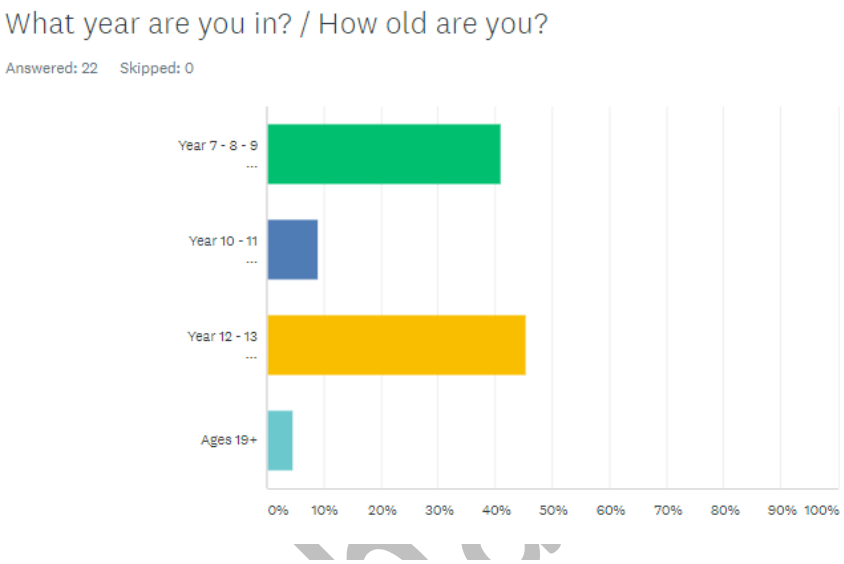
This final question has the aim of seeing how many people see better mental calculation speeds as beneficial. I chose to ask this question to get an approximation of the percentage of people who are willing to improve their mental calculation speeds. These are the people who will be willing to play my game.

Analysis of Survey

A total of 22 responses were gathered for this survey.

Question 1:

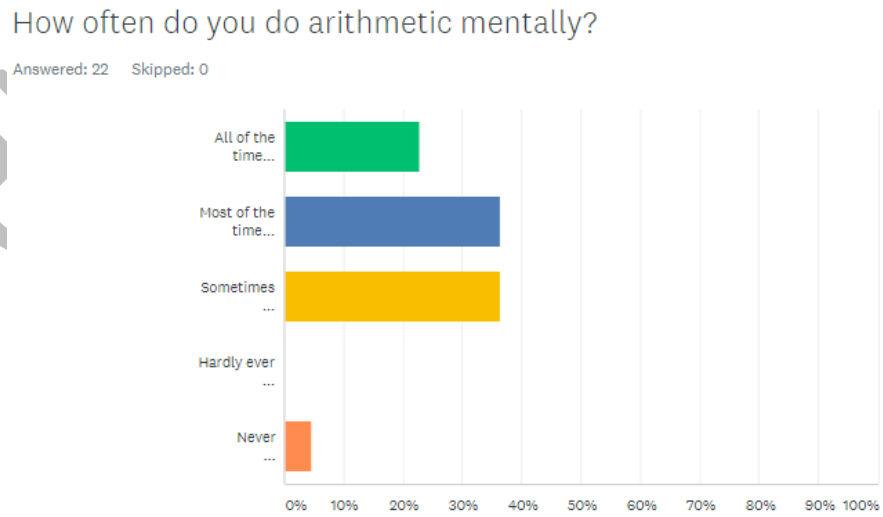
The data from this question shows the distribution of ages of the respondents to the survey. The majority of the respondents were students from Years 7 to 9, and students from Years 12 to 13. This is a good spread of respondents as it implies that the results for the remainder of the questions will be split between prospective users who have greater and less mental arithmetic experience when compared to each other.



ANSWER CHOICES		RESPONSES	
Year 7 - 8 - 9	Ages 11 - 14	40.91%	9
Year 10 - 11	Ages 15 - 16	9.09%	2
Year 12 - 13	Ages 17 - 18	45.45%	10
Ages 19+		4.55%	1
TOTAL			22

Question 2:

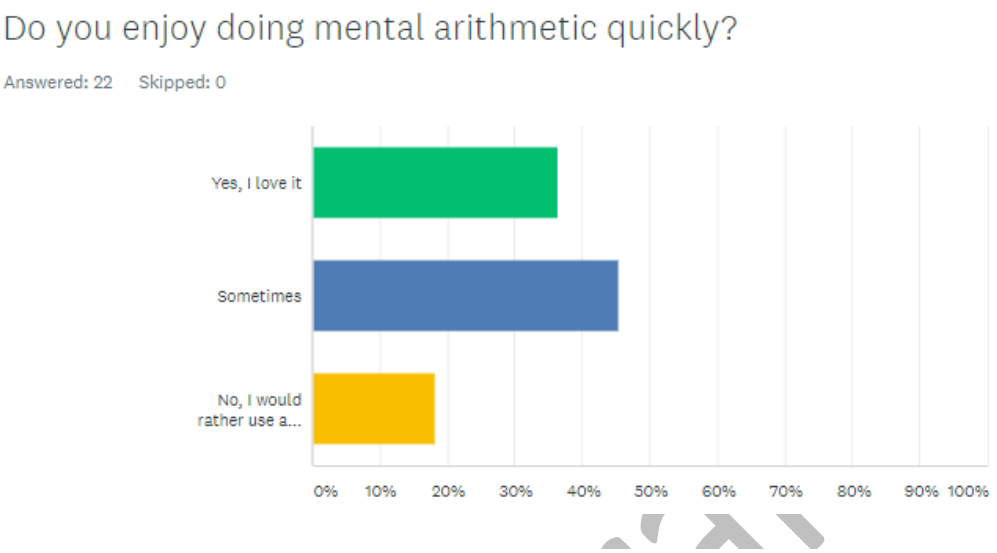
This data shows that more than 95% of the sample population used mental arithmetic occasionally, with over 59% using it most of the time. This shows that mental arithmetic is required by most students and so improving their speed to perform these calculations will help many students answer questions quicker. This will also mean that potentially, over 95% of people will be able to save time in every-day tasks which requires mental arithmetic by improving their speed.



ANSWER CHOICES		RESPONSES	
All of the time	90+%	22.73%	5
Most of the time	70-90%	36.36%	8
Sometimes	40-70%	36.36%	8
Hardly ever	10-40%	0.00%	0
Never	0-10%	4.55%	1
TOTAL			22

Question 3:

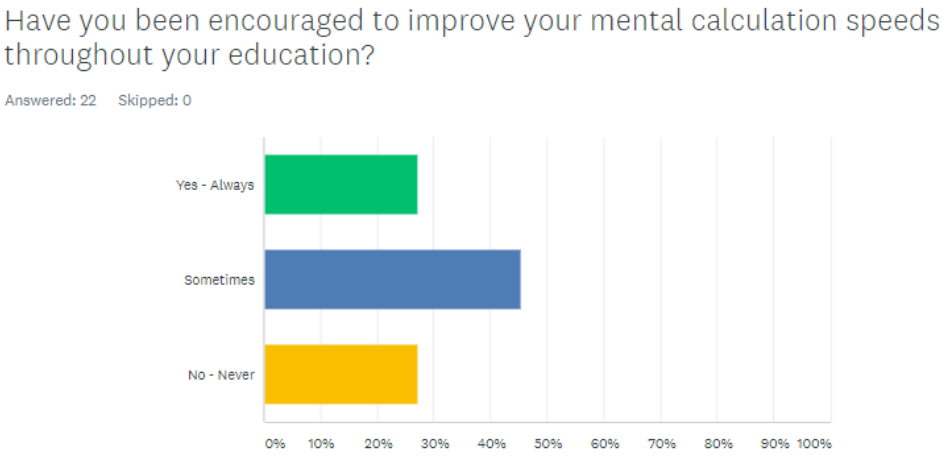
The data produced by this question suggests that over 80% of people enjoy the satisfaction produced by competing mental arithmetic quickly. This encourages the development of this project as it will further increase the speed at which these people can complete arithmetic.



ANSWER CHOICES	RESPONSES	
Yes, I love it	36.36%	8
Sometimes	45.45%	10
No, I would rather use a calculator	18.18%	4
TOTAL		22

Question 4:

This data shows that only 27% of the respondents have definitely been motivated to improve their mental calculation speeds throughout their education. This means that there is still a large proportion of students who need to be encouraged to improve this skill. My project will provide a fun and efficient method of achieving this task.

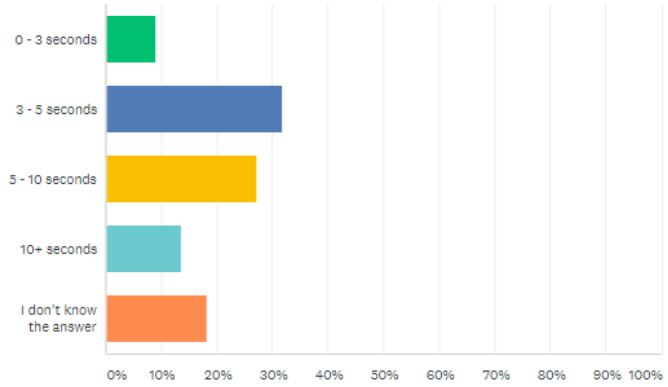


ANSWER CHOICES	RESPONSES	
Yes - Always	27.27%	6
Sometimes	45.45%	10
No - Never	27.27%	6
TOTAL		22

Question 5:

How long did it take you to answer this question? x +12 =
7x What is x?

Answered: 22 Skipped: 0



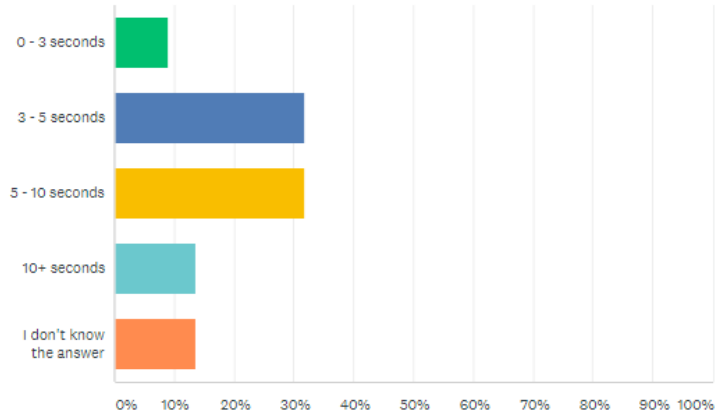
This question provided mixed results but an overall outcome that it took 68% of the respondents up to 10 seconds to answer this question. This question will fall into the easy question category therefore the easy questions should be given up to 10 seconds to answer the question. Additionally, 18% of the respondents couldn't calculate the answer. This is probably the students from the Years 7-9 category, implying that this question may even be too difficult for them. This fact should be taken into consideration as the easy questions in the program will be intended towards this age range of players.

ANSWER CHOICES	RESPONSES	
0 - 3 seconds	9.09%	2
3 - 5 seconds	31.82%	7
5 - 10 seconds	27.27%	6
10+ seconds	13.64%	3
I don't know the answer	18.18%	4
TOTAL		22

Question 6:

Previous question answer: x = 2How long did it take you to answer this question? 38x = 8 - 2x What is x?

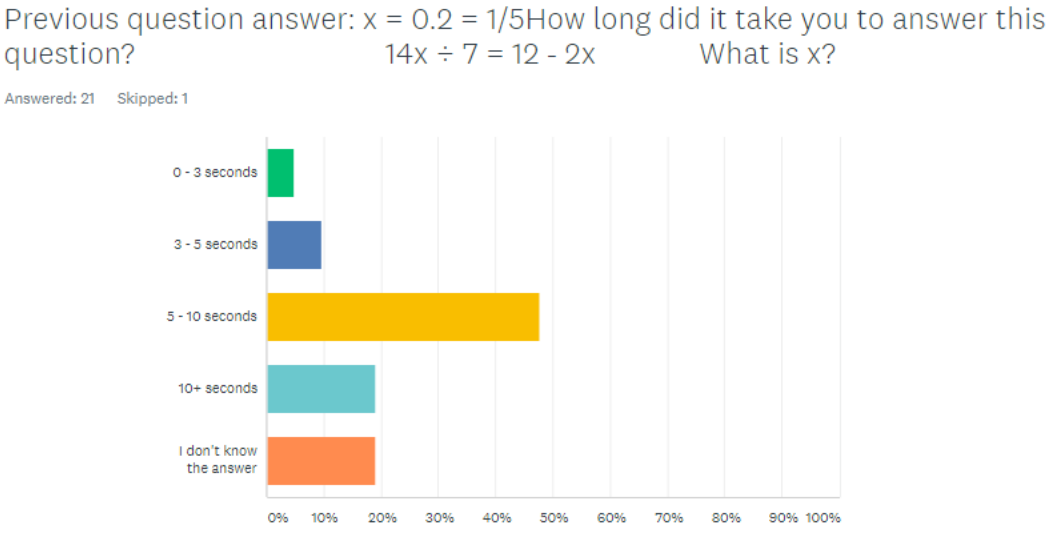
Answered: 22 Skipped: 0



This question will be considered to be an easy question. The data from the respondents indicated that this question also took upwards of ten seconds to answer. This confirms the data produced in the previous question and the decision to give the player 10 seconds to answer the easy questions in my game.

ANSWER CHOICES	RESPONSES	
0 - 3 seconds	9.09%	2
3 - 5 seconds	31.82%	7
5 - 10 seconds	31.82%	7
10+ seconds	13.64%	3
I don't know the answer	13.64%	3
TOTAL		22

Question 7:

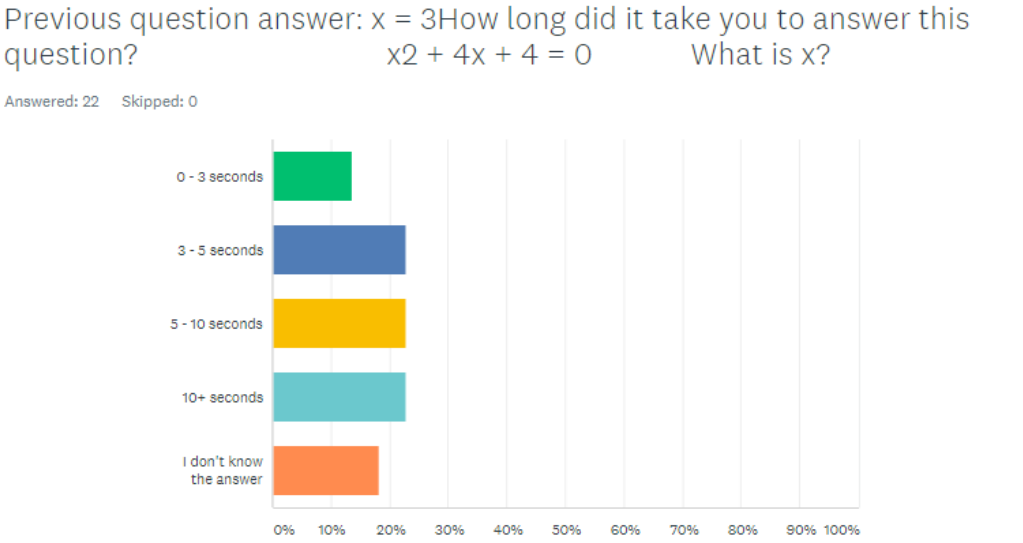


As previously explained, this is a more difficult algebraic question and so as expected, it took a longer duration of time to answer the question. The data from the respondents show that it took more than 5 seconds for 65% of them to calculate the answer. Therefore, the time limit for the intermediate or medium difficulty questions in my game will be 15 seconds.

.

ANSWER CHOICES	RESPONSES	
0 - 3 seconds	4.76%	1
3 - 5 seconds	9.52%	2
5 - 10 seconds	47.62%	10
10+ seconds	19.05%	4
I don't know the answer	19.05%	4
TOTAL		21

Question 8:



This would also be a medium difficulty question and the data shows mixed responses for the time taken to complete this. A plausible reasoning for this spread of data is because some respondents are more familiar with these styles of questions and therefore were able to quicker figure out the factorization procedure. Once again, the 18% of respondents who were unable to answer the question were probably younger students who have not yet been introduced to the ideas of factorization. This is the reason why I will have different difficulties in my game, in order to cater to the strengths of all age-ranges.

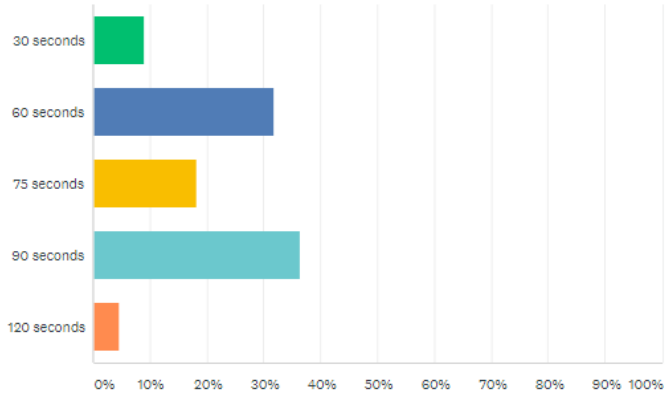
ANSWER CHOICES	RESPONSES	
0 - 3 seconds	13.64%	3
3 - 5 seconds	22.73%	5
5 - 10 seconds	22.73%	5
10+ seconds	22.73%	5
I don't know the answer	18.18%	4
TOTAL		22

Question 9:

This was an opinion-based question and the data suggests that a time of 60 or 90 seconds for the mini game is favourable. As 90 seconds was voted for by an additional 5%, I have decided to use this time for the min game in my project.

Previous question answer: x = -2In a game where you have to answer as many questions as possible within a fixed amount of time, how long do you think this time limit should be?

Answered: 22 Skipped: 0



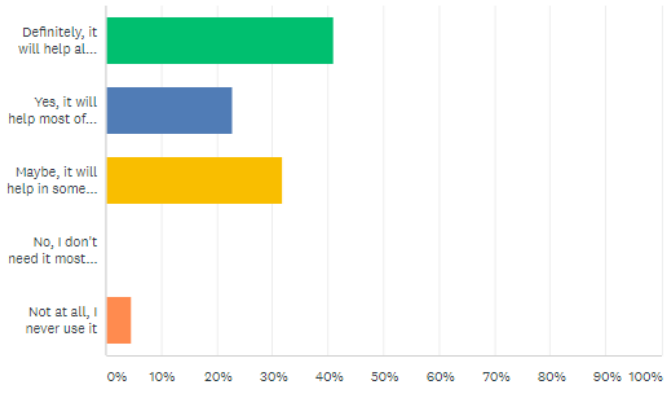
ANSWER CHOICES	RESPONSES	
30 seconds	9.09%	2
60 seconds	31.82%	7
75 seconds	18.18%	4
90 seconds	36.36%	8
120 seconds	4.55%	1
TOTAL		22

Question 10:

The data produced by this question show that more than 60% of the respondents believe that it will be beneficial for their mental arithmetic skills to be improved. This implies that lots of people will be willing to play my game out in order to improve their mental arithmetic calculation speeds whilst enjoying themselves and having competitions with their friends.

Do you think you will be benefited by improving your mental calculation speed?

Answered: 22 Skipped: 0



ANSWER CHOICES	RESPONSES	
Definitely, it will help all the time	40.91%	9
Yes, it will help most of the time	22.73%	5
Maybe, it will help in some situations	31.82%	7
No, I don't need it most of the time	0.00%	0
Not at all, I never use it	4.55%	1
TOTAL		22

Key features of the proposed solution

After analysing the results from the questionnaire and having a conversation with a maths teacher at my school, I have created a list of all the features that I want to include in my game. These features will be aid the development of the player's mental mathematical calculation speeds while also enhancing the player's experience during the game and keeping them wanting to continue playing.

Essential Features

Mathematic questions need to be available to the program so that the user's mental mathematical calculation speeds can be tested and eventually improved. There will need to be a large amount of questions for this game so that the user is not made to answer the same question too frequently. There will be three difficulties that the player can pick from: easy, intermediate and hard. The easy questions will be aimed at users who are aged between 13 and 14 and will mostly consist of simple single operation calculations. There could be some double operation or slightly more complicated questions such as expanding brackets occasionally. These questions will correspond to a higher score if answered correctly however if the player answers incorrectly, they will not be penalised heavily. As people between this age-range are not at GCSE level yet, these questions are purely intended to improve the user's mental mathematical calculation speeds in order to help in every-day activities. The intermediate questions will be aimed at users between the ages of 15 and 16. Users who fall into this bracket are currently doing GCSEs so the questions asked are more difficult. Such topics include trigonometric graphs and completing the square questions. The difficult category of questions will be aimed at users over the age of 17. This means that they could be doing AS/A Level Mathematics and/or Further Mathematics therefore, the questions and topics get more difficult. Such questions include: quick differentiation, integration, SUVAT questions and trigonometric identity questions. As the questions are helping students improve their calculation speeds, this game could also be thought of as a revision game. Questions will need to be selected and asked at certain points during each level.

The player needs to manoeuvre the car to the end of the track in order to complete the level. To do this the player needs to have certain controls of the car. These controls will include accelerate and brake. The accelerate button will cause the car's velocity to increase in the forward direction. There will be a maximum velocity for the car so that it cannot continuously accelerate as this would make the car very difficult to control. The brake button will gradually slow the car as long as the button is being pressed. The motion of the car will be calculated using equations of motion such as $s = ut + \frac{1}{2}at^2$, where s is displacement of the car, u is the car's initial velocity, $a$is the car's acceleration and t is the time taken.

All of the different screens and pages in the game need to have a common link so that the user can easily navigate around the application. This introduces the idea of having a main menu and different menu screens for pages such as settings. Upon launching the game, the player will be directed to the main menu, from which they can choose to continue playing the game or explore other elements of the applications. These other elements include: instructions of how to play the game, settings and game options. The instructions on how to play the game will be given to the player during the first levels however if they need to access this again, it will be available in the menu pages. In the settings and game options pages the user will be able to change certain settings to the game, such as game music level and sound effects level, and certain game options, such as the difficulty of the questions. When the game is first launched the user will be prompted to select this difficulty however, if they later change their mind they can alter this setting.

Extra/Desirable Features

Alongside the essential menu screens, leader board and statistics pages can be added to enhance the player's experience while playing the game. The leader board page will have a table of what score was achieved on each level. When a level is completed the score will be automatically added to the leader board, unless a higher score from the same player already exists. When the game is first launched, a name will be requested from the user. The leader board will show this name alongside the score achieved. At the end of the level, there will also be the option to submit the score as a different user. The player will need to enter another name and this result will be saved to the leader board. Adding this functionality will allow there to be leader boards and a competitive element even on a local (i.e. not internet based, as that will greatly increase the complexity) game. The score achieved by the player will be calculated based on how quickly they completed the level, the difficulty of the level and, the difficulty of the questions answered. The statistics page will contain facts about what the user has achieved while playing this game. This will include: the total number of questions answered, the total amount of coins collected and spent, how much progress has been made in the game and the total number of games played. This page will allow users to compare their statistics to other, which may motivate them to continue playing the game. Furthermore, it will give the user a sense of how much of the game they have so far completed.

When the game temporary pauses to let the player answer the question during a level, instead of having the car stationary in the background, I have decided to have an effect on the car. This would be a frozen-time/bullet-time effect, where the car will very slowly continue to move forward while the player is answering the question. The reason behind this is to give the player a sense that the game is still continuing in the background, insisting them to quickly answer the question so that they can regain control of the car.

Adding collectable items to a game is an easy method of keeping the player hooked. In my game, making the player play the game more will aid in achieving the goal of the project as every time they answer a question, they will slowly develop their cognitive thinking ability. As a result of this, I have decided to add coins to my game. Coins will be scattered around the level on the track and the player will need to drive the car into these coins to acquire them. Coins will also be rewarded to the player at the end of a level. The amount of coins that are rewarded will be calculated using the score achieved by the player. The player will be able to spend the coins in the in-game shop. Here they will be able to purchase vehicles (cars and bikes) as well as different themes to use in the game. They will also be able to buy power ups that can be used whilst playing. An example of such power up is to skip a question. At first this power up appears to defeat the purpose of the game however, if the player is stuck on a level because they are repeatedly being asked questions on a topic that they haven't been taught yet, they can use this power up to complete the level. It would be unfair to penalize the player due to their gaps in knowledge as the aim of the game is to improve the mental calculations speeds using their existing knowledge. Other features that the player can buy in the store include vehicle options such as a horn and lights. These would provide no practical advantage and only be aesthetic feature that the player can purchase for their own enjoyment.

An achievements page can be added to the game to award the player for progressing through the game. Examples of achievements include, completed 2 levels: reward - 40 coins and, answered 50 questions correctly: reward - 100 coins. These achievements will give the player a sense of accomplishment and set them goals to achieve in order to gain the rewards. Another method of keeping the user wanting to play the game is by providing rewards for playing it every day. This can be carried out by writing the current date to a file when the game is launched and checking if the current date if one day after the date on the file. Coin rewards can be given awarded in this situation, and the rewards can increase in size for consecutive days (i.e. 10 coins on day 1 and 20 coins on day 2) to keep the user wanting to return to the game.

A mini game could be added where the player attempts to answer as many questions correctly as possible in a fixed amount of time. A score will be calculated at the end of the mini game and a certain number of coins will be awarded to the player depending on this score. This score will then be added to the leader board and the player can attempt to beat their high score with every attempt. In order to remove the option of the user repeatedly guessing the answer, there could be a penalty for every incorrect answer such as a deduction in the time remaining to answer the questions. Additionally, there could be a penalty to the score (i.e. a certain amount is deducted from the score), which will affect the amount of coins earned. This mini game will directly be aiding the development of the user's mental calculation speeds, whilst also providing a method of progressing through the game by earning coins to spend in the in-game shop.

In addition to the user completing levels in the game an endless mode could be added. In this mode, the aim will be to see how far the player can travel on the track. Questions will be asked in this mode as well, with a major question asked after every ten minor questions. The concept of fuel is introduced to the game in this mode. Once again, how quickly the minor questions are answered determines how much time the player has to answer the major questions. Answering the major questions correctly will provide fuel for the car. The amount of fuel that is rewarded will be calculated using the difficulties of the questions, and the amount of time the player had reaming to answer the major question. When the fuel runs out, the game is over. After a certain distance, the speed of the car will slowly begin to increase, thus making the car more difficult to control. If this did not happen, controlling the car will stay at the same difficulty at all times, eventually making this mode tedious. Increasing the difficulty in manoeuvring the car, whilst also reducing the time available to answer the questions by a small amount, will keep the player under pressure and therefore keep this mode, and the game, entertaining. This mode will also have a section on the leader board page and the final score will be calculated using the total distance travelled by the player. What is engaging about this mode is the ability to pick-up and play without any need to complete levels in order. Also, if the player has completed all of the level, they can continue playing the endless mode. The lack of a finishing point in this mode is a benefit as it means that the player will never be able to finish the game.

Limitations to the program

There will be three different difficulties that the user can select however if the user is younger than the age range for the easiest difficulty, then they will not be able to answer the questions, and so will not be able to play the game. I will attempt to reduce the effect of this by making some of the questions in the easiest difficulty simple enough for younger players to understand.

People with a visual impairment will find it difficult to navigate the menu screens and play the game. I will attempt to make my game a better experience for these people by having a highly contrasting theme with vibrant colours. With this type of theme, people with minor visual impairment may be able to identify the objects on the screen and therefore will be able to play the game.

The inputs to this game will be through a keyboard or a touch interface. This means that people without hands will not be able to interact with the controls and so can't play the game. To minimize the effect of this, I will add the option to change the buttons associated with the controls so that handicapped users can make the game playable for themselves.

Scores achieved on the game on different devices cannot be saved on a common leader board. This is because this feature would require a database connected to the internet, and would need the device the game is being played on to have a constant connection to this database. This adds unnecessary complication to the game and would also mean that a dedicated server to store the database will need to be created.

Requirements for the solution

I will be making my game in two stages. The first stage will be using the programming language Python and the Python module Pygame, and for the second and final stage, I will be using the game creation software Unity and the programming language C#. I have decided to use Unity for my final game so that it can be available on multiple platforms, i.e. Android, iOS and Windows. A game created using Python and Pygame can be compiled to an executable format (.exe), which will allow the game to be played on any Windows machine without the source code present. However, the game cannot be compiled into an .apk (Android Package) or an .ipa (iOS App Store Package) format, meaning that the game cannot be distributed to mobile platforms. I want my game to be available on mobile devices as these are the platforms that people will be able to more easily access. This will allow my game to be played in many more situations (e.g. while in transport) as there is no requirement to have a computer present. Furthermore, the game will be easier to distribute on mobile platforms as the process of downloading games on these mobile operating systems are much more user friendly than on a computer.

I will make a fully functional, working solution of the game in Python using Pygame, and then start using Unity and C# to create the final version with better graphics and an improved user interface. The program created using Pygame will provide an understanding of the algorithms required to create the game. If a fully-function version of the game is created using Python, then essentially the algorithm needed in Unity will be the same after translating this code into C#. Despite this, previous experience with Unity has made evident that the game creation process is not entirely the same, meaning that other techniques will need to be used in this process. As the version of the game created using Python is going to be replaced by the version created using Unity, it would be pointless to make the Pygame version have beautiful graphics and smooth animations. Therefore, I will use simplistic character design and graphics for the Pygame version and aim on achieving all the functionality primarily. Touch control inputs can be easily implemented using Unity when making the mobile version. Additionally, at the end of the project the final game can also be compiled into an executable format by changing the touch controls to keyboard buttons and mouse movement.

Hardware and Software Requirements and Justification

For the first stage of the project, the computer must meet the following hardware specifications:

- ✓ Have a minimum processor clock speed of 1GHz
- ✓ Have a minimum of 1Gb of RAM
- ✓ Have a minimum of 100MB of free disk space

These are only the minimum requirement to run the program however having a more powerful computer will allow the game to run more smoothly at a high and consistent frame rate. Having more RAM will allow more of the game files, such as character images and the questions CSV file, to be stored in the faster RAM than the much slower secondary disk drive, affecting the performance of the game. Furthermore, having a CPU with a faster clock speed and more cores will allow the program to more quickly perform calculations and tasks such as rendering the frames.

The computer must meet these software requirements:

- ✓ Have an operating system that is either Windows, Mac or Linux
- ✓ Have the python module Pygame and Numpy correctly installed

The computer must be running these operating systems as they are supported by Python. Also, in order for the game to run, the pygame module needs to be correctly installed, which can be completed using the PIP

command. Some elements of the pygame module use the Python module numpy and therefore it needs to be installed on the computer as well.

For the second stage of the project, the computer must meet these following hardware specifications:

- ✓ Have an Intel Pentium 4 / AMD Athlon 64 CPU or above
- ✓ Have a minimum of 4GB of RAM
- ✓ Have a Nvidia GeForce 8500 GT / AMD Radeon HD 2000 GPU or above
- ✓ Have a minimum of 10 GB of free disk space

The CPU must meet this requirement as there needs to be support for the SSE2 instruction set. All CPU after the ones mentioned above support this instruction set and so can be used. When the game is compiled and run a large amount of memory is required to keep the game playing a consistent frame rate as otherwise, the CPU will spend valuable processing time paging the data to and from the slow, secondary storage drive. Additionally, a graphical processing unit with DirectX 10 and shader model 4.0 capabilities are required. The graphics cards stated above are the first cards to support both of these and any GPU released afterwards will be capable. Unity is a large file as there are many components of the game engine that need to be downloaded. Additionally, several development kits and other programs, mentioned below, need to be download in order to compile games for certain platforms. These are large files and so a minimum of 10 GB of free space is recommended.

The computer must meet the following software requirements:

- ✓ Have an operating system that is 64-bit version of Windows 7, 8, 8.1 or 10 or macOS 10.11+
- ✓ Have the Android SDK (software development kit) package downloaded
- ✓ Have the Java Development Kit (JDK) package downloaded
- ✓ Have a version of Visual Studio downloaded and installed

The computer must be running the specified operating systems as they are supported by Unity. The Android SDK and JDK packages are required by Unity to create and compile the game into an APK format. As the scripts in Unity will be written in C#, a C# compiler is required and this can be achieved using an IDE (Integrated Development Environment) called Visual Studio. For running the game a computer with the same hardware requirement as stated above is needed, and an Android device with the operating system Android OS 4.1 or later is required.

Other requirements include a keyboard, mouse and monitor for operating and interacting with the computer and an Android device with a touch screen to test the game. An app called Unity Remote can be used to run the game on a connected Android device without having to compile the game into an APK format. This will aid in the testing of the game and so should also be a software requirement.

System Requirements

Inputs

1. Accepts touch/mouse-button inputs to navigate through the menu screens.
2. Accepts touch/keyboard inputs to control the car throughout the level.
3. Wait for response to the asked question.
4. Wait for response to the major question.
5. Buttons to pause the game and/or go back to the main menu can be pressed during the game.
6. User inputs name to associate score with.
7. User answers whether they want to play the next level or exit to the main menu.
8. User navigates back through the menus.
9. For the mini game - wait for response to the asked question.

Processes

1. Loads the player into the chosen level.
2. Calculate the change in motion of the car and renders the changed frame
3. At specific points in the level, pause the game, choose a question randomly and ask it.
4. Keep track of time taken to answer question. Use this to calculate time for major question.
5. Determine the result of the answer (i.e. whether the player has answered correctly)
6. Resume game and give control back to the player.
7. Keep repeating this process to ask questions during the level.
8. For the major question, choose a random major question and ask it.
9. Check that the question was answered within the time allowed, if not fail the level.
10. Determine the result of the major question. If incorrect, fail the level. Otherwise complete the level.
11. Works out the player's statistics for the played level.
12. Determines the score and total coin amount for the level.
13. Ask for name to put score on leader board. Updates/saves the score and name to the leader board.
14. Ask if user wants to play the next level, if so launch the next level, otherwise open the main menu
15. For the mini game - start a countdown timer and ask the user a random question.
16. Works out whether the user has picked the correct answer.
17. Checks if there is time for another question, if so randomly picks another question from the chosen difficulty and ask it.
18. Calculate the score for the mini game and use that to calculate the reward.

Outputs

1. Displays the main menu screens.
2. Displays the loaded level and controls
3. Display the movement of the car throughout the level.
4. Displays any required effects during the game.
5. Show the question and possible answers when asked.
6. Visually show the result of the question and show whether the player answered correctly.
7. Display the major question and the time to answer it.
8. Visually show the result of the major question and show whether the player answered correctly.
9. Show game over/ level finished screen depending on result.
10. Display the score for the level and coins earned.
11. Launches either the next level or the main menu screen depending on the user's answer.
12. For the mini game - Display the question and possible answers.
13. Visually show the result of the question and show whether the player answered correctly.
14. Show the end screen with the score and number of coins earned.

Analysis of Requirements

Problem	Analysis	Solution
The same questions may be frequently asked in the same level/game mode.	Functions from the 'random' library in python and a similar library for C# can be used to randomize the order of the questions.	A record of the frequently asked questions will be kept and used to filter questions when one is picked. This will ensure that the same question is not asked too frequently. Even if this does happen, there will be more than 3 incorrect answers which will be randomly selected so that the options for the question will not be the same.
How long should the mini game mode last for?	I proposed this question in the survey to obtain information from the age groups of users who my game is aimed towards.	After analysing the data produced from the survey, I have decided to make the mini game last for 90 seconds.
Players might spams the answers	A check will be made in every frame of the game for any user input so it will be possible that events of quickly pressing buttons are created. It may be favoured by some players to spam the answers and attempt a brute force strategy in oppose to trying to find the answered to the asked question	Implement a procedure where no answers are accepted if there are multiple inputs at the same time. This would happen if the user attempts to press all of the answers simultaneously. Alternatively, another procedure where the user's first answer is accepted could be implemented. This would work because it is highly unlikely that the user will press the answers with 1/60 seconds, which is how long before the inputs are checked again.
The player might not know the answer to the question	There are different difficulty categories to pick however if within one of these categories there is a question that the user does know it would be unfair to punish them. They will be frustrated as they will not be to blame for not having covered the knowledge required to solve this question	Add a power up for skipping a question. As the aim of this game is to improve the player's mental calculation speed by asking questions, these power ups could be expensive so that they cannot be used frequently. This will ensure that the player keeps enjoying the game and does not resort to guessing and restricting their cognitive process.
Why should the user play the game	The players will need some sort of purpose to keep playing the game so that they return to it frequently. IF there was some sort of competitive element, the player will be fuelled to come back and replay levels and game modes by their ego. The player also needs to have fun.	The leader board page can be used by the player to view and analyse the scores that they have achieved on each played level. This will show how they scale with other players, making them want to replay levels where somebody has achieved a greater score. This will ensure that the player is having fun.

Success Criteria

No.	Success Criteria	Evidence
1	Load the questions from the CSV file and store the required data. (i.e. store the questions with the difficulty selected)	Run the program and check the local question storage to make sure only questions from the chosen difficulty are stored.
2	Create an arrangement of screens which can be linked to form a menu system. All the settings and features should be accessible from here.	Run the script and assess full functionality of all screens.
3	Level is loaded and the player's car and in-game items are spawned.	Make logs to the console/shell when items are spawned in. Run the game and check for the expected logs when the level is loaded.
4	Player provides input to control the car. The appropriate action is performed on the car. The new position of the car is rendered to the screen.	Run the game and make sure all the controls for the car are functioning correctly.
5	➢ Temporarily pause the game and ask the player a question at specific stages in a level. ➢ Initiate a countdown depending on the difficulty of the question. ➢ Wait for the player's response and determine their result. ➢ Resume the game	Run the program and check that it only pauses when a question is asked. Check that the question and possible answers are clearly readable. Make sure the program interprets the user's answer correctly by logging to the console/shell. Make sure the player has full control of the game when it is resumed.
6	The time to answer the final question in the level should decrease depending on the tine it took to answer the minor questions.	Whilst running the game, keep note of how quickly minor questions are answered and check if there is the appropriate time for the final question.
7	When the car is in the air, it will rotate. If the car lands with the roof on the track, it has crashed. The level is failed.	Run the game and purposely land the car on its roof. Check that the level fails.
8	The score for the level is determined from how quickly the level is completed. Score will increase if coins on the level are collected.	Complete the same level at different speeds and whilst collecting different amounts of coins. Check if the scores are different.
9	The next level will be unlocked only when the previous level has been completed.	Run the program and attempt to play an locked level.
10	Question asked will be random and the choice of answers will be random. (i.e the correct answer will not always be A)	Keep note of the position of the correct answer and questions asked to see if there are any patterns.
11	Mini game for answering questions correctly functions: ➢ Questions are only asked within the time period ➢ The player's result is determined at the end of the question ➢ Player is rewarded coins depending on the number of their correct answers. ➢ Incorrect answers will decrease time remaining ➢ Score is calculated depending on the difficulty and number of questions answered	Run the script, make sure only questions from the selected difficulty are asked. Use a stopwatch to check the time period is correct. Make sure the correct determination of the result is performed after each question. Identify that incorrect answers lead to time deduction by purposely answering questions incorrectly. Check whether the score is correctly calculated by running the mini game a number of times with different amounts of correct answers each time.

12	Endless mode of the game correctly functions: ➢ Questions are regularly asked ➢ Time for major question adjusted according to minor question answer times. ➢ Fuel is awarded for correct answering of major question ➢ Mode ends when player runs out of fuel ➢ Score is calculated depending on distance travelled	Run the game and check that the player has full control of the car. Make sure the correct questions are asked at the appropriate times and the time for the major question is changed. Do this by logging the calculated time to the console/shell. Purposely use up all the fuel to check that the mode ends. Check that the score is different by running the mode multiple times and achieving different distances each time.
13	All scores, for levels, mini game and endless mode, are correctly added to their respective leader boards. All leader boards are accessible from the leader board screen via the main menu.	Complete a level and add the score to the leader board under a name. Navigate to the leader board screen and make sure the name and score are shown on the leader board for that particular level.
14	The game statistics are correct and are accessible via the main menu. The game achievements are correctly unlocked as the game progresses. Coins can be used to purchase different items in the shop.	Note the statistics before playing the game and after to check that the information shown ins correct. Do this by noting the number of questions correctly answered. Make sure achievements are awarded at the correct points during the game. Note the total coins before collecting the reward for the achievement and make sure the after coin total is greater.
15	The game is entertaining for the player.	Make the menu system a user friendly interface with large buttons. Have aesthetically pleasing graphics and animations in the game.

This list of criteria alongside the system requirements represents the logical development process that will be followed in this project. Each success criteria point has been created using relevant system requirements. The success criteria have also been structured in a logical the way that will be followed throughout development. This is expressed through the essential features requiring the game to work being in the criteria before the extra and desirable features.

The evidence column represents the method that will be followed in order to test that this success criteria has been fully met. This will be used in the testing section after the game has been created however, it will also be used throughout development. This is because the development process is iterative and a component of the program needs to be fully functional before moving on as later parts of the program will rely on it.

Design

Decompose the problem

The main problem of making the game can be divided into smaller, more manageable sections to aid throughout development. This will help because these parts will be more manageable, allowing errors to be more easily identified. I have decomposed the problem into the following major sections which are suitable for computational solutions. Also, I have further divided the problem within each section and justified the decision made:

1. Create the main game loop
    - Create the class for the game and the main functions to update the game sprites and draw to the screen

    The justification for using a main game loop is so that the game follows logical order. Every frame of the game, functions will be called to collect any input data, update the game sprites using this data, and draw the updated sprites onto the screen. This logical order allows the program easier to trace and locate errors when necessary.

2. Create the main menu system
    - Create a class for creating buttons

    A class would be an appropriate computational solution for creating buttons as it will allow several instances of the class to be simultaneously running (i.e. when there are numerous buttons on one page). Classes provide better management of the data associated with each button, resulting in less code required to check the state (clicked or not clicked) of a button.
    - Create a method of managing the screens in the main menu system

    The main menu screens will need to be managed so that the code to create them is not unnecessarily repeated. Moreover, this solution would allow similar sections of code to be stored in separate files and be imported into the main game file, thus providing better readability of code.
    - Create each of the screens

    In order for the screens to be reused without repeated code, they will need to be created in functions. These functions can then be called whenever a certain button is clicked. An identifier variable can be provided as a parameter when creating a button to have a reference of which function (i.e which screen) needs to be called. Text can be drawn on the screens using a function for drawing text. This function can then be called every time text needs to be printed to the screen anywhere in the game. Additionally, the background of the screens could be solid colours and then advance to being images.

3. Create the fundamental game characters and mechanics
    - Create a class for the car

    Once again a class is used for storing all of the information regarding the player (such as their position and velocity) as it allows this data to be more easily managed. There will be several functions inside the class to add behaviours to the car, one such method is the update function. Here the game events will be checked to see which keys have been pressed and determine the appropriate direction to move the sprite.
    - Add movement functionality to the car class

    In order to move the sprite, the position of it will need to be changed. Once the direction of movement has been determined in the update function, the position will need to be updated before the sprite is drawn. Therefore this code will be executed every frame so that the movement of the car sprite is fluent.

4. Create/load a level

- Import and read the level

All the levels could either be loaded in when the game is first launched or individually when a particular level is loaded. In the first case, the code for loading the level will need to be executed before a new game is created. The advantage of this is that the levels will load faster however there may be performance issues if there are is restricted amount of memory. Despite this, the level data would not be excessively large and so performance issues would not be a concern. In the second case the code will need to be executed when the level is loaded using the screen management solution created previously.

- Draw the level

A class could be defined to create objects to act as platforms. When the car collides with this object, the car will remain on top of the platform. This object can now be used to create the level. After loading the level, it could be iterated over an at particular positions, a platform could be spawned.

- Create a Camera object

The level will be larger than the size of the screen and so there will need to be a scrolling effect as the car reaches either side of the screen. In order to do this, the camera view of the game needs to be adapted. This could be achieved by creating a class and storing the required functions there.

5. Add question asking functionality

- Load and read the questions

When the game is first launched al of the questions should be loaded and saved locally. Storing this information in a two-dimensional array will allow the same column and row numbers from the spreadsheet to be used to access the data, thus making the process of manipulating the data easier to implement. Any images or diagrams which are used in the question also need to be loaded and locally stored. In order to reduce memory requirements the pictures can be loaded in if the question is chosen.

- Choose the question to ask and the answers to propose

An appropriate question should be chosen from the question database depending on the level number and the chosen difficulty. A record could be kept of the previously asked questions to ensure that the same question is not repeated too often, as this will make the game repetitive and boring. There is one correct answer for every question and 4 incorrect answers. Three of the four incorrect answers will need to be randomly selected and the correct answer will need to be merged in with these answers in a random order. This process ensures that there are no patterns or similarities in the correct answer's order in relation to the other answers.

- Display the data to the screen

Once the question and answers have been chosen a function will need to be created to display them on the screen. The question will need to resize depending on its length therefore, the length of the text will need to be analysed to change the size of the font. Moreover, interactive buttons need to be created for the four answers. If the question uses any diagrams then this will also need to be drawn to the screen.

- Get the user's input and determine result

When a button is clicked, the function that it is linked with will be called. In this function checks can be made to see if the answer chosen matches the correct answer. If so, the code associated with answering correctly will be executed otherwise, the code which handles incorrectly answered questions will be executed.

6. Set the rules of the level

- Game over if car crashes

If the car's roof collides with the track then the level will be failed. This event can be checked in each frame by checking if the top if the car has collided with any track objects. If this does happen, the game over screen will be shown.

- Add question asking power-ups

Adding an item which the player will need to drive over to trigger the process of asking a question will make the player aware that they are going to be asked a question. This will also add a random element to the times when the questions are asked, giving the player a different experience every time they play a level. When the car collides with this object, the game will temporarily pause while the question is asked

- Incorporate coins

Coins can be added around the track and the coins will be collected by the car driving through them. A collision between the coins and the car will be checked and when this happens, the player's coin count will be increased and the coin will be deleted. The coins could randomly spawn around the track or have their positions hard coded into the levels. If the coin's position was linked to the level then when the level is created the coins will also be spawned in otherwise, the positions of the coin will be randomised when the level is launched. After this, the coin objects will be initiated.

7. Save and update level information

- Save the highscore

When a level is completed, the score will be calculated and this will be checked against the current highscore. If it is larger than the previous highscore, then this score will be save. Otherwise, the current score will be discarded after the game over screen is left. The highscore will be loaded and read from a text file when the game first loads up and stored in a local variable. Any changes made to the highscore will be updated by changing the value of the local variable, and then writing the new highscore to the text file when the game is closed.

- Save the completed level

A record of the locked and unlocked levels will need to be kept to ensure that the user cannot play levels which still have previous unfinished levels. The state of a level can be stored in a text file and loaded when the game first starts. Once again, this data can be stored locally and then the text file will be updated when the game closes. If the player finishes a level, the level's state will change from unlocked to completed, and the succeeding level will change from locked to unlocked.

8. Create remaining levels and modes

- Make the levels

Use the previously created level making method to create the remaining levels. Make the levels different so that the game doesn't feel repetitive. To this by adding the coins, questions and other game animations in different locations.

- Load and draw all of the levels

Implement a similar procedure to the first level to load the remaining levels into the game. This would require having a different camera object for each level as the dimensions of the map will be different. The levels then need to be correctly drawn to the screen.

- Create the mini game

As previously explained, the mini game will consist of a fixed period of time in which the player can answer questions to earn coins and compete with other scores achieved whilst playing the mini game. To do this, a timer can be started when the mode begins. Once this timer reaches zero, the mode must end and the final score for the user should be calculated. During this period, questions should be repeatedly asked and if answered correctly, the score will increase. However, if answered incorrectly, the time remaining will decrease. At the end, the number of coins awarded for the achieved score should be calculated and then the score can be recorded in the leader board tables.

- Create the endless mode

To do this, certain game features such as fuel and boost will be added. A continuous map or randomly generating level needs to be created. The player will control the car in this level and aim to answer the questions whilst also manoeuvring the level. Every five questions there will be a major question which will provided fuel. This needs to be answered correctly in order to progress on in the

mode. The player can continue playing the level until they answer one of these questions incorrectly or lose all of their fuel. At the end, their score should be added to the leader board for this mode.

9. Add graphics to the game

Graphics are required to enhance the player's feel for the game and make them want to keep playing it. They will improve the aesthetics of the game and therefore will also make it look more professional.

- Add images for the sprites

Each sprite in the game should be given an image. As object oriented programming will be used for this project, the image attribute of each instance of a particular game sprite can be the same. Up until this point, the aesthetics of the game is not essential as the fundamental functionality of the program will be under development. Therefore, the game sprites can be represented by rectangles with different colours.

- Make the map better aesthetically

The map should also up until this point will be basic as the functionality will be under development. After finishing this, the images used for the map can be enhanced. A possible method for this is to use multiple tile images and a tile image processing software to create the map using multiple images. This will also make it easier to create the maps, depending on the usability of the third-party software being used.

- Music can be added

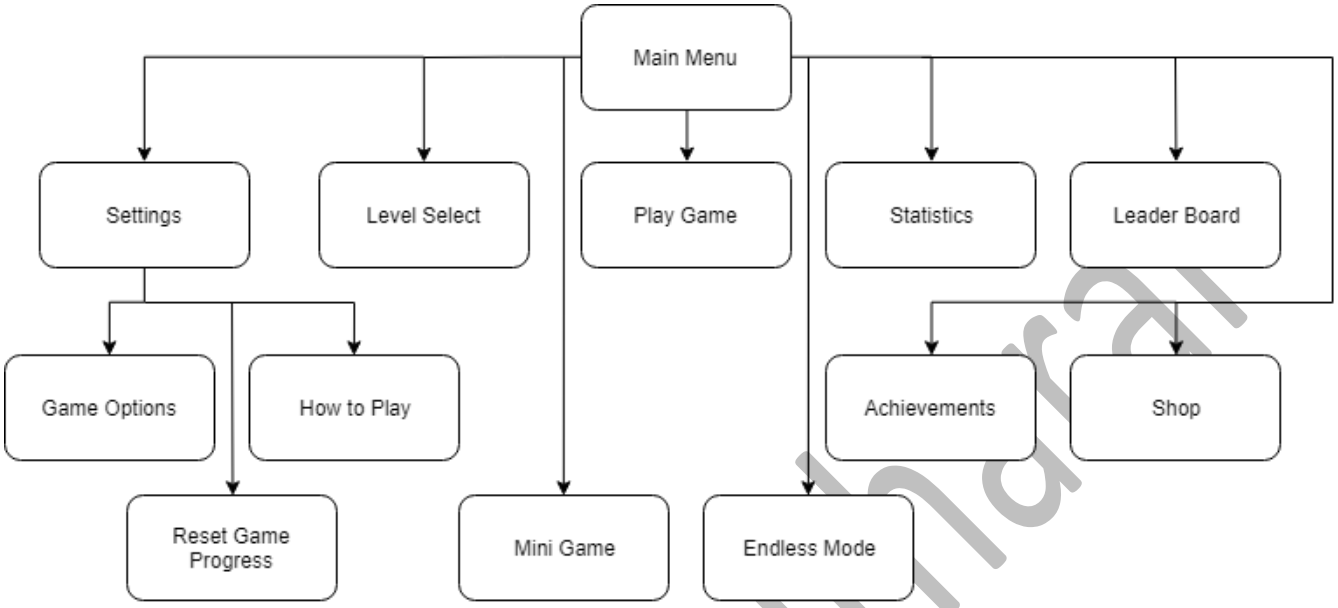
Music will be added also with the intention of enhancing the atmosphere whilst the user is playing the game. Music will entice them as it may leave them with a catchy tune whilst also immersing them into the game during the levels. In order to appeal towards all users, an option to switch off the music should be added.

The problem being approached by this project has now been broken down systematically into a series of smaller problems. Each of these smaller problems are simpler and therefore more suitable for a computational solution. The explanation and justification for the requirement of each stage has been included.

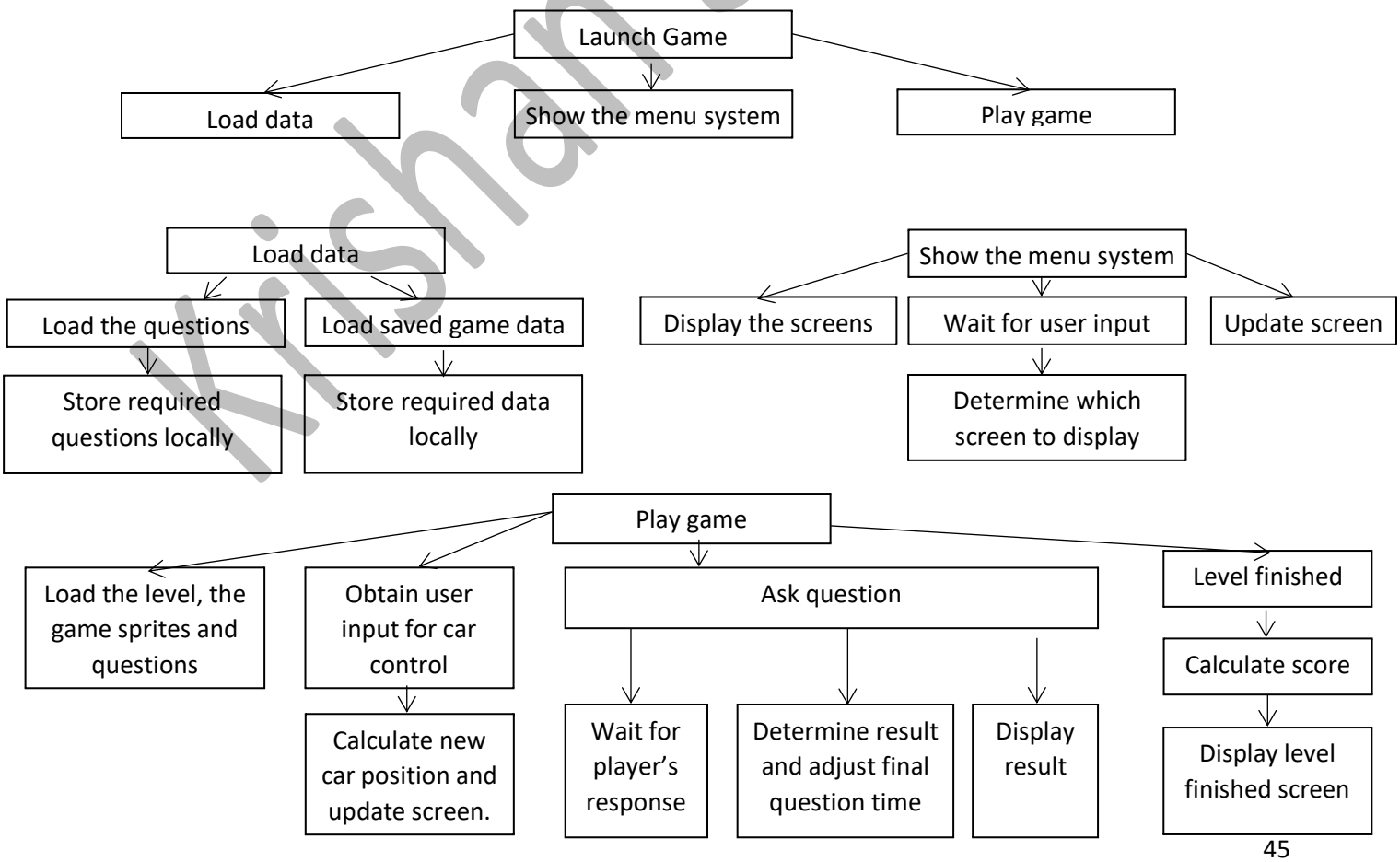
This breakdown can be used alongside the success criteria to structure this project. This breakdown has been ordered in the way that the project should be approached. This is a logical order as certain features such as the main game loop, and the question loading functionality (success criteria number 1), need to be developed before the menu system can be created and questions can be asked.

Top-Down Design

This is a top-down design flowchart showing a simple breakdown of the game. It shows the main screens and how they are linked.



This is a top-down design of the program.



Data Storage

The questions that are going to be asked to the user need to be stored in an appropriate file type so that new questions can easily be added and existing questions can be altered. They could be stored in a database however, as there would only be one table in the database, there will be no relationships in the database. A database with only one table is a flat-file database, which is essentially a spreadsheet. Therefore, I have decided to store the question in a CSV file as this way, a spreadsheet manipulation software can be used to manage the questions. The fields in the spread sheet will include: questionID, question, correct answer, wrong answer 1, wrong answer 2, wrong answer 3, wrong answer 4, difficulty, level number and isMajor question.

- QuestionID will essentially be the primary key in this flat-file database and will be used to uniquely identify each question. This value will be stored temporarily when a question is asked, and checked against when another question is selected. This check will make sure that the same question is not asked in the same level.

- The question field will have the question stored in a string format.

- All of the answers will be stored in a float format as not all questions will have integer solutions. The reason why there are four incorrect answers is so that even if the same question is asked in subsequent levels, the choice of answers will be different. Three of these incorrect answers will be randomly selected and added to the correct answer. The order of these questions will further be randomized before printing them to the screen.

- The difficulty field will identify the difficulty of the question (i.e. easy, intermediate or difficult).

- Some questions may only be asked on specific levels. For example, for the first few levels, easy questions will be asked to get the player familiar with the game and the controls. The level number field will be used to associate these questions with a particular level. If in this field a record has the value 0, the question doesn't have an associated question and so can be asked on any level. For the mini game and endless mode, the level number data will not be analysed.

- The last field will contain a Boolean value, which if True will cause that question to be asked at the end of a level. Questions with True in this field will usually be more difficult than the rest.

When reading in the CSV file, iterative functions can be used to search through the file and temporarily store all the questions with the chosen difficulty. If the question difficulty setting is changed in the settings, the CSV file will need to be re-loaded, and the new questions will then be temporarily stored. The questions that are going to be asked in a level will be selected during the loading phase of the level. This ensures that there is no unnecessary delay in the game for choosing the question. The questions can be stored in a 2-dimensional array data structure so that each piece of data can be accessed using the column number (field) and row number (record).

Data such as the player's statistics, number of levels unlocked, and coin count will be stored on a text file in the device's secondary storage when the game is closed so that no game progress is lost. These files will either need to be hidden from the user or well-protected so that people cannot cheat in the game by altering these game settings. A solution to this problem could be to encrypt the data however, this issue is not a massive concern, and so doesn't need to be solved until after all other aspects of the game are complete. All of this data will need to be loaded and locally stored (i.e. in variables in the program) when the game is launched. Similarly, when the game is closed, this data will need to be written back to the text files.

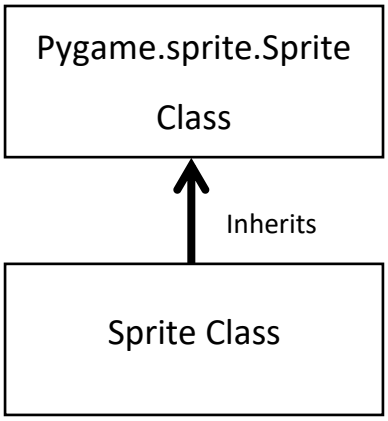
By using object oriented programming, instances of the sprites needed for the game can be created, reducing the overall storage required when compared to coding the game procedurally. Moreover inheritance can be in classes to reduce the repetition of code, making the overall code more efficient, whilst also reducing storage space required. This efficiency is a requirement when making a game as increasing the computational power required will take a toll on the FPS (frames per second) that the game is running at. A game with a slow FPS will feel 'choppy' as it will seem to be skipping frames, making the overall experience of the game not very fun. In order to avoid this, the code must be efficient and any compute intensive feature could be optional, i.e. be enabled or disabled in the game options.

Class inheritance diagram

It has now been established that I will be attempting to use object oriented programming for my project. A major component of OOP is the usage of classes and the attributes and methods they provide. Many instances of the same class can be created, for example, all the buttons required in the game can be created using one button class. This makes the code's performance and development cycle more efficient as changes only need to be made in one place to take effect in all areas.

Classes have the ability to inherit methods and attributes defined in other classes. The class being inherited is called the parent class, and the class inheriting is called the subclass. Inheritance allows previously created code to be efficiently re-used and therefore reduces the unnecessary repetition of code. There exists a simple sprite bass class for pygame's visible game objects. This base class has the functionality to add sprites into groups and control them in there. Pygame groups are very useful for sprites as it allows many sprites, usually instances of the same class, to be grouped together. This allows the same function to be called on all of the objects at the same time, and is also used for collision checks.

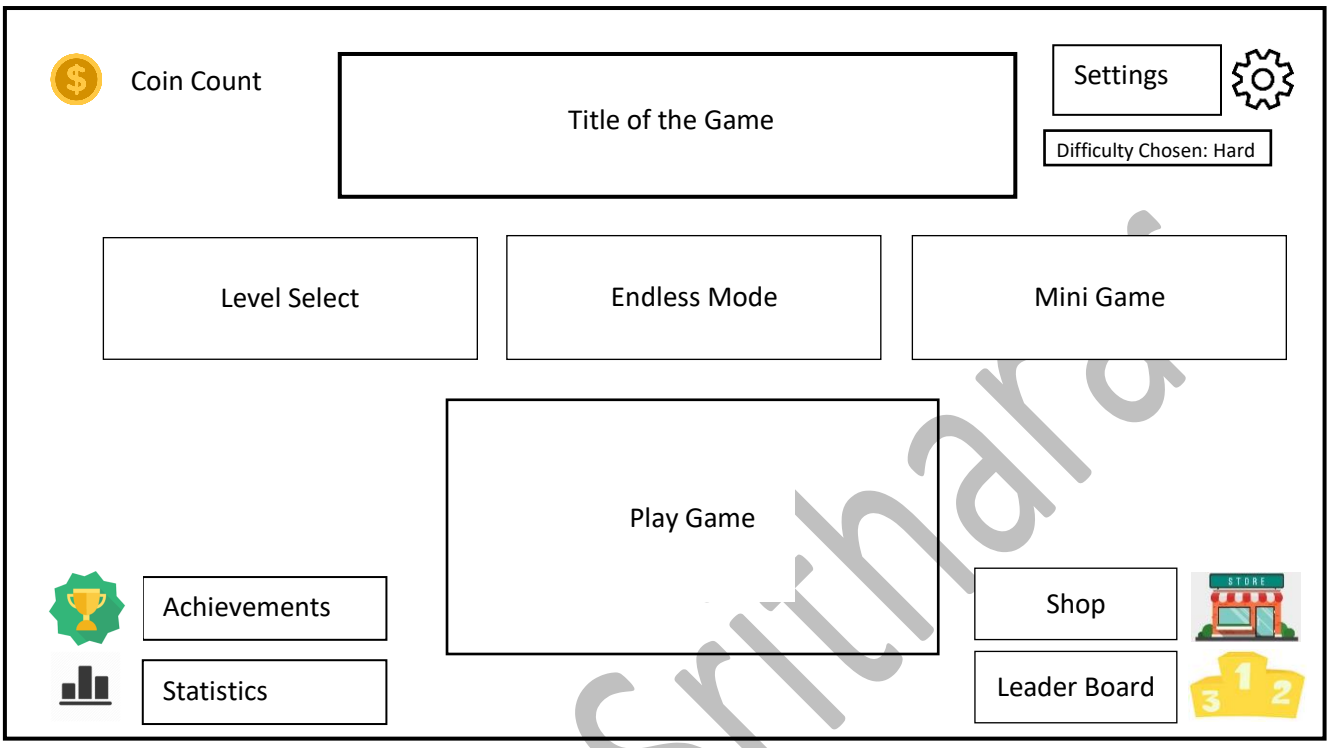
This base class is called 'pygame.sprite.Sprite'. The class has function: 'add', 'remove', 'kill', 'alive' and 'groups', which are used to: add a sprite to a group, remove a sprite from a group, remove a sprite from all groups, check if a sprite belongs to any groups and, return a list of the groups the a sprite belongs to respectively. After the class has been inherited, the 'ini' function inside this base class needs to be called providing the groups that the sprite needs to be added to as a parameter.



To the right is a class inheritance diagram of the sprite classes that will be used in my game. If any other classes in my game use inheritance, I will include a class diagram of that class when it is being explained in the development section.

User interface designs

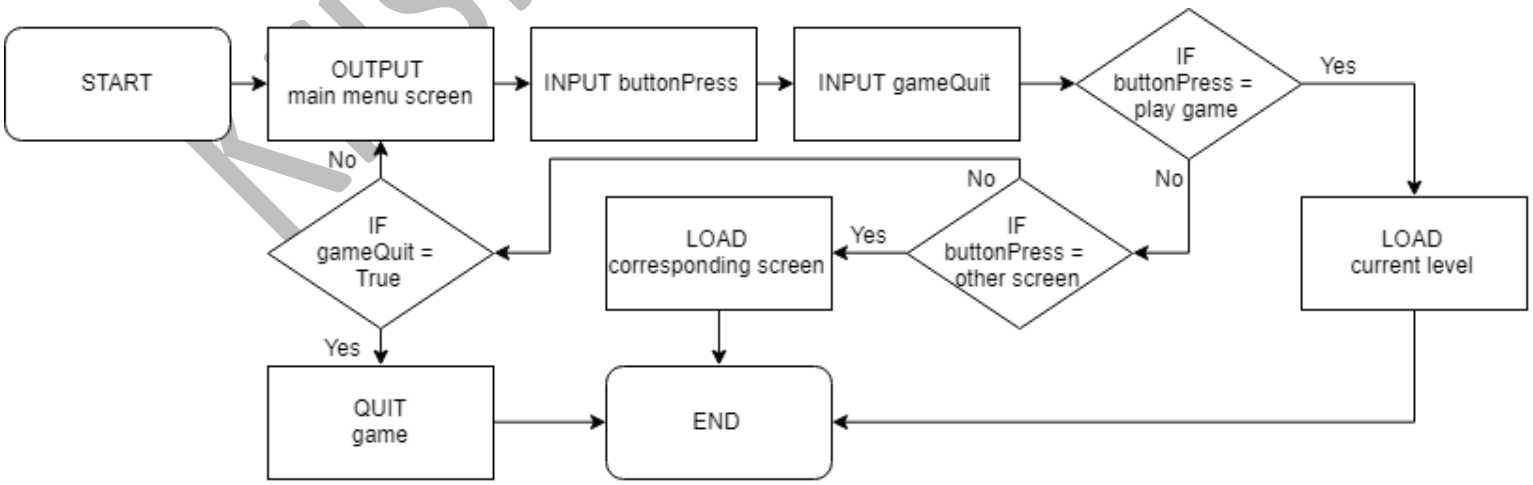
Main Menu Screen



Purpose: This is the main menu screen and is what the user is presented with upon launching the game. From this screen the user can navigate to all the other screens in the game.

Functionality: Pressing the Play Game button will resume the game from the last level the user played/unlocked. Pressing Endless Mode will start a game with the endless mode settings and selecting Mini Game will initiate a mini game. Pressing Level Select, Settings, Leader Board, Achievements and Statistics will take the user to the respective screens.

Algorithm Design -Flowchart



Algorithm Design -Pseudocode

```
function displayMainMenu() {
        OUTPUT main menu screen
        INPUT buttonPress
        INPUT gameQuit
        IF buttonPress = play game THEN
                LOAD current level
        ELSE IF buttonPress == level select OR achievements OR statistics OR shop OR leader board OR settings OR
        endless mode OR minigame THEN
                LOAD corresponding game screen
        END IF
        IF gameQuit = True THEN
                QUIT game
        END IF
endfunction
```

Key Variables

buttonPress: A variable which store the clicked state of the buttons. Each button will have this variable and checks will be made in each frame to see if it has been clicked. A class will be implemented to create the buttons, allowing a class variable called 'buttonPress' to be created for each instance of the class. A Boolean value of false will be initially stored in this variable and the state of this variable will be change to True when the button is clicked. A series of selection statements will then determine which screen to navigate to depending on which button has been pressed.

gameQuit: A variable which is used to check if the user has clicked the red cross in the top-right corner of the game window to close the game. A check for this variable will be made every frame and when this happens, a function will be called to safely close the game. This function will include saving the games progress, highscore, etc.

play game: This refers to the play game button. It will be checked whether this buttons has been clicked by using the tag of the clicked buttons and matching them against the tag for this button.

current level: This is a variable containing the highest level that is unlocked. The value of this variable will be passed in as a parameter to a function which loads a level.

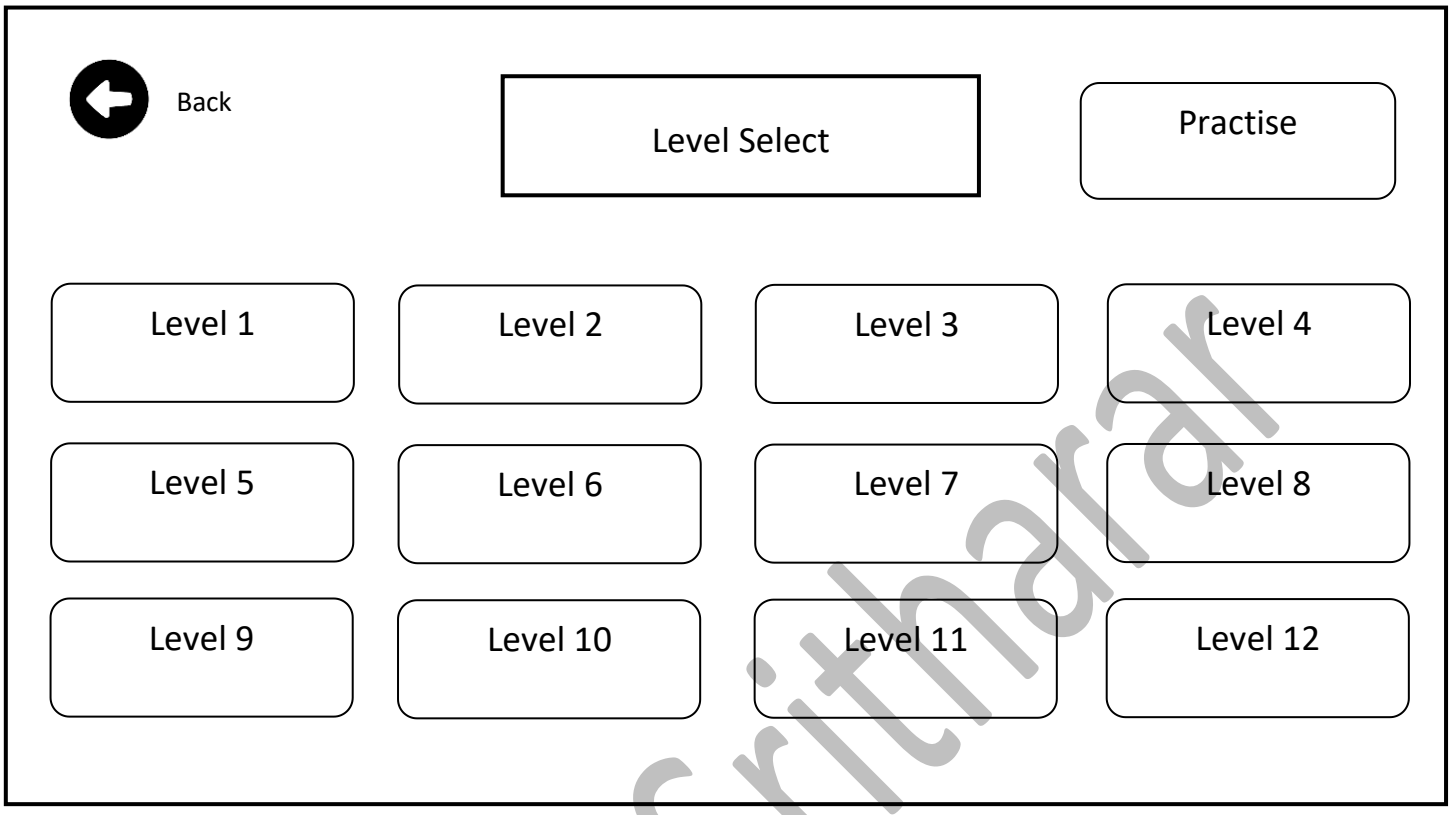
game: This is an instance of the game class, which will be the main class to run the game. When the QUIT command is executed on this instance of the game, it will be de-initialised, causing the program to end.

The names: level select, achievements, statistics, shop, leader board, settings, endless mode and mini game refer to the different screens and modes in the game. These will be the tags of the buttons that are on the main menu screen.

Test Data

Test Number	Test Description	Test Data	Test Type	Expected Outcome
1	Check the functioning of the Play Game button	Click on the Play Game button	Valid	Program launches the last played/unlocked level
2	Check the navigation of the Level select button	Click on the Level Select button	Valid	Game loads the Level Select screen
3	Check the functionality of the Mini Game button	Click on the Mini Game button	Valid	Game launches the mini game
4	Check the navigation of the Endless Mode button	Click on the Endless Mode button	Valid	Game launches the endless mode
5	Check the functioning of the Achievements button	Click on the Achievements button	Valid	Program loads the achievements page
6	Check if the Shop button loads the shop screen	Click on Shop button	Valid	Program loads the shop page
7	Check the functionality of the Statistics button	Click on the Statistics button	Valid	Game opens the statistics page
8	Check the navigation of the Leader Board button	Click on the Leader Board button	Valid	Program opens the leader board page
9	Check if the Settings button loads the settings page	Click on the Settings page	Valid	Program loads the settings page
10	Check that the difficulty shown on this screen changes when this setting is changed	Change the difficulty to another level in the settings	Valid	The difficulty shown will change according to the chosen level.
11	Test if the Coin Total value updates correctly	1. Spend some coins in the Shop 2. Earn some coins by playing a level or the mini game	Valid	1.The value of the coins should decrease 2.The coin value should increase
12	Test if the buttons are highlighted when the cursor hovers over them	Hover over all of the buttons	Valid	The buttons should change colour to identify that they are highlighted
13	Test if the buttons work when they are clicked at their edges	Click on the buttons at their edges	Valid Extreme	Game correctly executing the function associated with the buttons
14	Make sure that clicking anywhere which isn't a button on the screen doesn't open another page	Click on region on the screen which doesn't have any buttons	Invalid	Nothing should happen. No screens should be loaded.

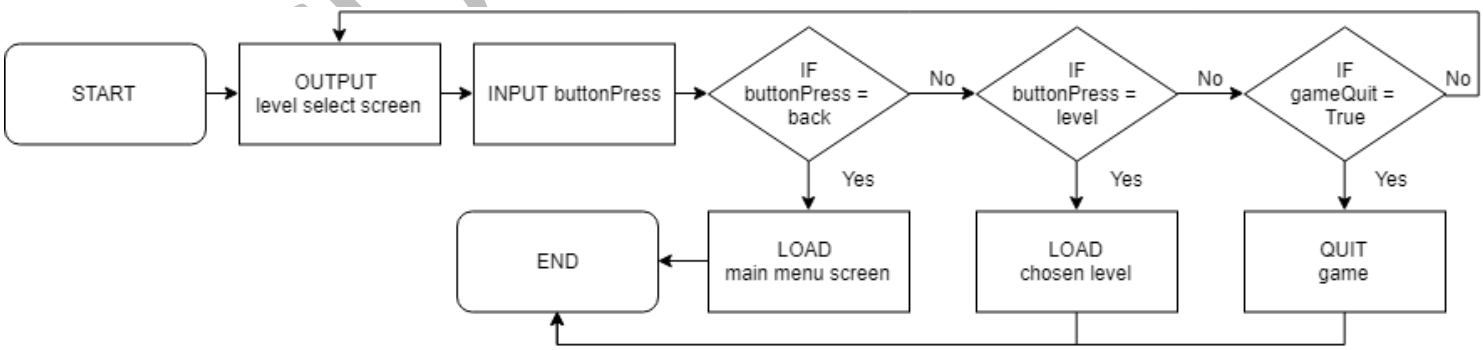
Level Select Screen



Purpose: This is the level select screen and is where the player can select which level they would like to play. All the level can be accessed from this screen as well as the practise level. There are 12 levels to pick from and levels only become unlocked once the player has completed the previous level. Locked levels will be clearly identified by being shaded out or having icons on them.

Functionality: Pressing the level buttons will launch each respective level and the practise button will launch the practise level, where the player can get familiar with the game controls. Pressing the back button will navigate the user back to the main menu screen.

Algorithm Design -Flowchart



Algorithm Design -Pseudocode

```
function displayLevelSelect() {
        OUTPUT level select screen
        INPUT buttonPress
        INPUT gameQuit
        IF buttonPress == back THEN
                LOAD main menu screen
        ELSE IF buttonPress == level THEN
                LOAD chosen level
        END IF
        IF gameQuit = True THEN
                QUIT game
        END IF
endfunction
```

Key Variables

buttonPress: As explained previously, this is a variable which store the clicked state of the buttons.

gameQuit: As explained previously, this is a variable which is used to check if the user has clicked the red cross in the top-right corner of the game window to close the game.

back: This is a variable storing the tag of the back button. This variable is used to check whether the back button has been pressed.

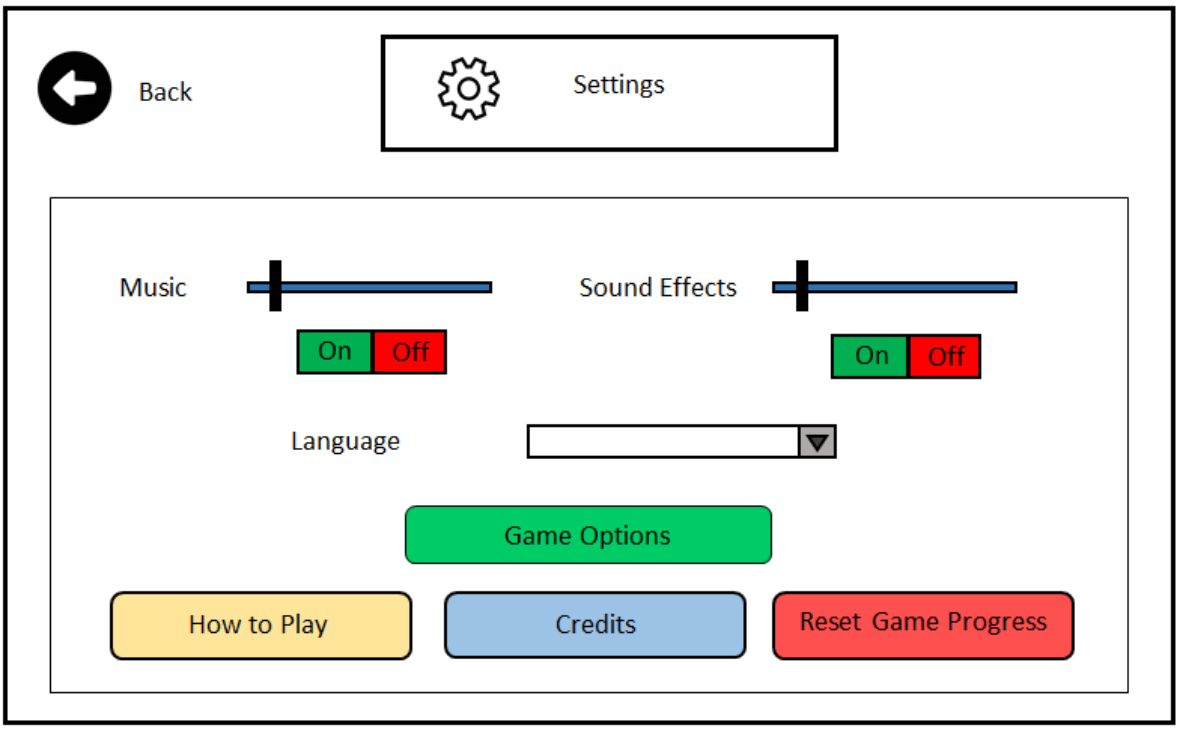
level: This relates to any of the level buttons on the level select screen. When any of these buttons are pressed, their tags will be analysed and the appropriate level will be loaded.

game: As explained previously, this is an instance of the game class.

Test Data

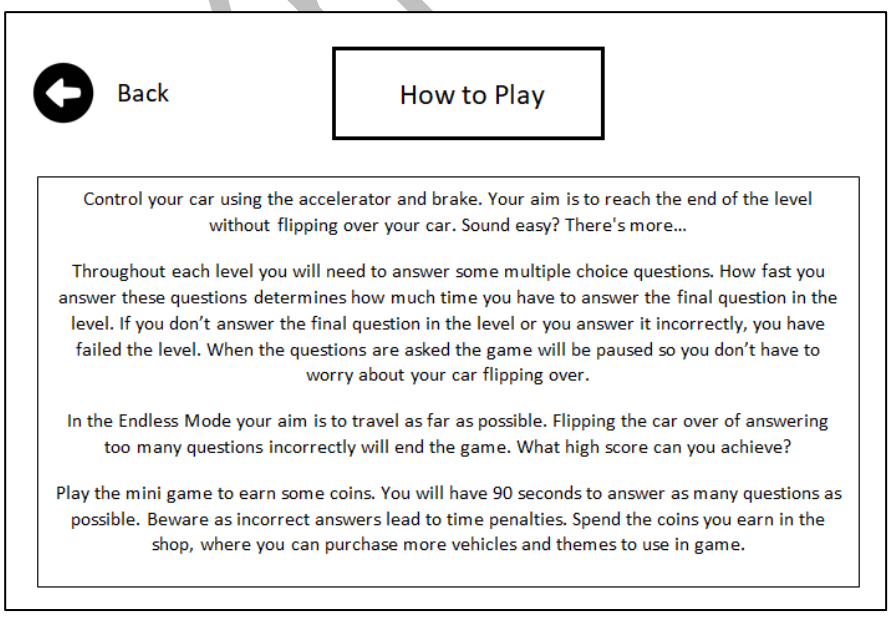
Test Number	Test Description	Test Data	Test Type	Expected Outcome
15	Check the functioning of the Back button	Click on the Back button	Valid	Game launches the main menu screen
16	Check the functionality of the Practise button	Click on the Practise button	Valid	Game loads the Level Select screen
17	Check the functionality of an unlocked level	Click on the Level 1 button	Valid	Game loads and launches level 1
18	Check the functioning of a locked level	Click on a locked level button such as the level 12 button	Valid	Game doesn't launch the level
19	Test if the buttons work when they are clicked at their edges	Click on the buttons at their edges	Valid Extreme	Game correctly performs the function associated with the buttons
20	Make sure that clicking anywhere which isn't a button on the screen doesn't open another page	Click on region on the screen which doesn't have any buttons	Invalid	Nothing should happen. No screens should be loaded.

Settings and Game Options



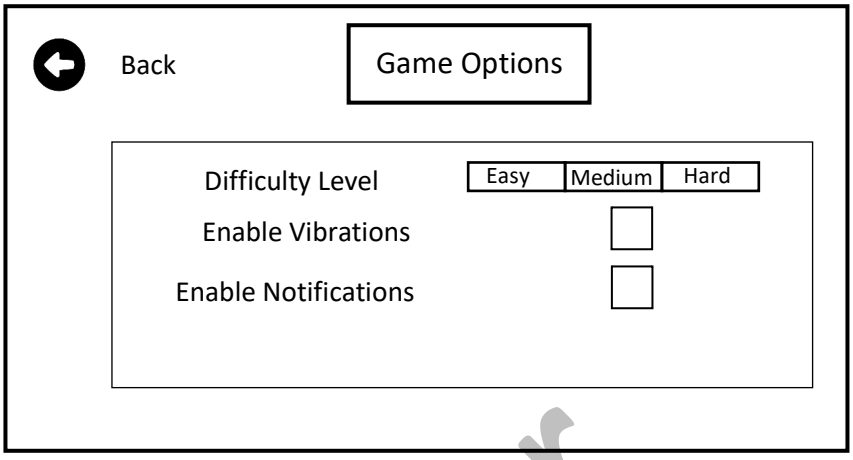
Purpose: This is the settings screen and is where the player can change the game's settings. From this screen, the game options and the how to play screens can be accessed. Sliding bars are used to change to volume of the game music and game sound effects. These provide a wider variety of volume choices whilst also being aesthetically better than increase and decrease buttons. Furthermore the on and off buttons are coloured so that their state can be easily identified. For example the user will know that the game music is off if the off button is red. The different languages is an optionally feature and will only be added after the rest of the game is complete. The languages can be selected from a drop down menu, allowing the user to see which options are available when picking.

Functionality: On this screen the functions possible include changing the volume of as well as enabling and disabling game music and game sound-effects, changing the language, viewing the instructions on how to play the game, viewing the credits and, resetting the progress of the game. The back button will take the player to main menu screen and the game options, how to play, credits and reset game progress buttons will take the user to their respective screens.



Purpose: Here the player can read through the instructions to learn their aims whilst playing the game. It also teaches them about how they can spend the coins that they collect to advance their fleet of vehicles.

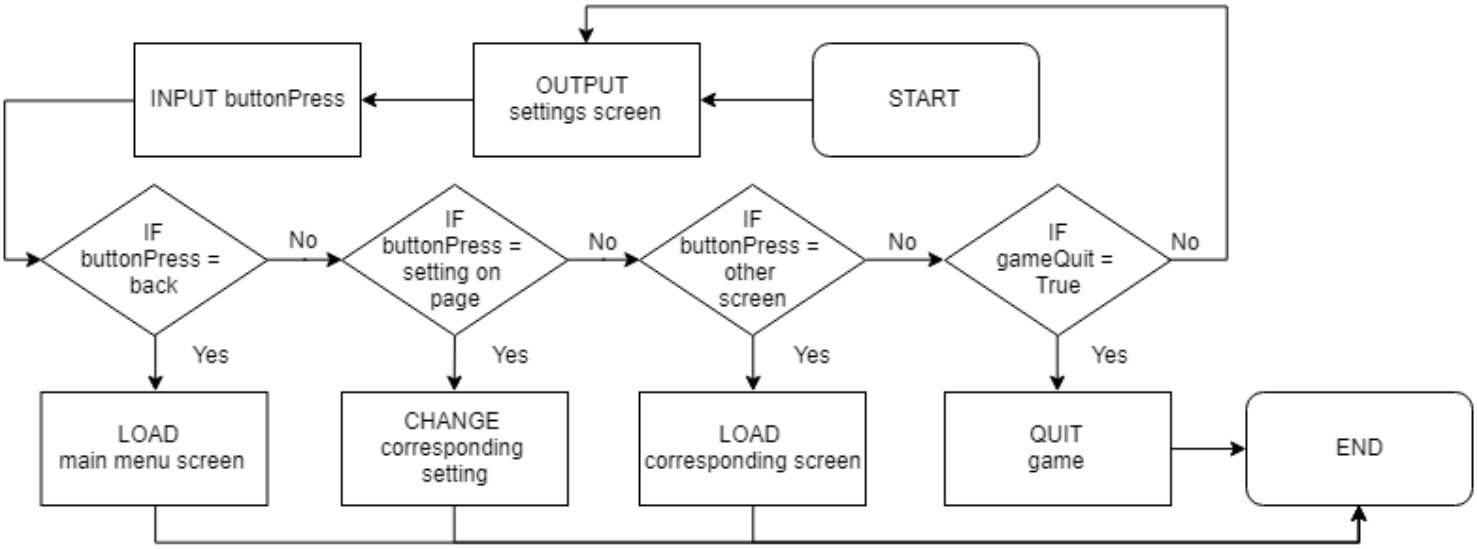
Functionality: The back button will take the user back to the settings page, from which they can go back again to the main menu page.



Purpose: Here the user can change further settings specific to the gameplay such as the difficulty of the questions being asked. Check box style indicators are used to let the user easily identify which options are currently activated. Clicking on the text indicating the current difficulty level on the main menu page will take the user to this screen.

Functionality: The difficulty level can be selected by choosing either the easy, medium or hard buttons. The other options can be activated and deactivated by clicking on the check box. The back button will navigate the user back to the settings screen, from which they can go to the main menu screen.

Algorithm Design - Flowchart



Algorithm Design - Pseudocode

```
function displaySettings() {
        OUTPUT settings screen
        INPUT buttonPress
        INPUT gameQuit
        IF buttonPress == back THEN
                LOAD main menu screen
        ELSE IF buttonPress == setting on page THEN
                CHANGE corresponding setting
        ELSE IF buttonPress == other screen THEN
                LOAD corresponding screen
        END IF
        IF gameQuit = True THEN
                QUIT game
        END IF
endfunction
```

Key Variables

buttonPress: As explained previously, this is a variable which store the clicked state of the buttons.

gameQuit: As explained previously, this is a variable which is used to check if the user has clicked the red cross in the top-right corner of the game window to close the game.

back: As explained previously, this is a variable storing the tag of the back button.

setting on page: This relates to any of the settings on the page. If any of these settings are changed, then the appropriate function is called to execute the change.

other screen: This is referring to any of the buttons on the page that link to other pages. This includes the How to Play button, the Credits button, the Game Options button and the Reset Game Progress button. When any of these are clicked, their respective screens are opened.

game: As explained previously, this is an instance of the game class.

Test Data

Test Number	Test Description	Test Data	Test Type	Expected Outcome
21	Check the functioning of the Back button	Click on the Back button	Valid	Game launches the main menu screen
22	Check that the Music slider is correctly functioning	Scroll the Music slider to different positions	Valid	Game changes the game volume accordingly
23	Check whether the Sound Effects slider is correctly functioning	Slide the Sound Effects slider to different positions	Valid	Game changes the Sound Effects volume respectively
24	Check the On Off buttons for the game music correctly functions	Click on the Off button for the game Music	Valid	Game stops playing music
25	Check whether the On Off buttons for the Sound Effects are correctly functioning	Click on the Off button for the game Sound Effects	Valid	Program stops playing the sound effects
26	Check is the selecting languages option is correctly working	Click on the languages box	Valid	Program opens a drop down box showing the available languages
27	Check the functionality of the Game Options button	Click on the Game Options button	Valid	Game opens the game options page
28	Check the navigation of the How to Play button	Click on the How to Play button	Valid	Program opens the how to play page
29	Check the functioning of the Credits button	Click on the Credits button	Valid	Program opens the credits page
30	Check if the Reset Game Progress button resets the game's progress	Click on the Reset Game Progress button	Valid	Game resets the game's progress
31	Test if the buttons work when they are clicked at their edges	Click on the buttons at their edges	Valid Extreme	Game correctly executing the function associated with the buttons
32	Make sure that clicking anywhere which isn't a button on the screen doesn't open another page	Click on region on the screen which doesn't have any buttons	Invalid	Nothing should happen. No screens should be loaded.

Name: Krishan Sritharar Candidate Number: 8422 Centre Number: 12304

Shop Screen

Back

Shop

Coin Count

Vehicles	Cost	Themes	Cost
Simple Car	40 Coins	Orange and Blue	100 Coins
Simple Bike	50 Coins		
Racing Car	70 Coins	Red and Orange	150 Coins
Racing Bike	80 Coins		
Moped	90 Coins	Blue and Green	200 Coins
Scooter	100 Coins		
Jeep	120 Coins	Orange and Black	250 Coins

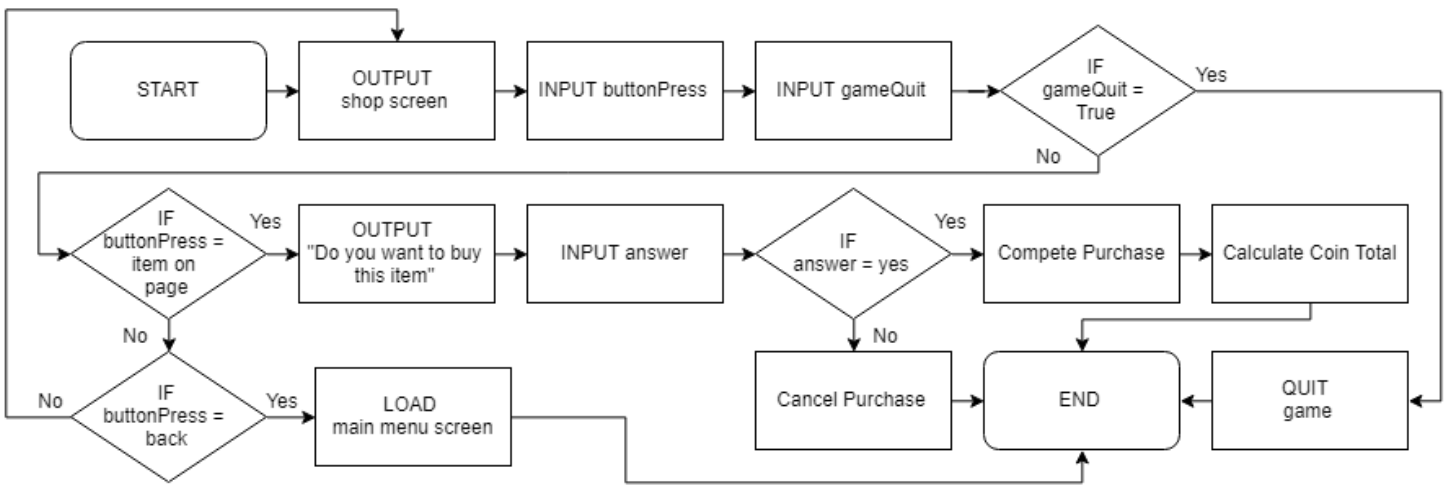
Power Ups	Current	Cost
Skip a Question	5	10 Coins
Bonus Time	3	15 Coins
Freeze Time	7	20 Coins
Second Life	14	15 Coins
Coin Magnet	2	30 Coins
Double Coins	4	35 Coins

Earn coins by playing more levels and answering the questions at the start correctly.

Purpose: This is the shop screen and is where the player can purchase items using the coins that they have earned while playing the various game modes. The current coin total is displayed in the top-right corner of the screen and this will be real time. Therefore, after the user has bought an item, the coin total will decrease. The items that the user has already bought will be clearly identified and the user will not be able to buy already purchased items. The current column in the power ups section of the page shows how many of that power up the use currently has in their inventory. More vehicles will be added to this list and the player will be able to scroll down on that section of the page to see the other vehicles.

Functionality: They player can purchase an item by pressing the on the item's name or its cost. This will open a pop-up box asking the user to confirm that they want to buy this item, where the user can click on yes to buy the item or no to cancel the purchase. Pressing on the back button will take the user back to the main menu screen.

Algorithm Design - Flowchart



Algorithm Design - Pseudocode

```
function displayShop() {
        OUTPUT shop screen
        INPUT buttonPress
        INPUT gameQuit
        IF gameQuit = True THEN
                QUIT game
        END IF
        IF buttonPress == item on page THEN
                OUTPUT "Do you want to buy this item"
                INPUT answer
                IF answer == yes THEN
                        COMPLETE purchase
                        CALCULATE coin total
                ELSE THEN
                        CANCEL purchase
                END IF
        ELSE IF buttonPress == back THEN
                LOAD main menu screen
        END IF
endfunction
```

Key Variables

buttonPress: As explained previously, this is a variable which store the clicked state of the buttons.

gameQuit: As explained previously, this is a variable which is used to check if the user has clicked the red cross in the top-right corner of the game window to close the game.

game: As explained previously, this is an instance of the game class.

item on page: This relates to any of the items on the shop page. If any of these items are clicked, further functions will be called to start the process of buying the item.

answer: This is a variable which stores the clicked state of the yes or no buttons. Validation is not required in this situation as only these two buttons will be available so any other input is not possible.

purchase: This relates to the item being bought and calls subsequent functions to determine the price of the item, add the item into the player's inventory and change the state of the item to purchased. The price of the item is deducted from the player's coin total in the following command.

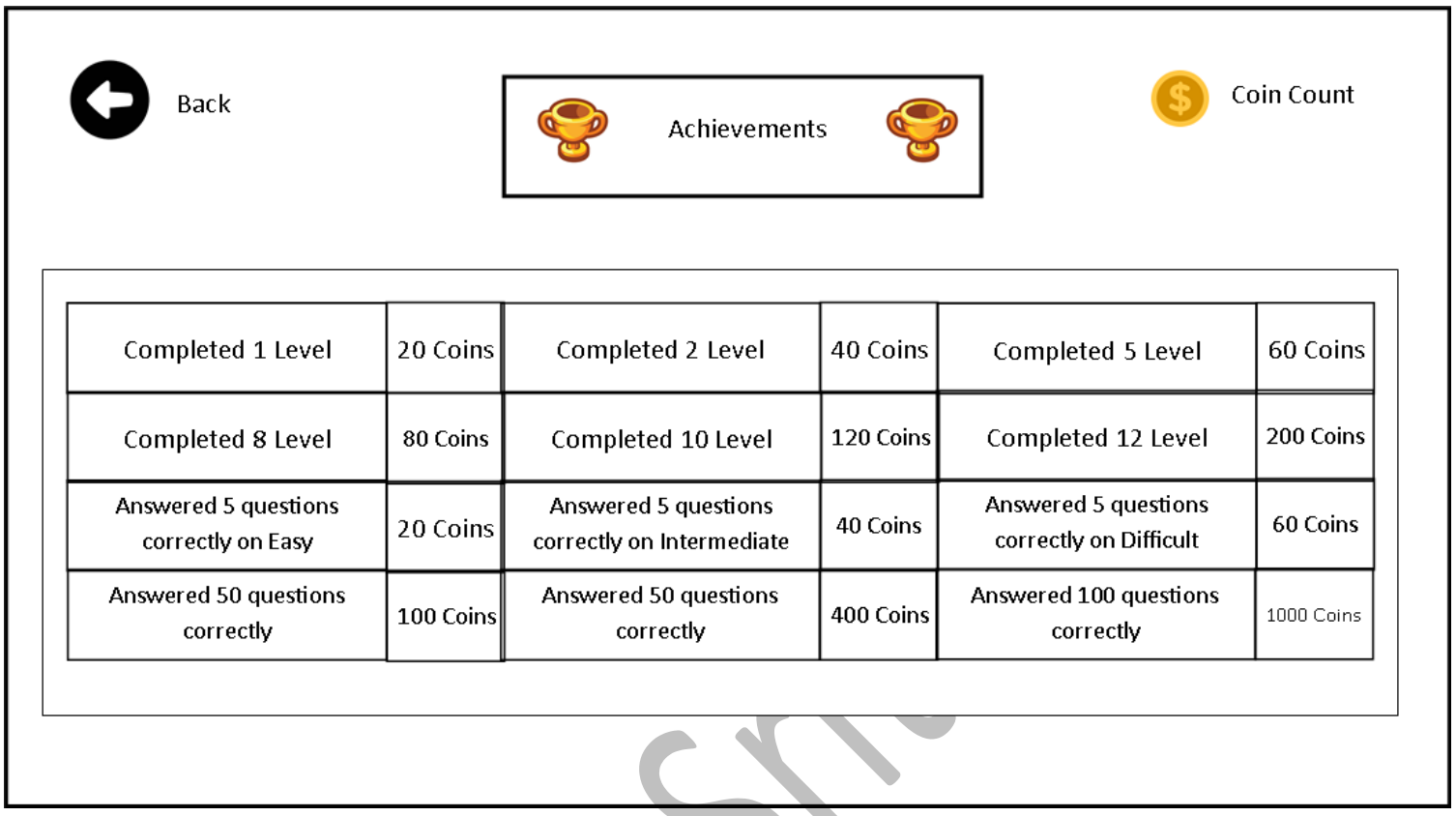
coin total: This is a variable storing the value of the amount of coins that the player has.

back: As explained previously, this is a variable storing the tag of the back button.

Test Data

Test Number	Test Description	Test Data	Test Type	Expected Outcome
33	Check the functioning of the Back button	Click on the Back button	Valid	Game launches the main menu screen
34	Check that the vehicles can be correctly purchased	Click on an affordable (i.e. there is enough coins to buy) vehicle	Valid	Game asks the user to confirm their purchase
35	Check the functioning of the yes button when purchasing an item	Click on the yes button when purchasing an item	Valid	Game completes the purchase of the item
36	Checking whether the no button works when buying an item	Click on the no button when purchasing a game	Valid	Game cancels the purchase of the item
37	Check that the themes can be correctly purchased	Click on an affordable theme	Valid	Game asks the user to confirm their purchase
38	Check that the power ups can be correctly purchased	Click on an affordable power up	Valid	Game asks the user to confirm their purchase
39	Check that the Coin Total correctly updates when an item is purchased	Click on an affordable item	Valid	The value of the coin total correctly decreases depending on the cost of the item purchased
40	Test if the buttons work when they are clicked at their edges	Click on the buttons at their edges	Valid Extreme	Game correctly executing the function associated with the buttons
41	Check that an item is not bought when there is not enough coins	Click on an item that is not affordable	Invalid	Game informs that user that they do not have enough coins for this item
42	Check that an already purchased item cannot be bought again	Attempt to purchase a purchased item	Invalid	Nothing should happen
43	Make sure that clicking anywhere which isn't a button on the screen doesn't open another page	Click on region on the screen which doesn't have any buttons	Invalid	Nothing should happen. No screens should be loaded.

Achievements Screen

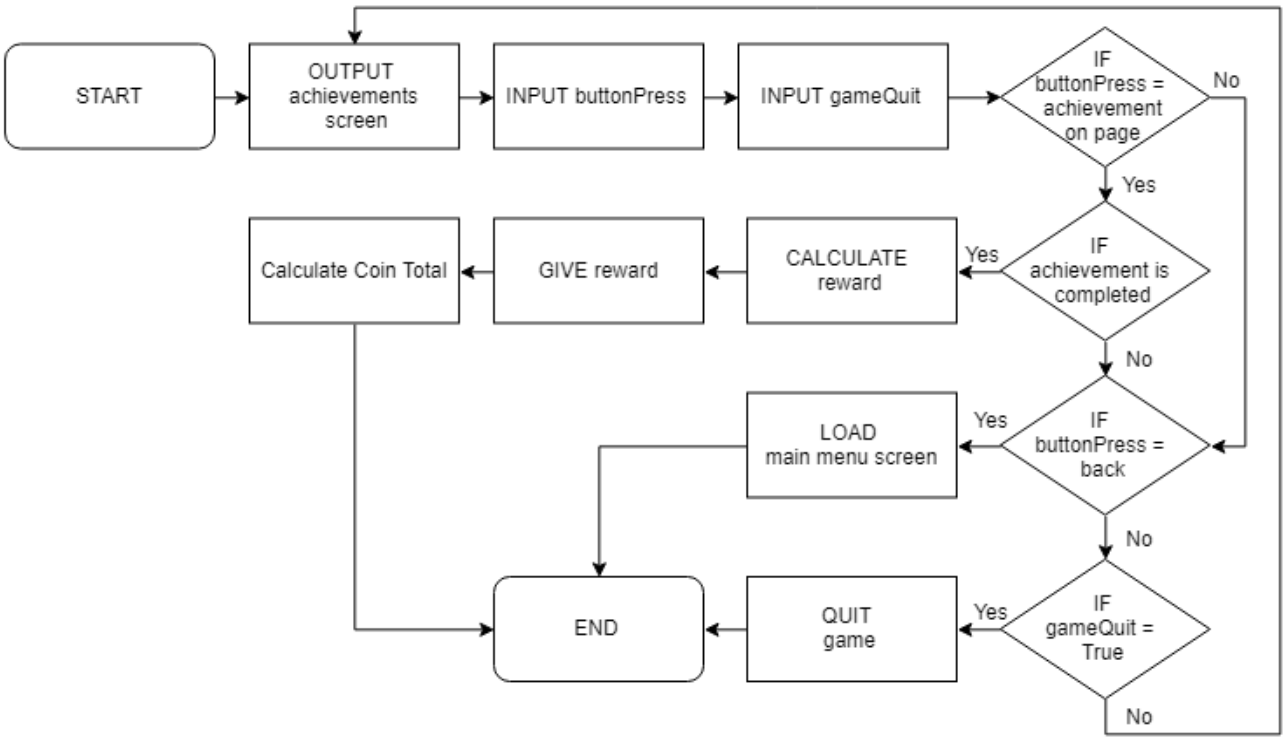


Completed 1 Level	20 Coins	Completed 2 Level	40 Coins	Completed 5 Level	60 Coins
Completed 8 Level	80 Coins	Completed 10 Level	120 Coins	Completed 12 Level	200 Coins
Answered 5 questions correctly on Easy	20 Coins	Answered 5 questions correctly on Intermediate	40 Coins	Answered 5 questions correctly on Difficult	60 Coins
Answered 50 questions correctly	100 Coins	Answered 50 questions correctly	400 Coins	Answered 100 questions correctly	1000 Coins

Purpose: This is the achievements screen and is where the player can collect the rewards for completing certain tasks or passing specific milestones in the game. This is where the player can also see what achievements that they can unlock, and how far they are in the process of completing a particular achievement. Achievements which take longer to complete are more rewarding. More pages of achievements will be added as the game gets more complicated. The total amount of coins will be updated as the user collects their reward. Similar to the leader board and statistics screens, this achievements screen is also intended to motivate the player to keep playing the game and keep improving their mental calculation speeds.

Functionality: They player can collect the coin reward by clicking on the completed achievement. Completed achievements will be indicated by having a green colour and locked achievements will be indicated by being shaded out. Pressing the back button will take the player to the main menu screen. When more achievements are added the player will be able to scroll down through the achievements to view all of them.

Algorithm Design - Flowchart



Algorithm Design - Pseudocode

```
function displayAchievements() {
        OUTPUT achievements screen
        INPUT buttonPress
        INPUT gameQuit
        IF buttonPress == achievements on page THEN
                IF achievement is completed THEN
                        CALCULATE reward
                        GIVE reward
                        CALCULATE coin total
                END IF
        ELSE IF buttonPress == back THEN
                LOAD main menu screen
        END IF
        IF gameQuit = True THEN
                QUIT game
        END IF
endfunction
```

Key Variables

buttonPress: As explained previously, this is a variable which store the clicked state of the buttons.

gameQuit: As explained previously, this is a variable which is used to check if the user has clicked the red cross in the top-right corner of the game window to close the game.

achievements on page: This relates to any of the achievements on this page. If any of these achievements are clicked, further functions will be called to start the collecting the reward for the achievement.

reward: This relates to the coin reward that is given for completing a specific achievement. All of the achievements will be stored in a file alongside the coin reward that is given for completing them. When the CALCULATE reward command is called, this file will be searched for the value of the reward to be given.

coin total: As explained previously, this is a variable storing the value of the amount of coins that the player has.

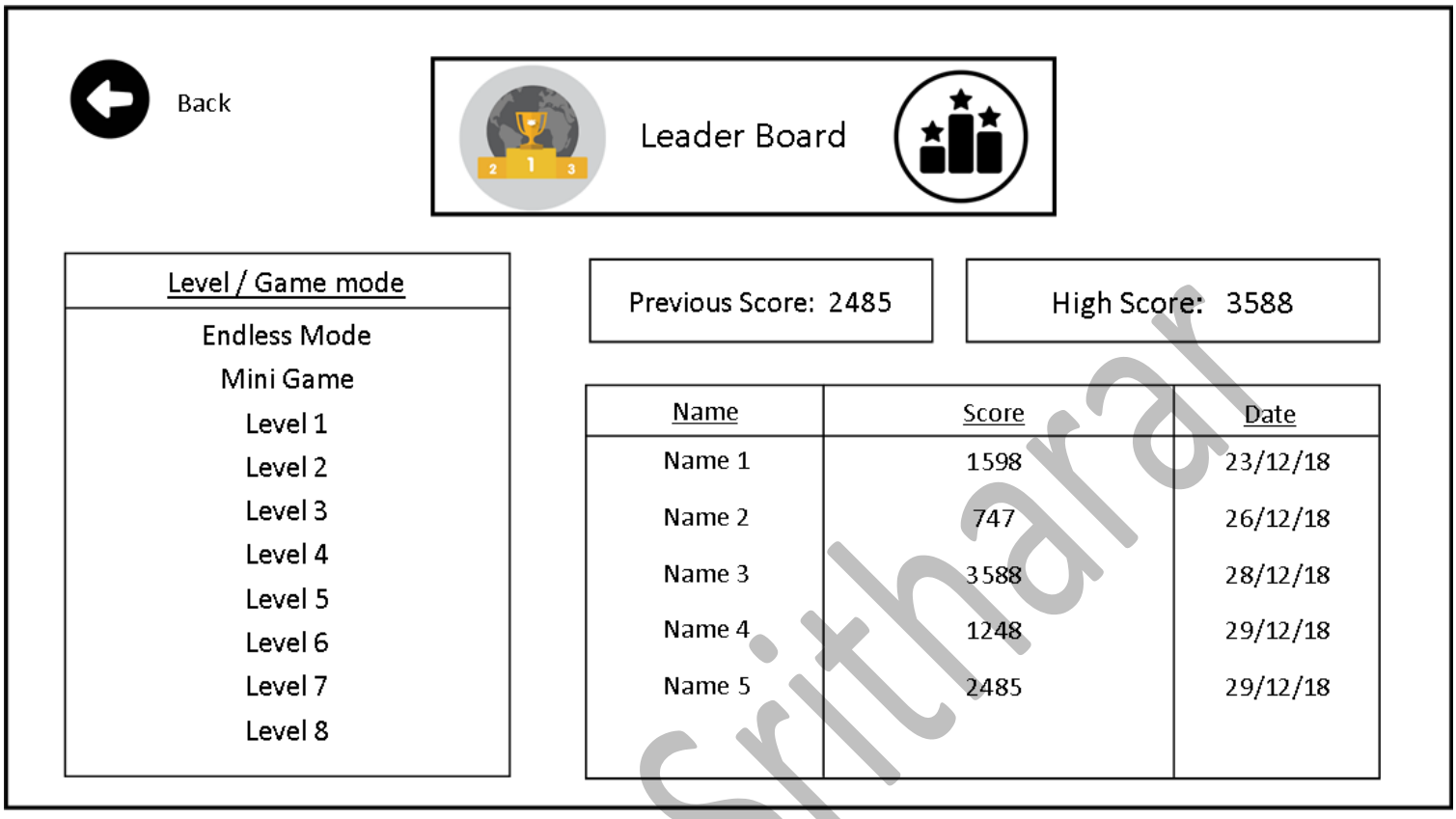
back: As explained previously, this is a variable storing the tag of the back button.

game: As explained previously, this is an instance of the game class.

Test Data

Test Number	Test Description	Test Data	Test Type	Expected Outcome
44	Check the functioning of the Back button	Click on the Back button	Valid	Game launches the main menu screen
45	Check that coins are correctly rewarded when an unlocked achievement is collected	Collect the reward of an unlocked achievement	Valid	Game correctly rewards the player the correct amount of coins
46	Check that the coin total is correctly updated when a reward is collected	Collect the reward of an unlocked achievement	Valid	Game correctly updates and increases the coin total
47	Test if the buttons work when they are clicked at their edges	Click on the buttons at their edges	Valid Extreme	Game correctly executing the function associated with the buttons
48	Check that coins are not rewarded when a locked achievement is pressed	Click on a locked achievement	Invalid	Game will not give the reward. A sound effect may play to indicate that the achievement is still locked
49	Make sure that clicking anywhere which isn't a button on the screen doesn't open another page	Click on region on the screen which doesn't have any buttons	Invalid	Nothing should happen. No screens should be loaded.

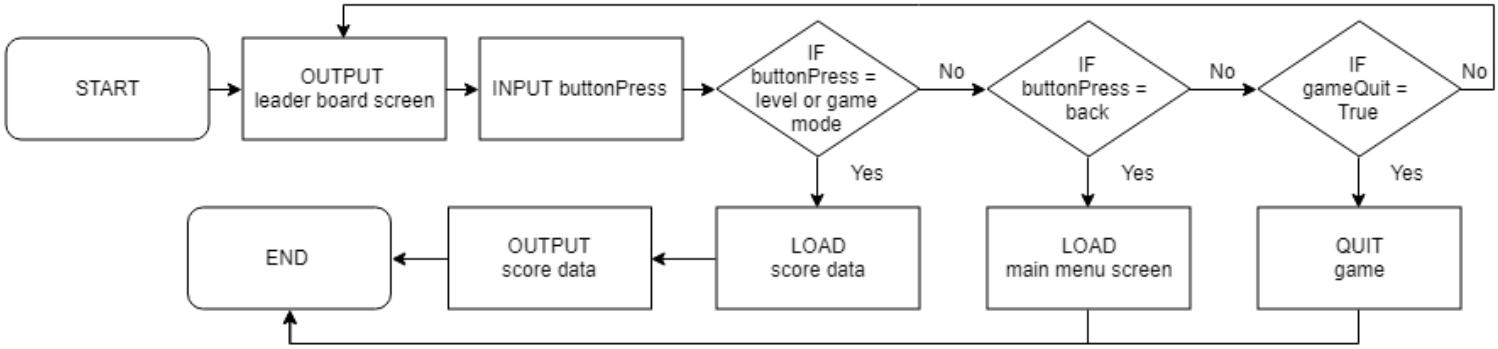
Leader Board Screen



Purpose: This is the leader board screen and is where the player can see the scores that they have achieved in the levels that they have played. They will be able to see the highest scores that have been achieved on each level alongside which player (identified by name) has obtained this score. The score achieved the last time that level was played as well as the highest score achieved separately. Similar to achievements and statistics screens, this leader board screen is also intended to motivate the player to keep playing the game and keep improving their mental calculation speeds, thus achieving the aim of this project.

Functionality: The player can switch to other levels and game modes by using the list on the left side of the page and selecting the desired option. They can the view all the scores achieved on this level/game mode on the table alongside the date that they were accomplished and the name of the person who completed the level/game mode. If they are lots of entries in this table then the player will be able to scroll the table to see the remaining scores. Pressing the back button will take the player to the main menu screen.

Algorithm Design - Flowchart



Algorithm Design - Pseudocode

```
function displayLeaderBoard() {
        OUTPUT leader board screen
        INPUT buttonPress
        INPUT gameQuit
        IF buttonPress == level or game mode THEN
                LOAD score data
                OUTPUT score data
        ELSE IF buttonPress == back THEN
                LOAD main menu screen
        END IF
        IF gameQuit = True THEN
                QUIT game
        END IF
endfunction
```

Key Variables

buttonPress: As explained previously, this is a variable which store the clicked state of the buttons.

gameQuit: As explained previously, this is a variable which is used to check if the user has clicked the red cross in the top-right corner of the game window to close the game.

level or game mode: This refers to the levels and the game modes on the left side of the page which the player can select.

score data: This is the data about the scores that are stored on a file to be loaded and read when this page is launched. This data includes the name of the player who set the score, the score itself and the data when the score was achieved.

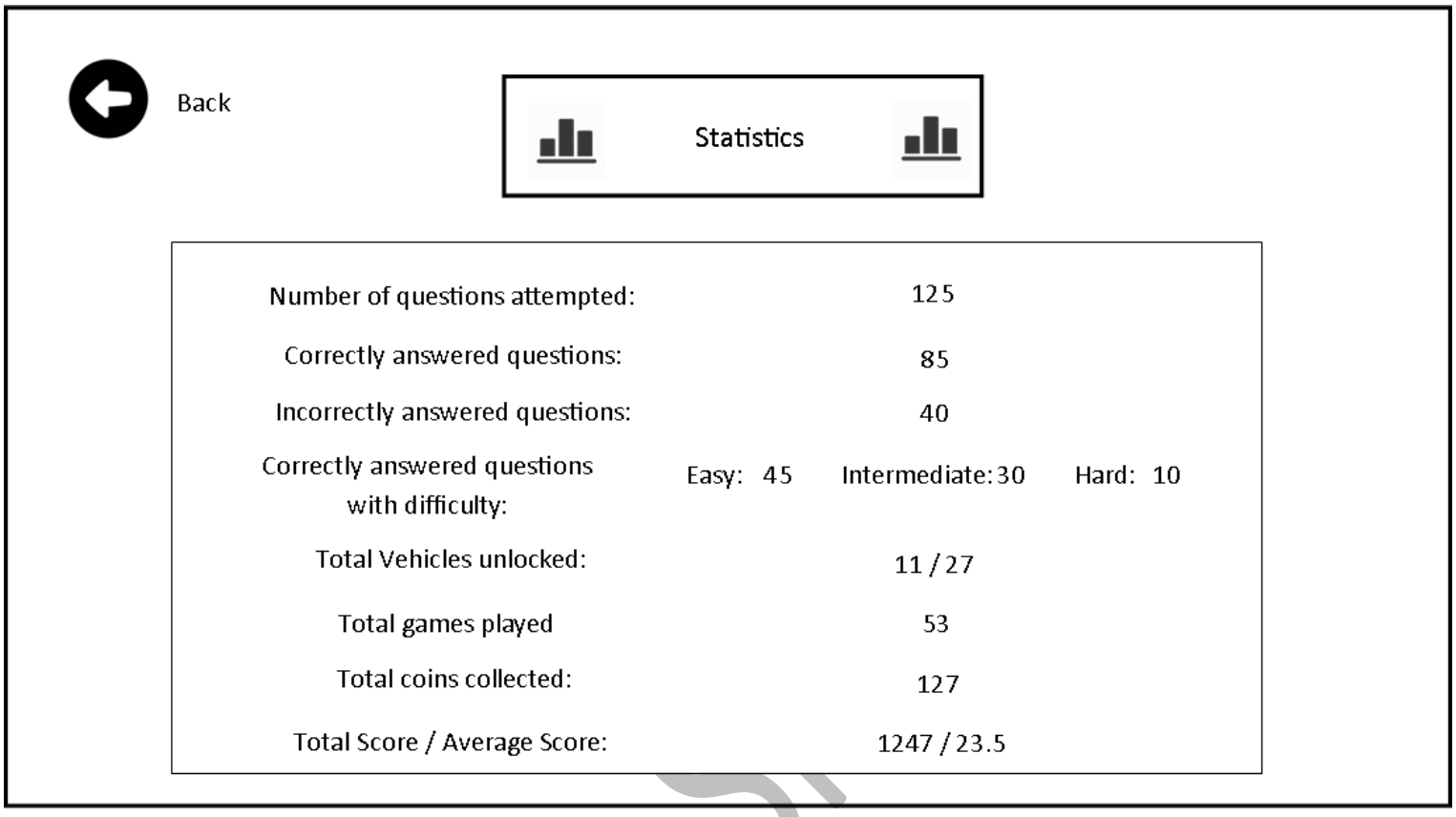
back: As explained previously, this is a variable storing the tag of the back button.

game: As explained previously, this is an instance of the game class.

Test Data

Test Number	Test Description	Test Data	Test Type	Expected Outcome
50	Check the functioning of the Back button	Click on the Back button	Valid	Game launches the main menu screen
51	Check that the scores correctly update	Play a level and record the score achieved. Then check on this page to see if the recently score is shown	Valid	Game correctly records the score achieved
52	Check that the scores correctly change to a different levels' when another level is chosen	Chose another level on the left side of the page	Valid	Game correctly changes the scores to those of the chosen level
53	Test if the buttons work when they are clicked at their edges	Click on the buttons at their edges	Valid Extreme	Game correctly executing the function associated with the buttons
54	Make sure that clicking anywhere which isn't a button on the screen doesn't open another page	Click on region on the screen which doesn't have any buttons	Invalid	Nothing should happen. No screens should be loaded.

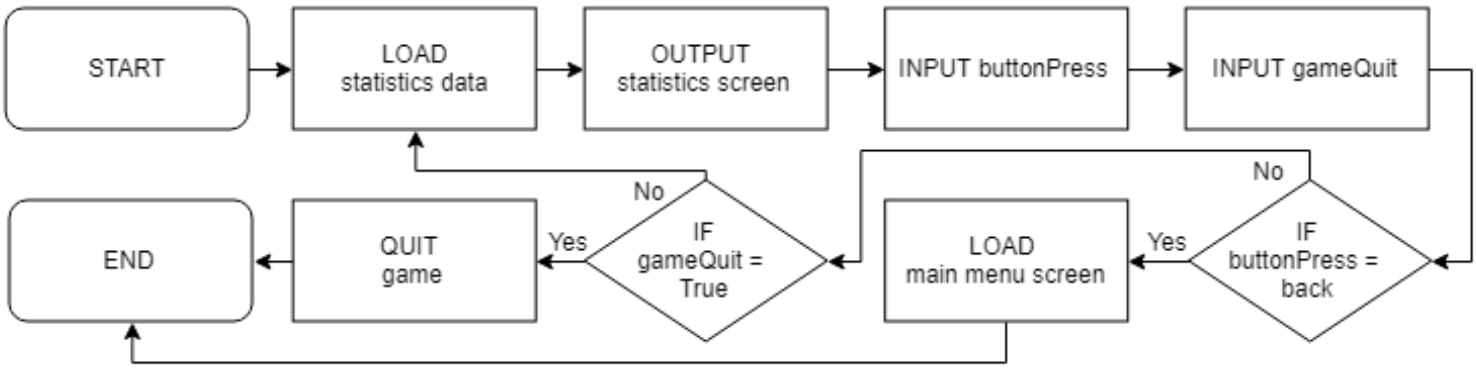
Statistics Screen



Purpose: This is the statistics screen and is where the player can view the facts and figures about their progress in this game. They will be able to see various statistics including the total number of question answered, the total number of these questions which were answered correctly and the correctly answered questions for each difficulty. They can also use this to analyse how much of the game they have completed as it shows them how many of the vehicles have been unlocked. Moreover it contains interesting information which the player can use to compare their skill in this game to other players. This includes the total number of games played, the total number of cons collected as well as the total and average score.. Similar to the leader board screen, the aim of this screen is to motivate the player to improve their statistics in the game if they are behind another player or, keep their statistics high if they are already ahead of another player. These statistics will be constantly updating as the player plays the game. When the game is launched, the text file containing this information will be loaded. Similarly, when the game is closed, the current information will be written to a text file to be stored. Resetting the game's progress will also reset the statistics on this page.

Functionality: The only interactive element on this page is the back button, which when pressed will take the user to the main menu screen. In order to reset these statistics, the player will need to navigate to the settings screen and select the Reset Game Progress button. There will be a scrollable feature on the table to view more statistical figures if more facts are added as the game gets more complicated.

Algorithm Design - Flowchart



Algorithm Design - Pseudocode

```
function displaySettings() {
        LOAD statistics data
        OUTPUT statistics screen
        INPUT buttonPress
        INPUT gameQuit
        IF buttonPress == back THEN
                LOAD main menu screen
        END IF
        IF gameQuit = True THEN
                QUIT game
        END IF
endfunction
```

Key Variables

statistics data: This refers to all the statistical data that is stored on a file and then loaded and read when this page is launched. The data in this file is updated after every level or game mode is played.

buttonPress: As explained previously, this is a variable which store the clicked state of the buttons.

gameQuit: As explained previously, this is a variable which is used to check if the user has clicked the red cross in the top-right corner of the game window to close the game.

back: As explained previously, this is a variable storing the tag of the back button.

game: As explained previously, this is an instance of the game class.

Test Data

Test Number	Test Description	Test Data	Test Type	Expected Outcome
55	Check the functioning of the Back button	Click on the Back button	Valid	Game launches the main menu screen
56	Check that the statistics data correctly updates after a level is played	Play a level and record the score achieved. Then check on this page to see if the total score and games played has increased	Valid	Game correctly updates the statistical data
57	Test if the buttons work when they are clicked at their edges	Click on the buttons at their edges	Valid Extreme	Game correctly executing the function associated with the buttons
58	Make sure that clicking anywhere which isn't a button on the screen doesn't open another page	Click on region on the screen which doesn't have any buttons	Invalid	Nothing should happen. No screens should be loaded.

Usability Features

As these are the menu pages for the game, the user will need to frequently use and interact with these screens. Taking this into consideration, I have made them easy to use by having large buttons to navigate to other game screens. This property of the buttons makes them easier to press, resulting in the process of manoeuvring through the game screes more seamless. Moreover, in certain screens there are images next to the buttons so that the users can easily identify the button's purpose. This alongside the larger text depicting the purpose of each button makes it easier for users with vision difficulties to use this program. This text will have a contrasting colour to the button to make it easy to read and aesthetically pleasing.

When the user hovers over or holds down (on mobile devices) the button, it will change colour to show that it is being highlighted. This lets the user know that they are about to click that button and go to a new screen, making the interface more user friendly. Bright, vibrant colours being used for the buttons regular and highlighted colours will aid visually impaired users to use the program. Furthermore, the strong, black border on the buttons will assist in conveying the interactive region of the buttons for users with vision difficulties, making them easier to locate and press. Moreover, when drawing objects or text to the screen antialiasing will be turned on to make the images and characters sharper and more detailed to further aid my program to be used by users with vision difficulties.

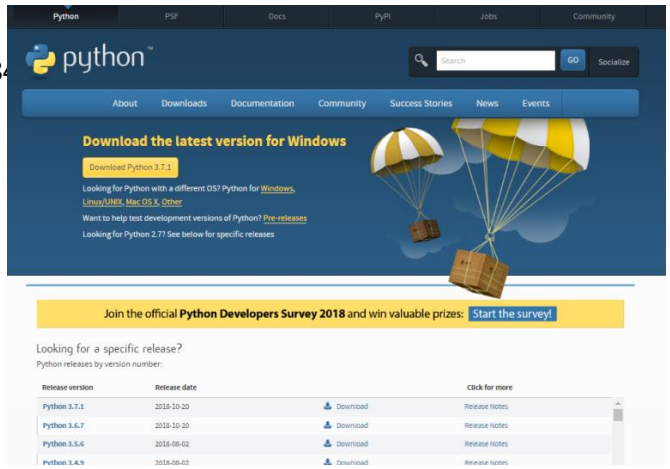
Users with hearing difficulties will not be at a disadvantage in my game as the purpose of the game is based completely on the visual elements. The sacrifice that these users will make is not being able to hear the background music that is used whist the user is in the menu screens and in the levels.

Links to success criteria:

- ✓ 2. Create an arrangement of screens which can be linked to form a menu system. All the settings and features should be accessible from here.
- ✓ Make the menu system a user friendly interface with large buttons.

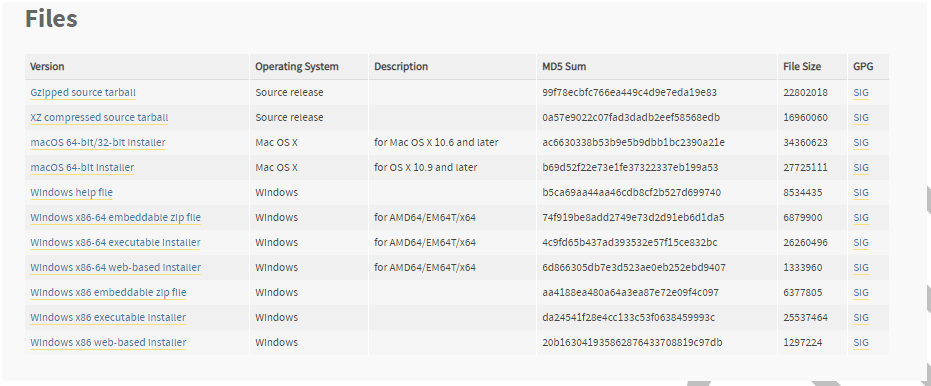
These are the initial ideas for the menu screens and further graphics and possibly animations will be added towards the later stages of the development process. These additions will make these pages more lively and exciting. There will also be a soundtrack playing while the user is in the menu screens to enhance their experience whilst navigating these screens.

Name: Krishan Sritharar Candidate Number: 8



Downloading Python and Pygame

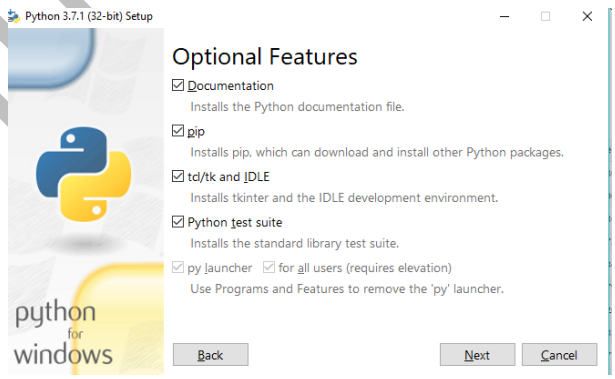
Python is available to download at https://www.python.org/downloads/ . The versions of Python that are available to download are listed on this page so any desired release can be installed. It would be recommended to install the most recent release of Python as this will include the latest versions of libraries and bug fixes. I installed the Python 3.7.1 release and selected 'Windows x86 executable installer' on the next page.
The other versions are for other operating systems and AMD CPUs as they will have a different instruction set to Intel CPUs, requiring different code for the Python interpreter.



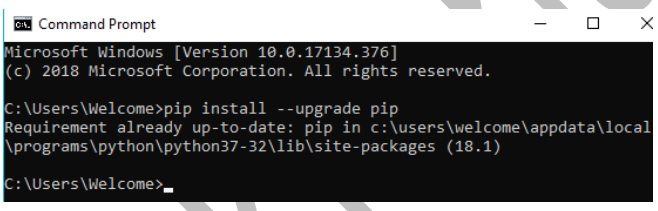
Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		99f78ecbfc766ea449c4d9e7eda19e83	22802018	SIG
XZ compressed source tarball	Source release		0a57e9022c07fad3dadb2eef58568edb	16960060	SIG
macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	ac6630338b53b9e5b9dbb1bc2390a21e	34360623	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	b69d52f22e73e1fe37322337eb199a53	27725111	SIG
Windows help file	Windows		b5ca69aa44aa46cdb8cf2b527d699740	8534435	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64	74f919be8add2749e73d2d91eb6d1da5	6879900	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	4c9fd65b437ad393532e57f15ce832bc	26260496	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	6d866305db7e3d523ae0eb252ebd9407	1333960	SIG
Windows x86 embeddable zip file	Windows		aa4188ea480a64a3ea87e72e09f4c097	6377805	SIG
Windows x86 executable installer	Windows		da24541f28e4cc133c53f0638459993c	25537464	SIG
Windows x86 web-based installer	Windows		20b16304193586287643370881 9c97db	1297224	SIG

In the installation window I selected 'Customize installation' and then selected all the optional features on the screen shown below. Pip is a package management system used in Python and comes included with the download for Python 2.7.9 or above for Python 2, and Python 3.4 or above for Python 3.

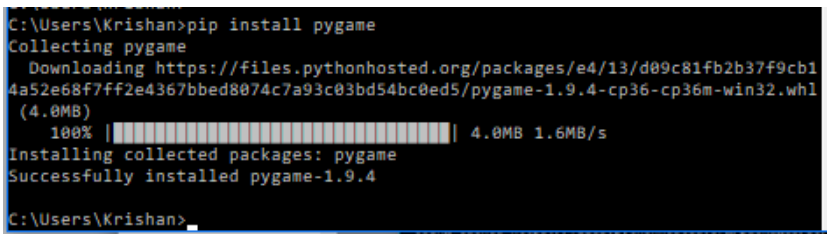


If it has not been included with the installation of Python, it can be downloaded from https://pypi.org/project/pip/#files and installed by running the 'get-pip.py' file.
Pip is used to download and install and manage Python packages. It reduces the complexities of installing additional modules by removing the requirement for the user to store modules in locations where multiple programs can access them. Pip commands are executed on command line interpreter applications (Command Prompt on Windows, and Unix command prompt, via the Terminal application, on Mac).



To update pip the command 'pip install –upgrade pip' is used. It is recommended to keep pip updated to avoid any errors and have the latest security fixes.

Pygame can be installed using the pip and the command 'pip install pygame'. The latest version of the pygame module that is compatible with the installed release of python is searched for online and downloaded. The package is then installed and saved in this directory: C:\Users\(Username)\AppData\Local\Programs\Python\Python36-32\Lib\site-packages . If there is no version of pygame that is compatible with the installed release of python then python should be upgraded to a newer version. Other packages can be installed using 'pip install <package name>. If a package needs to be uninstalled the command 'pip uninstall <package name>' is used.



The python shell can be used to check if a package has been correctly installed on the computer. A package has successful been installed if the code 'import <package name>' produces no errors. This welcoming message shows that pygame has been successfully installed onto my computer.

```
>>> import pygame
pygame 1.9.4
Hello from the pygame community. https://www.pygame.org/contribute.html
```

Development using Pygame

The Fundamental Game Loop

The following sections of code contain all the required functions to fundamentally run a game using pygame. I have decided to use an object oriented approach as it provides a modular structure to the program, making it easier to maintain and allowing faster development as objects (such as game sprites) can be reused. Splitting the program up into separate files aids towards better maintenance and debugging of all parts of the program as correctly functioning classes and function will not need to be analysed for errors.

Importing files and modules

The classes and functions defined within these separate files are imported into other files using python's inbuilt 'import' command. This command will first check the directory were the current file is stored for the module being imported. If no such file exists, it will check the directory on the device where python packages are installed. If it still doesn't find the file then 'ModuleNotFoundError: No module named '(file name)'' will be printed to the terminal/shell. If a module is successfully imported then the functions defined in that module will need to be addressed using the following command '(file name).(function name)'. If the file name is of considerable length and it would be bothersome to keep addressing the file name when a function defined in it is required, the 'import (file name) as (abbreviation)' command can be used. As shown above, the pygame module is imported as pg and all the functions defined in the pygame module can now be called using 'pg.(function name)' instead of 'pygame.(function name)'. To import all of the functions defined in a file, the command 'from (file name) import *' can be used. This will mean that these functions can now be called using their function name instead of also having to include the file name. Variables and constants (which in python are the same) are also imported from the files. If all of the functions/classes defined in a function are not required in the current file then the command 'from (file name) import (function) can be used to only import the required methods. The advantage of using this command is that it will avoid unnecessary errors if and when functions with the same name from different files are imported into the same file.

main.py

```
1  import pygame as pg#imports pygame package
2  import random#import random module
3  from settings import *#imports varaibles from settings file
```

Creating and initialising the Game class

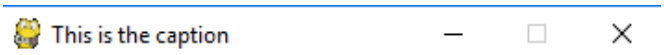
The game class is the main class in the project and will be where functions and classes will be called/initiated as required during the game. The class is defined using the command 'class (class name):'. Classes are beneficial as they provide a method of bundling data and functionality together. They allow multiple instances of themselves to be created at the same time, with each instance having all of the attributes and data defined in the class. Methods defined in parent classes can be inherited by child classes using the command 'class (class name) ((parent class name)):' when they are created. This helps reduces unnecessary repetition of code and as they are self-contained modules, correctly working sections of code will be easily identifiable and ignored while debugging and error.

```
5  class Game:
6      def __init__(self):
7          pg.init()#Initialises all imported pygame modules
8          pg.mixer.init()#Initilases modules for sound and music
9          self.screen = pg.display.set_mode((WIDTH, HEIGHT))#creates a game window
10         pg.display.set_caption(TITLE)#sets the caption of the window
11         self.clock = pg.time.Clock()#creates a Clock object to track time
12         self.running = True
```

Once a class is created it needs an initialisation function and this is defined using the command 'def __init__(self, (other parameters))'. The inbuilt python function 'def' is used to create new functions. This '__init__' function will be automatically called once an instance of the class is created. The parameter 'self' is used to represent the instance of the class and is required if other class functions are variables are needed within the function. Any parameters that are required for making the class need to be provided into this function in addition to the 'self'. Currently no such parameters are given however an example is the position on the screen of a game sprite. Inside the function, the command 'pg.init()' is first called. The purpose of this

function is to initialize all of the imported pygame modules such as the display and font modules. These modules must be first initialised for functions defined within these modules to work.
The next command is 'pg.mixer.init()' and this initialises the mixer modules which handles all of the in game music and sound effects. This modules is not automatically initialised by the 'init()' function and so needs to be performed separately.

The game screen is created in the next line. The command 'pg.display.set_mode()' creates a display surface (i.e. the game window) with the size that is passed in as the parameters to this functions. The constants WIDTH and HEIGHT are defined in the settings files and will be explained in further details later. The surface that is created is stored in the 'self.screen' variable. The reason why the variable is created using this syntax in oppose to just 'screen' is because using 'self.(variable name)' allows the variable to be accessible throughout the class. The variable 'screen' will be a local variable and can only be addressed within the 'init()' function. Therefore, variables that need to be accessible throughout the class are created in this way. The parameters provided when creating an instance of the class are usually assigned to variables of this format to make them accessible throughout the class. The command 'pg.display.set_caption()' is used to set the title of the game window. This title will be the name of the game and will be displayed on the game window as shown on the left.



The command 'pg.time.Clock()' creates a new Clock object which can be used to keep track of time during the game. This object is assigned to the variable 'self.clock' and it can be useful for controlling the game's framerate. A variable called 'self.running' is created and assigned the boolean value True to show that the game is currently running. When this variable is assigned the value False, the game will stop running.

New game and Run functions

The 'new' function is the first function which is called after a new game object is initiated. Inside this function, the groups for the sprites are created. The command 'pg.sprite.Group()' creates a container for sprite objects. Sprites can be added to these groups in the 'init()' function in their classes. The advantage of adding the sprites to these groups is that functions can be called all the sprites in the group at the same time. For example, each sprite will have a function called 'update()' which computes any changes that need to happen to the sprite/object, and a function called 'draw()' which will update the screen after making the changes. To call these functions without the use of these groups would mean that the 'update' and 'draw' function will need to be called on every instance of every sprite. This is a repetitive process and prone to leading to making mistakes. Functions can be called on the objects in the group using the command '(group variable.(function name)'. Here the 'allsprites' group is created and in this all of sprites created in the game will be added. An additional useful attribute to using groups for the sprites is that when checking for collisions, collisions between any sprites in two groups can be checked in oppose to checking every sprite against every other sprite. These groups of sprite therefore make managing multiple sprite objects easier and so the code more efficient. At the end of the 'new' function the 'run' function is called.

```
14    def new(self):      #start a new Game
15        self.allSprites = pg.sprite.Group()#Creates a group for game sprites
16        self.run()
17
18    def run(self):      #keeps the game running
19        self.playing = True
20        while self.playing:
21            self.clock.tick(FPS)#updates the clock
22            self.events()
23            self.update()
24            self.draw()
```

The 'run' function keeps the game running by calling the other function in an indefinite loop. A variable called 'self.playing' is created and assigned the Boolean value 'True'. This variable is used to check if the game is still running and an example of when this will be set to 'False' is when the game over screen needs to be displayed. An indefinite 'while' loop which repeats until the variable 'self.playing' is not equal to 'True' is created. Inside this loop there is the command 'pg.clock.tick(FPS)'. This command needs to be called once per frame and it computes the amount of time passed since the previous call of this function. The parameter 'FPS' is a value representing the frame rate that the game should be run at. By passing this value in, the

function limits the runtime speed of the game to so that the game only runs at the provided frame rate. For example if the frame rate provided is 30, this function will delay the game by $\frac{1}{30}$ seconds so that in one second, there can only be 30 loops of the 'while' loop. This function ensure that the experience of the game is the same across all devices. After this time delay function, the 'events', 'update' and 'draw' function defined in the Game class are called respectively.

Events, Update and Draw functions

The 'event' function handles searching for any user input that has been provided. The command 'pg.event.get' returns a queue of all the inputs that have been recorded. The 'for' loop is used to iterate through each item in this queue and the successive 'if' statements check if the recorded events match specific events. These include clicking on the mouse button, pressing down a key on the keyboard or releasing a key on the keyboard. Here it is being checked whether the program is trying to be closed by clicking on the cross in the top-right corner of the window. If this is pressed, then the variables 'self.running' and 'self.playing' are set to 'False', causing the game loop to end. Other key presses will be checked as the game increase in complexity.

```
26      def events(self):   #Gathers user input
27          for event in pg.event.get():
28              if event.type == pg.QUIT:#check for closing the window
29                  if self.playing:
30                      self.playing = False
31                      self.running = False
32
33      def update(self):   #Calculates changes in game
34          #Game Loop - Update
35          self.allSprites.update()#calls update function on all sprites in group
36
37      def draw(self):     #outputs changes made to screen
38          self.screen.fill(BLACK)#fills the screen with specified colour
39          self.allSprites.draw(self.screen)#calls draw function on all sprites in group
40          pg.display.flip()#updates changes made to the screen
41
```

The 'update' function is where changes in the game are computed. The events which have occurred and have been recorded will be used in this function to determine changes that need to be made to the sprites on the screen. Also, collision checks between sprites will take place here. Alongside this functionality, the 'update' function is called on all the sprites in the 'allsprites' group.

The 'draw' function is where the changes that have been made to the sprites are drawn to the screen. Here, the command '.fill(BLACK)' is used to fill the game window with the solid colour black. The constant BLACK has the value (0,0,0) which is the RGB (red green blue) value for the colour black. A second parameter could be provided to this function to specify the position and dimensions of a rectangle on this surface to fill. The absence of this parameter results in the whole surface being filled with the solid colour. The 'draw' function is then called on all of the sprites inside the 'allsprites' group, and the game window variable, 'self.screen', is provided as a parameter. After all the changes have been made to the screen the function 'pg.display.flip()' needs to be called. This will update the contents of the entire display so that the changes made become visible on the screen. If this function is not called, the changes will be made however as the display doesn't get updates, the changes will not be seen.

Creating an instance of the Game class

The functions 'showStartScreen' and 'showGameOverScreen' will contain code which draws text to the screen to welcome the user to the game and show them that the game is over. In this fundamental game loop, these aren't essential to get a functional game and so the pass statement is used to signify that no code needs to be executed.

```
43      def showStartScreen(self):
44          pass
45
46      def showGameOverScreen(self):
47          pass
48
49  g = Game()#creates an instance of the Game class
50  g.showStartScreen()
51
52  while g.running:
53      g.new()
54      g.showGameOverScreen()
55
56  pg.quit()
```

Outside of the class, an instance of the game called 'g' is created. The 'showStartScreen' function is called on this object and after this, the 'running' variable created in the 'init' function is used to create an indefinite loop which calls the 'new' function. This function then calls the 'run' function to keep the game running. When the game is over, the variable 'self.playing' will be set to 'false', causing the

'showGameOverScreen' function to be called If 'running' is 'false', then the 'pg'quit()' command will be executed. This un-initialises all the pygame modules and as a result of this, the game window will close.

Settings file

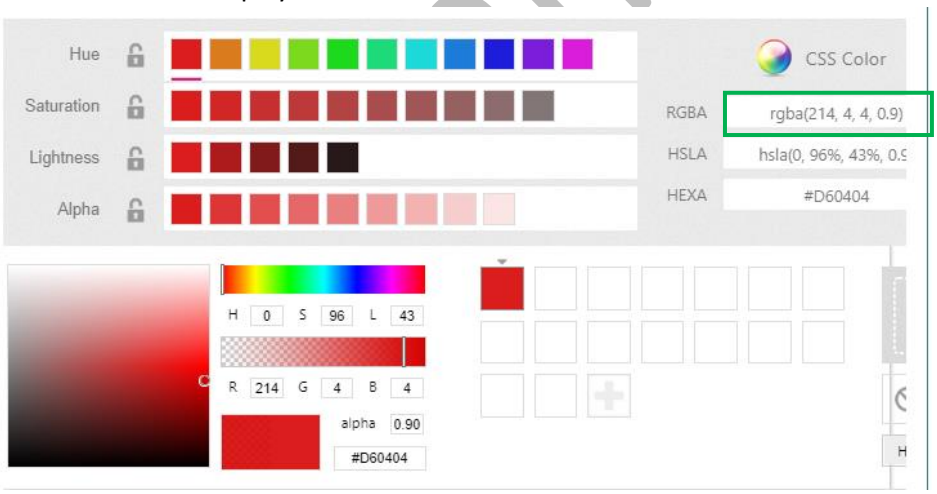
The settings file is where constants and frequently alterable variables are defined. By having the variables that need to be changed (such as jump height) in one place, the programmer only needs to make the change in one place. Alternatively, the change will need to be made in every occurrence of the variable, making it more likely for mistakes to be made, causing errors to be created. All of the variables created in this file have block capital names to easily identify them as constants in other files. The WIDTH and HEIGHT variables define the width and height of the game window in pixels. File names of images and sound tracks to be loaded into the game can be stored in a list here. Also, changeable features of the game, such as the value of the forward acceleration of the car when the accelerator button is pressed, can be stored here to be easily altered. Additionally, values of colours that need to be used to fill the screen and fill other game objects (such as buttons) can be stored here.

settings.py

```
#Game Settings

TITLE = "Demo"#for window's caption
WIDTH = 360#of game window
HEIGHT = 480#of game window
FPS = 30

# colours defined by RGB (Red, Green, Blue) value
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
```

Representing Colours

In pygame colours are defined using their Red, Green and Blue (RGB) value. This value is an indication of how bright each red, green and blue pixels which makes up the screen needs to be at a particular point. A value needs to be given for each colour so this is achieved using a tuple data structure. In this data structure, the data cannot be changed, which is ideal for a constant colour variable. Assigning a value of (0, 0, 0) means that the pixels at that particular location on the screen will not be lit, resulting in the pixel being black. As this value increases, more current will be provided to the pixel and therefore cause a different, brighter colour to be displayed. The maximum value for the colour is '255' as this is the highest value that can be represented using eight bits. If all of the pixels (red, green and blue) have a value of 255, then the overall pixel colour will be white. Using this representation of colours, $255 \times 255 \times 255 = 16581375$ colours can be created and displayed on the screen.




```
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
DARKGREY = (40, 40, 40)
LIGHTGREY = (100, 100, 100)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
YELLOW = (255, 255, 0)
CYAN = (0, 255, 255)
```

This colour picker tool available on https://mzl.la/296ZX3l can be used to pick the colours that are going to be used. The RGBA value is provided in the right side of the tool, indicated by the green rectangle. The A in RGBA stands for alpha value. This value defines how transparent the colour is. Similar to the other values, the alpha of a colour has a value between 0 and 255, with 0 meaning that the colour is completely opaque, and 255 meaning that the colour is completely transparent. A colour can be easily picked using this tool and the RGB value can be copied into the settings file of the game to be used.

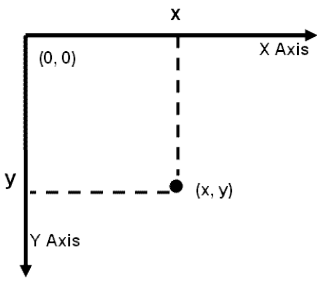
Monitoring the game's performance

As explained during the fundamental game loop, the game works by playing lots of frames at a very fast rate to create the impression of a moving picture. This frame rate is set using the 'clock.tick(FPS)' function and if the program was using a lot of computational resources then there will be a negative effect on the frame rate. The increased amount of computation will mean that the computer will struggle to maintain a consistent frame rate. With few animations on the screen this decrease in frame rate will be difficult to observe. Therefore, I have decided that it will be helpful to set the caption of the window as the current frame rate. When doing this, the frame rate can be easily observed and if it goes low after a change, then it would be clear that there has been a logical error causing more computation. To implement this I used the 'pg.display.set_caption' function, and got the current frame rate using the function 'clock.get_fps'. I then used the '.format' method to insert the frame rate into the '{ }' in the caption, and rounded the frame rate to two decimal places using ':.2f'.

```
def draw(self):
    pg.display.set_caption("{:.2f}".format(self.clock.get_fps()))
```

I chose to set the frame rate at 60 frames per second for my game. The reasoning for this is because this frame rate allows smooth animations while not requiring very intense computational power to render the frames every second.

Referencing coordinates on the screen



Referencing coordinates on a computer screen is different to referencing coordinates on the Cartesian axes as the origin for the axes on a computer is in the top-left of the screen. The reason for this is because old technology used to output to the pixels from left to right, and from top to bottom, making the top-left corner an ideal place to begin. This system keeps the coordinates positive integers and also allows use with all types of screens as the axes are just essentially extended.

Importing images and files

```
def load_data(self):
    game_folder = path.dirname(__file__)
    img_folder = path.join(game_folder, 'img')
    snd_folder = path.join(game_folder, 'snd')
    music_folder = path.join(game_folder, 'music')
```

Images, sound effects and music tracks need to be imported once at the start of the game. This can be efficient achieved by creating a 'loadData' function, and loading all of the images and files in there. The code 'path.dirname(__file__)' gets the path to the directory in which the code is saved. This function is useful as when this path is changed (for example when the code is executed on a different computer) the path is automatically returned and so no changes need to be made by the programmer as the path is not hard coded. Inside the current directory, there would be three folders called 'img', 'snd' and 'music', with each folder storing their respective files. The code 'path.join()' can now be used to add text to the path to the current directory. This means that these three folders can now be accessed. Their paths are stored in variables for later use.

Images are loaded using the '.load' function and the path to the picture. The 'path.join' function is used alongside the name of the image to create this path. The loaded images are stored in variables so that they can later be accessed. Sound effects are also loaded using the 'load' function and the path to the track. The command 'pg.mixer.music.load()' is used. There can only be one background music track playing at any time in pygame. The code 'pg.mixer.Sound()' is used to load the track and code 'pg.mixer.play(loops=-1)' will play the music. When the parameter to this function is -1, the music will keep on looping until the game is closed.

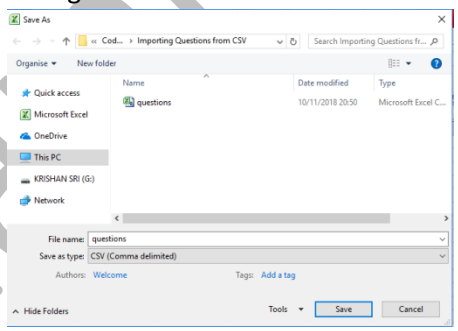
Using CSV files

Creating and importing the CSV file

I created the table below in the spreadsheet software Microsoft Excel. In row 1, the names of all the fields are given. This row is only required for maintaining the flat-file database and is not required by the program.

	A	B	C	D	E	F	G	H	I	J
1	questionID	questions	correctAnswer	wrongAnswer1	wrongAnswer2	wrongAnswer3	wrongAnswer4	difficulty	level	isMajor
2	1	2+2	4	3	2	6	5	easy	1	FALSE
3	2	5+7	12	13	25	10	15	easy	1	FALSE
4	3	15+17	32	30	42	23	34	easy	0	TRUE
5	4	25÷5	5	1	7	11	4	easy	1	FALSE
6	5	4x3	12	11	15	17	8	easy	1	FALSE

After creating this spreadsheet, I saved this as a CSV (comma separated value) file using the 'save as' feature in Microsoft Excel. During the process of making the spreadsheet, the cells had to be formatted to text when symbol "/" was being mistaken for defining a date. However after formatting the cell, I decided to replace the "/" symbol with "÷" as it is more easily identifiable as divide.



The CSV file can now be loaded into Python and manipulated. To do this, the modules 'csv' and 'path' are imported using the 'import' command. A function called 'reader' is needed from the 'csv' module to read the csv file, and functions from the 'path' module are needed to locate the path to the csv file. The command 'path.dirname(__file__)' is called and the returned value is stored in the variable 'main_folder'. This command returns the path to the directory where the current file is stored. As the CSV file is stored in the same directory, it can now be easily located. The command 'path.join(main_folder, 'questions.csv')' is called, and this joins the text provided with the path stored in the 'main_folder' variable. The new path is for the csv file and it is stored in the variable 'questions'. The inbuilt python function 'open()' is now used to open the csv file. The additional parameter provided to the 'open' function determines which state the file is opened in with 'r' meaning read only and 'w' meaning read and write. As the contents of the CSV file does not need to be changed, the file is opened only for reading. The command 'with … as' is used to store the opened csv file under a name. Here this name is 'questionFile'.

```
import csv, path

main_folder = path.dirname(__file__)
questions = path.join(main_folder, 'questions.csv')
with open(questions, 'r') as questionFile:
```

Reading the CSV file

After the CSV file has been opened, the function 'csv.reader(questionFile)' is called. This function returns a reader object which will iterate over the lines in the provided csv file. This reader object is stored in the variable 'reader'. The inbuilt python command 'next(questionFile)' is used to iterate over the first line in the CSV file. As mentioned previously, this first row contains the titles of the columns and so doesn't need to be read by the program. The data that is read from the CSV file will be stored in a two-dimensional list. The reasoning for this decision of data storage is that in a two-dimensional list, each row and column of the spreadsheet can be accessed by using two indexes for the list. A list called 'data' is created to store this data. A 'for' loop is used to iterate through each line in the csv file. A list called 'temp' is created to temporarily hold the stored data. In each row of the CSV file, the data type of the data in each column is specified and temporarily stored in a variable. After this, the 'extend' list operation is used to append a list of elements to the 'temp' list. The 'temp' list is then appended to the two-dimensional 'data' list. The reason for first using 'extend' is because 'append' doesn't allow multiple elements of data to be added at once. Therefore, this approach is required to achieve a two-dimensional list for the data from the CSV file.

```
with open(questions, 'r') as questionFile:
    reader = csv.reader(questionFile)
    next(questionFile)
    data=[]
    for line in reader:
        # Line = [QuestionID, Questtion, CAns, WAns1, Wans2, Wans3, Wans4, Diff, Level, isMaj]
        temp = []
        questionID = int(line[0])
        question = str(line[1])
        cAns = int(line[2])
        wAns1, wAns2, wAns3, wAns4 = int(line[3]),int(line[4]),int(line[5]),int(line[6])
        diff = str(line[7])
        level = int(line[8])
        isMaj = str(line[9])
        temp.extend((questionID,question,cAns,wAns1,wAns2,wAns3,wAns4,diff,level,isMaj))
        data.append(temp)#all the data is now held in a 2D array
```

Creating a simple question asking program using the CSV file

I decided to make a simple question asking program which reads the CSV file and asks the questions on it. The program picks three of the four incorrect answers and randomly inserts the correct answer with the incorrect answer. Additionally, the time taken to answer each question is recorded and this value is used to calculate the average time at the end of the program.

```
import csv, random, time
from os import path

def loadQuestions():
    global data
    main_folder = path.dirname(__file__)
    questions = path.join(main_folder, 'questions.csv')
    with open(questions, 'r') as questionFile:
        reader = csv.reader(questionFile)
        next(questionFile)
        data=[]
        for line in reader:
            # line = [QuestionID, Questtion, CAns, WAns1, Wans2, Wans3, Wans4, Diff, Level, isMaj]
            temp = []
            questionID = int(line[0])
            question = str(line[1])
            cAns = int(line[2])
            wAns1, wAns2, wAns3, wAns4 = int(line[3]),int(line[4]),int(line[5]),int(line[6])
            diff = str(line[7])
            level = int(line[8])
            isMaj = str(line[9])
            temp.extend((questionID,question,cAns,wAns1,wAns2,wAns3,wAns4,diff,level,isMaj))
            data.append(temp)#all the data is now held in a 2D array
```

First I created a function called 'loadQuestions' to load the questions from the csv file and store them in a two-dimensional list. The code used to do this has been explained previously. The two-dimensional list is made global in the program so that it can be accessed from outside of the function. As this is a simple program there is not a major issue in using global variables and alternatively, the list could have been returned by the function and stored in a variable when the function is called. The problem with using global variables in larger programs is that there is a possibility that two or more variables with the same name will be created, resulting in all variables apart from the most recently defined one to be overwritten.

```
def menu():
    print("Welcome to the mental maths speed test game")
    print("Try and answer the following questions as quickly as possible")
    print("To start playing the game press 1")
    print("To quit press 2")
    while True:
        ans = int(input("Enter your response: "))
        if ans == 1 or ans == 2:
            break
    if ans == 1:
        return True
    else:
        return False
```

Next, I created a function in which I used a series of print' statements to create a menu for this program. Pressing '1' starts the program and '2' will quit the program. There is validation for the user's response and any input which isn't either a '1' or a '2' will not break the 'while' loop, causing the question to be asked again. This function returns 'true' if the user chooses '1' and 'false' otherwise.

```
def main():
    loadQuestions()
    if menu():
        score = 0
        allQTime = 0
        for i in range(len(data)):
            print("What is {} = :".format(data[i][1]))
            nums = [3,4,5,6]
            answers = [2,]
```

This is the 'main' function where the questions are asked. First, the 'loadquestions' function is called to load the questions. Next, the 'menu' function is called and if this returns 'true', the program continues. Variables for keeping track of the score and the time to answer the questions are created and assigned a value of zero. There is then a 'for' loop which repeats for every row in the questions spreadsheet and first uses a 'print' statement to ask the question. As the question is the second column in the spreadsheet, it can be accessed in the two dimensional list using 'data[i][1]', where 'i' is the row number and '1' represents the second column, as python uses 0 based indexing. The '.format' method replaces the '{}' (curly brackets) in the string with the data provided in the parameters. This is a more efficient method of inserting multiple pieces of data into a string as the alternative would be separating the string by commas, using the 'str' function on the data, and then concatenating the data to the rest of the string. The incorrect answers are located in columns 4 to 7 in the spreadsheet and so are located in indexes 3 to 6 in 'data' (the two dimensional list). A list called 'nums' is created containing these indexes. Three of these numbers will be randomly selected and added to the 'answers' list, which already contains the index of the correct answer. The 'random' modules was imported at the start of the program so that the 'choice' function within this module can be used to randomly select a value in the 'nums' list. The chosen value is temporarily stored in the variable 'rand' before being removed from the 'nums' list using the '.remove' list operation. This value is now added to the 'answers' list and the loop

```
            for j in range(3):
                rand = random.choice(nums)
                nums.remove(rand)
                answers.append(rand)
            for k in range(4):
                rand = random.choice(answers)
                if rand == 2:
                    cAns = k+1
                answers.remove(rand)
                print("Press {} for {}: ".format(k+1,data[i][rand]))
```

repeats to randomly select two more numbers from the remaining three numbers. After this, an index is randomly chosen from the 'answers' list and a 'print' statement is used to output the four possible answers to the question to the user. The value 'k+1' is the key that will need to be pressed to choose that answer. For example, in the first loop, 'k' will be 0 so pressing '1' ('k+1') will correspond to the first choice. If the chosen index is equal to '2' (i.e. is the correct answer), the option number is stored in a variable called 'cAns' (short for correct answer), so that the user's answer can be checked against the correct answer.

```
        startTime = time.time()
        while True:
            while True:
                try:
                    userAns = int(input("Enter your answer: "))
                    break
                except ValueError:
                    print("That is not an integer, try again.")
            if userAns not in (1, 2, 3, 4):
                print("That is not 1,2,3 or 4, try again.")
            else:
                break
        endTime = time.time()
        totalTime = endTime - startTime
```

The 'time' module was imported at the start of the program and the 'time' function within this module is used to record the time at the particular instance when the function is called. This value is stored in the variable 'startTime' and will be used to calculate the time taken to answer the question. The 'input' function is used to ask the user for their answer. The 'int' function is used on the 'input' function to make sure that the provided answer is an integer. If this function returns an error, then the 'try' and 'except' block accepts the error and prints the error message as a result. If the provided answer is an integer but is not one of the options, the error message will be printed. It is only when an acceptable answer is inputted that both the validation 'while' loops will be stopped and the program will continue.

```
            endTime = time.time()
            totalTime = endTime - startTime
            if userAns == cAns:
                print("That is the correct answer, well done")
                score += 1
            else:
                print("That is the incorrect answer")
                print("The correct answer was {} = {}".format(cAns,data[i][2]))
            print("You took {} seconds to answer that question".format(round(totalTime,2)))
            allQTime += totalTime
            print()
        print("Your final score is: {}".format(score))
        print("Your total time to answer all the questions was: {}".format(round(allQTime,2)))
        print("Your average time to answer all the questions was: {}".format(round((allQTime/5),2)))
    else:
        print("Thank you for playing")

main()
```

After the user has provided an acceptable answer, the time at that instance is recorded and the time taken to answer the question is calculated by subtracting the finishing time from the starting time. The user's answer, which is stored in the 'userAns' variable, is checked against the 'cAns' variable to see if the correct answer has been inputted. If so, the 'score' variable is incremented by one. Otherwise, the option number and value of the correct answer will be printed. The total time taken to answer the question is outputted and then added to the 'allQtime' variable. The 'for' loop then repeats for the remaining questions. Afterwards, the final score is outputted, as well as the total time taken to answer all the questions and the average time take. This average time is the total time taken divided by the number of questions asked, which in this case is five.

```
Welcome to the mental maths speed test game
Try and answer the following questions as quickly as possible
To start playing the game press 1
To quit press 2
Enter your response: 1
What is 2 + 2 = :
Press 1 for 6:
Press 2 for 3:
Press 3 for 2:
Press 4 for 4:
Enter your answer: 4
That is the correct answer, well done
You took 2.89 seconds to answer that question
```

This shows the program being executed. The questions are being asked correctly and the time taken to answer the question is correctly being rounded to 2 decimal places using the 'round' function.

If the answers provided are no acceptable the program correctly identifies these answers and asks the user to try again. If the answer provided is incorrect, the program successfully outputs the correct answer and option number. After all of the questions have been answered, the total score, total time taken and the average time take to answer all the question is correctly output. The aim of this simple program was to become familiar with importing and manipulating CSV files with python in preparation for using it in my project.

```
What is 4 x 3 = :
Press 1 for 17:
Press 2 for 11:
Press 3 for 15:
Press 4 for 12:
Enter your answer: 4
That is the correct answer, well done
You took 1.88 seconds to answer that question

Your final score is: 5
Your total time to answer all the questions was: 12.43
Your average time to answer all the questions was: 2.49
```

```
What is 15 + 17 = :
Press 1 for 42:
Press 2 for 32:
Press 3 for 34:
Press 4 for 23:
Enter your answer: no
That is not an integer, try again.
Enter your answer: 9
That is not 1,2,3 or 4, try again.
Enter your answer: 3
That is the incorrect answer
The correct answer was 2 = 32
You took 14.26 seconds to answer that question
```

Developing the Menu System

Implementing a drawText function

In pygame the screen is the surface that was created at the start of the program using the command 'pg.display.set_mode(WIDTH, HEIGHT)'. Once objects have been rendered they need to be drawn to the screen. This is achieved using the 'blit(source surface, destination surface)' function in pygame. This function draws one image onto another. The image to be drawn first needs to be on its own surface, which happens during the process of rendering the object. An optional area can be passed into this function to represent a smaller portion of the source surface to draw to.

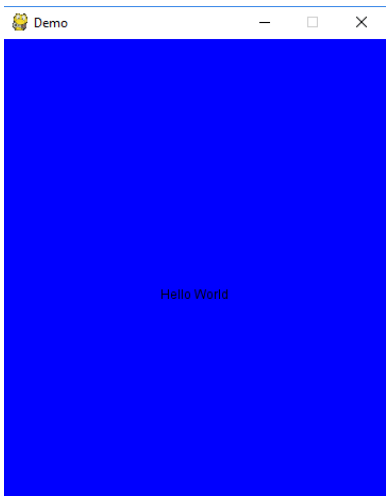
```
24      def drawText(self, text, size, colour, x, y, surf=None):
25          if surf == None:
26              surf = self.screen
27          fontType = "C:\WINDOWS\FONTS\ARIAL.TTF"
28          font = pg.font.SysFont('arial', size)
29          textSurface = font.render(text, True, colour)
30          textRect = textSurface.get_rect()
31          textRect.center = (int(x),int(y))#aligns the text to the center
32          surf.blit(textSurface, textRect)
```

This is the function that I have created for drawing text to the screen. This function is created in the Game class, allowing it to be accessed in other files and classes. The parameters for this function are the text, the size of the text, the colour of the text, the x and y position of where the text needs to be printed, and the surface that the text needs to be drawn onto. 'surf=None' means that if a surface is not provided when the function is called, the variable 'surf' is assigned the value 'None' instead of returning an error. If this is the case, then the game screen is used as the surface. This additional surface parameter provides more flexibility to this function.

The font of the text and its size now needs to be determined. There are two methods of achieving this: 'pg.font.Font(fontFilePath, fontSize)' creates a new font object of the specified size using the font at the given location, and 'pg.font.SysFont(fontName, fontSize)' also creates a font object but uses the system fonts. This means that for the latter function, the path to the font does not need to be specified and instead, the name of the font is given. Next, the 'render' function is called on the font to create a new surface with the specified text rendered on it. This function takes the text, colour and an option for antialias in its parameters. If this antialias argument is set to 'true' the characters in the text will have smooth edges.

The 'get_rect()' function is then used on the created text surface to get the rectangular area of the surface. This function returns a new rectangle covering the entire surface and the object can now be moved by addressing this rectangle. The command 'textRect.center' is set the x and y coordinates of the location of the text to the centre of the text surface rectangle. This aligns the text to the centre of the surface. Finally, the text is blitted to the desired surface using the command 'surf.blit(textSurface, textRect)'. As mentioned previously, this function will draw the 'textSurface' surface onto the 'textRect' surface, and then draw the result to the surface provided.

```
def showStartScreen(self):
    self.screen.fill(BLUE)
    self.drawText("Hello World", 12, BLACK, WIDTH/2, HEIGHT/2)
    pg.display.flip()
```



To test out the 'drawText' function I called it in the 'showStartScreen' function to print out 'Hello World' to the screen. The position of the text was set to 'WIDTH/2' and 'HEIGHT/2' which is the centre of the screen and the screen was filled with a solid blue colour before the text was printed. The game window to the right shows that the text was successfully and correctly printed.

Creating Buttons for the User Interface

An important component of the menu system will be navigation buttons. As a lot of button will need to be created, I decided to create a class to create many instances of buttons simultaneously. To keep the program organized, I created the class in a new file called 'button.py' and used the command 'from button import *' in the main file to import the button class. Firstly, pygame needs to be imported and alongside this, the constants and variables from the settings file are imported. The 'pg.math.Vector2' command is called and stored in the variable 'vec' to be referenced later. This 'math' module in pygame provides vector classes for managing data containing two dimensions. An example of this is position, which contains an x and y coordinate. When using vectors for storing this type of data, both pieces of data are stored in the same variable, and can be accessed using their index. A further advantage of using vectors is that already created functions can be performed on two or more vectors in a calculation. These functions include working out the magnitude of a vector, calculating the dot product and determining the angle between two vectors, which can be used to set the direction of a game sprite.

```
import pygame as pg
from settings import *

vec = pg.math.Vector2
```

Initialising the class

```
class Button(pg.sprite.Sprite):
    def __init__(self, game, tag, x, y, width, height, solidColour, highlightColour, text, textSize):
        self.groups = game.buttons
        pg.sprite.Sprite.__init__(self, self.groups)
```

When creating the 'Button' class, the 'pg.sprite.Sprite' class is inherited. This is a simple bass class for visible game objects that provides the functionality for adding sprites to groups. All of the parameters that are required when creating the button are shown in the '__init__' function for this class. 'game' is for providing an instance of the currently running game. This variable references the main file and will allow functions in the main file to be called, such as the 'drawText' function, and varaibles such as the screen to be used when blitting. The 'tag' variable is used to identify the button. If when clicked the button takes the user to the level select screen, then the tag of this button will be 'levelSelect'. This will be used to determine which button has been clicked. The 'x', 'y', 'width' and 'height' are the location and dimensions of the button. The 'solidColour' is the colour of the button when the mouse cursor is not hovering over it. When this is the case, the button will be coloured the 'highlightColour'. The 'text' variable will contain the text to be printed on the button, and the size of this text is given in the 'textSize' variable. A new group for the buttons is created in the 'new' function in the main game file. This group will be used to store the buttons that are created. This group can be referenced in the 'button' class using the code 'game.buttons', as the variable holding the buttons group in the main file is called 'self.buttons'. The initialise command is then called on the inherited class and within this function, the button object ('self') is added to the 'game.buttons' group.

```
self.buttons = pg.sprite.Group()
```

```
        self.game = game
        self.tag = tag
        self.width = width
        self.height = height
        self.solidColour = solidColour
        self.highlightColour = highlightColour
        self.text = text
        self.textSize = textSize
        self.pos = vec(x,y)
```

Next, all the data provided when creating the button object are stored locally using the word 'self'. This now allows this data to be accessed from outside the 'init' function. This means that 'self.game' can be used in the rest of the class however using 'game' will cause an error as 'game' is only defined within the 'init' function. The location of where the button is to be drawn in stored in a two-dimensional (2D) vector form. The 'vec' keyword, which was assigned the function 'pg.math.Vector2' at the start of the file, is used to achieve this. Two variables called 'self.clicked' and 'self.first' are created and given the values of 'false' and 'true' respectively. The 'self.clicked' variable will be set to 'true' when the button is clicked, and so can be used to determine the state of the button.

```
        self.clicked = False
        self.first = True
```

The 'self.first' variable is used to check if this is the first time the button is being drawn. If so, the text is also drawn on the button otherwise, the text is not drawn. This reduces the computational power required in this game as the text would not be unnecessarily re-drawn unless there is a change to the button.

```
        self.rectDimensions = (self.pos[0]-self.width/2, self.pos[1]-self.height/2, self.width, self.height)
        self.image = pg.draw.rect(self.game.screen, self.solidColour, self.rectDimensions , 0)
```

The button will be created by drawing rectangles to the screen. The dimensions of this rectangle will be the same so it will be efficient to create a variable for the rectangle's dimensions and pass that in where the dimensions are required. Doing this means if the dimensions of the rectangle need to be changed, the change only needs to be made in one place, making the process of debugging easier. The data required for the creation of a rectangle include the x and y coordinate of the top-let corner of the rectangle, the width of the rectangle and the height of the rectangle. The function 'pg.draw.rect' is used to create a rectangle in pygame. The parameters needed for this function are the surface to draw the rectangle to ('self.game.screen'), the colour of the button (by default 'self.solidColour'), the dimensions of the rectangle and the size of the outline. The created rectangle is stored in the variable 'self.image' to be used later.

The highlight function

```
    def highlight(self):
        mousePos = pg.mouse.get_pos()
        prevcol = self.game.screen.get_at((int(self.pos[0]-self.width/2), int(self.pos[1]-self.height/2)))
        colchange = False
```

The next function in this class is called 'highlight' and this function checks if the position of the mouse cursor is over the button and if the button is clicked. First, the current x and y position of the mouse cursor's relative to the top-left corner of the display is returned and stored in the variable 'mousePos' using the function 'pg.mouse.get_pos()'. The colour at a particular pixel on the screen is analysed next. The reason for this is to check if the colour of the button needs to be updated. The function '.get_at()' is performed on the screen surface, at the colour at the coordinates 'self.pos[0]-self.width/2' and 'self.pos[1]-self.height/2' is checked. 'self.pos[0]' is the x coordinate of the middle of the button. Subtracting half of the width from this

```
        if self.pos[0] - self.width/2  <= mousePos[0] <= self.pos[0] + self.width/2 and \
            self.pos[1] - self.height/2 <= mousePos[1] <= self.pos[1] + self.height/2:
            if prevcol[0:3] != self.highlightColour:
                self.image = pg.draw.rect(self.game.screen, self.highlightColour, self.rectDimensions , 0)
                colchange = True
            if pg.mouse.get_pressed()[0] == 1:
                self.clicked = True
        else:
            if prevcol[0:3] != self.solidColour:
                self.image = pg.draw.rect(self.game.screen, self.solidColour, self.rectDimensions , 0)
                colchange = True
```

goes to the left edge of the button. 'self.pos[1]' is the y coordinate of the middle of the button. Similarly, subtracting half of the height from here will go to the top edge of button. Therefore the position which satisfies both of these locations is the top-left corner of the button. This function returns the RGBA value of the colour at this pixel. A variable for checking whether the buttons colour has been changed is created. Next an if statement is used to check of the position of the mouse cursor is within the boundaries of the button. The variable 'mousePos' contains the coordinates of the curser with the x coordinate in the zero index and the y coordinate in the first index. The if statement checks if this x coordinate is between the coordinates of the left and right edge of the button. Additionally, it checks if the y coordinate is between the top and bottom of the button. If so, the button's colour needs to be changed to the highlighted colour. Before tis however there is a check to see if the current colour of the button is already the colour that it needs to be. If this check happens to be successful then there would be no need to change the button's colour, reducing the computational resources required during the execution of this game. As mentioned previously, the 'prevcol' variable contains the RGBA value of the colour at the top-left of the button. The code 'prevcol[0:3]' splits the list of the four integers contained in this variable into the first 3 integers (i.e. just the RGB values). The variable 'self.highlightColour' contains the RGB value of the colour of the button when highlighted. Therefore, the if statement returns true if the colour at the pixel on the screen is not the colour that it should be. This causes a new rectangle to be drawn, with all the same parameters apart from the different colour. The state of the variable 'colchange' is changed to true as the colour of the button has

been changed. The command 'pg.mouse.get_pressed( ))[0]' gets the current clicked state of the mouse. The alternative would be to use the 'pg.event.get' function, loop through all of the events and check for a MOUSEBUTTONDOWN event. This function returns the state of the all three mouse buttons therefore, the '[0]' is used to check the state of the left click button. If this is equal to 1, then it is being pressed down. This causes the variable which contains the clicked state of the button to be changed to true. If the mouse is not within the button's boundaries then there is a check to see if the colour of the button is the same as the 'self.solidColour' colour. If not, the mouse has just been moved off the button and therefore the colour of the button needs to be changed back to the solid colour. Once again, this is achieved by redrawing the

```
45          if colchange == True or self.first == True:
46              if self.first:
47                  self.first = False
48              self.game.drawText(self.text, self.textSize, BLACK, self.pos[0], self.pos[1])
```

rectangle with the different colour and then 'colchange' is set to true.
If the colour of the rectangle has been changed then the text drawn on the button has to be re-drawn. This is because the rectangle would have been drawn over the previous rectangle and text. Also, the button's text needs to be drawn when the button is first created. Therefore, an if statement is used to check if any of these conditions are true. If so and it was because the button has just been created then the value of the variable 'self.first' is changed to false. The 'drawText' function created in the main file is called using the instance of the game provided. The text is positioned at the centre of the button.

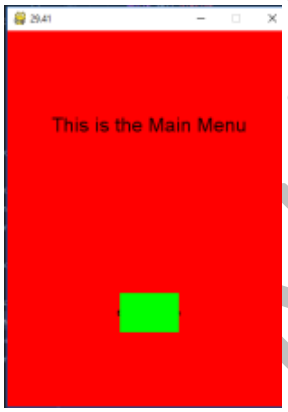
The update function

This is the 'update' function for the button class and firstly, the 'highlight' function is called. After this, the clicked state of the button is checked and if it is clicked, then the for loop is used to loop through the buttons group and delete all of the instances of buttons apart from the button which was clicked. This means that on the new page, the previous button will not be present. The function '.kill()' is used to delete the buttons from the group.

```
45  def update(self):
46      self.highlight()
47      if self.clicked:
48          for button in self.game.buttons:
49              if button != self:
50                  button.kill()
```

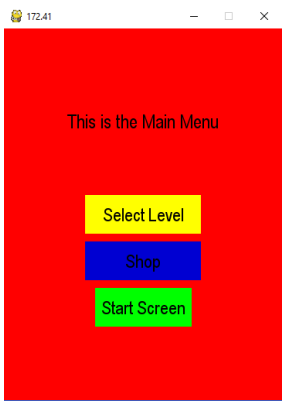
Errors during testing

I first tested this function by adding creating a button in the 'showStartScreen' function. However there was an error stating that the buttons group did not exist. I realised that the cause of this problem was that the 'showStartScreen' function was being called before the 'new' function, which was where the group was created. This was fixed by calling the 'showStartScreen' function after the within the new function and before the run function. At first there was also the problem that the text kept being drawn underneath the button (shown on the left). Initially, the button was being created using the 'pg.Surface' function but providing the new surface as a parameter to the 'drawText' function didn't resolve this problem. Also the surface object couldn't be added to the buttons group. This problem was solved by drawing a rectangle for the button instead of making a surface.



The image below shows the buttons and text correctly displayed, with the highlight function working correctly (the button change to blue when the mouse is hovering over it). However I noticed that the text kept getting bolder will the program was running, meaning that the text was constantly being printed to the screen. Using the 'self.tag' variable, I printed the tag of the button when the text was drawn to it. Text should only be drawn if the colour of the button has changed however, the terminal showed that the text for only one button ('startScreen') was being continuously drawn. I realised that this problem was due to the position of where the previous colour of the button was being checked. Initially, the colour was being checked at the centre of the button and for this button there happened to be text at the centre. This meant that the centre of the button was always the text colour and never either button colour. To fix this, I changed the colour check to the top-left of the button.

```
text drawn startScreen
text drawn shop
text drawn startScreen
text drawn startScreen
text drawn startScreen
text drawn startScreen
text drawn startScreen
text drawn startScreen
```



Designing a Scene Manager

Buttons can now be drawn to the screen however sets of if statements are needed to check if they have been clicked and if so, call the appropriate function. This is inefficient and a more systematic and organized method to checking if buttons are pressed if by creating a tool which manages the screens. I called this tool a scene manager and created a new class for it in a new file.

```
import pygame as pg
from settings import *
from button import *

class sceneManager():
    def __init__(self, game):
        self.game = game
```

Firstly, pygame as well as the variables and classes from the settings and buttons files are imported. Then the class is created with the only parameter being the instance of the current game. This is so that the game screen can be accessed and the 'drawText' function can be called.

```
    def loadLevel(self, level):
        self.level = level
        if self.level == "settingsMenu":
            self.settingsMenu()
        if self.level == "mainMenu":
            self.mainMenu()
        if self.level == "levelSelect":
            self.levelSelect()
        if self.level == "stats":
            self.stats()
        if self.level == "leaderboard":
            self.leaderboard()
        if self.level == "shop":
            self.shop()
        if self.level == "startScreen":
            self.startScreen()
        if self.level == "gameOverScreen":
            self.gameOverScreen()
```

Next a function called 'loadLevel' is created to call the functions for creating each level. The level to be loaded is provided into this function and this sting is analysed against a set of predetermined levels and the respective function is called if there is a match.

Creating the screens

```
    def settingsMenu(self):
        self.game.screen.fill((30, 210, 110))
        self.game.drawText("Change the Game Settings", 25, BLACK, WIDTH/2, HEIGHT*1/6)
        self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)

    def levelSelect(self):
        self.game.screen.fill((160, 160, 160))
        self.game.drawText("Select the Level you want to play!", 25, BLACK, WIDTH/2, HEIGHT*1/6)
        self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)

    def stats(self):
        self.game.screen.fill((220, 110, 250))
        self.game.drawText("View your statistics", 25, BLACK, WIDTH/2, HEIGHT*1/6)
        self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)

    def leaderboard(self):
        self.game.screen.fill((50, 250, 160))
        self.game.drawText("Leader Board Tables", 25, BLACK, WIDTH/2, HEIGHT*1/6)
        self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)

    def shop(self):
        self.game.screen.fill((255, 180, 0))
        self.game.drawText("Spend you coins in the shop", 25, BLACK, WIDTH/2, HEIGHT*1/6)
        self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)
```

After this, the functions for each level are created. Currently, the levels are basic with only some text and a few buttons however the complexities of these functions will increase as more items get added to the screens. In each screen, the colour of the screen is changed to a solid colour and some text is drawn to the screen to indicate what the screen is. The button is positioned by using fractions of the height and width of the screen instead of exact pixels so that if the screen size is changed, the location of the button also changes. All of these screens share a common back button which navigates back to the main menu screen.

```
    def mainMenu(self):
        self.game.screen.fill((255,90,70))
        self.game.drawText("This is the Main Menu", 25, BLACK, WIDTH/2, HEIGHT*1/4)
        self.startScreenButton = Button(self.game, "startScreen", WIDTH/4, HEIGHT*3/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Start Screen",25)
        self.levelSelectButton = Button(self.game, "levelSelect", WIDTH*3/4, HEIGHT*3/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Select Level",25)
        self.settingsMenuButton = Button(self.game, "settingsMenu", WIDTH/4, HEIGHT*5/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Settings",25)
        self.shopButton = Button(self.game, "shop", WIDTH*3/4, HEIGHT*5/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Shop",25)
        self.leaderboardButton = Button(self.game, "stats", WIDTH/4, HEIGHT*7/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Stats",25)
        self.statsButton = Button(self.game, "leaderboard", WIDTH*3/4, HEIGHT*7/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Leaderboard",25)
```

This is the function for the main menu screen. This is the screen from which all the other screens are accessed so therefore it has buttons to all the other screens. Theses buttons are temporarily positioned on the screen in a 2 by 3 table but will be repositioned to make the user interface designs.

```
63    def startScreen(self):
64        self.game.screen.fill((70,210,255))
65        self.game.drawText("Welome to my Game", 24, BLACK, WIDTH/2, HEIGHT*1/4)
66        self.game.drawText("Press a button to conitnue!", 12, BLACK, WIDTH/2, HEIGHT*3/4)
67        pg.display.flip()
68        self.waitForKey()
69        self.loadLevel('mainMenu')
```

This function is for the start screen. This will be the screen that the user first sees when launching that game. On this screen the user can click anywhere on the screen to continue to the main menu. To achieve this I created a function called 'waitForKey' which waits until the user provides any input. After this function completes, the user would have clicked on the screen or pressed a key and so the main menu is launched.

```
74    def waitForKey(self, key = True, click = True)
75        waiting = True
76        while waiting:
77            self.game.clock.tick(FPS)
78            for event in pg.event.get():
79                if event.type == pg.QUIT:
80                    waiting = False
81                    self.game.running = False
82                if event.type == pg.KEYUP and key:
83                    waiting = False
84                if event.type == pg.MOUSEBUTTONUP and click:
85                    waiting = False
```

Wait for response function

The 'waitForKey' function is shown below. This function has two parameters: key and click. By default both of these variables are set to true, meaning that the function will continue if a key or the mouse is pressed. These options can be changed by providing a false argument when the function is called. A while loop is used to keep looping until a key or the mouse is pressed and the variable 'waiting' is set to false. The function 'self.game.clock.tick(FPS)' so that the game keeps ongoing whilst it waits for an input. A for loop is used to iterate through the occurred events and section statements are used to check if these events are of type KEYUP or MOUSEBUTTONUP. The reason why the state when the buttons are released is checked is so that the screen doesn't instantaneously change when the input is provided. This could be a problem as if a button was in the same position on the successive page, it will also be triggered as the mouse button is held down. If an input is provided then the variable 'waiting' is changed to false to stop the loop. If the program is closed while on this screen then the event type will be 'QUIT', resulting in 'waiting' and the 'running' variable in the main file to be set to false, causing the program to quit.

```
93    def update(self):
94        self.game.buttons.update()
95        for button in self.game.buttons:
96            if button.clicked == True:
97                if button.tag == "startScreen":
98                    self.loadLevel('startScreen')
99                if button.tag == "levelSelect":
100                   self.loadLevel('levelSelect')
101               if button.tag == "shop":
102                   self.loadLevel('shop')
103               if button.tag == 'mainMenu':
104                   self.loadLevel('mainMenu')
105               if button.tag == "settingsMenu":
106                   self.loadLevel('settingsMenu')
107               if button.tag == 'leaderboard':
108                   self.loadLevel('leaderboard')
109               if button.tag == 'stats':
110                   self.loadLevel('stats')
111               button.kill()
```

Changing the screens

This is the 'update' function for the scene manager. Firstly the 'update' function is called on all of the buttons inside the buttons group. After this, a for loop is used to iterate through each button in the buttons group and check if it has been clicked. If this is true, the tag of the button is used to check which function needs to be called as a result of clicking that button. After this is determined the button that has been clicked is deleted from the buttons group using the '.kill' function. Inside the 'update' function in the button class, if a button was clicked, all the buttons apart from the clicked one was deleted from the buttons group. This means that the for loop would only have to loop through one button, thus making the code more efficient by reducing the number of checks required. Deleting the buttons is not a problem as the buttons will be created again when a page function is called.

```
sceneMan = sceneManager(self)
self.sceneMan = sceneMan
sceneMan.loadLevel('startScreen')
```

An instance of the scene manager was created in the 'new' function in the main game class. The 'self' parameter is an instance of the game. The start screen function is then called using the 'loadLevel' function in the scene manager class.

```
self.sceneMan.update()
```

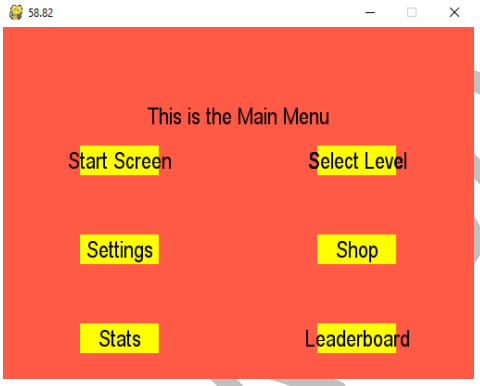
Inside the 'update' function in the main game class the 'update' function on the scene manager is called. Therefore, the buttons are updated every frame.

Review

Using a scene manager is a more efficient way of loading the levels than having functions in the main game loop as this is a self-contained module. This class reduces the amount of code on the main game file, making the remainder of the code easier to read and debug when necessary.

These are the screens that were made using the code above. Currently these screens are very basic however the buttons are functioning correctly so they can easily be made more ascetically pleasing. The buttons correctly change colour when the mouse cursor is over them and when clicked, the new page is loaded with the correct amount of buttons. Before I decided to remove the buttons from the group after one button was pressed, when new pages was loaded the buttons from the previous page remained on the screen and were functional.

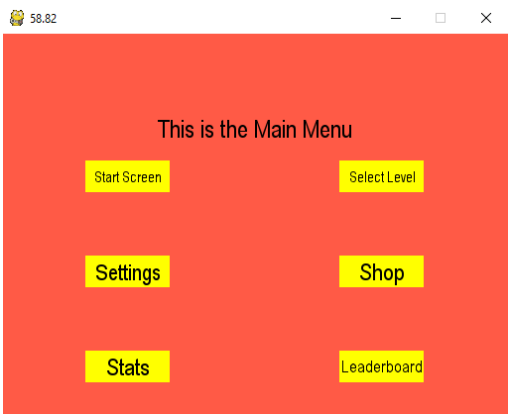
Errors during testing



When creating the buttons I had to make sure that the text fit inside the button. This was a trial and error process and I decided to make it more efficient by implementing a text size approximation function. This code is located in the initialisation function of the button class and is shown below.

```
if textSize == None:
    self.textSize = int(width/(len(text)*0.45))
    if self.textSize > 24:
        self.textSize = 24
else:
    self.textSize = textSize
```

The width of the button is used alongside the length of the text that needs to be drawn. This calculation is multiplied by the constant 0.45, which was arrived at by using trial and error. If the text size becomes larger than 24, then it is made equal to 24. This process only happens if a value for the text size is not provided when the button function is called. If it is, then the provided text size will be used. When using this function, the same screen above became the image shown on the left. As you can see, the size of the text gets smaller as the length of the text increases in order to fit into the same size button.



Changing the screens to user interface designs

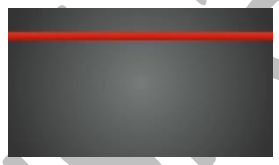
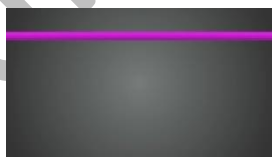
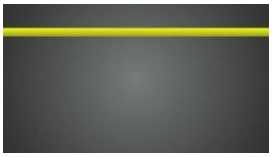
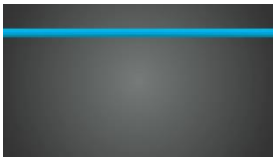
Now that all of the menu screens have been created their layouts need to be changed to the concepts created in the Design section. To do this images and fonts templates need to be imported using the 'loadData' function in the main game file.

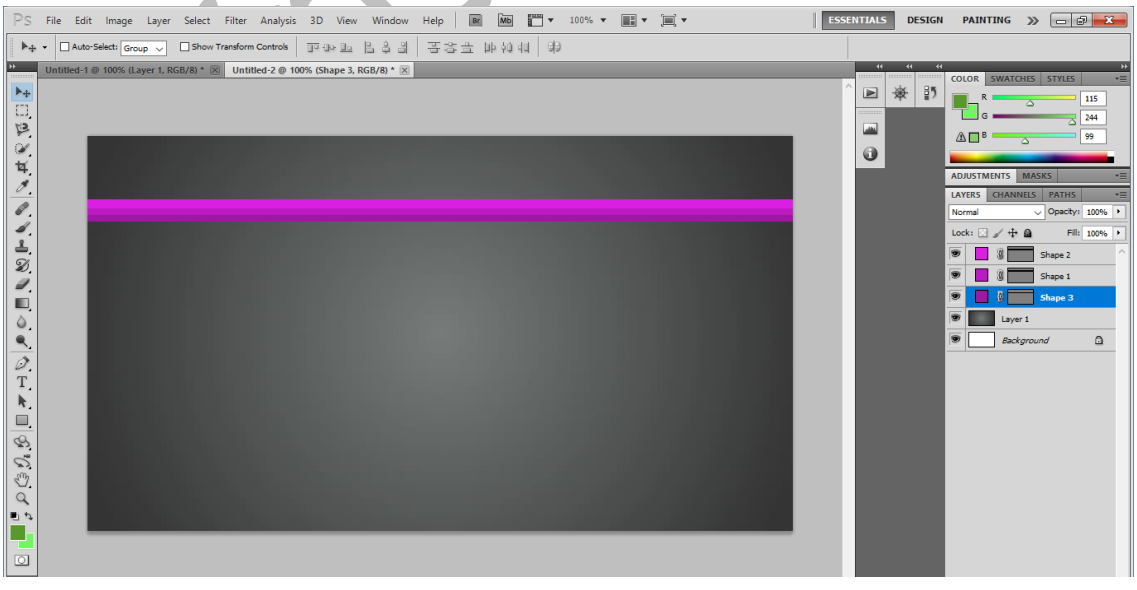
Loading the images and fonts

```
def loadData(self):
    gameFolder = path.dirname(__file__)
    imgFolder = path.join(gameFolder, 'img')
    self.menuButtonSolid = pg.image.load(path.join(imgFolder, 'blue_button01.png')).convert_alpha()
    self.menuButtonHighlight = pg.image.load(path.join(imgFolder, 'green_button01.png')).convert_alpha()
    self.interfaceFont = path.join(imgFolder, 'Future.ttf')
    self.buttonFont = path.join(imgFolder, 'PixelSquare.ttf')
    self.menuImages = {}
    self.menuImages["startScreen"] =  pg.image.load(path.join(imgFolder, 'grey_background.jpg')).convert_alpha()
    self.menuImages["mainMenu"] =  pg.image.load(path.join(imgFolder, 'blue_background.jpg')).convert_alpha()
    self.menuImages["settingsMenu"] =  pg.image.load(path.join(imgFolder, 'Grey_yellow_background.jpg')).convert_alpha()
    self.menuImages["levelSelect"] =  pg.image.load(path.join(imgFolder, 'Grey_blue_background.jpg')).convert_alpha()
    self.menuImages["stats"] =  pg.image.load(path.join(imgFolder, 'Grey_green_background.jpg')).convert_alpha()
    self.menuImages["leaderboard"] =  pg.image.load(path.join(imgFolder, 'Grey_violet_background.jpg')).convert_alpha()
    self.menuImages["shop"] =  pg.image.load(path.join(imgFolder, 'Grey_orange_background.jpg')).convert_alpha()
    self.menuImages["pause"] =  pg.image.load(path.join(imgFolder, 'Grey_red_background.jpg')).convert_alpha()
    self.pauseIMGWhite = pg.image.load(path.join(imgFolder, 'pauseWhite.png')).convert_alpha()
    self.pauseIMGBlack = pg.image.load(path.join(imgFolder, 'pauseBlack.png')).convert_alpha()
```

There are two images for the buttons, one for their normal non-highlighted colour and one for when the cursor is hovering over them. The images are loaded, as mentioned previously under 'Importing images and files', using the command 'pg.image.load()'. This command automatically creates a new surface object for the image and stores that in the variable. This means that surface object related methods, such as 'blit' and 'fill' can be called on these variables to refer to the images. The 'convert_aplha' method is called to convert the imported image to the same pixel format that is used by the screen. This ensures that performance is not lost when conversions take place in the process of drawing the images to the screen. There is another variant of this function called the 'convert' which removes the alpha (transparency) properties of the image however this command retains this information.

The fonts were not imported here but the file path to their location was stored in a local variable. This variable will be used later when the font is initialised in the 'drawText' function. I decided to store the images used for the menu screens in a list to organize the storage of them whilst also simplifying the code needed to determine which image to draw on which screen. The same command was used to load all of the background images for the screen and the pause button image.



These are the images for the menu screens that I imported. They are simple backgrounds which I created using Photoshop. As the game advances I will adapt their design if required.

Adding images to the buttons

```
if solidButtonImage == None:
    self.solidImage = pg.transform.scale(self.game.menuButtonSolid,(int(self.width), int(self.height)))
    self.highlightImage = pg.transform.scale(self.game.menuButtonHighlight,(int(self.width), int(self.height)))

else:
    self.solidImage = pg.transform.scale(solidButtonImage,(int(self.width), int(self.height)))
    self.highlightImage = pg.transform.scale(highlightButtonImage,(int(self.width), int(self.height)))
self.image = self.solidImage
```

This code was added to the '__init__' function of the 'Button' class. It replaces the previous code which created a rectangle and assigned it a solid colour. Two additional parameters named 'solidButtonImage' and 'highlightedButtonImage' were added to this class' '__init__' function. The purposes of these are for when different images are created for the buttons. The file path to the required images will be provided when the button instance is created. Temporarily, all of the buttons use the same image so the command 'if solidButtonImage == None' is used to check if this parameter is empty. If this is the case, the command 'pg.transform.scale' is used to resize the loaded button images to a new resolution. This is the size of the button, which is provided to this function in the form of 'self.width' and 'self.height'. The new images are stored in the local variables and then the image for the button's regular colour is assigned to the 'self.image' variable. The rest of the code functions as previously to change the image if the cursor is hovering over it and to draw the image to the screen.

Error Encountered - The text for the buttons was being drawn behind the images for the buttons. There was also a slight decrease in the frame rate, indicating that a part of the program was operating inefficiently and requiring excessive computation.

Solution - In the 'draw' function of the 'button' class the image of the button was being drawn to the screen every time the function was being called (which was in every frame).This command should only be called when the colour of the button image needs to be changed. Therefore, this command had to be moved into the 'if' statement which is executed if a change of image is required.

```
def draw(self):
    self.game.screen.blit(self.image,self.rect)
    if self.colchange == True or self.first == True:
        print("drawn", self.tag)
        if self.first:
            print(self.first, "self.first")
            self.first = False
        self.game.drawText(self.text, self.textSize, BLACK, self.pos[0], self.pos[1])
        self.colchange = False
```

Adding background images

This code was added to the 'update' function in the 'sceneManager' class. The reason for this is because the background images for the screens need to be drawn when the screen is changed and the code for changing the screens is present here. A variable called 'self.currentScence' is created to keep track of which screen is currently open. At the start the value of this variable is set to the 'startScreen' as this is the first screen that is shown to the user. If a button is pressed, the value of the current scene variable is set to the tag of the button that has been pressed. The image of the current screen is obtained from the main game file using the command 'self.game.menuImages[self.currentScene]'. This code can be used as the same naming scheme for the menu screens throughout the program. This is the benefit of storing the images for the game screens in a list otherwise, a series of selection ('if') statements will need to be implemented to determine the correct image to use. The 'transform.scale' function is used to resize the image to the size required for the screen. This required size is the dimensions of the game window, denoted by the constants 'WIDTH' and 'HEIGHT'. The rectangle covering the surface is created using the 'get_rect' function and stored in the variable 'rect'. This is then used to draw the image to the screen. This image does not constantly need to be drawn again as it is the background image. All of the other images are drawn on top of this and therefore, the code for drawing the background image only needs to be called once per screen.

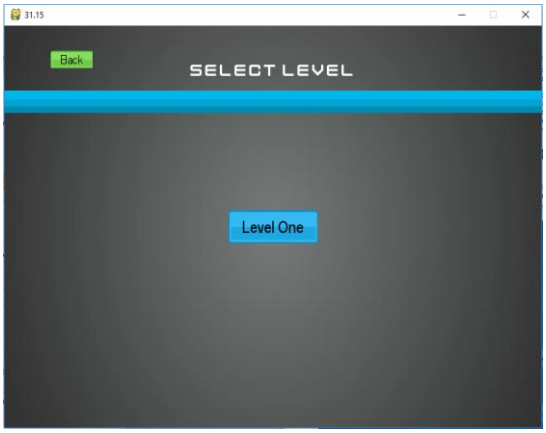
```
    for button in self.game.buttons:

        if button.clicked == True:
            self.currentScene = button.tag
            self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))
            rect = self.image.get_rect()
            self.game.screen.blit(self.image, rect)
```

Adding different fonts

An adaptation was made to the 'drawText' function in the main game file to incorporate font name as a parameter. Then this font name was used to initialise the font before rendering it to a surface. If this parameter is left empty then a default font of Arial is chosen. The variables assigned to the loaded fonts are used as parameters for the font name when calling this function. An example of when the 'drawText' function is called using the fonts is shown below. Two types of fonts have been loaded, one for the text used on the buttons and the other for the text used in the menu screens. As the game increases in complexity, more aesthetically pleasing fonts will be added to improve the appearance of the game.

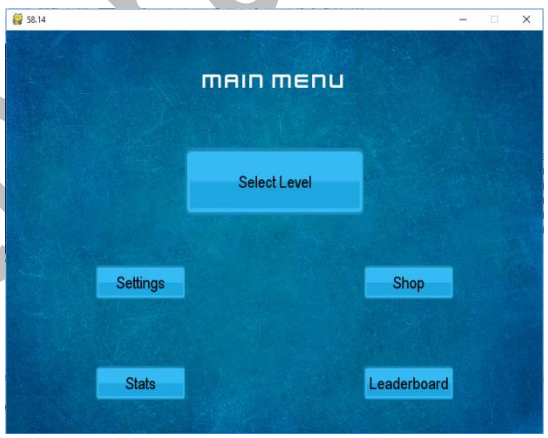
```
def drawText(self, text, size, colour, x, y, surf=None, align=None, fontName=None):
    if surf == None:
        surf = self.screen
    fontType = "C:\WINDOWS\FONTS\ARIAL.TTF"
    if fontName == None:
        font = pg.font.SysFont('arial', size)
    else:
        font = pg.font.Font(fontName, size)
    textSurface = font.render(text, True, colour)
    textRect = textSurface.get_rect()
```

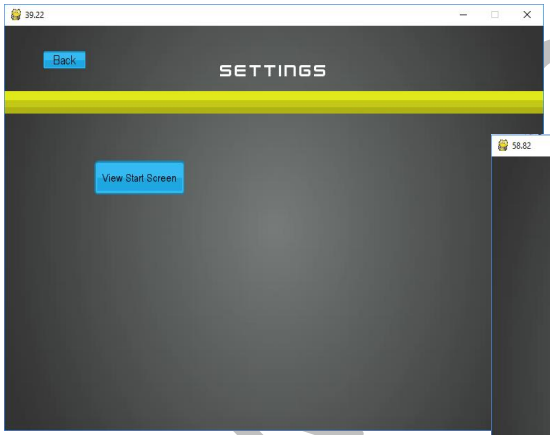
```
self.game.drawText("Select Level", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)
```



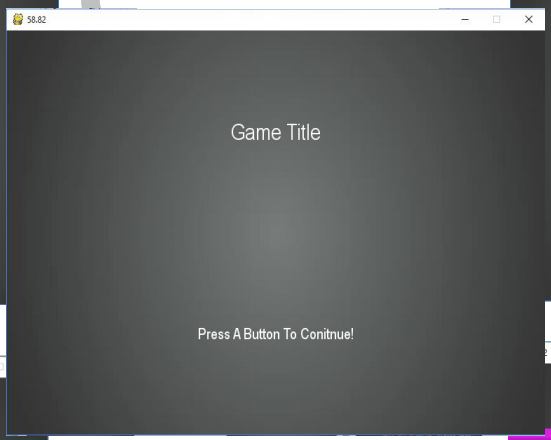
Review

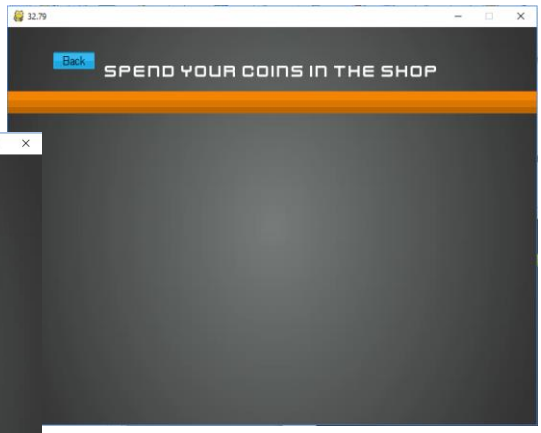
The main menu screens are not more similar to the designs of them created in the User interface designs section. The background image that has been created for each screen makes the game appear more polished as it provides a better aesthetic compared to a solid coloured background.

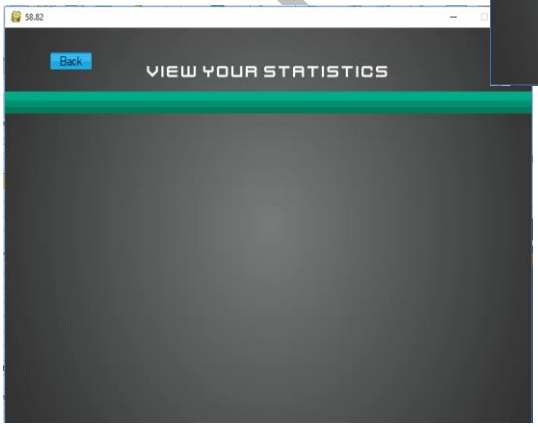


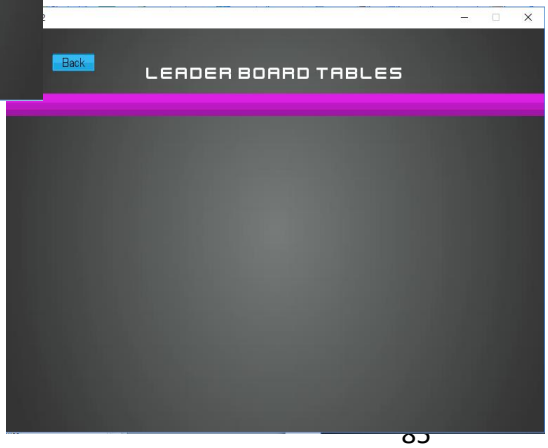


Start screen









Creating the car

The menu system has now been created and adapted to the designs that were initially created. Following the decomposition of the problem performed at the start of the Design section, the next stage of the development of the program is the creation of the car object. As this will be a new object in the game, it would be appropriate to create a class to manage all of its attributes and behaviours.

Creating the car class

This will be a new sprite to the game so this 'Player' class was created in the sprite file. Following the template for creating classes for game sprite, the simple bass class for visible game object ('pg.sprite.Sprite') is inherited from, and the sprite is added to the 'allSprites' group for simplified management. The parameters that are currently required when creating an instance of the player is an instance of the game, and the x and y coordinates of where the player will be positioned. For initial development a rectangle object with dimensions 40 pixels wide by 30 pixels tall is created to model the car. This rectangle was filled with the colour yellow and the rectangle covering the created surface is constructed. The position of the centre of the rectangle is set to the specified position. Next, variables to store the object's coordinate position, velocity and acceleration values are created. Vectors are created to store both components of these quantities in the same variable. The values for the velocity and acceleration are set to zero and the position of the object is set to the specified coordinates. A variable called 'direction' is created to keep track of which direction the car's motion is in. Left indicates a forward direction and right resembles a backwards/reverse direction.

```
class Player(pg.sprite.Sprite):
    def __init__(self, game, x, y):
        self.groups = game.allSprites
        pg.sprite.Sprite.__init__(self, self.groups)
        self.game = game
        self.image = pg.Surface((40, 30))
        self.image.fill(YELLOW)
        self.rect = self.image.get_rect()
        self.rect.center = (x,y)
        self.pos = vec(x,y)
        self.vel = vec(0,0)
        self.acc = vec(0,0)
        self.direction = "forward"
```

```
self.player = Player(self, WIDTH/2, HEIGHT/2)
```

This code was added to the 'new' function in the main file to create an instance of the player class. The initial coordinates of the car object are set at the middle of the screen.

Error Encountered - Now that an instance of the player class has been created, the yellow rectangle object is created and positioned at the middle of the screen however this is even visible when in the menu screens. The problem is clear as the player class has no awareness of the current scene.
Solution - A variable called 'showPlayer was created in the 'Scene Manager' class to determine when the player sprite should be drawn to the screen. A new screen called 'level1' was created where the car should be shown. The code shown was added to the end of the 'update' function in the 'Scene Manger' class and it sets the value of the 'showPlayer' variable to 'False' if the current screen is not the level screen. This ensures that the player object will not be drawn on the screens where it is not required. The update function of the player checks the state of this variable and only proceeds to update the player if this variable is 'True'.

```
if self.currentScene != "level1":
    self.showPlayer = False
```

Adding gravity to the car

To add an element of gravity to the car object the vertical acceleration of the object needs to be manipulated. It is not as simple as assigning the value $-9.81ms^{-2}$ to the y component of the acceleration as this would be too quick and thus seem unrealistic. It took a few attempts of trying out values for the gravity until I settled on a value which appeared to be suitable. I created a constant for this value and called it 'PLAYER_GRAV'. This is stored in the settings file and it has a value of 0.8. This value for the acceleration is later used when applying motion to the car.

```
def update(self):
    if self.game.sceneMan.showPlayer:
        self.game.screen.fill(BLACK)
        self.acc = vec(0, PLAYER_GRAV)
```

Creating a platform

Before the gravity feature can be tested a platform needs to be created for the car object to rest on when stationary. If this didn't exist the car will be created and then fall down continuously.

```
class Platform(pg.sprite.Sprite):
    def __init__(self, game, x, y, width, height, colour):
        self.groups = game.allSprites, game.platforms
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((width, height))
        self.image.fill(colour)
        self.rect = self.image.get_rect()
        self.rect.x, self.rect.y = x, y
```

As this will be a sprite a class called 'Platform' is created in the sprite file. A new group called 'platforms' is created under the 'new' function in the main file for the platforms. The platform is then added to the 'allSprites' and 'platforms' groups. The parameters that are used for this object include the coordinates of the position of the platform, the width and height of the platform and the colour of the platform. A rectangular surface is created to represent the platform and this is filled with the colour provided. The 'get_rect' function is then called on the surface and the position of the rectangle is set to the provided coordinates. An update function is not required for this game object as currently there is no need for the platforms to update in every frame.

```
def level1(self):
    self.showPlayer = True
    Platform(self.game, 0, HEIGHT*7/8, WIDTH, HEIGHT/8, BLUE)
    Platform(self.game, WIDTH, HEIGHT*7/8, WIDTH, HEIGHT/8, GREEN)
    Platform(self.game, 2*WIDTH, HEIGHT*7/8, WIDTH, HEIGHT/8, BLUE)
```

As mentioned, a new screen was created in the scene manager class for the levels. This is the code for level 1. The value of the 'showPlayer' variable is set to 'True' if this screen is loaded. Then three instances of the 'Platform' class are created. The platforms are positioned so that they line up with the bottom edge of the game window. I chose to implement three platforms of different colours to test this class as this difference in colour will indicate whether all the platforms have been correctly loaded.

Adding motion to the car (Equations of motion)

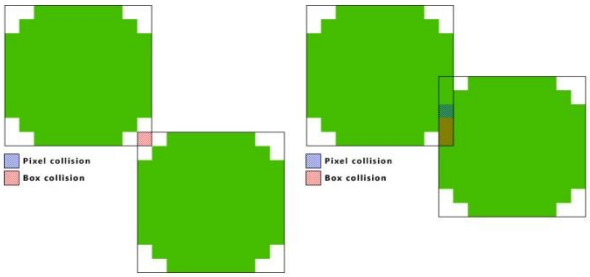
The equations that determine the movement of the car sprite are similar to the real world physics equations that are used to calculate displacement, velocity and acceleration. The commands will need to be executed in every frame of the game as they calculate the new position of the sprite using the inputs that have been provided. One such input is the coefficient of friction. This is a value that is used to replicate the effect that friction will have when travelling on a surface. In essence, this will eventually slow a sliding object down to a standstill. The value that I chose for this is –0.05.This value is multiplied with the velocity and then added to the acceleration. Afterwards, the value of the acceleration is added to the velocity. Following this the SUVAT equation $s = ut + \frac{1}{2}at^2$ is used to calculate the position of the sprite. Here the value of t is 1, making the equation 'self.pos += self.vel + $\frac{1}{2}$ self.acc'. After the position has been determined, the middle of the bottom of the bottom of the sprite is set to this new position. The sprite will then be drawn at this new position in the 'draw' function.

```
self.acc += self.vel * PLAYER_FRICTION

self.vel += self.acc
self.pos += self.vel + 0.5 * self.acc

self.rect.midbottom = self.pos
```

Adding collisions to the car

Now platforms for the car to rest on have been added to the level however currently, the car will simply pass over the platforms and continue falling. This is because there are currently no collision checks happening in the game. These checks will need to happen in every frame so the code for this goes in the 'update' function in the main file. A check needs to be made between the car and the platforms. Initial ideas of determining this collision seems inefficient and repetitive as a list of selection statements will need to be used to see if any collisions have occurred.

Pygame simplifies this procedure by using the 'pg.sprite.spritecollide()' function. A sprite and a group respectively are provided as parameters to this function. This command returns a list containing all the sprites in the provided group that intersect with the single sprite. These collisions are determined by comparing the '.rect' attributes of each sprite and checking if any of these rectangles overlap, which will indicate an intersection. There is an optional third parameter to this function which when set to the value 'True' will delete the sprite that has collided from the group by removing it from the group.



https://benjaminhorn.io/code/pixel-accurate-collision-detection-with-javascript-and-canvas/

There is a drawback to this method of checking for collisions and that is accuracy. The rectangle that is created for the sprites usually also covers a region which is not part of the sprite due to the nature of rectangles. As shown by the diagram on the left, this will mean that a box collision will be detected even when the sprites themselves actually have not collided. An alternative is to use pixel collisions however this much more resource and time intensive as many more checks will need to be made, especially if the group is large. The advantages of both types of collisions could be used by implementing box collisions and then when one is detected, use pixel collisions to check if the objects really have collided. I will first use the box collisions and then move onto implementing pixel collisions when I replace the car sprite's rectangle surface with images.

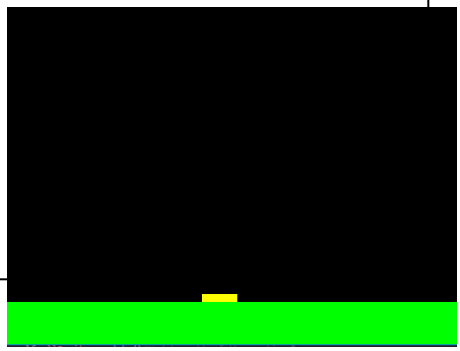
```
if self.sceneMan.currentScene not in MENU_SCREENS:
    hits = pg.sprite.spritecollide(self.player, self.platforms, False)
    if hits:
        self.player.pos.y = hits[0].rect.y
        self.player.vel.y = 0
```

This code was added to the 'update' function in the main file. A list of the names of the menu screens was created in the settings file as shown below. This now allows a check to be made using the 'currentScence' variable defined in the scene manager class of whether the current screen is a menu screen. This selection statement is used to reduce unnecessary computation of calculating whether any collisions are happening when the levels are not even being played. If the current scene is not one of the menu screens, i.e. a level is being played, then the 'spritecollide' function is called on the 'player' sprite and the 'platforms' group, with the platforms not being destroyed when a collision is detected. The command 'if hits:' will be executed if the variable 'hits' is not empty, which will only happen if a collision has occurred. In this situation, the y coordinate of the car's position is moved to the y coordinate of the collided platform's position. This is essentially putting the falling car sprite back to the top of the platform that it is attempting to fall through.

```
MENU_SCREENS = ["settingsMenu",
                "mainMenu",
                "levelSelect",
                "stats",
                "leaderboard",
                "shop",
                "startScreen",
                "gameOverScreen",
                "pause"]
```

Error Encountered - The car object kept traveling partially through the ground and then stopping. The car moved normally however it was not repositioning to the top of the platform that it has collided with. I made sure that car was in the platform by counting the visible pixels and there were 15 (half the height).

Solution - The code for the equations of motion was initially 'self.rect.center = self.pos'. This meant that the y coordinate of the car was acting at the centre of it and therefore the centre was positioned at the specified coordinate. This was fixed by changing the code from 'self.rect.center' to 'self.rect.midbottom' so that the middle of the bottom of the car was at the specified coordinate.



Error Encountered - The car object now rests above the ground line however it keeps going down and up really fast. It appears to be repeatedly going through the platform and then being reset back to the top again

Solution - The vertical velocity of the car had not been set to zero after the car had been repositioned to the top of the collided platform. This meant that the car retained its vertical velocity and used this to keep attempting to travel down. The collisions created were then detected and the car was repeatedly moved back to above the platform. This is what created that effect of the car going up and down really fast. This problem was fixed by adding the code 'self.player.vel.y = 0' to the end of the code after a collision with a platform was detected (which is the previously explained code).

Adding Forward, Brake and Reverse Keys

Now that code to move the car has been implemented, controls for manoeuvring the car can be coded. This will take place in the 'update' function of the 'player' class. First, the inputs that have been recorded in the previous frame need to be obtained. This is achieved using the 'pg.key.get_pressed()' command, and the recorded events are stored in the 'keys' variable. A list of the recorded events using a predefined template for the keys is stored in this variable.

```
def update(self):
    if self.game.sceneMan.showPlayer:
        self.game.screen.fill(BLACK)
        self.acc = vec(0, PLAYER_GRAV)
        keys = pg.key.get_pressed()
```

After this selection statements re used to determine if the right arrow key or the 'd' key have been pressed to move the car to this right. If 'if key[pg.K_RIGHT]' command checks if there is at least one element of 'pg.K_RIGHT' present in the list. When this happens, the x component of the acceleration of the car is changed to cause the equations of motion to create a forward movement. The value that the acceleration is changed to is 0.6 and this is a constant defined in the settings file. This value was arrived at by trying different values until I found one which felt appropriate. A higher value for the acceleration causes the car to move too quickly and a lower values makes the movement seem too slow. The value of the 'direction' variable is changed to 'forward' to indicate that the car is moving in that direction. When the left arrow key of the 'a' key is pressed, the car is moved to the left. This is achieved by again changing the x component of the acceleration to the -0.6. The direction variable is changed to 'reverse' to indicate that the car is travelling to the left.

```
if keys[pg.K_RIGHT] or keys[pg.K_d]:
    if self.direction == "stopped" or self.direction == "forward":
        self.acc.x = PLAYER_ACC
        self.direction = "forward"
```

```
if keys[pg.K_LEFT] or keys[pg.K_a]:
    if self.direction == "stopped" or self.direction == "reverse":
        self.acc.x = -PLAYER_ACC
        self.direction = "reverse"
```

The car can now move however when the left and right arrow are pressed quickly after each other there is an instantaneous change in direction. This is not realistic and makes the car much easier to control. In order to increase the difficulty of manoeuvring the car I decided to add a braking function. This will be allocated to the down arrow and the 's' key and when pressed will apply a negative acceleration on the car to slow it down. The value of this braking acceleration is 0.2 and the direction of this acceleration is changed depending on the direction that the car is travelling at. The value of the velocity is calculated to many decimal points so it is rounded to the nearest integer using the 'round' function. If the velocity is not equal to zero (i.e the car is in motion), then the acceleration is changed to negative 'BRAKE_ACC' if the car has a positive horizontal velocity and changed to positive 'BRAKE_ACC' if the car has negative horizontal velocity. The value of the direction variable is changed to 'stopped' if the car's horizontal velocity is zero.

```
#Player properties
PLAYER_ACC = 0.6
BRAKE_ACC = 0.2
PLAYER_FRICTION = -0.05
PLAYER_GRAV = 0.8
```

```
if keys[pg.K_DOWN] or keys[pg.K_s]:
    if round(self.vel.x, 0) != 0:
        if self.vel.x > 0:
            self.acc.x = -BRAKE_ACC
        else:
            self.acc.x = BRAKE_ACC
if round(self.vel.x, 0) == 0:
    self.direction = "stopped"
```

The direction variable is now used when the forward and reverse keys are pressed to check if the car's momentum. If the car is already moving in the required direction or it has stopped then the code to maintain the forward acceleration is executed. This ensures that a car moving in the forward direction cannot instantaneously start reversing. The car first has to come to a stop by braking and then it can change direction. This advanced motion implementation is more realistic and makes the car more challenging to control. This increased skill requirement will make the game more enjoyable to play.

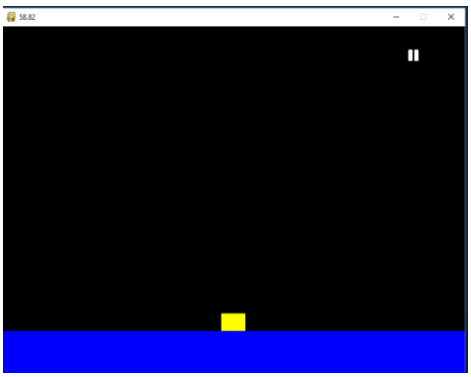
Testing the controls

For testing purposes I added a jump feature to the car. This is a simple function in the 'player' class which changes the vertical velocity of the car to '-20' when the space bar key is pressed. The code shown on the right was added to the 'events' function in the main file.

```
def jump(self):
    self.vel.y = -20
```

```
if event.type == pg.KEYDOWN:
    if event.key == pg.K_SPACE:
        self.player.jump()
```

I implemented some 'print' statements to print the direction, velocity and acceleration of the car to the console. This screen shot of the console log shows the changing direction, velocities and acceleration of the car when the direction control keys are pressed. The image of the game window below shows the car on the platforms that were previously created. It is difficult to show that the motion is working correctly in image form which is why I have used the values in the variables.

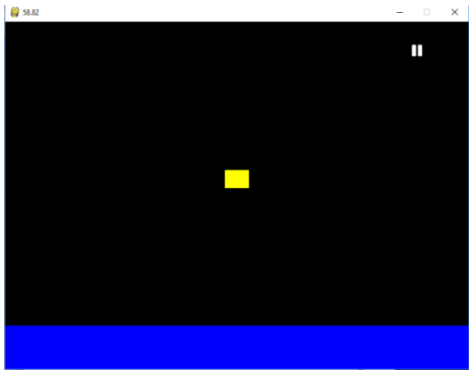


```
forward direction  Velocity: 17.0 Acceleration: 2.0
forward direction  Velocity: 16.0 Acceleration: 2.0
reverse direction  Velocity: 0.0 Acceleration: -2.0
stopping  Velocity: -1.0 Acceleration: 0.0
reverse direction  Velocity: -1.0 Acceleration: -2.0
stopping  Velocity: -3.0 Acceleration: 0.0
reverse direction  Velocity: -3.0 Acceleration: -2.0
stopping  Velocity: -4.0 Acceleration: 0.0
reverse direction  Velocity: -4.0 Acceleration: -2.0
stopping  Velocity: -6.0 Acceleration: 0.0
reverse direction  Velocity: -6.0 Acceleration: -2.0
stopping  Velocity: -7.0 Acceleration: 0.0
reverse direction  Velocity: -7.0 Acceleration: -2.0
stopping  Velocity: -8.0 Acceleration: 0.0
reverse direction  Velocity: -8.0 Acceleration: -2.0
stopping  Velocity: -9.0 Acceleration: 0.0
reverse direction  Velocity: -9.0 Acceleration: -2.0
reverse direction  Velocity: -10.0 Acceleration: -2.0
```

As shown, as the stopping key is held down the braking acceleration gradually gets added to the car's acceleration to decrease its speed, further slowing it down.

```
stopped direction  Velocity X: -0.0 Acceleration X: 0.0
                   Velocity Y: 1.0 Acceleration Y: 1.0
stopped direction  Velocity X: 1.0 Acceleration X: 2.0
                   Velocity Y: 1.0 Acceleration Y: 1.0
forward direction  Velocity X: 11.0 Acceleration X: -1.0
                   Velocity Y: 1.0 Acceleration Y: 1.0
forward direction  Velocity X: 7.0 Acceleration X: -0.0
                   Velocity Y: -18.0 Acceleration Y: 2.0
forward direction  Velocity X: 3.0 Acceleration X: -0.0
                   Velocity Y: -6.0 Acceleration Y: 1.0
forward direction  Velocity X: 1.0 Acceleration X: -0.0
                   Velocity Y: 2.0 Acceleration Y: 1.0
stopped direction  Velocity X: 0.0 Acceleration X: -0.0
                   Velocity Y: 7.0 Acceleration Y: 0.0
stopped direction  Velocity X: 0.0 Acceleration X: -0.0
                   Velocity Y: -7.0 Acceleration Y: 1.0
```

This is the console log when the 'jump' function is used to test the vertical motion and effects of the implemented gravity. As shown by the green box, the vertical velocity of the car correctly decreases to indicate that there is motion upwards. This is shown as negative velocity due to the nature of how the pixels on a computer screen are referenced. More information about this was explained in the 'Referencing coordinates on the screen' section. From this testing I can conclude that the motion and gravity implemented for the car correctly functions.



Review

Movement functionality has now been added to the car object. This means that the car can be navigated around the track by the player as required. The accelerator and brake controls need to be added as on screen controls for the mobile users of this program and this will be enforced after the levels have been designed, as this will reveal the space available for these controls. A method of loading and drawing more complex levels needs to be created. This includes angled platforms, with correctly functioning collision code, and possibly a smooth, curved track. This problem will be broken down to create possible solutions. Graphics also needs to be added and any deficiencies which are revealed will need to be optimised.

Implementing a Camera object

The car can now be moved however when it gets to the edge of the screen it disappears. This is because the game window is stationary and not showing the car as it travels beyond the width of the screen. This needs to be solved by implementing a camera for the game. The purpose of this will be to move the screen so that the car will always be visible. This allows levels to be created as the user can follow the car across the track.

Defining the camera class

This camera object will be a game management device and so I created the 'Camera' class in the 'sceneManager' file, which I renamed to 'management'. The command 'pg.Rect()' is used to create objects to store and manipulate rectangular areas, which in this case is the game window. The dimensions of the rectangle are provided as parameters to this function and they are the size of the game window. The 'width' and 'height' parameters are stored in local class variables. A camera object was initiated in the 'new' function in the main file. A width parameter of 10 times the size of the game window was provided. This means that the camera will keep following the car until it reaches an x coordinate of 10 times the size of the game window. After this point the car will not be followed however later code can be implemented to enforce a boundary at this point to not allow the car to be lost.

```
class Camera:
    def __init__(self, width, height):
        self.camera = pg.Rect(0, 0, width, height)
        self.width = width
        self.height = height
```

```
self.camera = Camera(10*WIDTH, HEIGHT)
```

Applying offset to objects

The concept of the camera object is to move the objects on the screen by applying a certain offset depending on the movement of the car. This essentially maintains the car at the centre of the screen and moves all the other sprites (such as the platforms) around it. A function for applying this offset is created, taking the sprite to be manipulated as a parameter. The 'rect' object of this sprite is repositioned using the 'move' function. This command returns a new rectangle that is moved by the offset provided as a parameter. This offset is the coordinates of the top left of the camera rectangle. This function is called when the sprites are being drawn in the 'draw' function in the main file. The command 'self.allSprites.draw(self.screen)' is replaced with what this command essentially performs. This is an iterative loop through all of the sprites in the 'allSpries' group and drawing each one to the screen using the 'blit' function. When performing this, the offset is applied to the rectangle of the image. This means that the sprites on the game window will now be drawn in relation to the position of the car sprite.

```
def applyOffset(self, item):
    return item.rect.move(self.camera.topleft)
```

```
if self.sceneMan.currentScene not in MENU_SCREENS:
    for sprite in self.allSprites:
        self.screen.blit(sprite.image, self.camera.applyOffset(sprite))
```

Error Encountered - The camera didn't scroll across the screen as expected. The offset was correctly being applied to the sprites in the game however no effect was being shown.

Solution - When the instance of the camera object was initialised in the 'new' function in the main file, the parameters that I provided were 'WIDTH' and 'HEIGHT'. This implies that the size of the map is the size of the game window, which is incorrect. The parameter should be '10*WIDTH' and 'HEIGHT' so that the map is larger in the horizontal direction. The car can now travel horizontally across the game window and the camera scrolls as expected.

Updating the Camera object

The camera object needs to be updated in every frame so that the position of the car can be tracked and monitored to move the camera object is necessary and apply offset. In the 'update' function the sprite to track (which is the car) is taken as a parameter. The x and y components of the car's position are subtracted from the x and y coordinates of the centre of the screen. This essentially moves the car to the centre of the screen as its initial positive is subtracted to zero. The succeeding statements using the 'min' and 'max' functions are designed for when the car is at the boundaries of the game window. At these points the car cannot be at the centre of the screen so the coordinates of the camera are recalculated. After this a new rectangle object is created for the camera with the x and y coordinates that have been calculated.

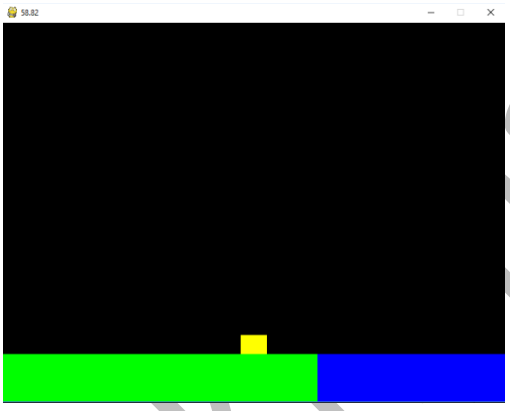
```
def update(self, target):
    x = -target.rect.centerx + int(WIDTH/2)
    y = -target.rect.centery + int(HEIGHT/2)
```

```
    #Limit scrolling to map size
    x = min(0, x) #Left
    y = min(0, y) #top
    x = max(-(self.width - WIDTH), x) #right
    y = max(-(self.height - HEIGHT), y) #bottom

    self.camera = pg.Rect(x, y, self.width, self.height)
```

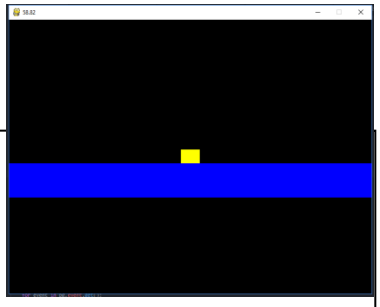
```
self.camera.update(self.player)
```

This command is called in the 'update' function in the main file, meaning that it is called in every frame. It calls the update function in the 'camera' class and provides the car sprite as a parameter. This means that the movement of the car sprite are tracked and the offset applied is revolved around the position of this sprite.

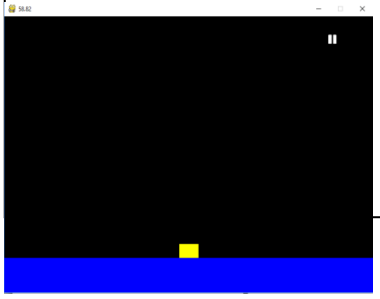
The properties of the other objects are not changed so this will not affect any of the previously written code, such as the collision code. From the point of view of the objects they are at the same location however they are being drawn at a different location due to the offset. Furthermore, this implementation of the camera means that any sprite can be tracked. If desired the parameter provided when updating the function can be changed to another game sprite and the camera will then move with the chosen sprite.



This screen shot of the game window shows that camera is functioning correctly and the car can now be seen when it travels across the different platforms.



Error Encountered - At the boundaries of the game window the car will still being positioned at the middle of the screen. The image shown is the position of the car after it is spawned. The car is positioned in the middle of the screen and this is causing the platform's position to be moved.

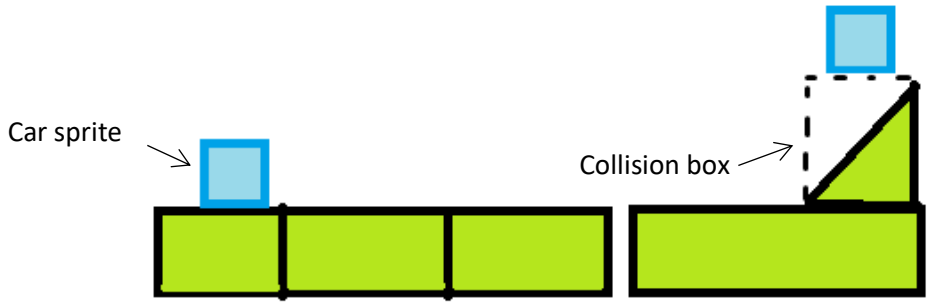


Solution - I added the 'min' and 'max' function to recalculate the offset at the boundaries of the game window. These function meant that the car wasn't at the centre of the screen anymore and the layout of the sprites was as expected.

Making the Levels

Breaking down the problem

An efficient method of creating and loading created levels needs to be added to the game. Problems which this method will need to solve include adding graphics to the levels and the background, correctly detecting collisions between multiple sprites and correctly moving the car sprite as a result of collisions. The collision logic that is correctly used works using rectangular objects. This works as expected with platforms which are correctly horizontal however levels created using only horizontal platforms are not difficult to complete as all the player will need to do is move the car to the right. Therefore angled platforms needs to be added so jumps and descents can be added to the levels. However rectangular collisions do not work with angled platforms as they will construct the 'rect' object for the angled platforms as follows:

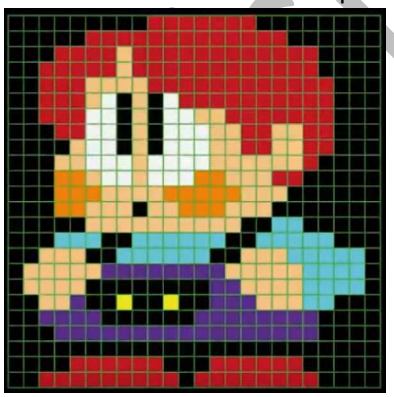


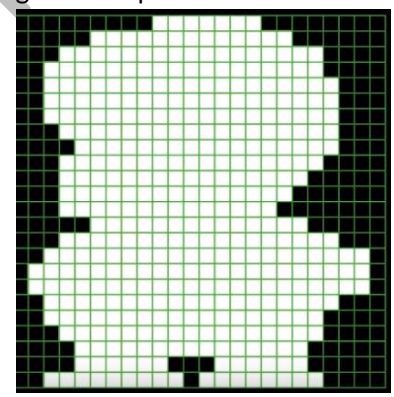
This diagram shows how when a rectangular collision box is used for angled platforms, the collision code doesn't function as required. When the car sprite is on the rectangular platforms, the collision boxes fit into the sprites image as both shapes are rectangular. Therefore the collisions between the car and the platforms are correctly detected and the car is moved to the top of the platforms. However when an angled surface is used, the rectangular collision box created for the sprite is no longer fully enclosed in the image of the sprite. This causes the current collision code to move the car sprite to the top-middle of the collision box (as shown above), creating an incorrect hovering car effect. This means that the car will not be able to smoothly slide on these angled surfaces.

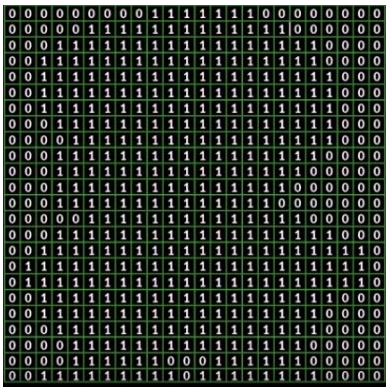
Using Masks and Pixel Perfect collision

A possible solution to this problem is by using masks and pixel perfect collision to detect the collisions between the car sprite and the platforms. Mask can be made from images and are used to detect whether two sprite images overlap.

Source - https://bit.ly/2QjV2fA





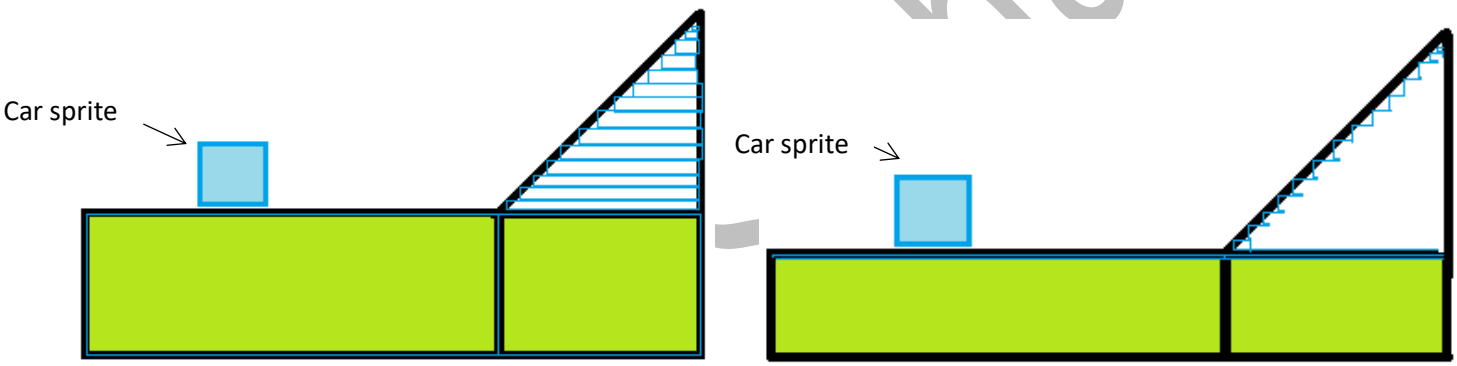


The pixels of the original sprite image and the box collider of this sprite are examined and a new image is created. In this new representation, the parts of the sprite are represented in white and the parts of the background are represented in black. As there are now only two colours, white is represented as a 1 and black is represented as a 0. A mask for the image has now been created. Masks for the other sprites are also created and then the 'and' bitwise manipulation is performed on all the sprites. If there is a 1 (from one sprite) AND 0 (from another sprite) then there is no collision. However if there is a 1 AND 1 then there is a collision.

The draw back to using this method for detecting the collision between the platforms and the car is that there will be a lot of platforms in the level meaning that a lot of comparisons will need to be made. This increased amount of computation may cause the program to slow down. This could be optimised by only checking for collisions between the car sprite and the platforms currently in the game window. Additionally, a method to only use the top layer of the platform images could be implemented to reduce the computation required. If boundaries are added at the vertical edges of the game window then the car will never be able to interact with the surface of the platforms below the top layer, meaning that its purpose will be entirely for decorative purposes.

Using Pixel-wide collision boxes

Another possible solution to this problem is by creating a lot of small rectangular platforms to symbolise a curve. The map could be drawn in a graphics editing program (such as Microsoft Paint or Adobe Photoshop) and saved as a bitmap file. This is a file format where the image is composed of a matrix of dots each representing a single pixel. If the track is drawn in only two colours, then it can be stored as a Monochrome Bitmap, allowing it to be loaded and read by pygame. Platform objects which are pixels wide can then be created at the points where the track is drawn. The rectangular box collisions can then be used to keep the car sprite on the platforms.

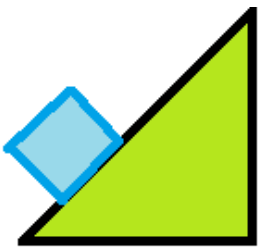


The blue boxes in the diagrams represent the collision boxes of the platforms. The first diagram shows how rectangular collision boxes can be added to angled surfaces. The collisions between the car sprite and the platforms can now be checked and the car can be moved to the top-left of the platform which it collided with. As the collision boxes will only be added to the top layer of the platforms, they will actually appear as shown in the diagram on the right.

With this method two maps of the level could be created. One map could be used by the program to perform collision checks using the car sprite. The other map could be for the player and contain all the graphics of the level, such as the images of the sprites and background elements, and could be the map that they interact with.

Another problem faced by both these method and the masks method is the rotation of the car sprite. When the car is on an inclined platform, its image will need to be rotated by a certain amount so that the base of the image aligns with the top of the inclined platform. Otherwise, the car sprite will appear as shown in the diagram on the left. The angle by which the car needs to be rotated needs to be calculated. This problem will be tackled after a successful method of creating and loading the levels is achieved. The current idea for the solution to this problem is by using vectors corresponding to the velocity and position of the car and the position of the collided platforms to calculate the dot product and then the angle to rotate.

Incorrect



Correct

All the diagrams on this and the previous page, excluding the credited one, were drawn by me.

Implementing a Map Object

After considering both of the solutions proposed above to the collision-detection problem I have decided to use the pixel-wide collision box approach. This is because the masks and pixel perfect collision method will be able to correctly determine the point of contact between the car sprite and the platform however, the problem of repositioning the car sprite will still be present. This is because the collision point will be inside the rectangular box for the platform and the car sprite cannot be repositioned to be inside the platform sprite. As a result of this, this approach will not solve the current problem however this will be used later in the program to detect whether the car has landed on its roof and failed the level.

In order to use the pixel-wide collision box method, a procedure of loading the levels needs to implemented. The simplest method of achieving this is by using the tile map technique to design the levels. This approach consists of building the level map out of small images called tiles which are the same pre-determined, square size. The benefit of this is that large image files for the entire level are not needed as the smaller images are used, resulting in better performance and memory usage gains. As the first step, I created a Map object.

Creating the Map class

This new class was created in the management file and takes a parameter for the location of the map file. The concept is that the map will be created in a text file using a series of characters, and later the text file will be analysed and several different objects will be spawned at various locations depending on the level data. An array called 'data' is created to store the data to be extracted from the text file. Next the map text file is opened in read text mode and saved as a local object called 'file'. Next every line in the text file is iterated over and the data is added to the 'data' array. The '.strip()' method is used to remove the whitespace from the lines in the text file and only store the useful data. After this, the '.close())' method is called on the file object to close the text file to prevent any accidental changes to the level data. The constant 'TILESIZE' was added to the settings file and the value 32 was assigned to it. This is the size of all the tiles that are going to be used.

```
class Map:
    def __init__(self, filename):
        self.data = []
        with open(filename, 'rt') as file:
            for line in file:
                self.data.append(line.strip())
        file.close()
```

```
TILESIZE = 32
```

After the data from the file has been extracted variables called 'tilewidth' and 'tileheight' were created to store the dimensions of the level in terms of the number of tiles. As the level data is stored in a two dimensional array, the length of the array determines the number of rows and thus the height of the level. The length of the first row determines the number of columns and thus the width of the level. These values are then multiplied by the TILESIZE constant and stored in new, separate variables to be used by the camera object.

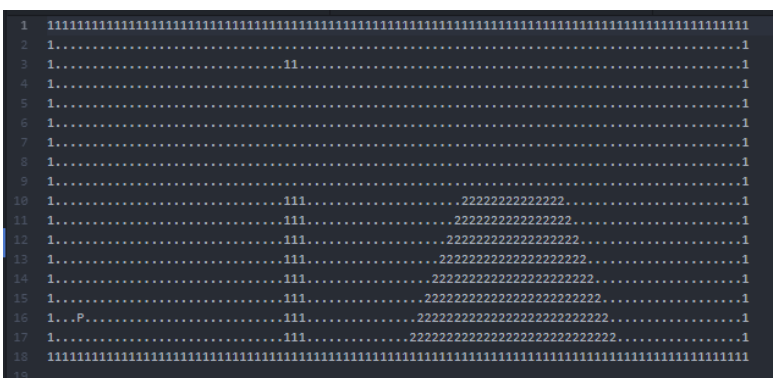
```
self.tilewidth = len(self.data[0])
self.tileheight = len(self.data)
self.width = self.tilewidth * TILESIZE
self.height = self.tileheight * TILESIZE
```

```
self.map = Map(path.join(self.gameFolder,'map2.txt'))
self.camera = Camera(self.map.width, self.map.height)
```

Inside the 'new' function in the main file an instance of the 'Map' class was created and stored in the variable 'self.map'. A camera object was then created using the dimensions of the level from the variables defined in the 'map' class.

Making the level in a text file

The screen shot on the right shows the text file that was used to create the level. Each row has 95 characters and there are 18 rows. 1 refers to each location on the map where there needs to be a wall/platform, 2 refers to where there needs to be a platform, and P is where the car sprite spawns. The dots are used to fill up the empty spaces between the other objects.



Loading and reading the level

```
def level1(self):
    self.showPlayer = True
    for row, tiles in enumerate(self.game.map.data):
        for col, tile in enumerate(tiles):
            if tile == "1":
                Platform(self.game, col, row, 50, 50, GREEN)
            if tile == "P":
                self.game.player = Player(self.game, col, row)
```

The 'level' function in the 'scenemanager' class was altered to adapt to the new map object. Instead of simply spawning three platforms, this function now iterates through the 2-dimensional array containing the level data. It assigns the value of the index of each array within the 2-dimensional array to the name 'row, and 'tiles' to the data of each row. The 'enumerate()' function is used to achieve the assignment of variables as this keeps a counter of the index when t iterating over an array. Next, each row is iterated over, assigning the value of the index to the name 'col' (short for column), and the data at this point to the name 'tile'. Following this, a series of selection statements are used to check if the character at that point on the level matches any of the pre-determined criteria. If there is a 1 then a platform object is created with the location at the values of 'col' and 'row respectively. If there is a P then the 'player' variable in the 'game' class is changed to a player object created using the x and y coordinates created using 'col' and 'row'. In order to achievement the assignment of the variable in the 'game' class, I set the value of the player variable in the 'new' function to None.

```
self.player = None
```

I ran the program to test the 'map' class however it did not work as expected. I had to solve a series of problems as before finally reaching a working solution

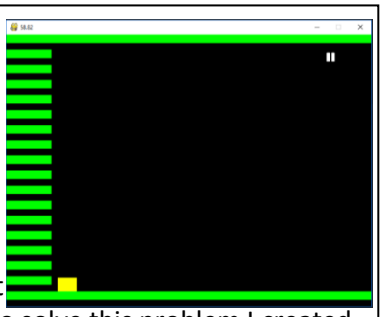
Error Encountered - The map has not loaded as expected. The platforms which have been spawned are in the incorrect location and appear to be squished together. The car sprite is correctly created but quickly falls and disappears off the screen due to gravity acting on it.

Solution - Due to the addition of the TILESIZE constant and the game window being split up into tiles, the incorrect coordinates were being provided when creating the platforms. The coordinates that were being provided were very close together, causing the platforms to be spawned on top of each other. This problem was fixed by multiplying the x and y coordinates which were parsed in the parameters by the TILESIZE constant.

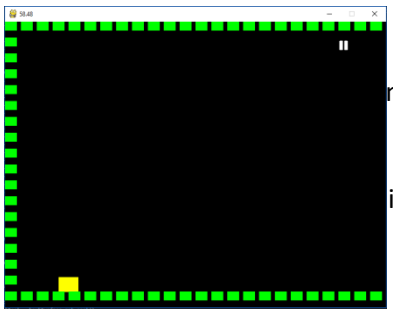
```
def level1(self):
    self.showPlayer = True
    for row, tiles in enumerate(self.game.map.data):
        for col, tile in enumerate(tiles):
            if tile == "1":
                Platform(self.game, col*TILESIZE, row*TILESIZE, self.game.map.tilewidth, self.game.map.tileheight, GREEN)
            if tile == "P":
                self.game.player = Player(self.game, col*TILESIZE, row*TILESIZE)
```

The values for the platform's width and height have also been changed to the number of tiles wide the level is and the number of tiles high the level is. The game window now looks like the screenshot on the left. The level looks more similar to how it should however there are still gaps between the platforms on the walls. I adapted the initialization code in the platform class to multiple the values of the x and y coordinates by TILESIZE before they are assigned to the platform's 'rect' object. This reduces the likelihood of an error occurring when a platform object is created.

Error Encountered - There gaps between the platforms on the walls and the platforms appear to have the wrong dimensions. This is because the platforms on the wall should only be 32 pixels wide but are actually longer than that.



Solution - The width and height values of the platforms are incorrect. This is because the number of tiles wide the level is, is not how wide each platform should be. The width of the platforms should be the total width of the level divided by the width of each tile.

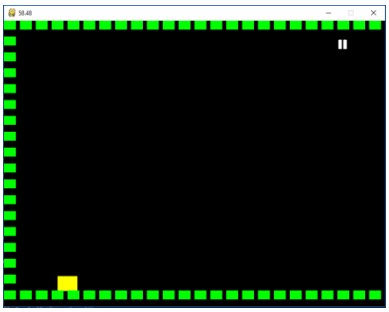


Similarly the height of each tile should be the total height of the level divided by the height of each tile. Therefore to solve this problem I created nstants in the settings file called GRIDWIDTH and GRIDHEIGHT and assigning them as the parameters for the platforms width and height. The platforms now appear to be the correct ions however there is a gap between each adjacent platform

```
GRIDWIDTH = WIDTH / TILESIZE
GRIDHEIGHT = HEIGHT / TILESIZE
```

```
for col, tile in enumerate(tiles):
    if tile == "1":
        Platform(self.game, col, row, GRIDWIDTH, GRIDHEIGHT, GREEN)
```

Error Encountered – The map is still not loading correctly. The only logical cause of the gaps between the platforms would be due to the dimensions of the platform being incorrect.

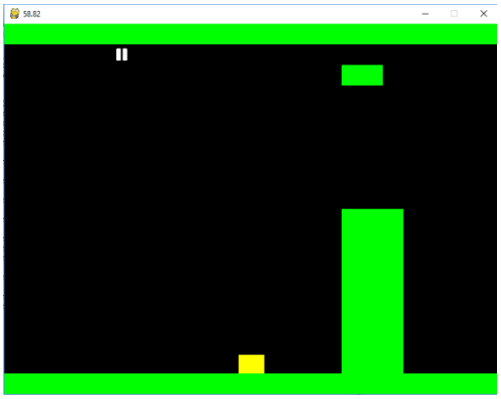


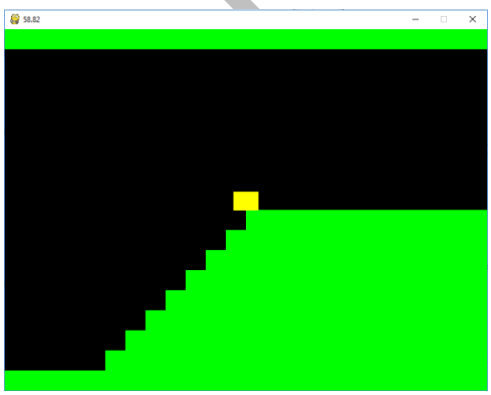
Solution -. After investigating the code surrounding the spawning of the platforms I realised that the width and height of the platforms need to be the size of the tiles. Therefore, I replaced GRIDWIDTH and GRIDHEIGHT in the parameters with TILESIZE. This solved the problem and now the level is correctly loaded.

```
for col, tile in enumerate(tiles):
    if tile == "1":
        Platform(self.game, col, row, TILESIZE, TILESIZE, GREEN)
```

Review

The map has now been successfully loaded and platforms have been spawned at all the required locations corresponding to the information in the map text file. As the camera object works using the dimensions of the map, as the car moves across and up the level it will be correctly followed.





The car can travel up the slope however the movement is bumpy and the graphics for the slope are not clear. As a method of created and loading game levels has been created, in the next stages of development, pixel-wide platforms need to be implemented. When this is achieved, the slopes in the level will be more gradual and genuine. Additionally if the car is rotated depending on the gradient of the slope then the movement will also appear to be more real.

Correcting the collisions between sprites

The collisions between the platforms and the car have been coded so that the car is always brought to the top of the platform when there is a collision. This works for the floor and the future track however this is not convenient on the walls or on the top of the level. This is because when the car collides with the sides or the top, it will be moved to the top of these platforms and then it cannot re-enter the level. To solve this issue, I created an adaptation of the platform object called a wall.

Creating a Wall object

The wall object is essentially the same as the platform object however the collision code between the walls and the car sprite is going to be different. In order for this to happen, I created a new group in the main file's 'new' function called 'walls', and added each instance of the wall object to that group.

```
class Wall(pg.sprite.Sprite):
    def __init__(self, game, x, y, colour):
        self.groups = game.allSprites, game.walls
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((TILESIZE, TILESIZE))
        self.image.fill(colour)
        self.rect = self.image.get_rect()
        self.rect.x, self.rect.y = x * TILESIZE, y * TILESIZE
```

```
self.walls = pg.sprite.Group()
```

The remainder of the code is the same as the code used to create a platform however the platform class could not be inherited as otherwise the wall object will be added to the platform group, causing the initial problem to return.

Collide with Walls function

This function was added inside the car class in the sprites file and the reasoning for this is because the car object's position will need to be changed depending on whether a collision occurs between it and a wall. A parameter called 'dir' (short for direction) is used in this function to signify whether to change the car's horizontal or vertical position.

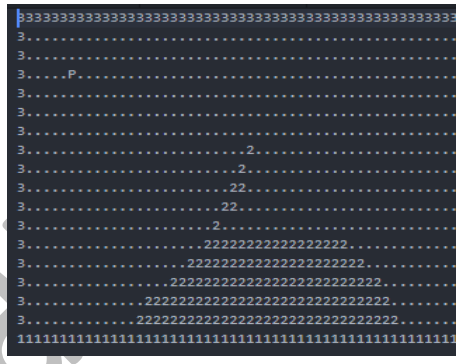
```
def collideWithWalls(self, dir):
    hitWall = pg.sprite.spritecollide(self.game.player, self.game.walls, False)
    if hitWall:
        if dir == "x":
            if self.game.player.vel.x > 0:#going to the right  [hit left of wall]
                self.game.player.pos.x = hitWall[0].rect.left - self.game.player.width / 2
            elif self.game.player.vel.x < 0:#going to the left [hit right of wall]
                self.game.player.pos.x = hitWall[0].rect.right + self.game.player.width / 2
            self.game.player.vel.x = 0
            self.rect.centerx = self.pos.x

        if dir == "y":
            if self.game.player.vel.y > 0:#going down [hit top of wall]
                self.game.player.pos.y = hitWall[0].rect.top - self.rect.height / 2
            elif self.game.player.vel.y < 0:#going up [hit bottom of wall]
                self.game.player.pos.y = hitWall[0].rect.bottom + self.game.player.height / 2
            self.game.player.vel.y = 0
            self.rect.centery = self.pos.y
```

Firstly, the 'spritecollide' function is used to check if a collision has occurred between the player sprite and any sprites in the wall class. Next, the value of the 'dir' parameter is checked and this could be either equal to 'x' (i.e. to check if the car has collided horizontally with a wall) or equal to 'y' (i.e. to check of the car has collided vertically with a wall). If the direction is horizontal ('x'), a check is made to determine the direction that the player is travelling, either forward (to the right) or backwards (to the left). This is achieved by checking the horizontal velocity of the player and if this is greater than zero, the car is travelling to the right otherwise, it is travelling to the left. If the player is travelling to the right then they have collided with the left side of the wall. The position of the car is calculated from the centre of the sprite and so the new position will be the left side of the wall minus half of the width of the car sprite. Similarly, if the player was traveling in the opposite direction the new position of the sprite will be the right side of the wall plus half of the width. If the direction is vertical ('y') , a check is made once again to determine whether the car is travelling up or down. Using this information, if the car hits the bottom of a wall (i.e. it was travelling up), the new position of the car will be the bottom of the wall plus half of the height of the sprite. If the car hits the top of a wall, the new position is the top minus half of the height. The velocity of the player in the corresponding direction is set to zero and the car sprite's rect object's coordinates are changed to the new calculated position.

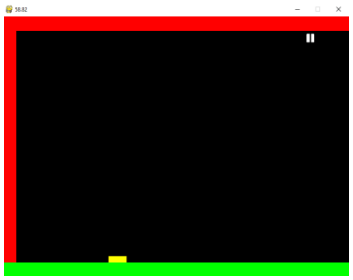
```
self.acc += self.vel * PLAYER_FRICTION
self.vel += self.acc
self.pos += self.vel + 0.5 * self.acc
self.rect.centerx = self.pos.x
self.collideWithWalls('x')
self.rect.centery = self.pos.y
self.collideWithWalls('y')
```

The 'collideWithWalls' function is called in the player class' 'update' function after the equations of motion have been used to calculate the position of the player. Before calling the function, the x and y coordinates of the position are assigned to the coordinates of the centre of the 'rect' object. He reason for this is so that the player's position is updated even if there is not a collision between the player and a wall. The 'collideWithWalls' function is called twice, once for each direction, and this code will be executed in every frame of the game as it is in the 'update' function.

Spawning the Walls



Before the wall objects could be spawned, they first need to be added to the level. This was done by adding a 3 in the text file in the place where a wall needed to be placed. After this, the code looking thorugh the level data and spawning the required objects needed to be adapated to include the walls. This was simple as only an additional selection ('if') statement needed to be added.



```
if tile == "3":
    Wall(self.game, col, row, RED)
```

This is how the game window looked after creating the wall objects (represented by the red tiles). The wall objects have correctly loaded however, a small fix needs to be made to move the car sprite to the correct position as currently, half of it is under and behind a platform.

Wall Collision Bug

Currently the collisions between the car sprite and other game sprites are being calculated using the middle of the bottom edge of the sprite's collision box. This however is producing the movement to not be smooth because the middle of the bottom edge of the car sprite is always shifted by a large amount. There is also a glitch currently where if the car is in contact with the sides of the walls and jumps, the car accelerates upwards and outside of the level. In order to fix these issues, the position from which the position of the car sprite is calculated needs to be changed.

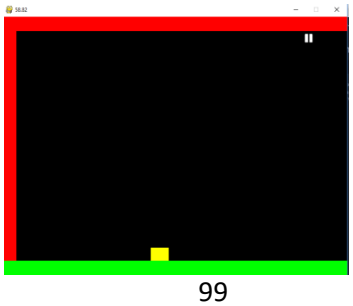
The 'collideWithWalls' function is calculating the player's collisions using the centre of the sprite and this has fixed the choppy movement along the top problem. This solution to the problem has worked because the sprite moves less when a collision occurs and so this movement is less noticeable.

```
if self.game.player.vel.y > 0:#going down [hit top of wall]
    self.game.player.pos.y = hitWall[0].rect.top - self.rect.height / 2
elif self.game.player.vel.y < 0:#going up [hit bottom of wall]
    self.game.player.pos.y = hitWall[0].rect.bottom + self.game.player.height / 2
self.game.player.vel.y = 0
self.rect.centery = self.pos.y
```

The accelerating up the side walls issue was caused by the highlighted code in the 'collideWithWalls' function being incorrect. This caused the player to be moved to the incorrect location, i.e. a position higher than where it collided with on the side, causing it to accelerate up the side and out the level. This problem was fixed by changing this code to the correct version as previously explained.

As the centre of the sprite is now being used to calculate the position, the collision code between the sprite and the platforms need to be changed. I changed the collision code in the 'update' function in the main file so that the player's 'y' position is calculated from the y position of the collided platform minus half of the player's height.

```
hitPlatform = pg.sprite.spritecollide(self.player, self.platforms, False)
if hitPlatform:
    self.player.pos.y = hitPlatform[0].rect.y - self.player.height / 2
    self.player.vel.y = 0
```



This solved the problem created after spawning the walls and now the car sprite is truly trapped inside the level.

Correcting the Player Movement

There is currently an issue with the car's movement where if the brake button is pressed alongside the forward button, the player slows down but keeps edging forward. The expected result is for the car to come to a complete stop and not move until the brake button is depressed. Additionally, if the brake button is pressed while travelling in reverse the car doesn't slow down.

Debugging the player movement

Another method I used for debugging the program and finding the cause of the error is shown below. This block of code prints the values of the car's friction, velocity and acceleration every 150 milliseconds. This condition was implemented as otherwise this will be printed out in each frame, making it more difficult to analyse the messages. The 'pg.time.get_ticks()' function was used to get the current time inorder to calculate when 150 milliseconds had passed.

```
now = pg.time.get_ticks()
if now - self.last > 150:
    self.last = now
    print("{} direction  Velocity X: {} Acceleration X: {}".format(self.direction, round(self.vel.x,0), round(self.acc.x, 0)))
    print("                Velocity Y: {} Acceleration Y: {}".format(round(self.vel.y,0), round(self.acc.y, 0)))
```

Fixing the braking while reversing issue

After reviewing the logic of the 'update' function in the 'player' class I realised what was causing this issue. The code for checking if the left arrow key or the 'a' key was being pressed to reverse was being executed after the code for going forward and braking. This meant that the code executed by the brake button was being overwritten in each loop of this function. The screenshot on the right shows the previous code where this issue existed. Therefore, this problem was fixed by moving the code for reversing above and before the code for braking.

Previous code

```
if keys[pg.K_RIGHT] or keys[pg.K_d]:
    if self.direction == "stopped" or self.direction == "forward":
        self.acc.x = PLAYER_ACC
        self.direction = "forward"
if keys[pg.K_DOWN] or keys[pg.K_s]:
    if round(self.vel.x, 0) != 0:
        if self.vel.x > 0:
            self.acc.x = -BRAKE_ACC
        else:
            self.acc.x = BRAKE_ACC
if round(self.vel.x, 0) == 0:
    self.direction = "stopped"
if keys[pg.K_LEFT] or keys[pg.K_a]:
    if self.direction == "stopped" or self.direction == "reverse":
        self.acc.x = -PLAYER_ACC
        self.direction = "reverse"
```

Fixing the stopping issue

I used the debugging shown above to find the cause of this issue and realised that the code for going forward or reversing is still executed when the car appears to have stopped. I presumed the cause of this will be that the 'direction' variable was not being set to 'stopped', which would be due to the velocity not rounding to zero. Therefore, I changed the code responsible for this to set the 'direction' variable to 'stopped' if the velocity becomes less than 0.5. This however still did not fully resolve the issue.

```
if abs(self.vel.x) < 0.5:
    self.direction = "stopped"
```

```
if keys[pg.K_DOWN] or keys[pg.K_s]:
    if self.direction != "stopped":
        if self.vel.x > 0:
            self.acc.x = -BRAKE_ACC
        else:
            self.acc.x = BRAKE_ACC
    self.braking = True
else:
    self.braking = False
```

Whilst attempting a different approach to solving this problem, I realised that the program is not aware of when the brake button is being pressed. I added a variable called 'braking' to the player class' initialisation function and set the value of it to 'False'. When the brake button is checked in the 'update' function, if the button is pressed, the value of this variable is set to 'True' otherwise, the variable is set to 'False'.

```
self.braking = False
```

Now in the code for the forward and reverse buttons, if they are pressed, the value of the 'braking' variable is checked. The code for changing the velocity of the sprite is only executed if the 'braking; variable is 'False', i.e. the brake button is no being pressed. This solved the problem and now the sprite will not move forward or reverse while the brake button is being held down.

```
if keys[pg.K_RIGHT] or keys[pg.K_d]:
    if self.direction == "forward" or self.direction == "stopped":
        if self.braking == False:
            self.acc.x = PLAYER_ACC
            self.direction = "forward"

if keys[pg.K_LEFT] or keys[pg.K_a]:
    if self.direction == "reverse" or self.direction == "stopped":
        if self.braking == False:
            self.acc.x = -PLAYER_ACC
            self.direction = "reverse"
```

Using Bitmap Images to create the track

As explained previously under the heading 'Using Pixel-wide collision boxes', bitmap images can be used to represent each pixel on the screen. In a monochromatic bitmap image there are only two colours and this can be used in this program to indicate where the track should be drawn. In order to do this, a method of loading and manipulating the bitmap images needs to be implemented.

Installing the Imaging Library

Firstly, a python image library called Pillow needs to be installed. This was accomplished using the procedure explained in the 'Downloading Python and Pygame' section, i.e. by using the command 'pip install pillow' in command prompt. This library has many classes and functions specifically for manipulating images and one such class is called 'Image'. Inside this class there is a function called 'getdata()' which returns the contents of the image as a sequence of pixel values. I decided to first test this out on a small scale to clearly view the manipulated data.

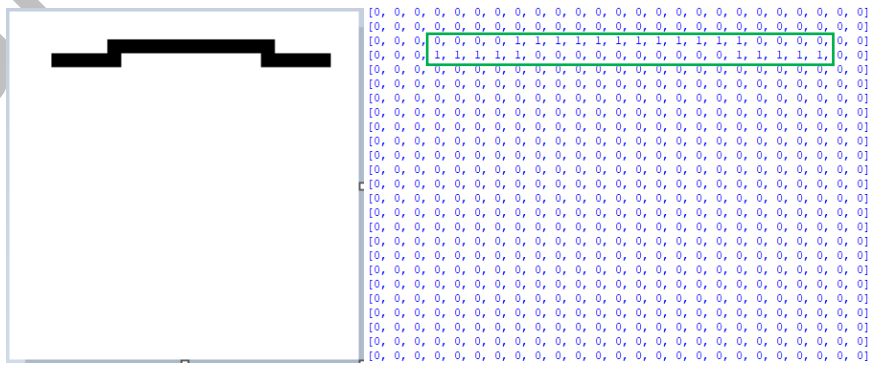
```
C:\Users\Krishan>pip install pillow
Collecting pillow
  Downloading https://files.pythonhosted.org/packages/21/fe/3df8a6faec6010ce2006758d22c
421164b16d9710fc8bd8e00dead22b510/Pillow-5.4.0-cp36-cp36m-win32.whl (1.7MB)
    100% |████████████████████████████████| 1.7MB 1.6MB/s
Installing collected packages: pillow
Successfully installed pillow-5.4.0

C:\Users\Krishan>
```

Getting the pixel data of an image

```
from PIL import Image

bmpImg = Image.open('MonoBitTest.bmp', 'r')
pixelValue = list(bmpImg.getdata())
print(pixelValue)
```

First the Image class from the PIL (Pillow) library was imported. Next, I created a monochromatic image using paint and saved it as a bitmap file. Now this image needs to be loaded into the program. To do this a function called 'open' from the 'Image' class was called, providing the file location (which in this case is the name of the image as both files are in the same directory) and read mode as parameters. The read mode is 'r' to signify the image needs to be opened in read only mode. This is because the image itself does not need to be changed in any way. The loaded image is stored in the 'bmpImg' (bitmap image) variable and the 'getdata()' function is performed on this to retrieve the image data. This is stored in the 'pixelValue' variable in the form of list with the help of the 'list' function. The image that I created is 25 x 25 pixels so the expected result of this program will be a one in the list when the pixel is black and a zero when it is white.

When I ran the program I was presented with this list of values and as shown, the pixel data has been correctly extracted.



This data can now be analysed and processed to locate the pixels where there is a one. These pixels will then be the locations to create the track.

Analysing the image data

The method of getting the pixel data shown previously does not provide a way to get the location of the pixel. As a result of this, after looking through the documentation of the Image module, I found a function called 'getpixel((x,y))'. This function performs the same as the 'getdata()' function however now a location to look in the image needs to be provided, meaning that the location of a pixel where there is a one is known. Conceptualizing the image as a two-dimensional array shows that each pixel in each row in each column needs to be analysed. This can be achieved by using definite loop providing the dimensions of the image are known. To get the values of the width and height of the image I used the 'size' function. This returns a tuple containing the two values therefore, I addressed the specific indexes to store the values of the width and height in two separate variables.

```
sizeWidth = bmpImage.size[0]
sizeHeight = bmpImage.size[1]
```

```
mapData = []
mapRow = []
blackPixel  = []
for col in range(sizeHeight):
    for row in range(sizeWidth):
        pixel = bmpImage.getpixel((row, col))
        mapRow.append(pixel)
        if pixel == 1:
            blackPixel.append((row, col))
    mapData.append(mapRow)
    mapRow = []
```

As the dimensions are now known, the image can be iterated over using two 'for' loops to analyse each pixel. Three arrays were created to store pixel data of the whole image, the pixel data of a row and, the location of the black pixels. The screen shot to the left shows the implementation of the loops. The pixel value at the location specified using the 'row' and 'col' variables is stored in the 'pixel' variable, and appended to the 'mapRow' array. Next, the value of the pixel is checked and if it is equal to 1 (i.e. if the pixel is black), the values of 'row' and 'col' are appended to the 'blackPixel' array. After each row of pixels, the contents of 'mapRow' are appended to 'mapData' and then the 'mapRow' array is re-declared as empty. This makes 'mapData' a two dimensional array, which will be useful for debugging as it allows the image data to be printed and shown in the same format as the original image. After a working solution is reached, only the location of the black pixels would be required so the other arrays can be removed to reduce the computational required.

Running this program on the test image returned this. There are 22 black pixels in the image and 22 coordinates have been printed. This means that the program is functioning correctly. These coordinates will be the positions to spawn the track in the game. Now this image processing program needs to be implemented into the game.

```
[(7, 2), (8, 2), (9, 2), (10, 2), (11, 2), (12, 2), (13, 2), (14, 2), (15, 2)
, (16, 2), (17, 2), (18, 2), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3), (18, 3),
(19, 3), (20, 3), (21, 3), (22, 3)]
```

Adapting the Map class

I decided that the most appropriate place to implement the previously shown code will be inside the 'Map' class. This is because this is where the data for creating the other parts of the level is read. I added an additional parameter to the initaialse function called 'trackfile' and this will contain the file path to the bitmap image. I added the code for manipulating the bitmap image after the existing code in this function.

```
self.trackData = []
self.trackImage = Image.open(trackfile)
self.trackWidth = self.trackImage.size[0]
self.trackHeight = self.trackImage.size[1]

for col in range(self.trackHeight):
    for row in range(self.trackWidth):
        pixelData = self.trackImage.getpixel((row, col))
        if pixelData == 1:
            self.trackData.append((row, col))
```

The Image class from the PIL library is imported at the top of the management file (which is where the Map class exists).An array for containing the locations of the black pixels was created. The bitmap image is opened and saved in the 'trackIamge' variable and, the dimensions of the image are retrieved and also saved in appropriate variables. After that the data from each pixel in the image is analysed and the location of the required pixels are appended to the 'trackData' array.

Spawning track at the required locations

The level data for different levels will be different therefore, a new map object will need to be created for every object. It would be unnecessary and inefficient to create a map object for all levels in the game when it is first loaded therefore, a new instance of the 'Map' class can be created at the start of each level function. The camera object requires the dimensions of the level defined in the 'Map' class as parameters, it too will need to be created at the start of each level. However, as code in the main file relies on the camera object, it will need to be created in the main file. This can be easily performed using the advantageous of object oriented programming. As an instance of the 'game' class is provided to all the other classes, the 'camera' variable from the 'game' class can be addressed and used to create a new camera object. In order to avoid errors before the camera object is created (i.e. whilst in the menu system), the variables 'self.map' and 'self.camera' are created in the 'new' function in the main file and assigned the values 'None'.

```
self.map = None
self.camera = None
```

```
self.game.map = Map(path.join(self.game.gameFolder,'map2.txt'), path.join(self.game.gameFolder,'LevelTest.bmp'))
self.game.camera = Camera(self.game.map.width, self.game.map.height)
```

This code is from inside the 'level1' function and shows the map and camera objects being created as previously explained. The code for the spawning the track is added after the 'for' loops iterating through the text file map data. This consists of a 'for' loop iterating over the array created in the 'Map' class which contains the locations of the black pixels. The coordinates are obtained and stored in temporary variables called 'platformx' and 'platformy'.

```
for coordinates in range(len(self.game.map.trackData)):
    platformx = self.game.map.trackData[coordinates][0]
    platformy = self.game.map.trackData[coordinates][1]
    Track(self.game, platformx, platformy, 1, 1, LIGHT_BLUE)
```

```
class Track(pg.sprite.Sprite):
    def __init__(self, game, x, y, width, height, colour):
        self.groups = game.allSprites, game.platforms
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((width, height))
        self.image.fill(colour)
        self.rect = self.image.get_rect()
        self.rect.x, self.rect.y = x  , y
```

I created a new class called 'Track' which was similar to the 'Platform' class but did not multiply the 'x' and 'y' coordinates by TILESIZE. After creating this class I realised as there will not be any differences between the collision code for track and platform objects, it will be more efficient to edit the already existing 'Platform' class than unnecessarily add another class which is essentially the same. To do this I added a parameter to the initialise function of the 'Platform' class called mode, and set the default value to 'platform'. Next, I added a selection statement which checks the value of this variable and performs [self.rect.x , self.rect.y = x * TILESIZE, y * TILESIZE] if the mode is 'platform' or, [self.rect.x , self.rect.y = x, y] (as shown by the green box above) if the mode is 'track'. Therefore, if a track object needs to be spawned, the 'mode' parameter will need to be set to 'track' when the object is created.

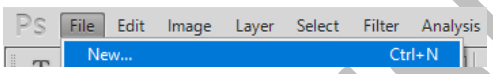
```
if mode == "platform":
    self.rect.x, self.rect.y = x * TILESIZE, y * TILESIZE
elif mode == "track":
    self.rect.x, self.rect.y = x, y
```

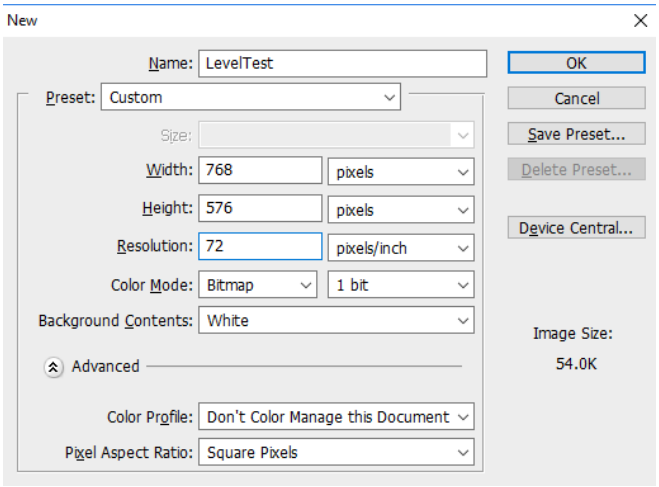
```
for coordinates in range(len(self.game.map.trackData)):
    platformx = self.game.map.trackData[coordinates][0]
    platformy = self.game.map.trackData[coordinates][1]
    Platform(self.game, platformx, platformy, 1, 1, LIGHT_BLUE, 'track')
```

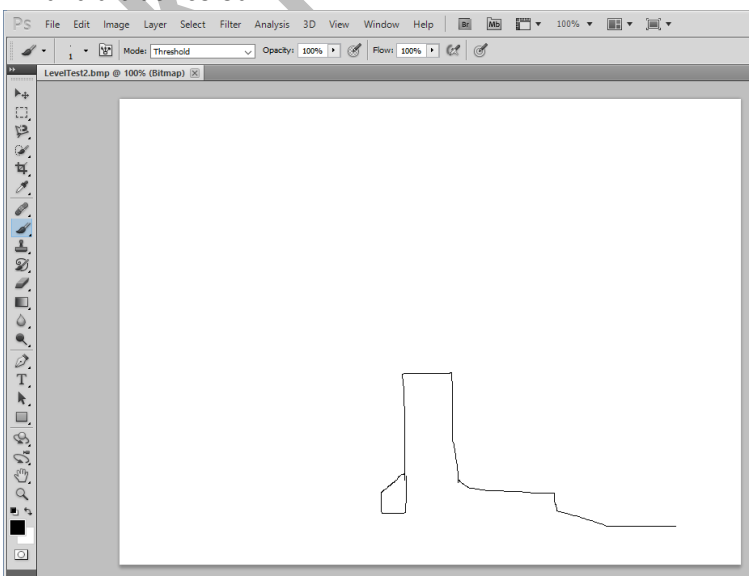
So, the 'platformx' and 'platformy' variables are used for the 'x' and 'y' coordinates to create platform objects. These platforms will be 1 pixel-wide square so the width and height will be 1.

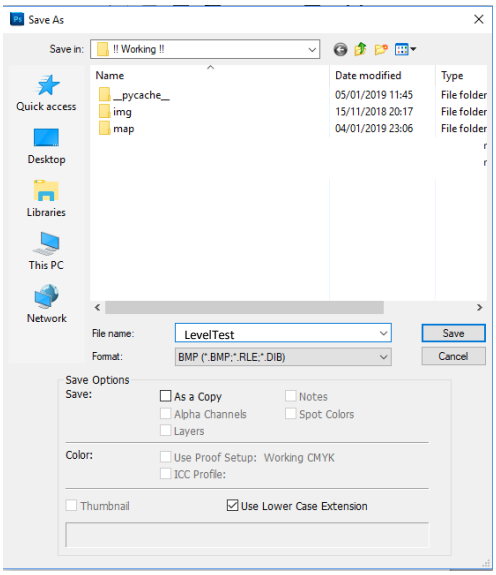
Creating a Bitmap image using Photoshop

Before I can test out the code that I have written for the bitmap image, I need to create a monochromatic bitmap image with specific dimensions. It not possible to create a specific size image in Microsoft Paint so I decided to use Adobe Photoshop to do this. The size of this first test bitmap image will need to be the size of the game window. This is so that each pixel in the bitmap image will correspond to each pixel in the game. I created a new file in Photoshop with the appropriate width, height and colour mode. After that I drew a track using the Pencil Tool and a black colour.

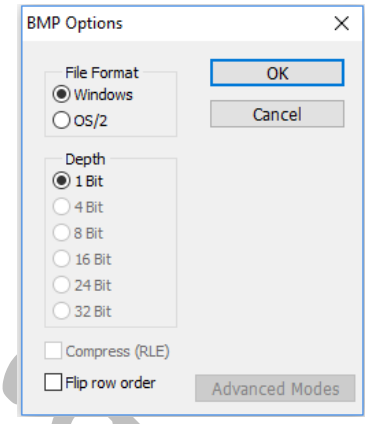






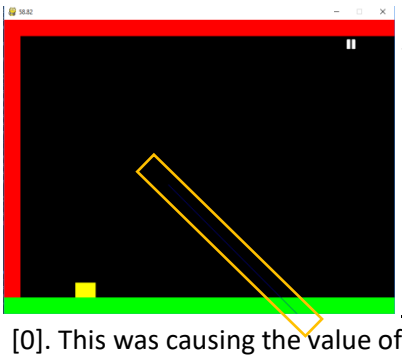


I saved the file as a bitmap image in the same directory as all of the code using the 'Save As' function. After this I was asked for the bit depth of the bitmap image and I chose 1 Bit. Presumably, if a bit depth of 4 or 8 bits was initially chosen when creating the page but a simpler colour depth was used to create the image, there will be the option to choose 1, 4 or 8 bits on this screen. This will reduce the file size of the image as fewer bits will be required to store the data of the colours making up the image.



As the image has now been created, the code can be run to check if it is functioning as expected.

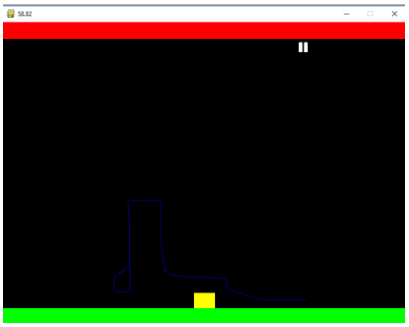
Error Encountered - The bitmap image did not spawn the platform objects in the expected locations. As shown by the orange rectangle, the blue coloured platform where being spawned in a diagonal line when they should be being spawned to resemble the image created. My first step to debugging this problem was to print the values of 'platformx' and 'platformy'. Soon after running the program again and viewing the debug messages, I realised the problem.



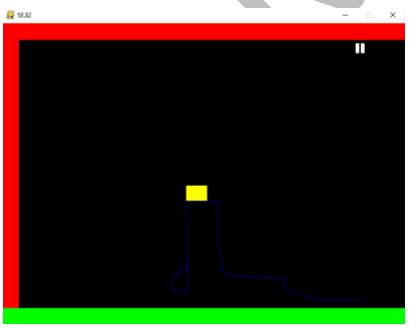
```
for coordinates in range(len(self.game.map.trackData)):
    platformx = self.game.map.trackData[coordinates][0]
    platformy = self.game.map.trackData[coordinates][0]
    print(platformx, platformy)
    Platform(self.game, platformx, platformy, 1, 1, LIGHT_BLUE, 'track')
```

Solution - The value assigned to 'platformy' was incorrect and should be [1] instead or [0]. This was causing the value of 'platformy' to be the same as 'platformx' and thus creating the diagonal line pattern. Therefore to fix solve the problem I change the [0] to [1] for 'platformy'.

Review

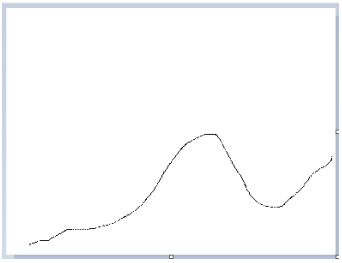


The bitmap image is now being correctly spawned into the level. The collisions between the platform objects and the car sprite work as expected. When the car sprite collides with the track, it is moved to the top of the track. Once more smooth curves are implemented, the player will not be able to get underneath the track so the immediate transportation to the top of the platform will be less obvious. Another method of making the transition less noticeable is by rotating the player so that the bottom edge of the sprite lines up with the curve of the track. This will make the movement of the car along the track more realistic.



There was a delay of about 2-3 seconds after selecting the level 1 button in the level select screen and the level actually loading. The cause of waiting could be related to the image processing for finding the locations of the black pixels. Currently 2-3 seconds is a short enough time to wait however if the problem escalates, a solution will need to be found. The next step is to create a more realistic level which has a larger width so the track continues as the camera moves.

Improving track implementation methods



After successfully loading in a monochromatic bitmap image, I created another image which further resembles a track. This image was also 768 by 576 pixels and so finished when the camera started to scroll. I decided to create a track which was longer and so continued as the camera scrolled. This will test the execution of the image processing and make transparent any performance issues.

Creating a second level

In order to create the second level certain lines of code need to be added to various places in the 'sceneManager' class. Firstly, the selection statement shown needs to be added to the 'loadLevel' function in the 'sceneManager' class. This ensures that the 'level2' function (when created), will be loaded when the appropriate button is clicked.

```
if self.level == "level2":
    self.level2()
```

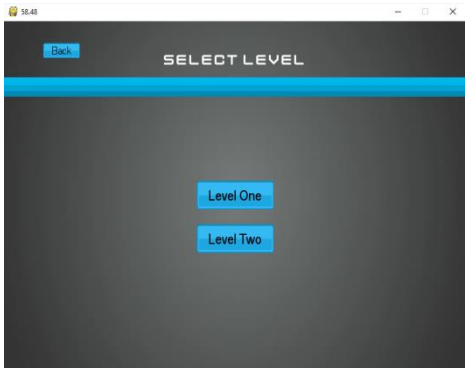
```
if button.tag not in ('level1', 'level2'):
    self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)
```

This code is in the 'sceneManger' class' 'update' function and 'level2'needs to be added to the not in section. This is responsible for loading the menu screen images and so should only be executed when the current scene is not a level. Similarly, 'level2' is added to the not in section in this code, which is added to the end of the 'update' function, to ensures that the 'showPlayer' variable is set to 'False' if the current screen is a menu screen.

```
if self.currentScene not in ("level1", "level2"):
    self.showPlayer = False
```

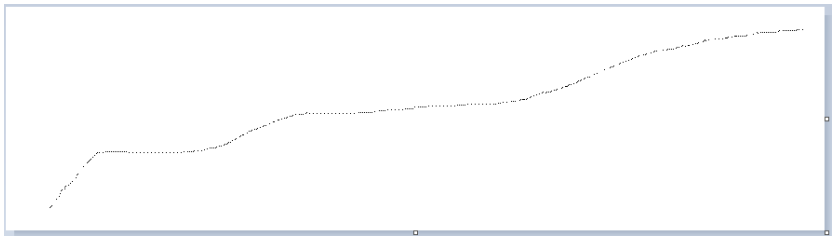
```
def levelSelect(self):
    self.game.drawText("Select Level", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)
    self.level1Button = Button(self.game, "level1", WIDTH*1/2, HEIGHT*1/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Level One")
    self.level2Button = Button(self.game, "level2", WIDTH*1/2, HEIGHT*5/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Level Two")
```

A new button is created in the 'levelSelect' button for the second level. The position of the button needs to be changed so that it doesn't overlap any other buttons on the page. The code shown on the left is added to the block of 'if' statements in the 'update' function and is responsible for calling the 'loadLevel' function to load the 'level2' function when a button with the tag 'level2' is pressed. Finally, a function called'level2' is created inside the 'sceneManager' class and this will contain all of the code required for the second level. This procedure will need to be repeated whenever a new level is added.




```
if button.tag == 'level2':
    self.loadLevel('level2')
```

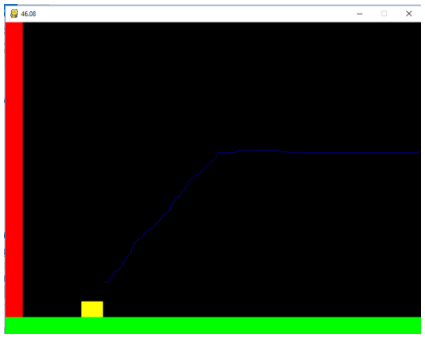
I created a new text file for a new level and made it 110 tiles wide and 30 tiles tall. There were 1's across the bottom to spawn platforms there, and 3's along the sides and top to spawn walls there. I calculated that the image needed to cover the length of this level will need to be 110 * 32 pixels wide and 30 * 32 pixels tall. I created an image that was 3520 pixels by 960 pixels using the formerly explained method. Next, added the level 2 code, which was essentially the same as the code used in the 'level1' function.



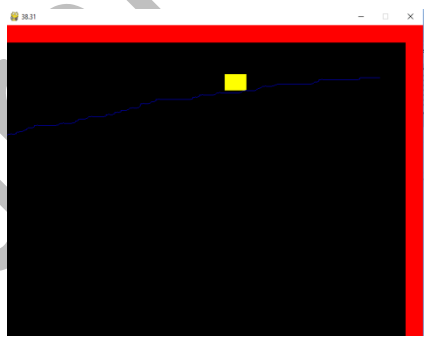
```
def level2(self):
    self.showPlayer = True
    self.game.map = Map(path.join(self.game.gameFolder,'map3.txt'), path.join(self.game.gameFolder,'Track4.bmp'))
    self.game.camera = Camera(self.game.map.width, self.game.map.height)
    #
    for row, tiles in enumerate(self.game.map.data):
        for col, tile in enumerate(tiles):
            if tile == "1":
                Platform(self.game, col, row, TILESIZE, TILESIZE, GREEN)
            if tile == "3":
                Wall(self.game, col, row, RED)
            if tile == "P":
                self.game.player = Player(self.game, col, row)

    for coordinates in range(len(self.game.map.trackData)):
        platformx = self.game.map.trackData[coordinates][0]
        platformy = self.game.map.trackData[coordinates][1]
        Platform(self.game, platformx, platformy, 1, 1, LIGHT_BLUE, 'track')
```

If the same code ends up being executed in all the levels then a separate function can be created to house this code and then be called in each level function. The only differences in the code were the filenames of the text file and the bitmap image.

Issues caused with larger bitmap images



When I ran the game and selected level 2, the level loaded correctly and the collisions were all functioning as expected. However, after clicking the level 2 button, there was a delay of approximately 25 seconds before the level loaded and the car could be controlled. Even after loading the level there were drops in the frame rate to averaging around 40 FPS. This meant that there is a problem in the program causing excessive computation and as a result impacting the smoothness of the game.

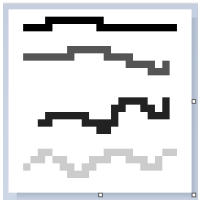


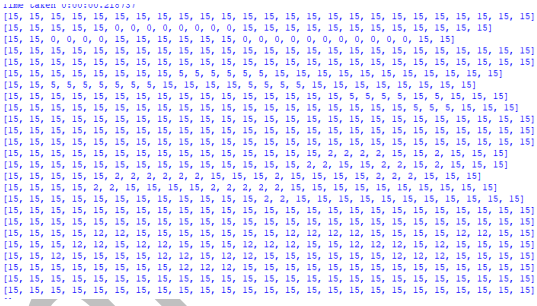
The long loading time problem is being caused by the increased amount of processing required to handle the larger image. There is no simple method of optimising this procedure so it could be separated from the program. This is possible because the level data doesn't change after the bitmap image has been created. Therefore, a separate program can be written to process the image and store the location of pixels where platforms need to be spawned on a text file. This "compiled" text file can then be loaded into the game and processed when a level function is called. As this text file will only contain essential data for loading the level, no time or computational resources will be wasted analysing unnecessary (blank) parts of the image. Moreover, this solution will also solve a problem created by using the pause button. Currently the pause button has been programmed to reload the level function when 'Resume' is pressed. This means that the image processing code is re-run and it takes 25 seconds for the game to resume.

The excessive computation problem is being caused by the increased number of platforms on the screen. In this bitmap image there are 3463 black pixels, meaning that 3463 checks are being made in every frame to see if the car has collided with a platform. This was the drawback of using the pixel-wide collision box method however there are method of optimising this. One such method is to only check the collisions between the car sprite and the platforms that are currently in the game window. This logic originates from the fact that the car sprite cannot possibly collide with a platform object that is currently not inside the game window. Therefore, implementing this would reduce the checks that will need to be made in each frame and possibly return the frame rate to 60 FPS as a result.

Using a 4 bit bitmap image

Having two separate blocks of code for spawning the objects needed to create the level seems inefficient. Therefore, I decided to use a 4 bit bitmap image to store more information in the image and use different colours to spawn different items. Different colours will be used to spawn the track, walls and the area under the track. This area doesn't need to be filed with platform objects as the player will only come into contact with the top layer. Doing this will unnecessarily increase the computational resources required and possibly slow down the operation of the game. Therefore, rectangles can be drawn over these pixels with the same colour as the track to create a more realistic appearance.

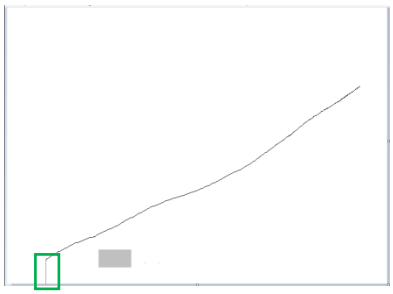




Firstly, I created this image using Photoshop with the colour mode set to Bitmap 4 Bit. I then loaded this image into the pixel data testing program that I created when testing the monochromatic bitmap images. I didn't make any changes to the program and when I ran it I was given the output shown. The difference between this and the monochromatic bitmap image is that now 4 bits are being used to represent the colours, allowing 16 colours to be represented. Therefore, I could theoretically use 4 bit bitmap images to represent 16 different game objects. As 4 bits are now being used to represent black and white, the values for them change from 1 and 0 to 0 and 15 respectively. I also used this program to identify the values of the three grey colours in the test image. Using these values, I created this key table.

Colour	Function	Denary Value	Hexadecimal Colour Code
Black	Spawn Track	0	000000
White	Nothing	15	FFFFFF
Dark Grey	Spawn Wall	5	5e5e5e
Light Grey	Draw rectangle	12	cacaca

Adding 4 bit image processing



Using the previously identified key, I created a new track using Photoshop. The straight edge identified by the green box is dark grey and so walls will be spawned there. Next, I created a new level (level 3) using the method explained under the heading 'Creating a second level'. The 4 bit bitmap image will be processed and loaded on this level. In order to add the 4 bit image processing without altering the code that has already been written for the first level, I decided to add a parameter called 'bitSize to the 'Map' class, and have the default value set to 4. This is required as the values corresponding to the same colours are different due to the colour depth. Next I created two arrays to store the walls spawning locations and the rectangle drawing locations.

```
self.trackData = []
self.wallData = []
self.fillData = []
```

After this, I used an 'if statement inside the 'for' loops to check the state of the 'bitSize' variable. If it was equal to 1 (i.e. a 1 bit bitmap image is being used) then a 1 will correspond to a black pixel so the location of any pixel with a 1 is added to the 'trackData' array. If the value of 'bitSize' is instead 4, then a 0 will correspond to a black pixel and so any pixel with a 0 is added to the relevant array. No further processing is required for 1 bit bitmap images therefore the other two checks are implemented under another 'if' statement. Locations of pixels with values 5 and 12 will be added to the 'wallData' and 'fillData' arrays respectively.

```
for col in range(self.trackHeight):
    for row in range(self.trackWidth):
        pixelData = self.trackImage.getpixel((row, col))
        if bitSize == 1:
            if pixelData == 1:
                self.trackData.append((row, col))
        else:
            if pixelData == 0:
                self.trackData.append((row, col))
        if bitSize == 4:
            if pixelData == 5:
                self.wallData.append((row, col))
            if pixelData == 12:
                self.fillData.append((row, col))
```

```
for coordinates in range(len(self.game.map.trackData)):
    trackx = self.game.map.trackData[coordinates][0]
    tracky = self.game.map.trackData[coordinates][1]
    Platform(self.game, trackx, tracky, 1, 1, LIGHT_BLUE, 'track')

for coordinates in range(len(self.game.map.wallData)):
    wallx = self.game.map.wallData[coordinates][0]
    wally = self.game.map.wallData[coordinates][1]
    WallFile(self.game, wallx, wally, 1, 1, LIGHT_RED)

for coordinates in range(len(self.game.map.fillData)):
    fillx = self.game.map.fillData[coordinates][0]
    filly = self.game.map.fillData[coordinates][1]
    pg.draw.rect(self.game.screen, YELLOW, [fillx, filly, 1, 1], 1)
```

Next, I had to add the track and wall spawning code to the 'level3' function. The code to do this was the same as the code for spawning in the track however now, the 'x' and 'y' coordinates are used for spawning wall objects instead. I used the 'pg.draw.rect()' function to draw a rectangle with dimensions 1 pixel by 1 pixel and colour yellow. I set the width argument of the rectangle to 1. When I ran the game and selected level 3, there was a wait of approximately 4 seconds before the level loaded. The track and wall objects were spawned correctly however the rectangles intended to fill the area under the track were not drawn. The orange box shows that the wall objects (indicated by their red colour) have correctly been created. I suspected that the cause of the rectangle not being drawn were due to the width argument and so removed it. This however did not resolve the problem. I printed the values of the variables 'fillx' and 'filly' to check that they were correct and they were.

Solving the rectangle drawing problem

After reviewing the logic of the program during the levels, I remembered that the whole screen was being filled black in each frame in the 'update' function of the 'player' class. As the rectangles were only being drawn once, the 'fill' command being called in every frame immediately draws over the rectangles. It does draw over the wall and the track but these are re-drawn in each frame after this command

```
def update(self):
    if self.game.sceneMan.showPlayer:
        self.game.screen.fill(BLACK)
```

and so appear to not have been overwritten. I removed this line of code from the 'update' function and the rectangles were correctly drawn. However, I remembered the reason why this line of code was used as the previous images of the sprite from past frames remained on the screen. This makes the game un-playable as there is no indication of the current position of the car. The problem causing the rectangles to not be drawn has been resolved however now a solution needs to be created which solves this problem without causing the initial problem to return.

```
class Rectangle(pg.sprite.Sprite):
    def __init__(self, game, x, y, width, height, colour):
        self.groups = game.allSprites
        pg.sprite.Sprite.__init__(self, self.groups)
        pg.draw.rect(game.screen, colour, (x, y, width, height))
```

I decided to create a class for making rectangle objects. This class was created in the 'sprites' file and takes in the same parameters required by the 'pg.draw.rect()' function. The reason why I chose to create an object for this problem is because the objects can be added to the 'allSprites;' group and then drawn in every frame. This would mean that the rectangles will be drawn on the new black background in each frame and so will be visible. I used the 'pg.draw.rect()' function inside the class to create the rectangles. Next, I created an instance of the 'Rectangle' class in the same loction when the rectangles were drawn in the 'level3' function.

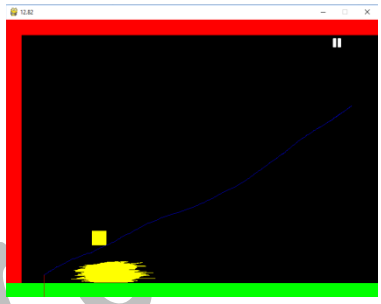
```
Rectangle(self.game, fillx, filly, 1, 1, YELLOW)
```

```
    self.draw()
  File "C:\Users\Krishan\Documents\Krishan\AS and A Level\Computer Science
amming Project\Code Development\!! Working !!\main.py", line 136, in draw
    self.screen.blit(sprite.image, self.camera.applyOffset(sprite))
AttributeError: 'Rectangle' object has no attribute 'image'
>>>
```

When I ran the program and chose the third level, the level did not load and an Attribute Error message was displayed. This message explained how an attribute called 'image' is required by all of the sprites in the 'allSprites' group in order to blit the sprite to the screen. I attempted to fix this program by saving the rectangle in a variable called 'self.image' however this did not work as a 'rect' attribute was then required, and this could only be taken from a surface object.

```
class Rectangle(pg.sprite.Sprite):
    def __init__(self, game, x, y, width, height, colour):
        self.groups = game.allSprites
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((width, height))
        self.image.fill(colour)
        self.rect = self.image.get_rect()
        self.rect.x, self.rect.y = x, y
```

As a result of the series of error messages, I finally created a surface object for the rectangle using the 'width' and 'height' parameters. This object was filled with the provided colour and then the 'rect' attribute of this object was used to position the rectangle and the required location.



When I ran the code, the rectangles were correctly being spawned in however there was a severe impact on the frame rate. The increased number of sprites being drawn in each frame is the cause of this. The frame rate, which should be at 60 FPS, has dropped to around 20 FPS. A possible solution to this problem is, after all the objects used in the level have been finalised, a background image can be created using the track bitmap image as a template. In this background image the area under the track can be drawn, meaning that only one image will need to be re-drawn in every frame in oppose to the current solution.

Solving the previous level sprites problem

If the pause button was clicked during a level, followed by the main menu button, level select button and then another level, the platforms and walls and car sprite from the previous level are still visible. The previous car object falls down off the screen as it is now not controlled. I realised that the cause of this problem was because the objects in the platforms and walls groups were not deleted after a level is exited. To solve this problem I added these 'for' loops in the 'update' function in the 'sceneManager' class and it will be executed if a level button is clicked.

```
for wall in self.game.walls:
    wall.kill()
for platform in self.game.platforms:
    platform.kill()
```

These 'for' loops iterate through the 'walls' and 'platforms' groups and 'kill' (i.e. delete) each of the objects inside them.

When I ran the code the walls and platforms from the previous level were correctly removed however the rectangles drawn remained. In order to fix this problem a new group to hold the rectangles needed to be created. Therefore, I created a new group for the rectangles in the 'new' function in the main file,

```
self.rectangles = pg.sprite.Group()
```

```
self.groups = game.allSprites, game.rectangles
```

and altered the first line of code in the 'Rectangle' class to include the 'rectangles' group. After this I added the code shown below to the 'sceneManager' class' 'update' function to remove all of the rectangle objects when a new level is launched. I tested out the code and it worked as expected.

```
for rectangle in self.game.rectangles:
    rectangle.kill()
```

The final object to remove when a new level is launched is the car sprite. In my first attempt at solving this problem I used the code [self.game.player = None] after the three 'for' loops defined above. This did not solve the problem. I then changed the code to [self.game.player.kill()] however this brought up an error stating that a 'NoneType object has not attribute 'kill''. Using this information, I used an 'if' statement before the 'kill()' function is called on the player sprite to check if the value of 'self.game.player' is equal to 'None'. This solved the problem and now when a new level is loaded, no objects or sprites from the previosu levels remain.

```
if self.game.player != None:
    self.game.player.kill()
```

Review

4 bit bitmap images have now been successfully added to the game and all the problems that arose have been solved. This means that essentially only one image is required to draw the whole level. The next stage in development will be to start incorporating the mental arithmetic questions into the game.

Implementing question asking functionality

Previously, code for loading questions from a csv file was created. This now needs to be implemented into the game so that the loaded questions are printed to the screen. However, before the questions are displayed, the game needs to be temporarily paused. Currently, the 'pause' function is programmed as a new screen, causing the current level data to be deleted when the 'pause' button is pressed. This needs to be changed so that the level data is only deleted if the 'mainMenu' button is pressed.

Improving the 'pause' function

My first thought in solving this problem was to create a temporary surface for the pause screen which will be displayed and removed as required. The elements of the pause screen, such as the text and buttons, will be added to this temporary surface. First, I created two variables in the 'new' function of the game class for determining whether the pause button has been pressed. The buttons for the pause screen need to be made in the first frame after the pause button is pressed. This is required only once and the 'pauseScreenPrinted' variable is used to control this. I decided that a method for freezing the time (which is essentially what is required by the pause feature) is by stopping the 'update' function in the 'Game' class from being called. Therefore, I added an 'if' statement in the 'update' function so that it is only executed if the 'pause' variable is equal to False. I decided to allow the 'sceneManager' object to update as this will be required to determine when the buttons have been pressed.

```
self.paused = False
self.pauseScreenPrinted = False
```

```
def update(self):
    #Game Loop - Update
    self.sceneMan.update()
    if not self.paused:
        self.allSprites.update()
```

```
self.pauseScreen = pg.Surface(self.screen.get_size()).convert_alpha()
self.pauseScreenImage = pg.transform.scale(self.menuImages['pause'], (WIDTH, HEIGHT))
self.pauseScreenImageRect = self.pauseScreenImage.get_rect()
self.pauseScreen.blit(self.pauseScreenImage, self.pauseScreenImageRect)
self.drawText("Pause Menu", 25, WHITE, WIDTH/2, HEIGHT*1/9, surf=self.pauseScreen, fontName=self.interfaceFont)
```

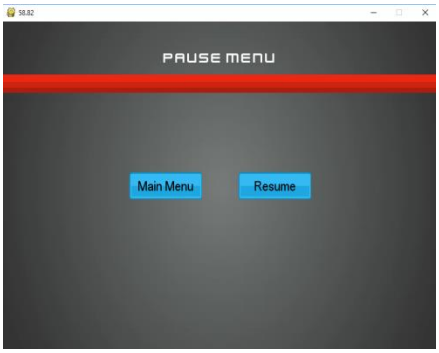
The pause menu surface was created in the 'loadData()' function using the 'pg.Surface' command. The size of this temporary surface is the same as the dimensions of the main game screen. The image that was previously used for the pause screen is loaded and the 'rect' object of this image is obtained. This image is then blitted to the pause screen surface before blitting the text 'Pause Menu' to the screen last. There was an error with the text not displaying which was caused by the image being blitted after and on top of the text.

```
if self.paused:
    if self.pauseScreenPrinted == False:
        self.sceneMan.loadLevel('pause')
        self.pauseScreenPrinted = True
    for button in self.buttons:
        button.draw(self.pauseScreen)
    self.screen.blit(self.pauseScreen, self.pauseScreen.get_rect())
else:
    for button in self.buttons:
        button.draw(self.screen)
    if self.sceneMan.currentScene not in MENU_SCREENS:
        for sprite in self.allSprites:
            self.screen.blit(sprite.image, self.camera.applyOffset(sprite))
pg.display.flip()
```

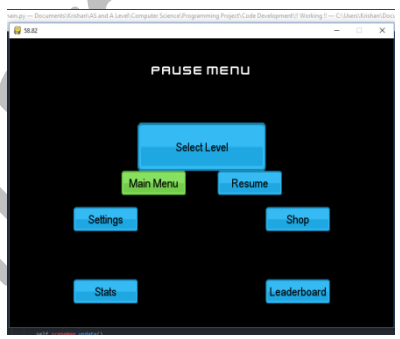
Next, I changed the 'draw' function to print the temporary pause surface to the screen when the pause button is pressed. The state of the 'pause' variable is checked and if it is True, the buttons are created and then the 'pauseScreenPrinted' variable is set to False. The buttons that have been created need to be blitted to the new surface. Currently the buttons were programmed to blit to the game screen so I had to change this function. A new surface parameter was added to the button's draw function and this provided surface is what the buttons are blitted to. Finally, the pause screen surface is blitted to the screen). If the 'pause' variable is not equal to 'True', then the code for drawing the buttons and the sprites in the 'allSprites' group is executed.

```
if event.type == pg.KEYDOWN:
    if event.key == pg.K_SPACE:
        self.player.jump()
    if event.key == pg.K_p:
        self.paused = not self.paused
        if self.paused == False:
            for button in self.buttons:
                button.kill()
        self.pauseScreenPrinted = False
```

I tried using the existing pause button image however this was causing persistent errors (which are explained later) so I decided to use the 'P' button to trigger the pause screen. This code was added to the 'events' function as this is where the inputs are recorded. If a keydown of the letter 'P' event is recorded then the state of the 'pause' variable is reversed. This allows the 'P' button to be used to pause and resume the game. If it is being used to resume, then the buttons on the screen need to be removed and the 'pauseScreenPrinted' variable needs to be reset.



I tried out the pause menu by running the game and pressing the 'P' button whist inside a level. As expected the game loaded the pause screen surface with the images and the buttons. The buttons on this screen did highlight when the cursor was over them however, the resume button did not have any functions linked to it, causing the program to crash when it was pressed. Additionally, when the 'main menu' button was pressed, the output was not as expected, as shown on the right. The solution to this problem is explained later.




```
self.secondPrevScence = self.prevScence
self.prevScence = self.currentScene
self.currentScene = button.tag
```

I added these variables in the 'update' function of the 'sceneManager' to act as trackers of the previous game screens. At the start of the program, there will be no values for 'self.secondPrevScene' and 'self.prevScene' therefore, to prevent an error, I defined them in the 'init' function of the class with the value None. Now when the pause screen is displayed, the screen from which it was called is stored and can be used by the resume function to correctly set the value of the 'currentScene' variable.

```
self.prevScence = None
self.secondPrevScence = None
```

```
if button.tag == 'resume':
    self.game.paused = False
    for button in self.game.buttons:
        button.kill()
    self.game.pauseScreenPrinted = False
    self.currentScene = self.prevScence
```

This was added to the 'update' function of the 'sceneManger' class to be executed if the 'resume' button is pressed. When this happens, the 'paused' variable in the game class is set to False and all of the current button objects are killed. Next, the 'pauseScreenPrinted' variable is set to False and the 'currentScene' variable is set to screen which was displayed before the pause screen.

```
if button.tag not in ('level1', 'level2', 'level3', 'pause', 'resume'):
    self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)
```

The 'pause' and 'resume' tags were also added to this 'if' statement which is used to check if background images are needed.

```
if button.tag == 'mainMenu':
    if self.game.paused:
        self.game.paused = False
    self.loadLevel('mainMenu')
```

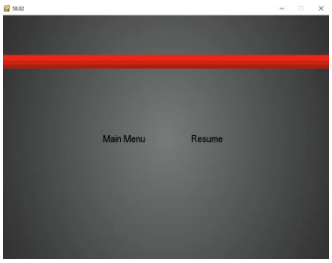
Finally, this was added when the main menu button is pressed and the purpose is to reset the state of the 'paused' variable if this button is pressed in the pause screen. This ensures that the 'else' block of code is executed in the 'draw' function to correctly draw the main menu screen instead of causing the error shown above.

The pause screen was now fully functional with correctly working buttons. This screen is entered by pressing the 'P' button and can be left by pressing the 'P' button again, or pressing the resume button. Alternatively, the main menu screen can be using pressing the respective button.

Problems with the pause screen

This implementation was not a smooth process however, as there were many errors which needed to be overcome, and they will now be explained.

Error Encountered - The buttons were not being displayed to the pause menu surface. The text that was provided for the creation of the button was being displayed. This meant that the problem must be with the blitting of the button image.



Solution - The problem was with the 'draw' function of the 'button' class as this was blitting the button images to the game screen be default. This was fixed by adding a 'surface' parameter which will be the surface which the button objects need to be blitted to. The required surface is now provided when calling the 'draw' functions for the buttons.

```
def draw(self, surface):
    if self.colchange == True or self.first == True:
        surface.blit(self.image,self.rect)
```

```
for button in self.buttons:
    button.draw(self.pauseScreen)
```

Error Encountered - The buttons still were not being correctly displayed. This is now not an issue connected with the surface as this has been already corrected. This is currently the code used to display the pause menu surface.

```
self.screen.blit(self.pauseScreenImage, self.pauseScreenImageRect)
```

Solution - The code to blit the temporary surface is incorrect as instead of drawing the surface to the screen, the image used in the surface is being drawn to the screen. This meant that the buttons were being drawn to the pause screen surface however, they were not being displayed as that surface was not being drawn. This was fixed by correctly blitting the surface to the game screen.

```
self.screen.blit(self.pauseScreen, self.pauseScreen.get_rect())
```

Error Encountered – The resume button was not functioning as expected. Instead of resuming the level, when it was pressed, the game crashed.

```
Doing Update function
Doing Update function
Doing Update function
drawn this button pause
Doing Update function
Doing Update function
Doing Update function
Doing Update function
Doing Update function
Doing Update function
drawn this button mainMenu
True self.first
drawn this button resume
True self.first
drawn this button mainMenu
True self.first
drawn this button resume
True self.first
drawn this button mainMenu
drawn this button mainMenu
drawn this button mainMenu
drawn this button mainMenu
drawn this button resume
drawn this button resume
```

Solution - I created some debug messages to inform of when the 'update' function was being resumed and the state of the 'paused' variable throughout this process. Analysing these messages made me realise that the update function was not being called again once the resume button is pressed. I noticed that the message to declare the creation of a button object is being printed twice.Two instances of the main menu and resume buttons were being created, once when the pause screen surface was drawn in the 'draw' function, and once in the 'pause' function in the 'sceneManager' class.

```
if self.paused:
    if self.pauseScreenPrinted == False:
        self.sceneMan.mainMenuButton = Button(self, "mainMenu", WIDTH*3/8, HEIGHT*1/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Main Menu")
        self.sceneMan.resumeButton = Button(self, "resume", WIDTH*5/8, HEIGHT*1/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Resume")
        self.pauseScreenPrinted = True
```

I fixed this by removing the code to create the buttons from the 'draw()' function. This however didn't fully resolve the problem.

Error Encountered - The scene is not changing to the scene from which the pause button was pressed when the resume button is pressed. I added a debug message to be printed when the resume button is clicked. This indicated that the resume function was being called however the scene was not changing as expected.

```
print("resume function executed")
```

Solution - I added some tracker variables to store the previous scenes. When the resume button is now pressed, the value of the 'currentScene' variable is changed to the scene two screen before. This is required as once the resume button is pressed, the current scene nowbecomes 'resume'.

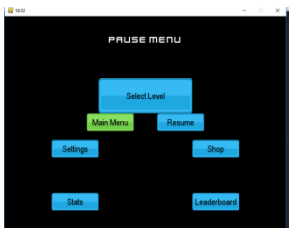
```
self.secondPrevScence = self.prevScence
self.prevScence = self.currentScene
self.currentScene = button.tag
```

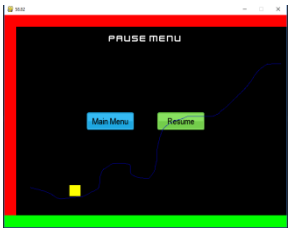
```
self.currentScene = self.secondPrevScence
```

```
if self.currentScene  not in ("level1", "level2", "level3"):
    self.showPlayer = False
    print("showPlayer :", self.showPlayer)
else:
    self.showPlayer = True
```

Also, I added an 'else' condition to this 'if' statement in the 'update' function of the 'sceneManager' class to tell the 'player class's 'update' function to render the player movements after the resume button is pressed.

Error Encountered - When resume button is pressed the pause screen surface is not cleared from the display. The buttons are also not cleared when the main menu button is pressed. On top of this, the background image of the main menu screen is not being blitted. The car sprite can be controlled, meaning that the previous addition to the 'else' statement to change the state of the 'showPlayer' variable is working as expected.



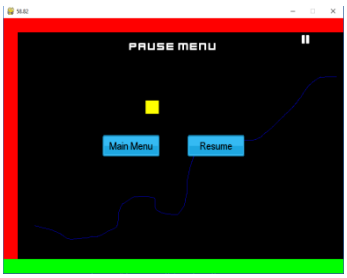


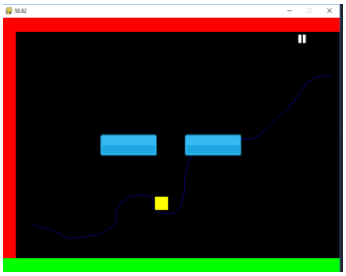
Solution - I added a 'for' loop to iterate through the 'buttons' group and 'kill' all of the button objects when the resume button is pressed. For the main menu screen problem, I set the value of the 'paused' variable to False if it is True when the button is pressed. Now when the main menu and resume buttons are pressed the previous buttons are deleted from the screen. The resetting of the 'paused' variable in the 'mainMenu' function allowed the background image for this screen to be correctly drawn.

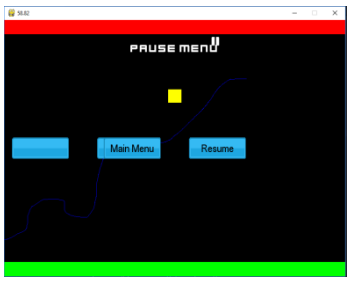
```
for button in self.game.buttons:
    button.kill()
```

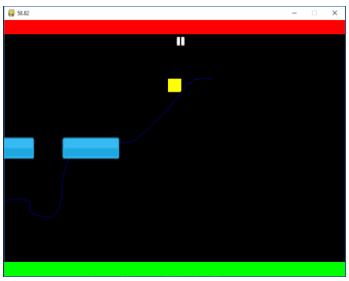
```
if button.tag == 'mainMenu':
    if self.game.paused:
        self.game.paused = False
    self.loadLevel('mainMenu')
```

Error Encountered - When the 'P' button was used to resume the game the button images were not correctly removed from the game. When the 'P' button is pressed again after this, two new button objects and images are created. The pause screen surface is not loaded. However, when the 'P' button is then pressed to resume the game, the button s which were created on the second loop are correctly deleted from the screen.









Solution - The problem exists in the 'events' function in the 'game' class.
When the 'P' button is pressed to resume the game, the buttons on the screen need to be removed and the 'pauseScreenPrinted' variable needs to be reset to False. The buttons are not deleted as when the resume button is pressed as these two sections of code are independent of each other. Therefore, this problem was fixed by adding the same code to deleted the buttons of the screen and make the 'pauseScreenPrinted' equal to False. If this code needs to be used gain in more sections of the code, then a function can be created instead of copying the code each time.

```
if event.key == pg.K_p:
    self.paused = not self.paused
    if self.paused == False:
        for button in self.buttons:
            button.kill()
    self.pauseScreenPrinted = False
```

All aspects of the pause function are now working as expected. This updated pause screen can now be used to pause the screen when a question needs to be asked.

Loading the questions

The code explained in the 'Using CSV files' section can now be added to the game. The csv file will need to be loaded before data can be extracted from it therefore, the code to do this is implemented into the 'loadData' function.

```
def loadData(self):
    self.gameFolder = path.dirname(__file__)
    self.imgFolder = path.join(self.gameFolder, 'img')
    self.mapFolder = path.join(self.gameFolder, 'map')
    self.loadQuestions()
```

To make the management of the code easier, I have decided to keep the same 'loadQuestions' function and simply call that in the 'loadData' function. The 'loadQuestions' function is the same as the code explained previously with the only differences being with the variable holding the data and the closing of the file after the data has been extracted. This variable has been made a class variable so that it can be accessed from other parts of the program if required.

```
def loadQuestions(self):
    questionCSV = path.join(self.gameFolder, 'questions.csv')
    with open(questionCSV, 'r') as questionFile:
        reader = csv.reader(questionFile)
        next(questionFile)
        self.questionData=[]
        for line in reader:
            temp = []
            questionID = int(line[0])
            question = str(line[1])
            cAns = int(line[2])
            wAns1, wAns2, wAns3, wAns4 = int(line[3]),int(line[4]),int(line[5]),int(line[6])
            diff = str(line[7])
            level = int(line[8])
            isMaj = str(line[9])
            temp.extend((questionID,question,cAns,wAns1,wAns2,wAns3,wAns4,diff,level,isMaj))
            self.questionData.append(temp)
    questionFile.close()
```

```
[[1, '2 + 2', 4, 3, 2, 6, 5, 'easy', 1, 'FALSE'], [2, '5 + 7', 12, 13, 25, 10, 15, 'easy', 1, 'FALSE'],
[3, '15 + 17', 32, 30, 42, 23, 34, 'easy', 0, 'TRUE'], [4, '25 ◆ 5', 5, 1, 7, 11, 4, 'easy', 1, 'FALSE'],
[5, '4 x 3', 12, 11, 15, 17, 8, 'easy', 1, 'FALSE']]
```

To test if the file has been correctly loaded in I added a statement to print the contents of the CSV file to the console. The output was as expected and showed that the required data had been correctly extracted. The division symbol '÷' was not loaded correctly but this must be a problem with the IDE as when I used IDLE to run the program the symbol was correctly displayed.

```
print(self.questionData)
```

```
'25 ÷ 5'
```

Creating a question surface

When the questions need to be asked, a question could be selected from the two-dimensional 'questionData' array and blitted to a separate surface. This surface, which will be called the question surface, can then be blitted to the main screen. This method of displaying the questions on the screen is the simplest as it is difficult to remove objects from the screen once they have been blitted. By using a separate surface, the screen could be overwritten with the objects on it before the question was asked still present.

```
self.loadQuestions()
self.questionSurface = pg.Surface(self.screen.get_size()).convert_alpha()
```

Stopping the game to ask a question

The updated pause function can be used to temporarily stop the game whilst the question is asked. However, additional variables and conditions need to be added to the existing pause code to differentiate a pause triggered by the 'P' key with a pause for the questions. Otherwise, the pause screen will be displayed when the game is stopped for asking a question. To do this, I first created a variable called 'askQuestion' to identify when a question needs to be asked. I also created a variable called 'questionScreenPrinted' to identify when to output the objects which only need to be processed once. This includes the text for the question surface.

```
self.askQuestion = False
```

```
self.questionScreenPrinted = False
```

After this, I modified the code executed after the 'if self.paused:' condition to execute a different block of code if the 'self.askQuestion' variable is True. This means that all the other aspects of the game think the pause button has been pressed when in fact a question is being asked. Code to select the questions and display the required data to the screen will go here.

```
def draw(self):
    if self.paused:
        if self.askQuestion:
            pass
```

To test whether the game will be paused when a question needs to be asked, I decided to temporarily map this functionality to the 'Q' button. Therefore, when the 'Q' button is pressed, the state of the 'askQuestion' variable will reverse.

```
if event.key == pg.K_q:
    self.askQuestion = not self.askQuestion
```

```
if self.askQuestion:
    self.paused = True
else:
    self.paused = False
```

When the 'askQuestion' variable is True, the game needs to be paused therefore, the 'self.paused' variable is made True. Similarly, when the 'askQuestion' variable is False, the game needs to be resumed so the 'self.paused' variable is made False.

The 'questionScreenPrinted' variable also needs to be reset to False 'askQuestion' is False so code to do this is included here.

```
else:
    self.paused = False
    self.questionScreenPrinted = False
```

```
self.screen.blit(self.questionSurface, self.questionSurface.get_rect())
```

Finally, the question surface needs to be blitted to the screen when the 'askQuestion' variable is True. Therefore, this statement is added to the 'draw' function to achieve this purpose.

When I ran the game and pressed the 'Q' button from inside a level, the question surface was successfully blitted to the screen. The text that was added after the creation of the surface in the 'loadData' function was correctly displayed. When the 'Q' button was pressed again, the level screen was resumed and the car sprite could be controlled.

Error Encountered - The game was not pausing as expected when the 'Q' button was pressed.

Solution - I added a debug message to print when the 'Q' button was pressed. Alongside this, I added a debug message to print the state of the 'askQuestion' variable. The output from these messages made it clear that the button press was being detected but the screen was not changing. I realised that the game was not being paused when the 'Q' button was pressed, meaning that the state of the 'askQuestion' variable was never being checked.

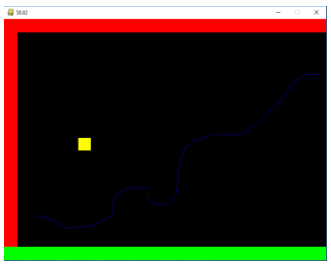
```
if event.key == pg.K_q:
    print("Q button pressed")
    print("self.askQuestion is: ", self.askQuestion)
```

```
if self.askQuestion:
    self.paused = True
else:
    self.paused = False
```

```
self.askQuestion is:  True
self.askQuestion is:  False
self.askQuestion is:  True
self.askQuestion is:  False
```

To fix this, I added code to change the 'paused' variable depending on the state of the 'askQuestion' variable.

Error Encountered - When the 'Q' button was pressed, the game now paused however the question surface was not being displayed. The car sprite image froze, as the player movement calculations were suspended due to the state of the 'paused' variable. When the button was pressed again , the game resumed.



Solution - This was being caused by the question surface not being blitted to the screen. Therefore, this problem was fixed by blitting the question surface to the game screen from inside the 'draw' function.

```
self.screen.blit(self.questionSurface, self.questionSurface.get_rect())
```

Choosing and asking the question

Questions will need to be asked numerous times throughout the game so I decided to implement the code to select the questions inside a function called 'getQuestion()'. First, a question number needs to be picked. This would be simple if there is an array containing all of the question numbers. As a result of this, I modified the 'loadQuestions()' function by creating a new array called 'questionID' and appending the data in the questionID coloumn of the csv file to this array.

```
self.questionID = []
```

```
self.questionID.append(questionID)
```

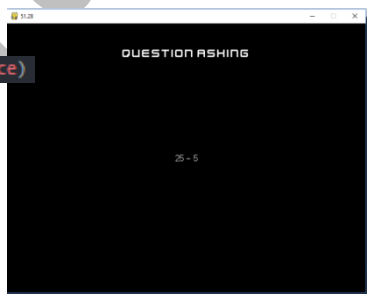
```
def getQuestion(self):
    questionNumber = random.choice(self.questionID)
```

I then used this array alongside the 'choice' method in the 'random' module to randomly select a question number. This question number was stored in the local variable 'questionNumber'.

```
self.questionID.remove(questionNumber)
questionNumberIndex = questionNumber - 1
```

To prevent the same question from being repeatedly asked, the 'questionID' value of the chosen question (i.e. the chosen question's number) is removed from the 'questionID' array. Additionally, due to the zero-based indexing used in Python, the index of the questions is one less than the value of their question number. Therefore a new variable for storing the index of the chosen question is created and assigned the value of 'questionNumber - 1'.

```
self.drawText(str(self.questionData[questionNumberIndex][1]), 20, WHITE, WIDTH/2, HEIGHT/2, surf=self.questionSurface)
```



Finally, the 'drawText' function is used to draw the question (which is extracted from the 'questionData' array using the 'questionNumberIndex') to the question surface. Now when the 'Q' button is pressed, a question is randomly chosen and printed to the screen. Shown on the left is the '25 ÷ 5' question being displayed, indicating that the division symbol is correctly loaded.

Error Encountered - A list index out of range error was being caused occasionally when the 'loadQuestions' function was executed.

```
    correctAns = self.questionData[questionNumber][2]
IndexError: list index out of range
```

Solution - This error message implies that an index in which there is no data is being provided. Upon analysing the code, I realised that the index being used currently was the question numbers in oppose to the python based indexing needed. Therefore, I fixed this problem by adding a new variable to use when extracting data from the two-dimensional array. This new variable had a value which was equal to 'questionNumber - 1'.

```
questionNumberIndex = questionNumber - 1
```

Selecting and outputting the answers

In the CSV files, there are four incorrect answers for each question. Only three of these are needed as they are mixed in with the correct answer to have a total of four options. I have decided to include the extra step of choosing three of these four wrong answers to make the question less familiar and easier to memorise as the possible answers will be different.

Two arrays to store the possible answers were created. The 'wrongAns' array stores the indexes of the incorrect answers while the 'chosenAns' array stores the indexes of the three randomly chosen incorrect answers and the correct answer.

```
wrongAns = [3,4,5,6]
chosenAns = [2,]
```

This 'for' loop is used to randomly select three of the four incorrect answers. When an answer is selected, it is removed from the 'wrongAns' array, to avoid being chosen again, and appended to the 'chosenAns' array. After this, the 'shuffle' function from the 'random' module is called to randomise the order of the answer indexes inside the 'chosenAns' array. This ensures that even when the same question is asked, the correct answer is not always in the same position.

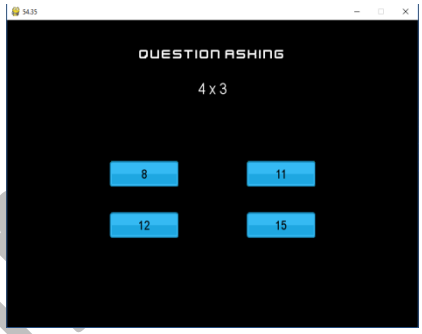
```
for i in range(3):
    randomIndex = random.choice(wrongAns)
    wrongAns.remove(randomIndex)
    chosenAns.append(randomIndex)
random.shuffle(chosenAns)
```

```
answer1 = Button(self, str(chosenAns[0]), WIDTH/3, HEIGHT/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, str(self.questionData[questionNumberIndex][chosenAns[0]]))
answer2 = Button(self, str(chosenAns[1]), WIDTH*2/3, HEIGHT/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, str(self.questionData[questionNumberIndex][chosenAns[1]]))
answer3 = Button(self, str(chosenAns[2]), WIDTH/3, HEIGHT*2/3, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, str(self.questionData[questionNumberIndex][chosenAns[2]]))
answer4 = Button(self, str(chosenAns[3]), WIDTH*2/3, HEIGHT*2/3, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, str(self.questionData[questionNumberIndex][chosenAns[3]]))
```

The answers need to be interactive so I have decided to make them buttons. I used the index of the answers for the button's tags. This simplifies the checking of whether the correct answer has been clicked as this would only be the case if the clicked button has the tag '2'. The text for each button is extracted from the 'questionData' array using the 'questionNumberIndex' index and the answer index.

```
for button in self.buttons:
    button.draw(self.questionSurface)
```

The buttons need to be drawn to the question surface so that they can be seen on this screen. To do this I called the 'draw' function on each of the buttons, using 'self.questionSurface' as the surface parameter.



I ran the game and pressed the 'Q' button from inside a level. The answers to the chosen question were successfully and randomly selected and blitted to the screen as button objects. When the mouse cursor was over the buttons they highlighted as expected. Now functionality needs to be added to check which button is clicked and display an appropriate message afterwards.

Error Encountered - When I first created the button objects for the answers there was an error message stating that one of the parameters provided towards the creation of the button object was a None type object.

```
    answer1 = Button(self, str(chosenAns[0]), WIDTH/3, WIDTH/3,
TypeError: 'NoneType' object is not subscriptable
```

Solution - I checked all of the parameters again and found that I had not missed out any essential ones. I then used a debug message to print the button tag before the button was created as I suspected this was causing the error. Running the program again caused the name error but now on the newly added 'print' statement. I decided to print the contents of the 'chosenAns' array before creating the button and found that its contents was None. To investigate, I printed the 'chosenAns' array before the 'shuffle' command was called and found that the contents were as expected. The problem was with the implementation of the 'shuffle' function. I was initially calling the function like this: but this was incorrect.

```
print(str(chosenAns[0]))
```

```
TypeError: 'NoneType' object is not subscriptable
```

```
print("chosenAns", chosenAns)
```

```
chosenAns None
```

```
chosenAns [2, 5, 3, 6]
chosenAns None
```

```
chosenAns = random.shuffle(chosenAns)
```

The function needed to be called like this: . This solved the problem.

```
random.shuffle(chosenAns)
```

Checking the answer

All the button objects in the game are checked if clicked in the 'sceneManager' class's 'update' function. Therefore, this will be where the code to check if the correct answer has been selected will need to go. To avoid these buttons interfering with the 'pause' function and other elements of the program incorporated with the button objects, I added an 'if' statement to filter out button clicks linked to the answers. All the existing code in this function will only be executed if the clicked button's tag is not either 2,3,4,5 or 6.

```
for button in self.game.buttons:
    if button.clicked == True:
        if button.tag not in ("2", "3", "4", "5", "6"):
            self.secondPrevScence = self.prevScence
            self.prevScence = self.currentScene
            self.currentScene = button.tag
```

I created three new variables at the start of the 'loadQuestions' function. The purpose of these variables is to identify when an answer button has been clicked, whether the chosen answer is correct and, the actual answer which is chosen.

```
self.answerCorrect = False
self.answerClicked = False
self.selectedAns = None
```

```
else:
    self.game.answerClicked = True
    self.game.selectedAns = self.game.questionData[self.game.questionNumberIndex][int(button.tag)]
```

`The 'else' condition to the created 'if' statement will be executed if the button clicked is for an answer. Therefore, when this is executed, the 'answerClicked' variable in the 'Game' class is made True. The actual answer which was chosen is extracted from the 'questionData' array using the 'questionNumberIndex' index and the 'button.tag' index. To do this I had to make the 'questionNumberIndex' variable into a class variable (i.e. 'self.questionNumberIndex'), allowing it to be accessed from other parts of the program. As the button's tag is a string, the 'int' function is called on it to change it to an integer type, which is required for the index.

```
if button.tag == "2":
    self.game.answerCorrect = True
else:
    self.game.answerCorrect = False
```

After this, the tag of the button is analysed and if it is "2" then the correct answer has been chosen. In this case, the 'answerCorrect' variable is made True. Otherwise, the 'answercorrect' variable is made False.

Following this, the remaining buttons are removed from the screen using a 'for' loop and the 'kill()' function.

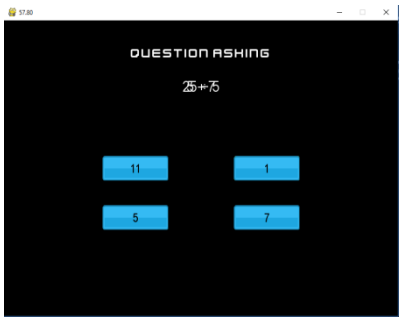
```
for button in self.game.buttons:
    button.kill()
```

```
if self.answerClicked:
    if self.answerCorrect:
        print("CORRECT ANSWER CHOSEN: {}".format(self.selectedAns))
    else:
        print("INCORRECT ANSWER CHOSEN: {}".format(self.selectedAns))
    self.askQuestion = False
    self.questionScreenPrinted = False
    self.paused = False
```

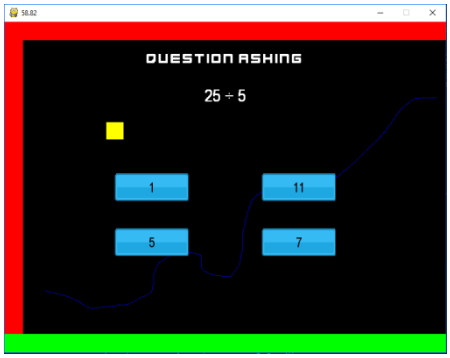
Now theses variables need to be checked in the main file. I added this code to the 'draw' function and it is only executed when the 'answerClicked' variable is True (i.e. after an answer has been chosen). If the 'answerCorrect' variable is True, then the correct answer has been chosen. Otherwise, an incorrect answer was chosen. Print statements are temporarily used to check the functionality of the program. In the final program , code to increment the score and calculate the time taken to answer the question will be here instead. After this, the question surface needs to be closed so the 'askQuestion' variable is made False, alongside the 'questionScreenPrinted' and 'paused' variables.

```
text to be printed 4 x 3
The question: 4 x 3 The correct answer: 12
drawn this button 4
drawn this button 2
drawn this button 5
drawn this button 6
drawn this button 2
CORRECT ANSWER CHOSEN: 12
```

I ran the program and correctly answered a question. The question surface closed as expected and control of the car sprite was given back. The message to say the correct answer was chosen was printed to the console, showing that the program is functioning as expected.



When the 'Q' button is pressed again the new question is blitted on top of the previous question, making it unreadable. This is happening because the question surface is not being cleared after a question is asked. Therefore, I used the 'fill()' function on the question surface to clear the surface. This code was executed under the 'if' statement checking if 'questionScreen Printed' was False, meaning that it will be executed at the start of when a question is being asked. The clearing of the screen means that the 'Question Asking' text will be overwritten. To solve this problem, I moved the statement writing this text after the fill statement.




```
if self.questionScreenPrinted == False:
    self.questionSurface.fill(0)
```

Now, when the questions are asked, instead of loading a black surface on the screen, the questions were asked with the background being the level being played. I prefer this behaviour and it is the requirement of the program as stated by Success Criteria 5. This encourages the player to answer faster as they can see their game progress in the background.

Error Encountered - Instead of resuming the game when a question is answered more questions were asked. If the mouse button was held down on an answer, that answer was selected for the next question.

Solution - The code to resume the game after answering a question was incorrect. These were the statements in the 'sceneManager' class's 'update' function. The correct variables were not being changed. The variables should be 'self.game.askQuestion' and 'self.game.paused'. Changing this solved the problem and the game resumed after an answer was chosen.

```
for button in self.game.buttons:
    button.kill()
self.askQuestion = False
self.game.questionScreenPrinted = False
self.paused = False
```

Creating Question Items

Functionality to ask questions has now been added. The questions will need to be asked when the car collides with a question item. To do this a new game sprite needs to be created.

```
class Question(pg.sprite.Sprite):
    def __init__(self, game, x, y):
        self.groups = game.allSprites, game.questionItems
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((50, 50))
        self.image.fill(WHITE)
        self.rect = self.image.get_rect()
        self.rect.center = (x, y)
```

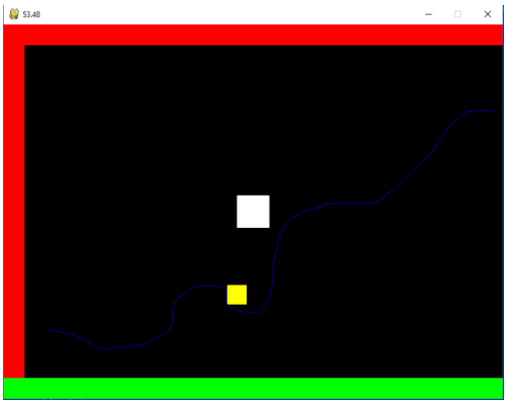
A new class called 'Question' was created in the 'sprites' file. Similarly to the classes of the other game objects, the parameters required when creating this object are: an instance of the game and the coordinates of its location. The object is added to the 'allSprites' group and a newly created 'questionItems' group. This will be required when testing for collisions.

```
self.questionItems = pg.sprite.Group()
```

For testing purposes I created a 50 pixel by 50 pixel surface to represent this object. In the final program, this object will have its own image and possibly have an animation. The temporary surface is filled white and the 'rect' object is obtained. Finally the 'rect' object's centre (i.e. the centre of the surface) is moved to the specified coordinates.

```
Question(self.game, WIDTH/2, HEIGHT/2)
```

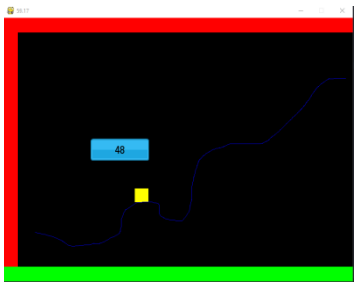
I created an instance of the 'Question' class in the 'level1' function to test if it has worked and when running the game, the question item was correctly loaded in. Now the collision code to detect when the player has collided with the question object needs to be implemented.



```
hitQuestion = pg.sprite.spritecollide(self.player, self.questionItems, True)
if hitQuestion:
    self.askQuestion = True
    self.paused = True
```

This is added inside the 'update' function. The 'spritecollide' function is used to check if there is a collision. When there is, the collided 'Question' object is removed from the group and so from the screen. Also, when there is a collision, the 'askQuestion' and 'paused' variables are made True to initiate the process of asking the question. I tested out the collision and they worked as expected.

However, when a question was asked the screen was once again filled with a black surface instead of having a transparent background. I tried to debug this but found that when the background was transparent, the buttons were not being cleared from the screen after the game was resumed. When the black surface was displayed this issue did not exist so the program is worked as programmed as the behaviour to have a transparent background was not specifically coded.

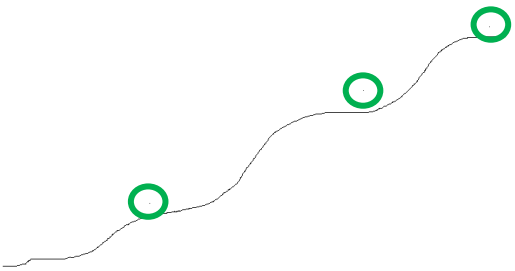


Adding Question Items to the level

The locations where the question items need to be spawned can be incorporated with the loading of the level. This allows the question items to be easily added and spawned when creating and loading the level.

I have chosen this colour to represent the questions in the level data.

Colour	Function	Denary Value	Hexadecimal Colour Code
Dark Grey	Spawn Question	2	222222



I created a new track image using Photoshop and added a dot with the colour '222222' in each of the three circles. The question objects will spawn here.

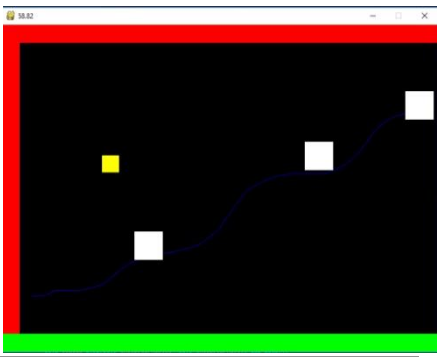
To add a new key to the level the 'Map' class needed to be modified. I created an array called 'questionData' to store the coordinates of the map where question objects will need to be spawned. Next, I added this statement to append the coordinates of the pixel whose colour matches the colour used for the question objects.

```
self.questionData = []
```

```
if pixelData == 2:
    self.questionData.append((row, col))
```

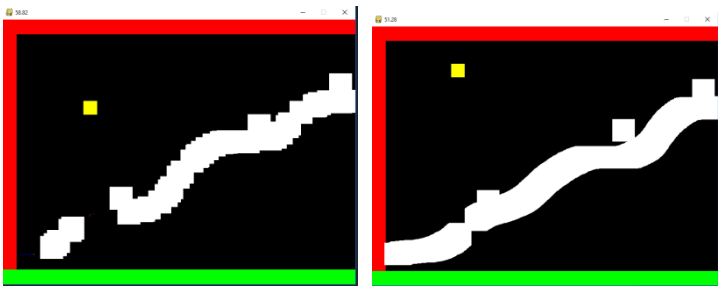
```
for coordinates in range(len(self.game.map.questionData)):
    fillx = self.game.map.questionData[coordinates][0]
    filly = self.game.map.questionData[coordinates][1]
    Question(self.game, fillx, filly)
```

Inside a new level ('level4') I added the code to look through the 'questionData' array and spawn a question at each location.



The question objects were correctly loaded into the map and the specified locations. The collisions between the objects correctly work as well.

Error Encountered - The question objects did not load correctly. Only three objects needed to be spawned but a lot more were created.



Solution - The problem was in creation of the track in Photoshop. Using the pen tool turned on anti-aliasing and added other colours to the image, interfering with the creation of game sprites. The second error was caused by using the wrong colour to draw the track. The colour with code 222222 was used instead of the required black (000000). This caused the whole track to be spawned from question objects.

Review

Question asking functionality has successfully been incorporated into the game. The question objects will be incorporated into all of the levels, making the player have to answer the questions in order to proceed in the level. Success Criteria number 5 has been fulfilled. Next, the player needs to be timed during the process of answering the question. This time will then be used to calculate a score for the level. Also text needs to be displayed to identify the result of the question answer as the player is currently unaware of whether they have answered correctly. After these functionalities have been included, all of the game's graphics (including the sprite images) will be improved to enhance the feel of the game.

Calculating the Score

A score now needs to be calculated for the completion of each level. This score will depend on how quickly the questions in the level are answered. Therefore, first the time taken to answer the questions needs to be calculated and stored.

Getting the time taken to answer the question

The current time can get gotten with the command 'pg.time.get_ticks()'. The time taken to answer the question can be calculated from the end time minus the start time. The time needs to start from when the player is asked the question, which is from the 'getQuestion' function. I have made this class variable called 'startTime' to store the time when the question answering process was started. Similarly, the time needs to be stopped after the question is answered. As a result of this, I have added a class variable called 'endTime' after the 'if self.answerClicked' statement as this will only be executed when an answer has been selected.

```
self.startTime = pg.time.get_ticks()
```

```
if self.answerClicked:
    self.endTime = pg.time.get_ticks()
```

```
self.timeTaken = round((self.endTime - self.startTime) / 1000, 2)
```

The time taken to answer the question can now be calculated using the previously explained method. The 'pg'.time.get_ticks()' function measures time in milliseconds therefore, I have divided the calculated time by 1000 to convert it into seconds. Additionally, I have used the 'round' function to round the time taken to two decimal places as sometimes it was an unnecessarily long decimal and the ultimate precision is not essential.

Creating a calculateScore function

The score awarded for answering the question can be based on the time taken to answer the question. It is preferred to answer the question quicker so the remaining time can be used to calculate the score. Therefore, answering quicker yields a longer remaining time and thus a larger score. The time allowed for each question can be calculated using the difficulty of the question. The value for this can be adjusted later if required. The difficulty of each question is stated in the CSV file in the 8th column (7th index). This can be used to set the value of a variable called 'timeAllowed'. This code was added to a new function called 'caculateScore'. The time taken to answer the question is also calculated in here. The values for the time are constants which can be easily adjusted in the settings file.

```
if self.questionData[self.questionNumberIndex][7] == "easy":
    self.timeAllowed = TIME_FOR_EASY_Q
if self.questionData[self.questionNumberIndex][7] == "medium":
    self.timeAllowed = TIME_FOR_MEDIUM_Q
if self.questionData[self.questionNumberIndex][7] == "hard":
    self.timeAllowed = TIME_FOR_HARD_Q
```

```
TIME_FOR_EASY_Q = 10
TIME_FOR_MEDIUM_Q = 15
TIME_FOR_HARD_Q = 20
```

If the question difficulty is 'easy' then 10 seconds are allowed to answer the question. Similarly, if the question difficulty is 'medium' and 'hard', the time given is 15 and 20 seconds respectively. I decided that to vary the score for each question, there should be a multiplier in the calculation. The value of this multiplier should range from 2 – 10 but it should be less common to get higher values. To do this I used the 'random.choice' function with a sequence of numbers. In this sequence there are more lower numbers than higher numbers, making them more likely to be picked.

```
multiplier = random.choice([2,2,2,2,3,3,3,3,3,4,4,4,4,4,4,5,5,5,5,5,5,6,6,6,6,6,7,7,7,7,8,8,8,8,9,9,10,15])
```

Next, I calculated the remaining time, by deducting the time taken from the allowed time, and then multiplied this by the randomly chosen multiplier. This result was rounded to two decimal places as the time calculation was more precise than required.

```
scoreIncrease = round((self.timeAllowed - self.timeTaken) * multiplier, 2)
self.score += scoreIncrease
```

A variable called 'self.score' was created in the 'new' function to keep track of the score. The calculated score is then added to the 'self.score' variable to increase the score.

```
self.score = 0
```

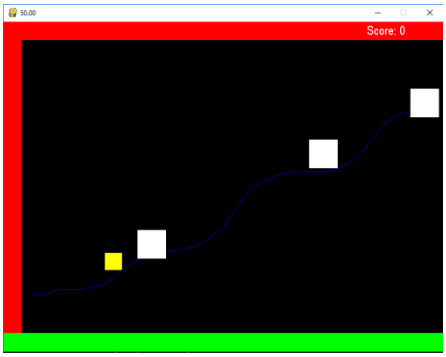
```
if self.answerCorrect:
    print("CORRECT ANSWER CHOSEN: {}".format(self.selectedAns))
    self.calculateScore()
```

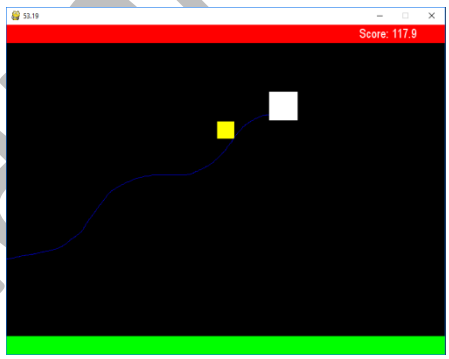
The 'calculateScore' function now needs to be called. The score should only be increased if the player has correctly answered the question. Therefore, this function is called if 'self.answerCorrect' is True. I used print statement to display the value of the score variable to the console and found that as expected, the score increased after answering questions correctly. The score now needs to be printed to the screen to make it visible to the player.

Displaying the Score

The 'drawText' function can be used to draw the score to the screen. The 'str()' function needs to be used convert the integer value of the score to a string before it can be printed. The text is positioned in the top-right corner of the screen. The score only needs to be displayed when the player is inside a level therefore, this statement is added in the 'else' block in the 'draw' function, under the 'if' statement which is executed if the current scene is not one of the menu screens.

```
self.drawText("Score: " + str(self.score), 22, WHITE, WIDTH-100, 15)
```





The score is correctly displayed at the start of the level as zero. When a question is correctly answered, the score is increased in accordance with how quickly it was answered.

Displaying the result of the question

The player is currently unaware of whether they have answered a question correctly. A message could be printed to the surface stating the result of their answer. However, as the 'paused' and 'askQuestion' variables are changed shortly afterwards, the player will not be able to see this message as the surface will quickly be changed. This can be solved by using the 'delay' function to pause the screen for a short time, enough for the user to read the message.

```
if self.answerCorrect:
    print("CORRECT ANSWER CHOSEN: {}".format(self.selectedAns))
    self.calculateScore()
    self.drawText("Correct Answer", 35, GREEN, WIDTH/2, HEIGHT*5/6, surf=self.questionSurface)
else:
    print("INCORRECT ANSWER CHOSEN: {}".format(self.selectedAns))
    self.drawText("Incorrect Answer", 25, RED, WIDTH/2, HEIGHT*5/6, surf=self.questionSurface)
```

These messages will be printed depending on the state of the 'asnwerCorrect' variable. The 'drawText' function is used to display the text 'Correct Answer' in a green colour to the question surface. Alternatively, if the player answered the question incorrectly, the text 'Incorrect Answer' is printed to the question surface in a Red colour.

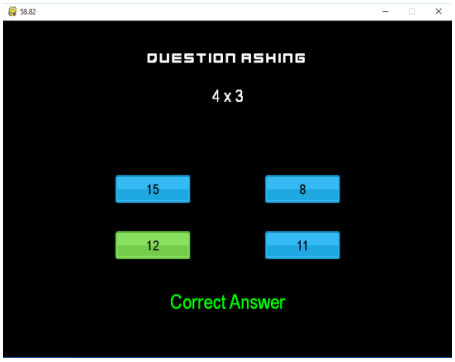
If the player answered incorrectly, the correct answer is also printed to the screen. The

```
self.drawText("Correct Answer was {}".format(self.questionData[self.questionNumberIndex][2]), 20, WHITE, WIDTH/2, HEIGHT*5/6 + 50, surf=self.questionSurface)
```

'questionNumberIndex' and '2' indexes are used to retrieve the correct answer from the 'questionData' array. After the 'if' statement, the 'time'.delay' function is used to freeze the screen so that the printed message can be read by the player. The input into this function is the length of time in milliseconds to delay the program. I decided to freeze the screen for 1.5 seconds and so provided a delay time of 1500 milliseconds.

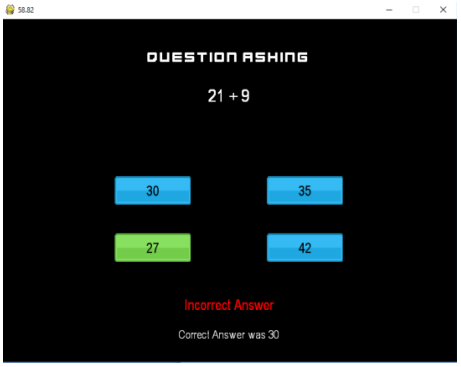
```
pg.time.delay(1500)
```

```
self.screen.blit(self.questionSurface, self.questionSurface.get_rect())
pg.display.flip()
```



Before the screen is frozen, the question surface needs to be updated so that the printed message is displayed to the player. Following this update, the 'pg.display.flip()' function is called to update the display.

The text is correctly displayed when the correct answer and incorrect answer are chosen.



Finishing the level

Currently there is no end point to the level so the player cannot complete it. An item could be added which when collides with the player the level ends. Alternatively, the walls on the furthest right side of the level could cause the level to end. When the level is finished the user will be redirected to the level select screen after a message stating the completion of the level is displayed.

End level collisions

The wall objects that are spawned at the furthest right size of the level can be added to another group to check collisions with the car sprite. I changed these walls to the number 4 in the text file that is used to make the level. Walls will still need to be spawned here, to keep the player confined inside the level, however these walls also need to be added to another sprite group. Instead of creating a new object (and so new class) to do this I modified the existing 'Wall' class. An additional parameter called 'mode' was added and if this was equal to "end", then the 'self.groups' variable (which are the groups to which this object is added) includes an additional 'endWalls' group.

```
class Wall(pg.sprite.Sprite):
    def __init__(self, game, x, y, colour, altColour=None, mode=None):
        if mode == "end":
            self.groups = game.allSprites, game.endWalls
        else:
            self.groups = game.allSprites, game.walls
```

This group was created in the 'new' function in the main file

```
self.endWalls = pg.sprite.Group()
```

I modified the code for loading the level in the level functions in the 'sceneManager' class to create a 'Wall' object with 'mode="end"' if the tile in the text file was a 4.

```
if tile == "4":
    Wall(self.game, col, row, GREEN, altColour=ORANGE, mode="end")
```

```
hitLevelEnd = pg.sprite.spritecollide(self.player, self.endWalls, True)
if hitLevelEnd:
    print("Hit Wall")
```

The collision code was then added in the 'update' function in the main file.
A test was carried out to see if the collisions where detected printing "Hit Wall" if there is a collision. When I ran the code however the collisions were not being detected. I suspected that the error must be linked to the collisions code between the walls interfering with this collision as that is calculated first. That collision code moves the player sprite so that it is no longer colliding. This meant that when this collision check took place, the sprites where no longer colliding. I fixed this problem by moving the end walls one tile before the end of the level. Now the end walls will be independent of the walls.

```
..43
..43
..43
..43
```

```
if mode != "end":
    self.image.fill(colour)
else:
    self.image.fill(altColour)
```

I used the 'mode' parameter to fill the end walls with a different colour for testing purposes. This alternative colour was provided as a parameter called 'altColour' when creating an instance of the class. Now when the program is run, the 'endWall' walls are correctly displayed in an orange colour. When the player collides with these walls the "Hit Wall" message is correctly displayed.

```
ORANGE = (255, 165, 0)
```

```
Hit Wall
```

Now when the player collides with the end walls a level complete screen needs to be called. On this screen there will be options to navigate to different game screens, such as the main menu and the level select screens.

Level complete screen

This screen will be called at the end of the level to signify the completion of the current level. There will be numerous pieces of the information on this screen including the score achieved on the level and the highscore for that level. As this screen will not be loaded with a button press the 'secondPrevScene', 'prevScene 'and 'currentScene' variables will not be changed. Therefore these variables are set before the 'loadLevel' function from the 'sceneManager' class is called to load the level complete screen.

```
hitLevelEnd = pg.sprite.spritecollide(self.player, self.endWalls, True)
if hitLevelEnd:
    self.sceneMan.secondPrevScence = self.sceneMan.prevScence
    self.sceneMan.prevScence = self.sceneMan.currentScene
    self.sceneMan.currentScene = "levelComplete"
    self.sceneMan.loadLevel("levelComplete")
```

This is the 'levelComplete' function inside the 'sceneManager' class. First the image for this screen is loaded and blitted to the screen.

```
def levelComplete(self):
    self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)
```

```
self.menuImages["levelComplete"] =  pg.image.load(path.join(self.imgFolder, 'blue_background.jpg')).convert_alpha()
```

A new image for this screen was loaded in the 'loadData' function.

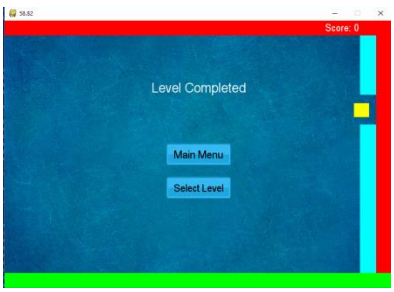
```
self.game.drawText("Level Completed", 30, WHITE, WIDTH/2, HEIGHT/4)
```

After the image is displayed the text level completed is printed to the screen in white.

```
self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/4, HEIGHT*5/8, WIDTH/5, HEIGHT/6, YELLOW, LIGHT_BLUE, "Main Menu")
self.levelSelectButton = Button(self.game, "levelSelect", WIDTH*3/4, HEIGHT*5/8, WIDTH/5, HEIGHT/6, YELLOW, LIGHT_BLUE, "Select Level")
```

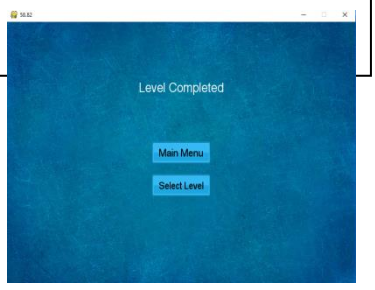
After this, two buttons are created to navigate to the main menu screen and the level select screen. These buttons are positioned towards the bottom of the screen to allow for other information to be displayed in the middle.

Error Encountered - When the program was run and the end of the level was reached, the level complete screen loaded however the level was still running and visible in the background. The buttons on the level complete screen were functional and were correctly highlighting when the cursor was over them.



Solution - The level was being updated due to the 'update' function being called on the player sprite. This is programmed to only happen when the current scene is not one from the MENU_SCREENS array in the 'settings' file. The level complete screen was not added to this array and so was causing the level to keep updating. Therefore, this problem was fixed by adding 'levelComplete' to the MENU_SCREENS array.

```
MENU_SCREENS = ["settingsMenu",
                "mainMenu",
                "levelSelect",
                "stats",
                "leaderboard",
                "shop",
                "startScreen",
                "gameOverScreen",
                "pause",
                "levelComplete"]
```



The level complete screen is now correctly loaded when the end of the screen is reached. More information can now be displayed on this screen.

Next Level button

It would be convenient to access the next level from the level complete screen as otherwise the player will need to navigate to the level select screen and pick the next level. To create this button the tag of the button

```
nextLevelTagNumber = int(self.prevScence[-1]) + 1
```

needs to be generated using the current level's tag. As the'currentScene' variables contents are currently 'levelComplete' the level that was just completed is stored in the 'prevScene' variable. The level number is the last character of the string and so can be accessed using the '-1' index. This level number is converted into an integer and incremented by one to have the level number of the next level. This value is stored in the variable 'nextLevelTagNumber' for use later.

```
levelList = list(self.prevScence)
```

Strings are immutable in python so to manipulate the 'prevScene' variable is contents are first converted into a list using the 'list' function.

The number of the next level is converted to a string using the 'str' function and replaced with the last character in the recently created list.

```
levelList[-1] = str(nextLevelTagNumber)
```

Now the 'levelList' list has the tag of the next level but the problem is that it is in list formed but required in string form. This problem was solved by using the '.join' method to join the characters in the list together into a string and store it the 'nextLevel' variable.

```
nextLevel = "".join(levelList)
```

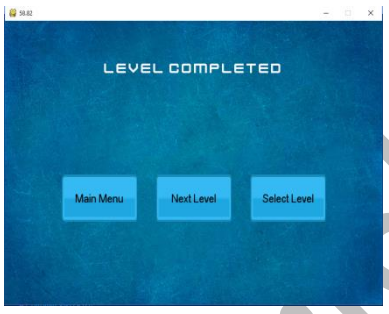
```
print("Next Level: ", nextLevel)
```

I used a print statement to print this 'nextLevel' variable to check that the program has manipulated the data as expected. The message in the console was as expected and showed that the program was correctly working.

```
Next Level:  level2
```

```
self.nextLevel = Button(self.game, nextLevel, WIDTH /2, HEIGHT*5/8, WIDTH/5, HEIGHT/6, YELLOW, LIGHT_BLUE, "Next Level")
```

Following this, the next level button was created in the 'levelComplete' function. The 'nextLevel' variable was used as the tag for this button.



The next level button was positioned in the middle of the main menu and level select buttons as this will be the most frequently pressed buttons. Moreover, the buttons' sizes have been increased to make them easier to press.

The next level button works as expected and correctly loads the next level.

Review

A level can now be completed by the player and they can navigate to other game screens at the end of the level. The score achieved on the current level and the highscore for the current level can be displayed on this screen. Moreover, the leader boards for this current level can also be linked from this screen or shown on this screen. Next the highscore for each of the levels will be implemented, followed by adding that information to the level complete screen.

Creating the highscore

The highscore achieved on each level needs to be saved so that this data is not lost when the program is not running. If the score achieved on a level is higher than the score saved as the highscore for that level then the highscore data needs to be updated. First, the highscore needs to be loaded into the game.

Creating the highscore file

The highest score that is achieved on a level needs to be saved and loaded the next time the level is played. There will be numerous values for the highscore, one for each level, making it difficult to manage the data if the scores are saved to a text file as an array. The most appropriate data type to store the highscore in is a dictionary. The level numbers can be used as keys for the scores. To save this dictionary whilst the program is not running it needs to be saved to a file.

```
import pickle
```

In a separate program, a library called 'pickle' can be used to store and retrieve this dictionary in and from a file. The first time the dictionary is saved to the file a procedure needs to be followed to construct the file. In this procedure, the dictionary is first created. To this dictionary, twelve levels have been added and the scores for each level have been set as zero. The pickle file is then created using the 'open' function. The file created is called 'highscore.pickle' and the file is opened in 'wb' mode to write to the file in bytes. Next, a function called 'dump' is called from the 'pickle' library to export the dictionary to the created file. Following this the file is closed to avoid any accidentally manipulation of the file data. The file has been created and can now be loaded in the game to access the highscore data. If the highscore for each level needs to be reset then this code can be run to create a new file and erase the existing one.

```
highscoreDict = {1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0, 9:0, 10:0, 11:0, 12:0}
exportDict = open("highscore.pickle", "wb")
pickle.dump(highscoreDict, exportDict)
exportDict.close()
```

Loading the highscore

As the highscore will be different for each level, a function was created to load the highscore for a specified level. This 'loadHighscore' function takes a parameter 'level', which is the level for which the highscore data is required. Inside this function the pickle file is opened and the dictionary inside is loaded into the class variable 'self.highscoreDict'. The file is then closed to avoid any accidental changes to it. Next a 'highscore' variable is created to store the highscore value for the current level. This value is extracted from the highscore dictionary using the 'level' key which was provided as a parameter to the function. The 'highscore' variable can now be used in the rest of the program. This function is called in the 'sceneManager' class's 'update' function when the clicked button is a level button as this is when the highscore is required. The 'currentScene' variable would have been set to the new level so the 'level' parameter for the 'loadHighscore' function is extracted from the 'currentScene' variable using the '-1' index.

```
def loadHighscore(self, level):
    highscoreFile = open("highscore.pickle", "rb")
    self.highscoreDict = pickle.load(highscoreFile)
    highscoreFile.close()
    self.highscore = self.highscoreDict[level]
```

```
self.loadHighscore(int(self.currentScene[-1]))
```

```
print("highscoreDict: ", self.highscoreDict)
print("highscore", self.highscore)
```

To test this function I printed the 'highscoreDict' and the 'highscore' variable and found that their values were as expected.

```
highscoreDict:  {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0}
```

I played the first level and set a highscore. When the program was next run, this highscore was correctly saved to the 'highscoreDict' file and the highscore for the current level was correctly loaded.

```
highscoreDict:  {1: 123.71, 2: 70.56, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0}
highscore 123.71
```

Updating the highscore

The highscore data for the current level will need to be updated if the score achieved on the level is higher than the score saved in the highscore file for that level. The parameters for this function are the score achieved on the level and the level which has been played. The old highscore is saved to a variable to be used to be printed to the level complete screen if required. The value of the data stored under the 'level' key in the 'highscoreDict' dictionary is changed to the new high score. This highscore is then saved to the 'highscore' variable. Now the file needs to be updated so that the score is saved. To do this, the highscore file is first opened in write mode. Following this, the 'dump' function from the 'pickle' library is used to write the 'highscoreDict' dictionary to the file. The file is then closed.

```
def updateHighscore(self, level, score):
    self.oldHighscore = self.highscoreDict[level]
    self.highscoreDict[level] = score
    self.highscore = score
    highscoreWriteFile = open("highscore.pickle", "wb")
    pickle.dump(self.highscoreDict, highscoreWriteFile)
    highscoreWriteFile.close()
```

```
if self.game.score > self.highscore:
    self.updateHighscore(nextLevelTagNumber-1, self.game.score)
```

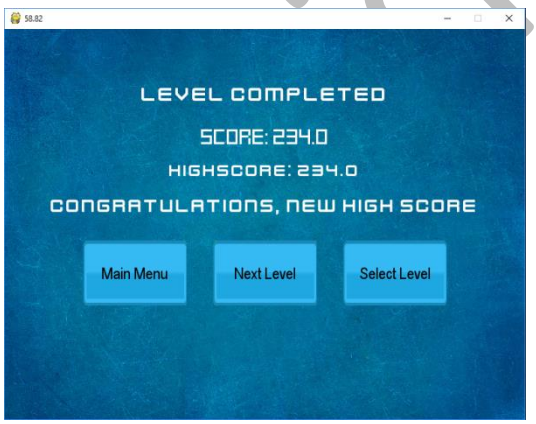
This function is called in the 'levelComplete' function if the score achieved in the level is higher than the highscore for this level. The 'nextLevelTagNumber' variable containing the integer value of the next level is used for the 'level' parameter for this function. The score achieved on the level is called from the main file using the provided instance of the game ('self.game.score').

Displaying the highscore

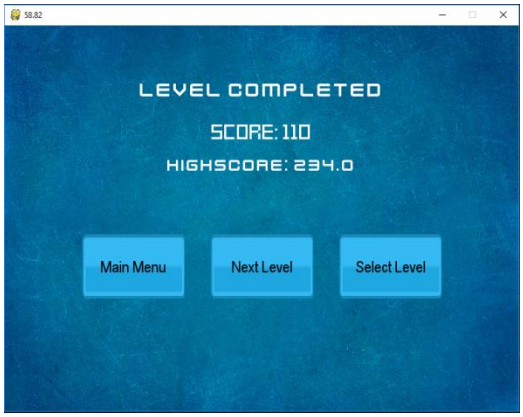
The highscore now needs to be drawn to the level complete screen so that the player can view if they have beaten their previous highscore.

```
if self.game.score > self.highscore:
    self.updateHighscore(nextLevelTagNumber-1, self.game.score)
    self.game.drawText("Congratulations, new High Score", 28, WHITE, WIDTH/2, HEIGHT*5/11, fontName=self.game.interfaceFont)
self.game.drawText("Score: {}".format(str(self.game.score)), 25, WHITE, WIDTH/2, HEIGHT*3/11, fontName=self.game.buttonFont)
self.game.drawText("Highscore: {}".format(str(self.highscore)), 25, WHITE, WIDTH/2, HEIGHT*4/11, fontName=self.game.interfaceFont)
```

This was achieved by using the 'drawText' variable to blit the score achieved on the level and the highscore to the level complete screen. If the player has achieved a new highscore on the level then the message "Congratulations, new High Score" is also displayed to the screen. I changed the 'round' function used for the score calculation to round to the nearest integer as this will be more memorable for the player



When a high score is achieved on a level the congratulations message and the values of the score and highscore are correctly displayed on the level complete screen.



I noticed that even though the score was being rounded to the nearest integer, the fact that the value was being saved as a float was causing the '.0' to be displayed at the end of the number. This issue was fixed by using the 'int' function to convert the score to an integer when it is calculated. The game was then run again and this time the score was printed without the additional '.0' and the highscore for the level was correctly loaded.

Countdown Timer for questions

A timer needs to be implemented when the questions are asked to restrict the time allowed to answer the question and also display the time remaining to answer the question. Currently if the player exceeds the time allowed for the question, the question is not passed and the player can take as long as required to answer the question. The drawback is that in the calculation of the score a negative is formed, leading to a negative score.

Creating the countdown timer function

The timer function will need to be called in every frame and the time remaining displayed on the screen will need to be updated. Despite this, the code to display the time to the screen only needs to be executed every second. This is achieved by getting the current time using the 'pg.time.get_ticks()' function and subtracting the time of the last update from it. If the answer is greater than 1000 milliseconds (1 second), then the timer needs to be updated. The variable containing the time of the last update is created and initialised in the 'new' function with the value zero. The other condition required to be met in order to execute the 'if' statement is for the time remaining to be greater than or equal to zero. This ensures than the timer doesn't count into negative numbers. Inside the 'if' statement, the 'lastCountdownTime' variable is set to the current time.

```
def countdownTimer(self):
    now = pg.time.get_ticks()
    if now - self.lastCountdownTime > 1000 and self.timeRemaining >= 0:
```

```
self.lastCountdownTime = 0
```

```
self.lastCountdownTime = now
```

The calculation of the time allowed is currently happening in the 'calculateScore' function however this information will be required in this function, which will be executed before the 'calculateScore' function. To solve this issue, I moved the code to work out the time allowed into a separate function called 'getTimeAllowed' and called it after the 'getQuestion' function, meaning that it is only executed once. This code could have been moved to the 'countdownTimer' function however this would cause unnecessary execution of code each time the function is called. A variable called 'timeRemaining' was created to store the remaining time. Initially the value of this variable is the same as the value of 'timeAllowed' therefore, the 'timeRemainin' variable is created and initialized in the 'getTimeAllowed' function. Moreover, a variable called 'timeOut' was created to identify when the time has finished. This variable is also created and initialized in this function with a value of False.

```
def getTimeAllowed(self):
    if self.questionData[self.questionNumberIndex][7] == "easy":
        self.timeAllowed = TIME_FOR_EASY_Q
    if self.questionData[self.questionNumberIndex][7] == "medium":
        self.timeAllowed = TIME_FOR_MEDIUM_Q
    if self.questionData[self.questionNumberIndex][7] == "hard":
        self.timeAllowed = TIME_FOR_HARD_Q
    self.timeRemaining = self.timeAllowed
    self.timeOut = False
```

```
self.getTimeAllowed()
```

As the text of the time remaining is blitted to the screen every second, if the current text is not removed then the text will begin overlapping and become unreadable. It is difficult to remove an item from the screen once it was been blitted to it. The solution which I came up with was to fill the screen with the background colour in the rectangle in which the text will be. This will overwrite the current text and allow the new text to be displayed on a clear background. For testing purposes, the colour of this rectangle which will be filled is set to white. The text was then drawn to the question surface using the 'drawText' function. The value of the 'timeRemaining' function was used to achieve this.

```
self.questionSurface.fill(WHITE, (WIDTH*7/8-80, HEIGHT/8-25, 160, 50))
```

```
self.drawText("Time Remaining: {}".format(self.timeRemaining), 20, BLUE, WIDTH*7/8, HEIGHT/8, surf=self.questionSurface)
```

```
    self.timeRemaining -= 1
if self.timeRemaining == -1:
    self.timeOut = True
    self.answerClicked = True
```

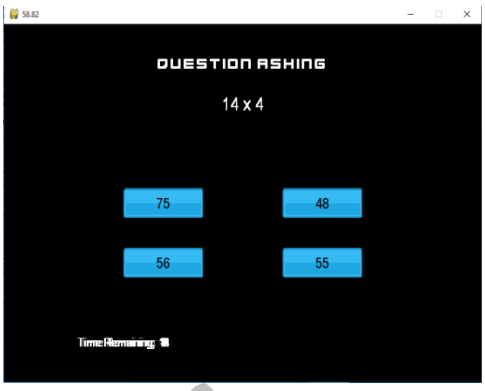
Now that the remaining time has been blitted, the time is decreased by 1. A check if made to see if the remaining time is equal to '-1' and if so, the 'timeOut' variable is made True, and the 'answerClicked' variable is made True. This will trigger conditions to stop the question being asked and display a suitable message.

This function was called in the 'else' block of the 'if' statement checking if the answer has been clicked. This means that this function will be called as long as the answer is not chosen.

```
else:
    self.countdownTimer()
```

This function worked as expected. The white rectangle being blitted to the screen successfully cleared the previous text and the time correctly decreased every second to zero.

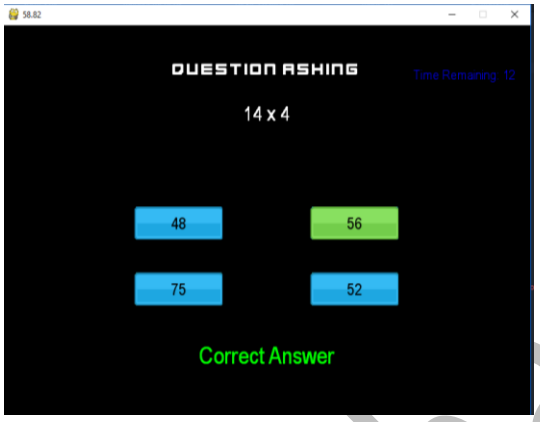
The image on the right shows the result of not clearing the screen. As shown, the text is unreadable.



Time out message

The question now needs to stop being asked when the time reaches zero. The question has been stopped by making 'answerClicked' True. An additional statement was added to the 'if' statement checking whether the question has been answered correctly to execute when the time is up. If the 'timeOut' variable is 'True' then a "Time Out" message and the correct answer is printed to the screen.

```
elif self.timeOut:
    self.drawText("Time Out", 25, RED, WIDTH/2, HEIGHT*5/6, surf=self.questionSurface)
```



When the time remaining reaches zero, the "Time Out" message is correctly printed to the screen alongside the text revealing the correct answer. The screen is then paused for 1.5 seconds before it returns to the level screen.

Review

```
self.game.score = 0
```

High scores for each level, printing the scores to the level complete screen, implementing a countdown timer in the question screen and restricting the player from answering when the time is up are all elements of the program which have been successfully added. When the wall and platform objects are deleted for the next level, the score variable in the main file is also reset to zero. Next, the data for the statistics screen needs to be collected, stored and printed to the statistics screen. Following this, the leader board functionality needs to be added to the game. It might be useful to use the dictionary-in-a-file feature provided by the 'pickle' library in these tasks to store the permanent data in a dictionary file. Information about the number of coins collected will need to be displayed on the statistics screen therefore it will be useful to first implement the coin objects into the game.

Adding coin objects

Coins will be rewards that can be collected by the player. These objects will be scattered across the map in the level and colliding with them will collect the coin. Coins will also be awarded as a reward when a level is completed depending on the score achieved. These coins will then be able to be traded in the in-game shop to buy different vehicles. As the coins will be a game sprite their class is created in the 'sprites' file.

Creating the Coin class

The parameters for the 'Coin' class are an instance of the game, and the coordinates at which the coin will need to be displayed. A new group is created in the 'new' function in the main file for the coins. The coin objects are then added to the 'allSprites' and 'coins' groups. Following this, for testing purposes, a surface is created for the coin objects. In the final program there will be an image for the coins that will be animated. This surface is filled with an orange colour and the 'rect' object of the image is retrieved. The coordinates at which the coin needs to be displayed are then used to locate the 'rect' object at this position.

```
self.coins = pg.sprite.Group()
```

```
class Coin(pg.sprite.Sprite):
    def __init__(self, game, x, y):
        self.groups = game.allSprites, game.coins
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((30,30))
        self.image.fill(ORANGE)
        self.rect = self.image.get_rect()
        self.rect.center = (x, y)
```

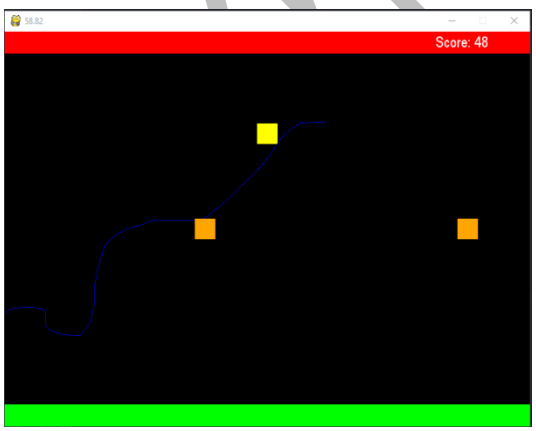
A variable for storing the amount of coins collected was created in the 'new' function in the main file and initialized with a value of zero. In the final program the previous coin total will be loaded in and this variable will be initialised with that value.

```
self.coinAmount = 0
```

The collisions between the player and the coin objects are then checked in the 'update' function on the main file. The 'pg.sprite.spritecollide' function is used to achieve this, and when there is a collision, the coin object is deleted from the coins group. This causes the coin to no longer be displayed to the screen, which is what is required as the coin has been collected. A collision triggers the 'coinAmount' variable to be incremented by one. The contents of this variable were printed to the console to check if the collisions were being detected. Instances of the coin objects need to be created before the game is run. To do this, I called the 'Coin' class in the 'level1' function at various locations across the track.

```
hitCoin =  pg.sprite.spritecollide(self.player, self.coins, True)
if hitCoin:
    print("A Coin has been hit")
    self.coinAmount += 1
    print("Total Coins: {}".format(self.coinAmount))
```

```
Coin(self.game, WIDTH/4, HEIGHT/2)
Coin(self.game, WIDTH*3/4, HEIGHT/2)
Coin(self.game, WIDTH*5/4, HEIGHT/2)
Coin(self.game, WIDTH*7/4, HEIGHT/2)
Coin(self.game, WIDTH*9/4, HEIGHT/2)
```



When the program was run and the level one function was called the coins correctly loaded. When the player sprite collided with the coin object, the coin was deleted from the screen as expected and the message stating that a coin has been hit and the coin total was printed to the console.

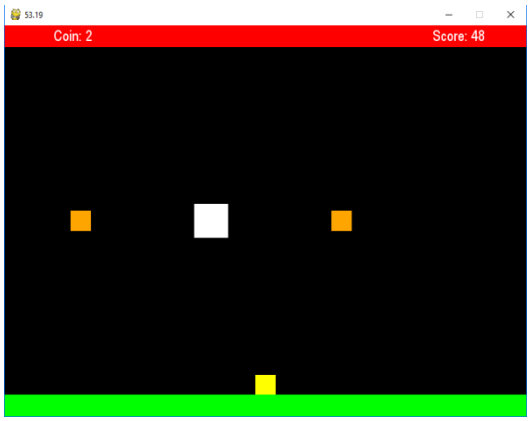
```
A Coin has been hit
Total Coins: 1
```

The coin total now needs to be blitted to the screen so that the player can view this total as they play.

Displaying the coin total

Similar to the score total, the coin total needs to be displayed while the user is playing the level.

```
self.drawText("Coin: " + str(self.coinAmount), 22, WHITE, 100, 15)
```



This was achieved by using the 'drawText' function to blit the 'coinAmount' in the top-left corner of the screen in white. This statement was added to the 'draw' function to be executed when the current screen is a level screen.

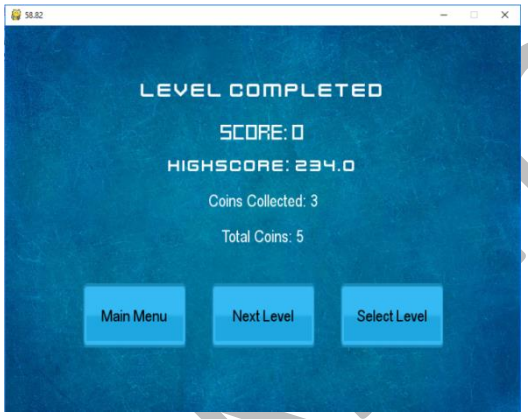
When the program was run, the coin total was correctly displayed to the screen and increased as coins were collected. It would also be informative to display the coin information to the level complete screen so that the player can analyse their performance during level.

I decided to calculate the number of coins collected in the current level, as this will be different to the total number of coins. To do this a new variable called 'coinCollected' was created in the 'sceneManager' class. Now when there is a collision, this variable is also incremented by one. When a new level is loaded, this variable is reset to zero. This means that this variable will store the number of coins collected during the current level. This variable was then used when printing the coin totals in the 'level complete' screen.

```
self.game.score = 0
self.coinCollected = 0
```

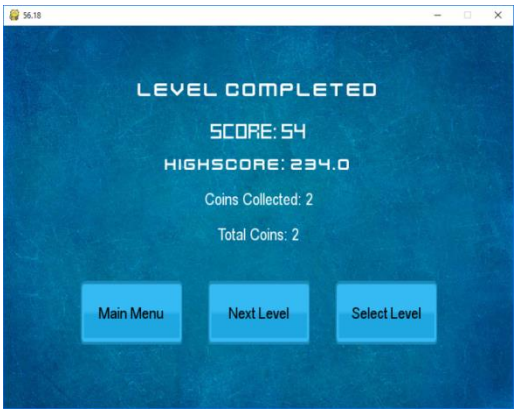
```
self.game.drawText("Coins Collected: {}".format(self.coinCollected), 25, WHITE, WIDTH/2, HEIGHT*5/11, fontName=None)
self.game.drawText("Total Coins: {}".format(self.game.coinAmount), 25, WHITE, WIDTH/2, HEIGHT*6/11, fontName=None)
```

The buttons on the level complete screen were rearranged by moving them down. This gives more space on the screen for other information.



The values of the total coins and the coins collected for the current level were correctly displayed to the level complete screen.

I tested out the program by completing another level and found that the coins collected amount was different to the total coins, as expected.



Inside the shop screen, clicking on buttons to buy different items will decrease the coin amount by a specific amount. This functionality will be implemented at a later stage. Now, this coin data can be used, alongside the other data produced in the game, to produce the statistics screen.

Creating the Statistics screen

As explained in the Design section when creating the statistics screen, this screen will contain interesting and useful facts about the player regarding their progress through the game. This will include: the total number of questions they have answered, the total number of question which they have got correct/incorrect, the total number of games played, the total number of coins collected/spent and, the total and average score. This information will need to be saved in a file when the program is not running and loaded when it is to retrieve the data. It will be useful to store this information together therefore, I decided to create another dictionary using the 'pickle' library.

Creating the information dictionary

```
def statistics():
    statsDict = {"gamesPlayed":0, "questAnswered":0, "correctAnswerQuesEasy":0, "correctAnswerQuesMed":0, \
                 "correctAnswerQuesHard":0, "vehicleUnlock":0, "coinTotal":0, "coinSpent":0, "totalScore":0}
    exportDict = open("stats.pickle", "wb")
    pickle.dump(statsDict, exportDict)
    exportDict.close()
```

This code was added to the file that was used to create the 'pickle' dictionary for the high scores. I implemented this code in a separate function, meaning that if the statistics data needs to be reset, this function would simply need to be called. The dictionary is first created and saved in the variable 'statsDict'. I realised that certain pieces of information such as the number of incorrectly answered questions and the average score can be calculated using the other information in the dictionary. Therefore, it would be unnecessary to store this data as it would increase the file size, making it take longer to read and write to. A new file called 'stats.pickle' was created and opened in write bytes mode. The 'dump' function was then used to save the dictionary to the 'pickle' file. After this, the opened file was closed.

Loading the statistics data

The statistics data can now be loaded into the game and certain variables, such as the 'coinAmount' variable, can be initialized with their correct amounts. A function was created to load the statistics data and this function was called in the 'loadData' function. The 'stats.pickle' file was opened in read bytes mode using the 'open' function. The dictionary was loaded using the 'pickle.dump' function, and saved to the class variable 'self.statsData'. As a test, the 'coinAmount' variable was initialized with the value for the coin total in the dictionary. I used some print statement to show the contents of the 'statsDict' dictionary and the value of the 'coinAmount' variable to check that this code as functioning as expected.

```
def loadstatsData(self):
    statsFile = open("stats.pickle", "rb")
    self.statsData = pickle.load(statsFile)
    statsFile.close()
    self.coinAmount = self.statsData["coinTotal"]
```

```
print("Loaded stats fle")
print("statsDict: ", self.statsData)
print("highscore", self.coinAmount)
```

```
Loaded stats fle
statsDict:  {'gamesPlayed': 0, 'questAnswered': 0, 'correctAnswerQuesEasy': 0, 'correctAnswerQuesMed': 0,
'correctAnswerQuesHard': 0, 'vehicleUnlock': 0, 'coinTotal': 0, 'coinSpent': 0, 'totalScore': 0}
```

As expected, the dictionary containing the statistics data was correctly loaded and printed to the console. Each of the keys of the dictionary had a value of zero as this was the first time the file was being used in the game. The 'highscore' variable also correctly printed with the expected value of zero.

```
coinAmount 0
```

The data now needs to be written to the 'stat-pickle' file when the game is closed to save the data.

Updating the statistics data

The code for updating the 'stats-pickle' file was implemented inside a function called 'updateStatsData' as this function will need to be called in all locations from which the game can be closed. The file is opened in write bytes mode using the 'open' function. For testing, I printed the value of the 'coinTotal' key in the dictionary before and after it was changed. The value of the 'coinTotal' in the dictionary was then changed to the value in the 'coinAmount' variable. This updated dictionary was then 'dumped' to the file.

```
def updateStatsData(self):
    statsFile = open("stats.pickle", "wb")
    print("Coin total before update", self.statsData["coinTotal"])
    self.statsData["coinTotal"] = self.coinAmount
    pickle.dump(self.statsData, statsFile)
    print("coin total after update:", self.statsData["coinTotal"])
    statsFile.close()
```

This function was called in the 'events' function when the red-cross in the top-right corner was pressed.

```
if event.type == pg.QUIT:
    self.updateStatsData()
```

```
Coin total before update 0
coin total after update: 2
```

I then ran the program, collected some coins and closed the game. The coin updated messages printed to the console with different values as expected. I then ran the program again to analyse the 'statsDict' dictionary and found that the update had successfully worked as the value of the 'coinTotal' in the dictionary was different ('2').

```
Loaded stats fle
statsDict:  {'gamesPlayed': 0, 'questAnswered': 0, 'correctAnswerQuesEasy': 0, 'correctAnswerQuesMed': 0,

        'correctAnswerQuesHard': 0, 'vehicleUnlock': 0, 'coinTotal': 2, 'coinSpent': 0, 'totalScore': 0}
```

Implementing statistics data collectors

Variables now need to be implemented in multiple parts of the game to track the data required for the game statistics. I modified the 'loadStatsData' function to use all of the data in the 'stats-pickle' file. Now all of the data is stored in these class varaibes to be changed at certain places. Similarly, the 'updateStatsData' function has been modified to write all of the updated values of these class variables to the 'stats-pickle' file.

```
def loadstatsData(self):
    statsFile = open("stats.pickle", "rb")
    self.statsData = pickle.load(statsFile)
    statsFile.close()
    self.coinAmount = self.statsData["coinTotal"]
    self.gamesPlayed = self.statsData["gamesPlayed"]
    self.questAnswered = self.statsData["questAnswered"]
    self.correctAnswerQuesEasy = self.statsData["correctAnswerQuesEasy"]
    self.correctAnswerQuesMed = self.statsData["correctAnswerQuesMed"]
    self.correctAnswerQuesHard = self.statsData["correctAnswerQuesHard"]
    self.vehicleUnlock = self.statsData["vehicleUnlock"]
    self.coinSpent = self.statsData["coinSpent"]
    self.totalScore = self.statsData["totalScore"]
```

```
def updateStatsData(self):
    statsFile = open("stats.pickle", "wb")
    self.statsData["coinTotal"] = self.coinAmount
    self.statsData["gamesPlayed"] = self.gamesPlayed
    self.statsData["questAnswered"] = self.questAnswered
    self.statsData["correctAnswerQuesEasy"] = self.correctAnswerQuesEasy
    self.statsData["correctAnswerQuesMed"] = self.correctAnswerQuesMed
    self.statsData["correctAnswerQuesHard"] = self.correctAnswerQuesHard
    self.statsData["vehicleUnlock"] = self.vehicleUnlock
    self.statsData["coinSpent"] = self.coinSpent
    self.statsData["totalScore"] = self.totalScore
    pickle.dump(self.statsData, statsFile)
    statsFile.close()
```

```
self.game.gamesPlayed += 1
```

The 'gamesPlayed' variable is increased by one in the 'sceneManager'' class's 'update' function when the button selected has a tag which is a level screen.

The 'questAnswered' variable is incremented by one when a question is answered, which is when the 'answerClicked' variable in the 'draw' function is True.

```
if self.answerClicked:
    self.questAnswered += 1
```

```
if self.answerCorrect:
    if self.questionDiff == "easy":
        self.correctAnswerQuesEasy += 1
    elif self.questionDiff == "medium":
        self.correctAnswerQuesMed += 1
    elif self.questionDiff == "hard":
        self.correctAnswerQuesHard += 1
```

The 'correctanswerQuesEasy', 'correctanswerQuesMed' and 'correctanswerQuesHard' variables are also incremented by one when if the answer clicked is correct, which is when the 'answerCorrect' variable is True. When the time allowed for the question is determined, a variable called 'questionDiff' is created to store the difficulty of the question which has been asked. This variable is used to check the difficulty of the question here to increment the appropriate variable.

The 'totalScore' variable is updated by adding the 'score' variable to it when the level has finished by colliding with the end walls.

```
if hitLevelEnd:
    self.totalScore += self.score
```

```
Loaded stats fle
statsDict:  {'gamesPlayed': 1, 'questAnswered': 2, 'correctAnswerQuesEasy': 2, 'correctAnswerQuesMed': 0,

          'correctAnswerQuesHard': 0, 'vehicleUnlock': 0, 'coinTotal': 4, 'coinSpent': 0, 'totalScore': 0}
```

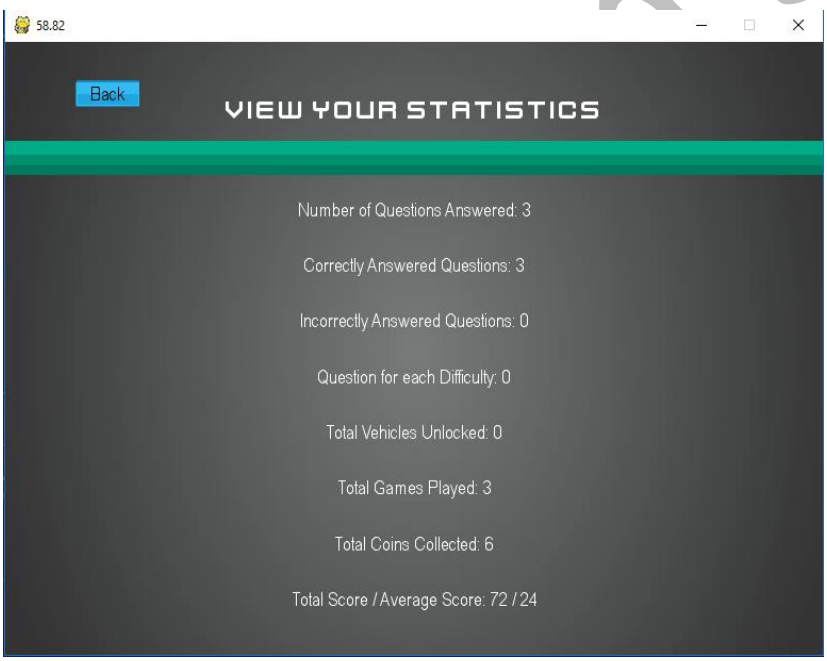
I ran the program and played one level, in which I collected 4 coins and answered two questions. I closed the game and then ran it again to see the contents of the 'statsDict' dictionary. The values of all of the keys of the dictionary were successfully changed to their expected values. Now this data can be displayed on the statistics screen.

Adding the information to the Statistics screen

Adding the collected information to the statistics screen was simply a case of using the 'drawText' function to blit text containing the information to the screen.

```
def stats(self):
    self.game.drawText("View your statistics", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)
    correctAnswerQuest = self.game.correctAnswerQuesEasy + self.game.correctAnswerQuesMed + self.game.correctAnswerQuesHard
    incorrectAnswerQuest = self.game.questAnswered - correctAnswerQuest
    averageScore = int(round(self.game.totalScore / self.game.gamesPlayed,0))
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)
    self.game.drawText("Number of Questions Answered: {}".format(self.game.questAnswered), 18, WHITE, WIDTH/2, HEIGHT*3/11)
    self.game.drawText("Correctly Answered Questions: {}".format(correctAnswerQuest), 18, WHITE, WIDTH/2, HEIGHT*4/11)
    self.game.drawText("Incorrectly Answered Questions: {}".format(incorrectAnswerQuest), 18, WHITE, WIDTH/2, HEIGHT*5/11)
    self.game.drawText("Question for each Difficulty: {}".format("0"), 18, WHITE, WIDTH/2, HEIGHT*6/11)
    self.game.drawText("Total Vehicles Unlocked: {}".format("0"), 18, WHITE, WIDTH/2, HEIGHT*7/11)
    self.game.drawText("Total Games Played: {}".format(self.game.gamesPlayed), 18, WHITE, WIDTH/2, HEIGHT*8/11)
    self.game.drawText("Total Coins Collected: {}".format(self.game.coinAmount), 18, WHITE, WIDTH/2, HEIGHT*9/11)
    self.game.drawText("Total Score / Average Score: {} / {}".format(self.game.totalScore, averageScore), 18, WHITE, WIDTH/2, HEIGHT*10/11)
```

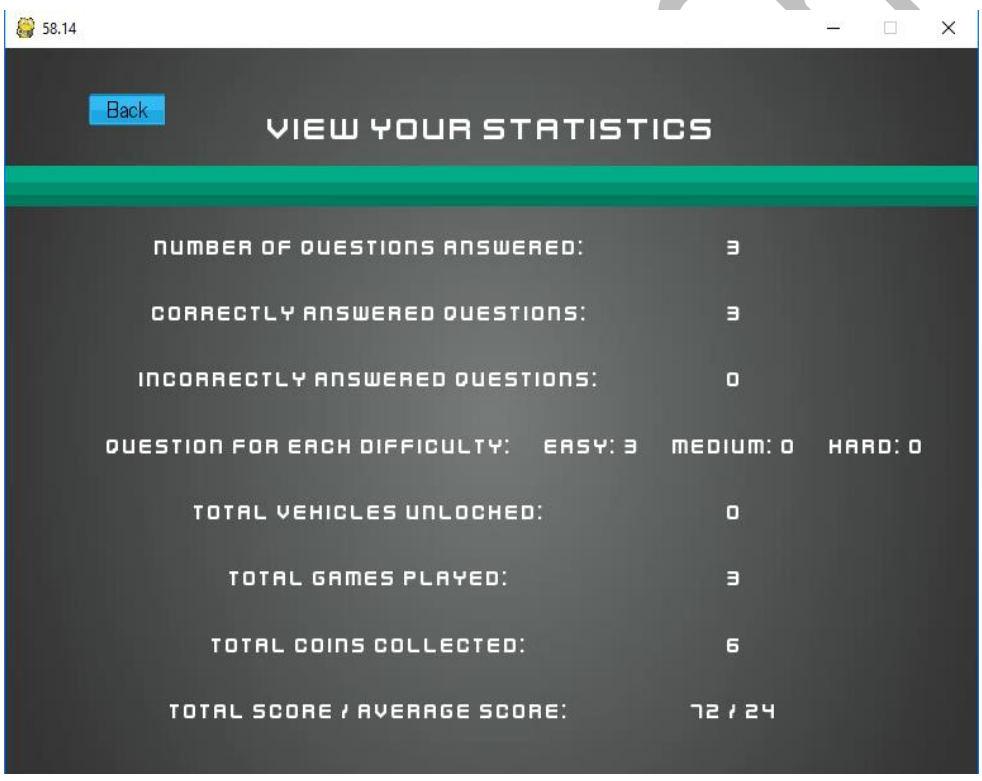
Despite this, some calculations were made for the before mentioned facts which could be calculated. The total number of correct answers was calculated by adding the number of correct easy, medium and hard questions. This amount was then used to calculate the number of incorrectly answered questions, by subtracting this variable from the total number of questions answered. The average score was calculated by dividing the total score by the number of games played, and rounding this answer to the nearest integer. These variables were then used to display all the information to the screen. Currently no coins have been spent and no vehicles have been collected therefore, a zero is printed for these values.



All the information has been displayed to the screen as expected. However, the text looks squashed together, making it difficult to read. As drawn when designing the level in the 'Design' section, it would be more clear if the fact titles were towards the left side of the page and the actual values were towards the right. This additional space will yield better readability of the text.

```
def stats(self):
    self.game.drawText("View your statistics", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)
    correctAnswerQuest = self.game.correctAnswerQuesEasy + self.game.correctAnswerQuesMed + self.game.correctAnswerQuesHard
    incorrectAnswerQuest = self.game.questAnswered - correctAnswerQuest
    averageScore = int(round(self.game.totalScore / self.game.gamesPlayed,0))
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)
    self.game.drawText("Number of Questions Answered:", 18, WHITE, WIDTH*3/8, HEIGHT*3/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(self.game.questAnswered), 18, WHITE, WIDTH*3/4, HEIGHT*3/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Correctly Answered Questions:", 18, WHITE, WIDTH*3/8, HEIGHT*4/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(correctAnswerQuest), 18, WHITE, WIDTH*3/4, HEIGHT*4/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Incorrectly Answered Questions:", 18, WHITE, WIDTH*3/8, HEIGHT*5/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(incorrectAnswerQuest), 18, WHITE, WIDTH*3/4, HEIGHT*5/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Question for each Difficulty:", 18, WHITE, WIDTH*5/16, HEIGHT*6/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Easy: {}     Medium: {}     Hard: {}".format(self.game.correctAnswerQuesEasy, self.game.correctAnswerQuesMed,\
                        self.game.correctAnswerQuesHard), 18, WHITE, WIDTH*3/4, HEIGHT*6/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Total Vehicles Unlocked:", 18, WHITE, WIDTH*3/8, HEIGHT*7/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format("0"), 18, WHITE, WIDTH*3/4, HEIGHT*7/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Total Games Played:", 18, WHITE, WIDTH*3/8, HEIGHT*8/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(self.game.gamesPlayed), 18, WHITE, WIDTH*3/4, HEIGHT*8/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Total Coins Collected:", 18, WHITE, WIDTH*3/8, HEIGHT*9/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(self.game.coinAmount), 18, WHITE, WIDTH*3/4, HEIGHT*9/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Total Score / Average Score:", 18, WHITE, WIDTH*3/8, HEIGHT*10/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{} / {}".format(self.game.totalScore, averageScore), 18, WHITE, WIDTH*3/4, HEIGHT*10/11, fontName=self.game.interfaceFont2)
```

The text on the screen has now been rearranged to improve the aesthetics of the screen whilst also achieving a clearer presentation of the information. The data itself for each fact is printed at a different location (closer to the right side of the page) than the text defining the fact. Additionally, a new font was loaded into the game in the 'loadData' function and saved in the variable 'self.interfaceFont2'. This font was used when blitting the text to the screen. The size of the text was also modified until I found a size in which the text appeared the correct size in relation to the page. This desired font size was 18. Another difference in the code now is that the values for the easy, medium and hard questions are being blitted.



I ran the program and loaded the statistics screen from the main menu. The screen now appeared much better than before. The information on the screen is more clear and easier to read. I adjusted the spacing between the easy, medium and hard words until there was a sufficient amount of space in between them. Moreover all of the facts are perfectly spaced out vertically in accordance with the design of the background image used for this page.

Review

The statistics screen and all the information required for it has successfully been implemented into the game. Next the leader board functionality needs to be added to the game. After this, more questions can be added to the question database, and the functionality for asking specific difficulty questions can be added.

Creating the Leader board screen

The leader board screen will contain the top ten scores achieved on all of the levels. The player will be able to easily view this information and analyse their performance. As mentioned during the Analysis and Design sections, the purpose of these screens will be to encourage the player to replay the level and achieve a higher score. During this process, they will answer more questions which will improve their mental calculation speeds and ultimately achieve the purpose of the game. To aid this motivation, a name for the achieved score will be provided and saved. This means that multiple players on the same device will be able to record and view their progress throughout the game compared to the other players. In additional to this, the data when the score achieved will also be saved. First, a method of storing the top ten results will be implemented.

Storing the top ten scores

Currently, the highscore data for each level is stored in one 'pickle' dictionary file. Adding more keys to this dictionary so that each level has ten scores will make this dictionary very large and thus difficult to manage. Therefore, I decided to create a separate 'pickle' dictionary for each level and only load the required file.

```
def level1():
    level1Dict = {1:[0, "name","time"], 2:[0, "", ""], 3:[0, "", ""], 4:[0, "", ""], 5:[0, "", ""], 6:[0, "", ""], 7:[0, "", ""], \
                  8:[0, "", ""], 9:[0, "", ""], 10:[0, "", ""], 11:[0, "", ""], 12:[0, "", ""]}
    exportDict = open("level1.pickle", "wb")
    pickle.dump(level1Dict, exportDict)
    exportDict.close()
```

In the same file where the other 'pickle' dictionaries were created (which will be called the 'dictionary' file from here onwards), I created a dictionary to store the scores achieved for the first level. The keys of this dictionary are number 0-9, to be in-accordance with Python's zero-based indexing, and will store the scores in descending order. The score held in the zero index will be the highest and the score stored in the ninth index will be the lowest. The values for the keys are arrays with three element, the score, the name of the player achieved and the data and time achieved. I used the same previously explained code to create this 'pickle' dictionary.

```
def loadLevel1():
    exportDict = open("level1.pickle", "rb")
    dict = pickle.load(exportDict)
    print(dict)
    print(dict[0][0])
    print(dict[0][1])
    print(dict[0][2])
```

I created a function to load the created file to test if it had successfully been created. In this test, I also checked that the each element of the value could be correctly addressed using the appropriate indexes. The output to the console was as expected and now this dictionary can be used in the game.

```
0
name
time
```

```
{1: [0, 'name', 'time'], 2: [0, '', ''], 3: [0, '', ''], 4: [0, '', ''], 5: [0, '', ''], 6: [0, '', ''],
 7: [0, '', ''], 8: [0, '', ''], 9: [0, '', ''], 10: [0, '', ''], 11: [0, '', ''], 12: [0, '', '']}
```

Loading the new scores dictionary

The code for loading the highscore is in the 'loadHighscore' function in the 'sceneManager' class. The file name of the 'pickle' dictionary will vary according to the level therefore, a method of opening any level's dictionary file needs to be implemented. I approached this problem similar to how the 'nextLevelTag' was created for the next level button. An array containing all of the characters in the filename apart from the level number was created and saved as 'fileNameList'. The level number is provided as a parameter to this function. This value is inserted into the index position five in the list using the 'insert' function. The list of individual charaters is then joined into astring using the 'join' function. This string is saved in a class varaible called 'self.levelFileName'. This name is then used to open the scores file as before.

```
def loadHighscore(self, level):
    fileNameList = ["l","e","v","e","l",".","p","i","c","k","l","e"]
    fileNameList.insert(5, str(level))
    self.levelFileName = "".join(fileNameList)
    highscoreFile = open(self.levelFileName, "rb")
```

Updating the dictionary with the achieved score

The score achieved at the end of a level needs to be added to the scores dictionary is required. It will only need to be added if the achieved score is greater than the value of the lowest score in the dictionary. In this case, the position where the score needs to be inserted needs to be determined. The dictionary stores the score data in descending order therefore, after the insertion point is determind, all the values below this position need to be shifted down without losing any information. The 'updateHighscore' function was modified to implement this procedure.

```
index = 11
for j in range(10):
    if score > self.highscoreDict[9-j][0]:
        index = 9-j
```

A variable called index is created to store the index of the position where the current score will need to be added in the dictionary. This variable is initialized with the value 11. A 'for' loop is used to check each score value inside the dictionary. The '9-j' index is checked because this will mean that the check begins from the bottom. This way if the score is bigger than the 7th index score but not the 6th index score, then the index value will be 7. If the scores were checked from the top then it will not be possible to pinpoint the index. If after this 'for' loop the value of the 'index' variable is 11, the score is too low to need to be added to the dictionary file.

```
addedScore = True
if index != 11:
    for i in range(10-index):
        if addedScore:
            tempScore = self.highscoreDict[index+i][0]
            tempName = self.highscoreDict[index+i][1]
            tempDate = self.highscoreDict[index+i][2]
            self.highscoreDict[index][0] = score
            self.highscoreDict[index][1] = self.scoreName
            self.highscoreDict[index][2] = self.scoreDate
            addedScore = False
        else:
            tempScore1 = self.highscoreDict[index+i][0]
            tempName1 = self.highscoreDict[index+i][1]
            tempDate1 = self.highscoreDict[index+i][2]
            self.highscoreDict[index+i][0] = tempScore
            self.highscoreDict[index+i][1] = tempName
            self.highscoreDict[index+i][2] = tempDate
            tempScore = tempScore1
            tempName = tempName1
            tempDate = tempDate1
```

This index is now used to update the 'highscoreDict' dictionary, in preparation for writing this dictionary to the 'pickle' file for the current level. A variable called 'addedScore' is created to check when the current score has been added to the dictionary. The concept of shifting all the values down lies with looking at the values which will be overwritten and storing them in temporarily variables. The values inside the next index will then also be stored in temporary variables whilst the values of the initial temporary variables are written to this index. The number of times this loop needs to happen depends on the 'index' variable's value. This is because the close towards the zero index that the score needs to be inserted, the more values needs to be moved. Therefore, the number of times this loop is needed is calculated using '10-index'.

In the first iteration, the current score needs to be written to the location instead of the temp variables like for the other elements. The purpose of the 'addedScore' variable is to identify this case. The temporarily variables containing the data to added to the next index need to be calculated and stored first. This is because once the current score data is written to this index, the original data is overwritten and lost. Temporary variables are created for the score, name and the date. In the 'else' case, the data in the next index is going to be written with the temporary variable data. However the data at this index need to also be stored. If this data is saved to the temporary variables, then the original data in the temporary variables is lost without it being added to the dictionary. Alternatively, if the temporary value data is added to the dictionary, then the data which will need to be stored in the temporary variables will be overwritten.

I solved this issue by adding a second set of temporary variables. After the data of the first set of the temporary variables have been added to the dictionary, the values of the second set of temporary variables are copied to the first set. I used some print statements to print the 'highScoreDict' dictionary to the console so that the changing values could be analysed. Before the game can be run however, the method of calling this function needs to be modified.

Currently, the 'updateHighscore' function is only called if the 'self.score' value is greater than the 'self.highScore' value, i.e. if the score achieved in the level is greater than the highscore. This needs to be changed so that this function is always called as the score achieved could be in the top ten scores. To do this, I removed the 'if' statement determining whether this function should be called. Variables for the name and the date were created for testing. After this, I tested out the program by running the game and completing level 1 with a score greater than 10. The score achieved on the level was 52.

```
self.scoreName = "test"
self.scoreDate = "1/1/1"
```

52

```
Before Update:  {0: [100, 'name', 'time'], 1: [90, '', ''], 2: [80, '', ''], 3: [70, '', ''],
4: [60, '', ''], 5: [50, '', ''], 6: [40, '', ''], 7: [30, '', ''], 8: [20, '', ''], 9: [10, '', '']}
```

```
{0: [100, 'name', 'time'], 1: [90, '', ''], 2: [80, '', ''], 3: [70, '', ''], 4: [60, '', ''],
5: [52, 'test', '1/1/1'], 6: [40, '', ''], 7: [30, '', ''], 8: [20, '', ''], 9: [10, '', '']}
```

```
{0: [100, 'name', 'time'], 1: [90, '', ''], 2: [80, '', ''], 3: [70, '', ''], 4: [60, '', ''],
5: [52, 'test', '1/1/1'], 6: [50, '', ''], 7: [30, '', ''], 8: [20, '', ''], 9: [10, '', '']}
```

```
{0: [100, 'name', 'time'], 1: [90, '', ''], 2: [80, '', ''], 3: [70, '', ''], 4: [60, '', ''],
5: [52, 'test', '1/1/1'], 6: [50, '', ''], 7: [40, '', ''], 8: [20, '', ''], 9: [10, '', '']}
```

```
{0: [100, 'name', 'time'], 1: [90, '', ''], 2: [80, '', ''], 3: [70, '', ''], 4: [60, '', ''],
5: [52, 'test', '1/1/1'], 6: [50, '', ''], 7: [40, '', ''], 8: [30, '', ''], 9: [10, '', '']}
```

In the

```
{0: [100, 'name', 'time'], 1: [90, '', ''], 2: [80, '', ''], 3: [70, '', ''], 4: [60, '', ''],
5: [52, 'test', '1/1/1'], 6: [50, '', ''], 7: [40, '', ''], 8: [30, '', ''], 9: [20, '', '']}
```

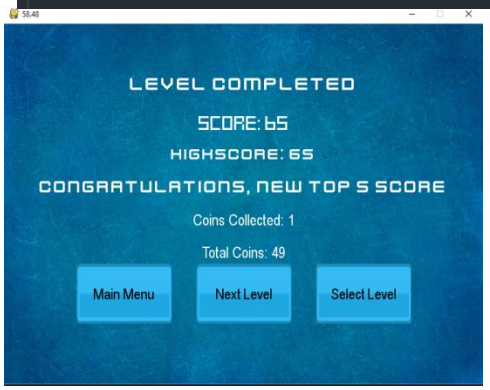
first iteration, the score is added to the correct position in the dictionary. In the second iteration, the value that the score overwrote is written to the next index. This procedure repeats until the end of the dictionary is reached. This is exactly the behaviour that was expected meaning that the program is working as expected.

```
highscoreWriteFile = open(self.levelFileName,"wb")
pickle.dump(self.highscoreDict, highscoreWriteFile)
highscoreWriteFile.close()
```

The 'levelFileName' variable is used to open the 'pickle' dictionary file for the current level. The updated 'highscoreDict' dictionary is then 'dumped' to this file to save the data.

As a result of moving the calling of the 'updateHighscore' function, the text displaying a congratulations message for beating the highscore also needs to be moved.

```
if index == 0:
    self.game.drawText("Congratulations, new High Score", 28, WHITE, WIDTH/2, HEIGHT*5/11, fontName=self.game.interfaceFont)
else:
    self.game.drawText("Congratulations, new top {} Score".format(index+1), 28, WHITE, WIDTH/2, HEIGHT*5/11, fontName=self.game.interfaceFont)
```



I moved this statement to inside the 'if index != 11' block of code. If the index if equal to zero, the congratulations on achieving the highscore message is displayed otherwise, congratulations on achieving top x, with x being the index value plus one, is displayed. I ran the program to check if this was working and it was. I noticed that the highscore was not the correct value. This statement had not been updated to use the new dictionary so I changed the value which replaces the '{}' to be 'self.highscoreDict[0][0]'.

```
self.game.drawText("Highscore: {}".format(str(self.highscoreDict[0][0])), \
        25, WHITE, WIDTH/2, HEIGHT*4/11, fontName=self.game.interfaceFont)
```

Error Encountered - When the check was being made to see if the current score was in the top ten scores an error causing a "KeyError: 0" was persistently arising. At that time, the keys of the dictionary were numbered 1-10.

```
    if score > self.practiceDict[9-i][0]:
KeyError: 0
```

Solution - The problem was with the numbers used for the keys in the dictionary. As the key were numbered 1-10, when the key with value '9' was used, 9-9 = 0 but there was no key with value 0. This is what was meant by KeyError: 0

Error Encountered - The shifting of the values inside the dictionary was not working as expected. Instead of writing the temporary variables to the next index, the remainder of the values remained constant.

```
{0: [100, 'name', 'time'], 1: [90, '', ''], 2: [80, '', ''], 3: [70, '', ''], 4: [60, '', ''], 5: [50, '', ''],
 [60, '', ''], 5: [50, '', ''], 6: [45, 'test', '1/1/1'], 7: [30, '', ''], 8: [20, '', ''], 9: [10, '', '']}
```

As shown, after 45 was added to the dictionary, 30 should have been replaced with 40, 30 with 20, and 20 with 10 however this did not happen.

Solution - This a logical error caused by the implementation of the temporary variables. I was not using a second set of temporary variables and the order of the statements meant that the data to be transferred was deleted before it could be stored temporarily as explained previously. I fixed this problem by changing the code to the form it was previously shown as.

```
else:
    self.highscoreDict[index+i][0] = tempScore
    self.highscoreDict[index+i][1] = tempName
    self.highscoreDict[index+i][2] = tempDate
    tempScore= self.highscoreDict[index+i][0]
    tempName= self.highscoreDict[index+i][1]
    tempDate= self.highscoreDict[index+i][2]
```

Getting the player's name

The player will need to provide their name so that the score that they achieved on the level is stored with their name. To do this an interactive input box needs to be added to the game. The 'input' function could have been used to get the user input however this would have been inconvenient as the player would have to interact with the console. This input box will need to be displayed on the level complete screen and if no name is provided "N/A".

Creating the Input Box class

```
class InputBox(pg.sprite.Sprite):
    def __init__(self, game, x, y, width, height, text='', textSize=32, fontName=None):
        self.groups = game.allSprites, game.userInputBox
        pg.sprite.Sprite.__init__(self, self.groups)
        self.game = game
        self.rect = pg.Rect(x, y, width, height)
        self.colour = RED
        self.text = text
        if fontName == None:
            self.font = pg.font.SysFont(None, textSize)
        else:
            self.font = pg.font.Font(fontName, textSize)
        self.textSurf = self.font.render(text, True, self.colour)
        self.boxActive = False
```

To implement this, I first created an 'inputBox' class in the 'sprites' class. Inside this class, the input box objects are first added to a new sprite group called 'userInputBox'. This group was created in the 'new' function of the main file.

```
self.userInputBox = pg.sprite.Group()
```

The object is also added to the 'allSprites' group. A rectangle is created using the 'pg.Rect' function with the dimensions provided in the parameters. A 'colour' class variable is created and assigned the colour Red. Additionally, the optional 'text' parameter, which by default is set to an empty string, is stored in the class variable 'text'. The font is then initialised using the 'fontName' parameter. After this, the text surface is using the initialised font and the 'textSize' parameter. A variable called 'boxActive' is then created and assigned the value False. This variable will be used to determine when the input box needs to accept keyboard input.

```
def inputKeys(self, event):
    if event.type == pg.MOUSEBUTTONDOWN:
        if self.rect.collidepoint(event.pos):
            self.boxActive = not self.boxActive
        else:
            self.boxActive = False

        if self.boxActive:
            self.colour = GREEN
        else:
            self.colour = RED
```

A function called 'inputKeys' is next created to handle the keyboard inputs to the input box. This function takes the pygame events as a parameter. If this event is of type 'MOUSEBUTTONDOWN', i.e. the mouse button is being clicked, then the 'collidepoint' function is used to check if the position of the mouse is within the boundaries of the rectangle. This function takes the location of the mouse as a parameter. If the mouse is inside the boundaries, then the user wishes to write to the input box therefore, the 'boxActive' variable's state is reversed. This allows the player to click again in the box to deactivate it. If a mouse click in another location is detected, then the 'boxActive' variable is made False to stop it detecting character input. The colour of the input box needs to be changed depending on the state of the 'boxActive' variable. This was achieved using an 'if' statement to check if the 'boxActive' variable is True and if so, change the 'colour' variable to green. Otherwise, it is changed to Red.

```
if event.type == pg.KEYDOWN:
    if self.boxActive:
        if event.key == pg.K_RETURN:
            print("This is the text: {}".format(self.text))
            if self.game.sceneMan.currentScene == 'levelComplete':
                self.game.sceneMan.updateHighscore(self.game.sceneMan.nextLevelTagNumber-1, self.game.score)
            self.text = ''
        elif event.key == pg.K_BACKSPACE:
            self.text = self.text[:-1]
        else:
            self.text += event.unicode
        self.game.screen.fill(WHITE, (WIDTH*5/8, HEIGHT*7/11, self.rect.width+5, 32))
        self.textSurf = self.font.render(self.text, True, self.colour)
```

Inside the same function a 'KEYDOWN' event is checked. If there is such event, the state of the 'boxActive' variable is checked and the remainder of the code is only executed if this variable is True. The return key is used to submit the name. When this is pressed, a message is printed to the console containing the 'self.text' variable to check that the function has correctly worked. The current scene is checked and if this is the level complete screen, the 'updateHighscore' function is called using the 'nextLevelTagNumber' variable from the 'sceneManager' class and the score from the 'game' class. This variable will use the 'self.text' variable so after it has executed, the 'text' variable is reset to an empty string.

If the backspace key is pressed, the previously entered character is deleted from the string. This is achieved using string slicing. The 'self.text' variable is set to 'self.text[:-1]' which resets the variable to store all the characters in the original variable apart from the final one. For any other keyboard input, the value of the key is obtained using the statement 'event.unicode' and then added to the 'self.text' variable.

After this, the text now needs to be displayed to the screen. To do this a new text surface is created using the same method. The problem is that the new text will be blitted on top of the old text without it being cleared, causing the text to become unreadable. This can be fixed by blitting all of the objects to the screen again however the 'levelComplete' function is only called once. Therefore, I fixed this problem by using the 'fill' function to only colour the screen at a particular location. The rectangle on the screen which is filled has the width 'self.rect.width+5', which is the length of the input text box plus five pixels, and a height of 32 pixels. The top-left corner of this rectangle has coordinate 'WIDTH*5/8, HEIGHT*7/11'.

```
def update(self):
    newWidth = max(150, self.textSurf.get_width()+10)
    self.rect.width = newWidth
```

The 'update' function of the input box is created next. This function is responsible for increasing the width of the text inside the box increases beyond the width of the text box. To do this, the width of the text surface is obtained using the 'get_width()' function on the text surface. The 'max' function is then used to determine whether 150, which is the original width of the box, or the width of the text is larger. This width of the rectangle is then updated to this new calculated width.

Finally the 'draw' function for input box object is created. In this function, the text surface is blitted to the provided surface parameter at 5 pixels to the right and below the top-left corner. The rectangle itself is then drawn to the surface using the 'colour' variable, the rectangle's dimensions and, an outline of 2 pixels along the border. Now the player can input their name in the level complete screen.

```
def draw(self, surf):
    surf.blit(self.textSurf, (self.rect.x+5, self.rect.y+5))
    pg.draw.rect(surf, self.colour, self.rect, 2)
```

Adding the Input Boxes to the game

An instance of the 'InputBox' class was created in the level complete function. The name of the player is only required if the score is going to be added to the top ten scores dictionary. This will only be required if the score achieved by the player is higher than the lowest score inside the dictionary. The method used to determine the index to place the score in the highscore dictionary was used to solve this problem. If after this loop the value of 'index' is 11, then the score is not in the top ten and the input box doesn't need to be created.

```
index = 11
for i in range(10):
    if self.game.score > self.highscoreDict[9-i][0]:
        index = 9-i
```

```
if index != 11:
    if index == 0:
        self.game.drawText("Congratulations, new High Score", 28, WHITE, WIDTH/2, HEIGHT*5/11, fontName=self.game.interfaceFont)
    else:
        self.game.drawText("Congratulations, new top {} Score".format(index+1), 28, WHITE, WIDTH/2, HEIGHT*5/11, fontName=self.game.interfaceFont)
    self.game.drawText("Enter name for Leader Boards:", 25, WHITE, WIDTH*3/4, HEIGHT*6/11, fontName=None)
    self.inputBox = InputBox(self.game, WIDTH*5/8, HEIGHT*7/11, 150, 32, text='')
else:
    self.inputBox = None
```

The code to perform this task was added to the block of code printing the congratulations message as this also checks the value for the 'index' variable. If 'index' is not equal to 11, then an 'InputBox' object is created with the necessary parameters and stored in the 'self.inputBox'. Otherwise, the 'self.inputBox' variable is set to 'None'.

```
if self.inputBox != None:
    self.scoreName = self.inputBox.text
else:
    self.scoreName = "N/A"
```

Next, in the 'updateHighscore' function, the 'text' variable of the 'inputBox' object is checked. If it is not equal to None, the 'scoreName' variable, which is the variable holding the name of the player' is made equal to this text. Otherwise, this variable is made equal to "N/A". This will happen if the player leaves the level complete screen without providing a name. To determine if this has happened, I added a variable called 'updatedHighscore' which becomes True if the 'updateHighscore' function has been called. Then, in the block of code checking which screen to load when a button is clicked, if current scene is the level complete screen and the 'updatedHighsore' variable is False (i.e. the player is leaving the screen without providing a name), the 'updateHighscore' function is manually called.

```
if self.currentScene == "levelComplete":
    if not self.updatedHighscore:
        self.updateHighscore(self.nextLevelTagNumber-1, self.game.score)
```

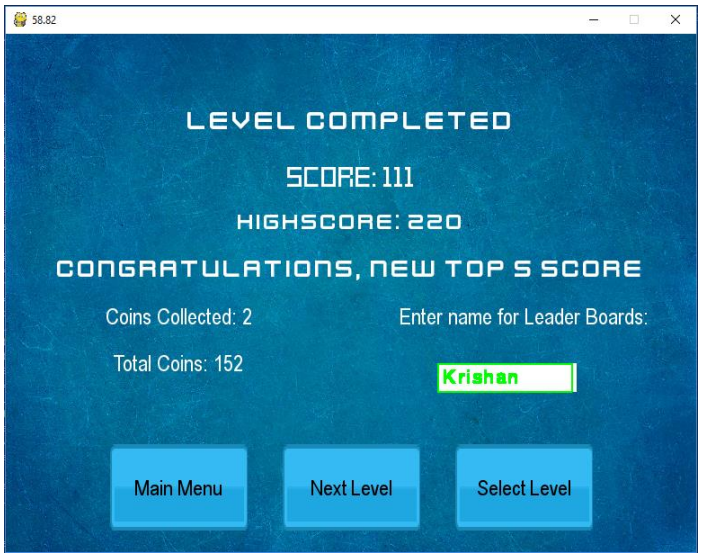
The 'inputKeys' function of the 'inputBox' objects is called in the 'events' function for each pygame event that occurs. Following this, their 'update' function and 'draw' function are called in the 'update' and 'draw' functions of the main class respectively.

```
for box in self.userInputBox:
    box.inputKeys(event)
```

```
self.userInputBox.update()
```

```
for box in self.userInputBox:
    box.draw(self.screen)
```

Testing out the Input Boxes



I ran the first level in the game and achieved a score greater than the lowest score in the top ten. As expected, the input box was correctly displayed in the level complete screen and when I typed into it, the character input successfully worked. Additionally, when the enter/return key was pressed, the text box cleared, which is a sign that the highscore has been updated. The statement confirming the text that has been saved by the input box is printed to the console and the result is correct.

```
This is the text: Krishan
```

Also, the scores dictionary was correctly updated using the provided name.

```
{0: [203, 'Krishan', '1/1/1'], 1: [195, '', '1/1/1'],
```

Issues with the Input Box

There were a few error and problems that needed to be solved to reach this solution as I will now explain.

Error Encountered - When text was entered into the box, the game crashed and there was an error stating that 'font' in the 'inputKeys' function is not defined.

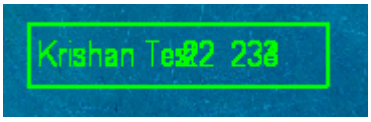
```
    self.textSurf = font.render(self.text, True, self.colour)
NameError: name 'font' is not defined
```

Solution - The 'font' variable in the 'InputBox' class was not a class variable therefore, when the variable, which was created locally in the 'init' function, was used in the 'inputKeys', the variable was not defined and so caused this error. I fixed this problem by making this into a class variable.

Error Encountered – When the name was typed in and the enter/return key was pressed, the text cleared however the highscore file was not successfully updated. The text typed into the input box was successfully recorded however this was not used in the scores file.

Solution - The 'updateHighscore' function was being called instantly after the level complete function was finish. This meant that sufficient time for the player to enter their name was not being given. The 'updateHighscore' function was not using the input box name and so was saving the scores with an empty string name. This was fixed by calling this function when the return key was pressed to submit the text provided in the input box. This meant that the function could not be run until the text was provided. The program then worked as expected and the name was correctly saved to the file.

Error Encountered - When the backspace key was used to delete the characters in the text the letters appeared to have not been removed. However when a new character was written in the same location, the new letter was blitted on top of the old letter, making them both unreadable.



Solution - The problem is with not updating the screen. When the 'InputBox' detects a KEYDOWN event, a new surface is created using the new text and blitted to the screen. The existing text on the screen needs to be removed so that the new text can be displayed on a clear background. If this is not done, as shown by the problem, the characters end up blitting on top of each other. This problem can be solved by clearing the screen every time new text needs to be drawn to the screen. To do this the 'fill' function was used, alongside the dimensions of the input box object, to only fill the area under the input box in white. Loading the whole screen again each time would cause unnecessary computation and so was avoided. The input box worked correctly after this. When the backspace key was pressed, the Last character was correctly removed from the screen.

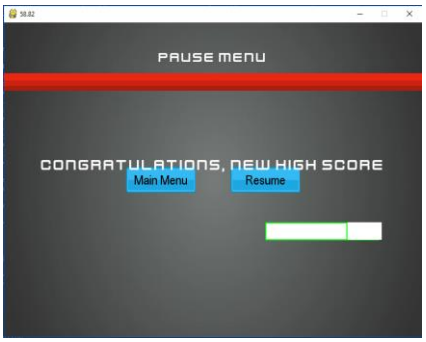
Error Encountered - Now the text was correctly being cleared however when the size of the input box is increased due to the length of the text, the text stops updating correctly.

Solution - This problem is caused by the restrictions of the previously defined 'fill' function. Only the area under the input box is being filled. This needs to be changed so that the width of the input box is filled in. To do this, I changed the width of the

```
self.game.screen.fill(WHITE, (WIDTH*5/8, HEIGHT*7/11, self.rect.width+5, 32))
```

fill rectangle to 'self.rect.width+5'. These additional 5 pixels are to keep filling the extension to the rectangle when its width is increased. This fix worked as expected. The only problem is that the white fill is not removed from the screen if it is not needed. This is not a big issue as the text will still be visible.

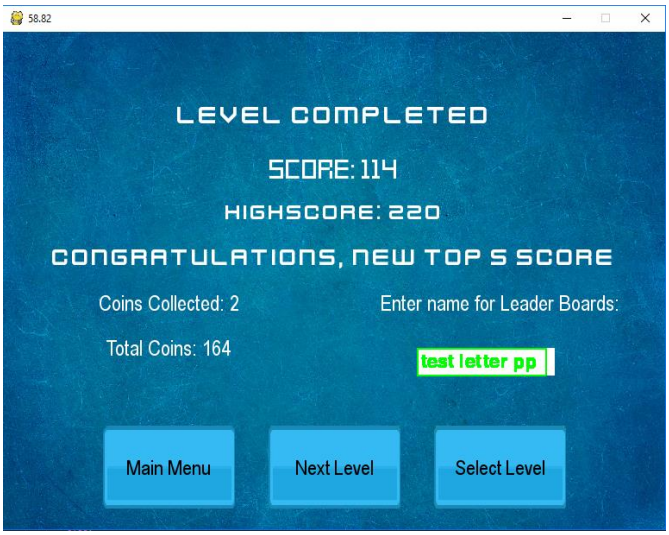
Error Encountered - When the 'p' button is pressed while typing into the input box the pause screen is loaded. When the 'p' button is pressed again to un-pause the screen, or the resume button is pressed, the screen doesn't clear and the level complete function doesn't re-load.



Solution - The simplest solution will be to avoid this situation in the first place. This can be achieved by only being able to pause whilst playing a level. I created a list in the 'settings' file to store all of the level names. This can now be used when checking I the current scene is a level. I changed the code in the 'sceneManager' class's 'update' function to work with this list. Inside the 'events' function, if the 'p' button press event is recorded then a check is made to see if the current scene is a level. If so, the remainder of the code to change the 'paused variable state is executed. If not, then the game is not paused. This means that the game can now only be paused while in a level, which is an appropriate condition.

```
LEVEL_SCREENS = ["level1",
                 "level2",
                 "level3",
                 "level4",
                 "level5",
                 "level6",
                 "level7",
                 "level8",
                 "minigame"]
```

```
if event.key == pg.K_p:
    if self.sceneMan.currentScene in LEVEL_SCREENS:
```



The 'p' button can now be typed into the input box without pausing the screen as the current scene will be 'levelComplete'.

Getting the Date

Information about the current date and time can be obtained using the 'datetime' module. The current date and time are retrieved using the code 'datetime.datetime.now()'. This gets the year, month and day alongside the current hour, minute, second and millisecond. The individual elements of this data can be obtained by calling methods on this data. The program shown on the left has the output shown on the right.

```
2019-03-01 09:20:23.558528
```

```
import datetime
currentDate = datetime.datetime.now()
print(currentDate)
print(currentDate.year)
print(currentDate.month)
print(currentDate.day)
print(currentDate.hour)
print(currentDate.minute)
print(currentDate.second)
```

```
2019-03-01 09:20:23.558528
2019
3
1
9
20
23
```

For my game, the date including the day, month and year will need to be extracted and stored alongside the top ten scores.

```
date = currentDate.strftime("%Y-%m-%d %H:%M:%S")
print(date)
```

```
2019-03-01 09:23:01
```

One method of organizing the data in a preferred format is to use the 'strftime' function. This takes in predetermined keys as inputs to create a particular representation of the date time information.

```
currentDate = datetime.datetime.now()
self.scoreDate = currentDate.strftime("%Y-%m-%d")
```

This code was added to the 'updateHighscore' function. The current date and time information are retrieved and stored in the 'currentData' variable. This variable is then used alongside the 'strftime' function to obtain only the year month and day information. The code for adding this date to the scores dictionary has already been implemented. To test if the date was correctly being stored, I played the level and viewed the scores dictionary after it has been updated. The date was successfully store alongside the score and the player name.

```
After Update:  {0: [112, 'Krishan', '2019-03-01'],
```

Adding information to the Leader Board screen

As explained and shown in the Design section when creating designs for the various game screens, the leader board screen will have buttons which when clicked will change the current information shown on the page to the data associated with the selected level. Therefore, button which don't change the scene when clicked need to be implemented.

Adding the level buttons

Before the buttons are added to the leader board screen the functionality to not launch a different screen needs to first be implemented. This was performed by checking if the current scene is the leader board screen first when a button is clicked. If this is not the case, then the remainder of the code executes as previously programmed. Otherwise, the 'else' statement is executed to make the class variable 'self.leaderboardLoadLevel' equal to the tag of the button. This variable will be used in the leader board function to identify which level's data to display. This variable was created and initialised in the 'init' function with the value 'level1Score'. Any buttons remaining on the screen are then removed. The leader board screen is then loaded again using the 'loadLevel' function.

```
if self.currentScene != "leaderboard":
```

```
else:
    self.leaderboardLoadLevel = button.tag
    for button in self.game.buttons:
        button.kill()
    self.loadLevel("leaderboard")
```

```
self.leaderboardLoadLevel = "level1Score"
```

As the leader board screen is loaded again, the background image needs to be blitted again. However, the code to perform this is not executed as a result of the previously defined condition when a button is clicked. This problem is solved by moving the code to load the screen's image into the leader board function.

```
def leaderboard(self):
    self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)
    self.game.drawText("Leader Board Tables", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)
```

The buttons are then created. They all have the same dimensions and x coordinate however their y coordinate vary by 1/12. The tags of the buttons were created to be different to the tags of the actual level buttons to help differentiate these if required later in the program.

```
self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)
self.level1Score = Button(self.game, "level1Score", WIDTH/5, HEIGHT*3/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 1")
self.level2Score = Button(self.game, "level2Score", WIDTH/5, HEIGHT*4/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 2")
self.level3Score = Button(self.game, "level3Score", WIDTH/5, HEIGHT*5/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 3")
self.level4Score = Button(self.game, "level4Score", WIDTH/5, HEIGHT*6/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 4")
```

```
fileNameList = ["l","e","v","e","l",".","p","i","c","k","l","e"]
levelNumber = self.leaderboardLoadLevel[:-5][-1]
fileNameList.insert(5, str(levelNumber))
self.leaderboardLevelFileName = "".join(fileNameList)
scoreFile = open(self.leaderboardLevelFileName, "rb")
self.leaderboardDict = pickle.load(scoreFile)
scoreFile.close()
print("leaderboardDict: ", self.leaderboardDict)
```

Following this, I analysed the 'self.leaderboardLoadLevel' variable. The level number needs to be extracted from this string so that it can be used to load the scores file. To do this, I used string slicing. The code '[:-5]' was used to remove the last 5 characters from the string. Now the 'Score' text has been removed from the string. The level number is now the last element in the string. Therefore, the '[-1]' index is used to look at this value and it is stored in the 'levelNumber' variable. A similar method to open the 'pickle' file containing the scores is used to load the required file.

Error Encountered - When the back button in the leader board screen was pressed there was an error showing that the program has attempted to load a 'pickle' file but no such file exists.

```
    scoreFile = open(self.leaderboardLevelFileName, "rb")
FileNotFoundError: [Errno 2] No such file or directory: 'level1.pickle'
```

Solution - As this is a button in the leader board function, the game assumes that it is a leader board button so executes saves its tag to the 'self.leaderboardLoadLevel' variable and re-loads the leader board screen. This function then uses the string slicing to get the level number (which in this case is the letter 'i') and loads the 'pickle' file, which doesn't exist. Therefore to avoid the back button being restricted, the following statement was used:

```
if self.currentScene != "leaderboard" or (self.currentScene == "leaderboard" and button.tag == "mainMenu"):
```

Now when the back button is pressed, the original code to load the correct screen is executed.

I ran the program, loaded the leader board screen and clicked on the level button, When analysing the data printed to the console, I found that the dictionary containing the level 1 data was correctly printed as

```
leaderboard Load level:  level1Score
leaderboardDict:  {0: [112, 'Krishan', '2019-03-01'], 1: [100, 'name', 'time'], 2: [90, '', ''],
```

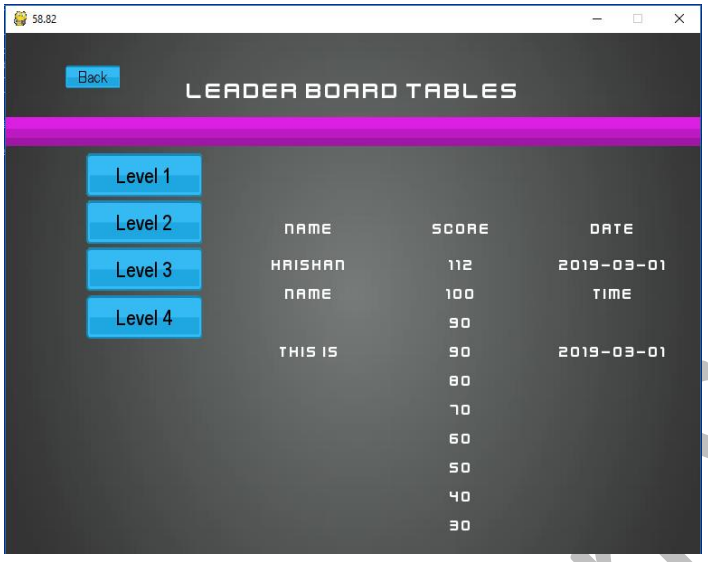
expected. This means that the remainder of the buttons can be created and the data can be displayed to the screen.

Displaying the score information

The information will be displayed in the form of three columns titled ‘Name’, ‘Score’ and ‘Date’. I used the ‘drawText’ function to draw these titles to the leader board screen.

```
self.game.drawText("Name", 22, WHITE, WIDTH*7/16, HEIGHT*3/8, fontName=self.game.interfaceFont2)
self.game.drawText("Score", 22, WHITE, WIDTH*21/32, HEIGHT*3/8, fontName=self.game.interfaceFont2)
self.game.drawText("Date", 22, WHITE, WIDTH*7/8, HEIGHT*3/8, fontName=self.game.interfaceFont2)

for i in range(10):
    self.game.drawText(str(self.leaderboardDict[i][0]), 18, WHITE, WIDTH*21/32, HEIGHT*(8+i)/18, fontName=self.game.interfaceFont2)
    self.game.drawText(str(self.leaderboardDict[i][1]), 18, WHITE, WIDTH*7/16, HEIGHT*(8+i)/18, fontName=self.game.interfaceFont2)
    self.game.drawText(str(self.leaderboardDict[i][2]), 18, WHITE, WIDTH*7/8, HEIGHT*(8+i)/18, fontName=self.game.interfaceFont2)
```

After this, I used a definite loop with 10 iterations to blit the score information to the screen. The recently loaded in and saved ‘leaderboardDict’ dictionary has all of the data about the level. The titles have a larger size than the data. The index of the iteration is used to change the y coordinates of the text. The x coordinate of the score was calculated by finding the midpoint of the x coordinates of the name and time.



The data is correctly loaded however there are lots of empty spaces which are where no name or data has been provided. To test if the buttons are correctly working, text data for other levels need to be created. To do this, more ‘pickle’ dictionary files need to be created to store their highscore. This will be performed in the ‘dictionary’ file.

Creating score files for more levels

The existing function to create score dictionaries could have been copied for the other levels but instead, I decided to modify the function to take in a level parameter. This parameter will then be used to create the name of the ‘pickle’ file. The dictionary that is ‘dumped’ to the file is not empty this time but filled with test data. Now it will be clear if the leader board screen is functioning as expected as this test data should be shown. I called this function providing ‘2’ as a parameter to create a scores ‘pickle’ file for level 2.

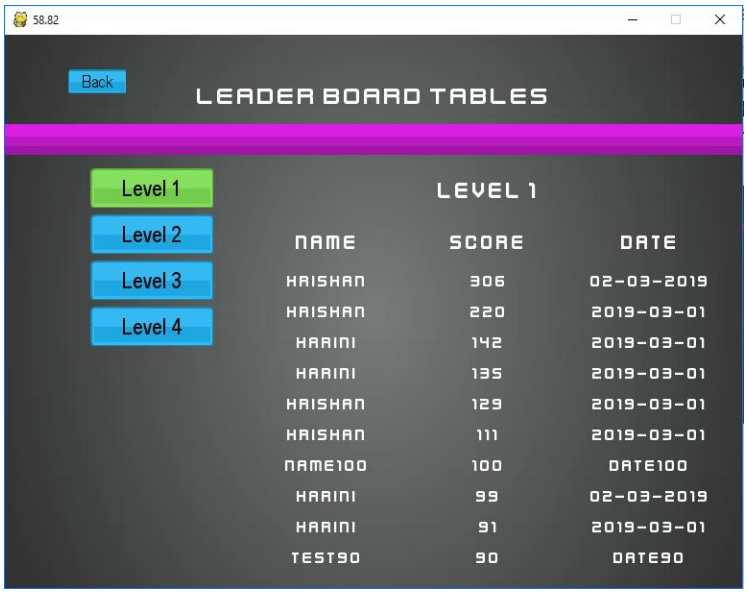
```
def createScoreDict(level):
    levelDict = {0:[100, "name100","date100"], 1:[90, "test90", "date90"], 2:[80, "name80", "date80"], 3:[70, "test70", "date70"], \
                 4:[60, "name60", "date60"], 5:[50, "test50", "date50"], 6:[40, "name40", "date40"],   7:[30, "test30", "date30"], \
                 8:[20, "name20", "date20"], 9:[10, "test10", "date10"]}
    fileNameList = ["l","e","v","e","l",".","p","i","c","k","l","e"]
    fileNameList.insert(5, str(level))
    pickleFile = "".join(fileNameList)
    exportDict = open(pickleFile, "wb")
    pickle.dump(levelDict, exportDict)
    exportDict.close()
```

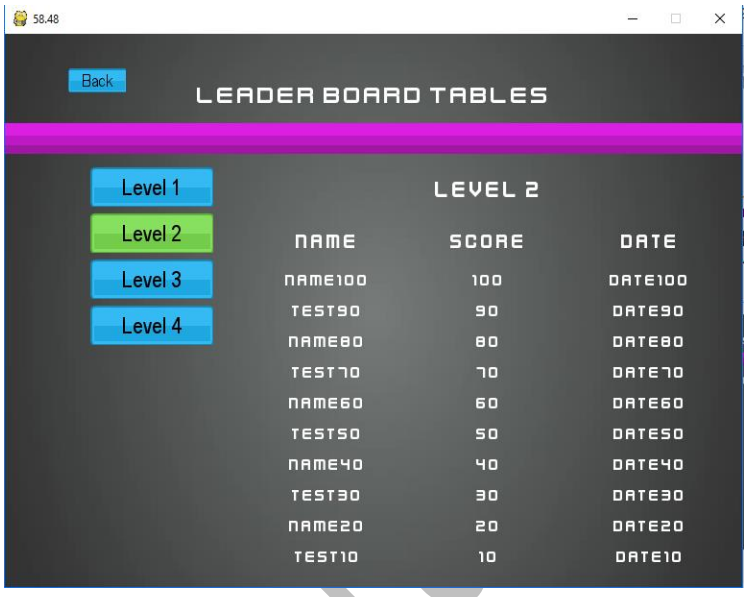
```
createScoreDict(2)
```

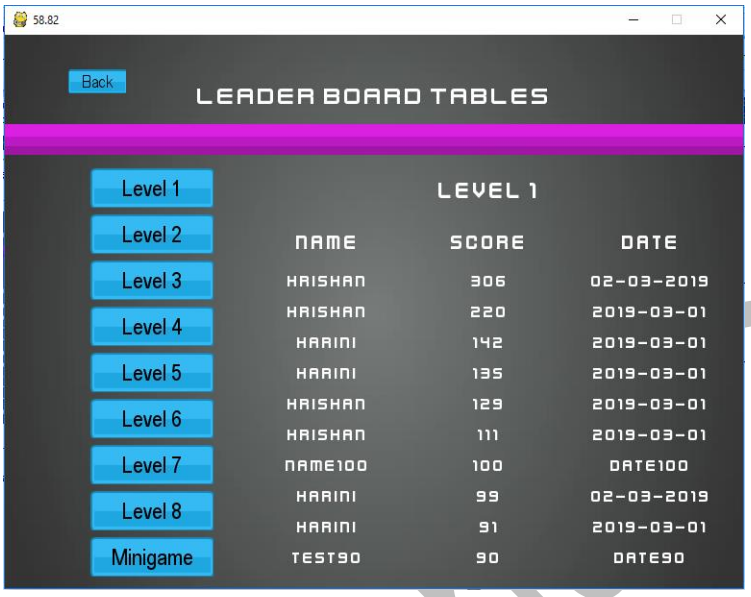
Testing the buttons

Beore testing the buttons, I decided to blit the level of the data being shown to make it clearer. This was achieved by simply calling the ‘drawText’ function to print the text ‘Level’ and the value of the variable ‘levelNumber’. This was a recently created variable to store the level number after the string slicing was performed to the ‘self.leaderboardLoadLevel’ variable. Now the buttons can be tested with better recognition.

```
self.game.drawText("Level {}".format(str(levelNumber)), 25, WHITE, WIDTH*21/32, HEIGHT*9/32, fontName=self.game.interfaceFont2)
```







The leader board data is correctly changing between the levels when the buttons are pressed. The test data is correctly showing up in the second level. The screen is working exactly as intended.

The remainder of the buttons for the remaining levels have now been added. The level number for the mini game is 0. I added an 'if' statement to check the value of the 'levelNumber' variable and if it is 0, instead of printing 'Level' and the level number, it will print mini game.

```
if levelNumber == 0:
    self.game.drawText("Mini Game"), 25, WHITE, WIDTH*21/32, \
    HEIGHT*9/32, fontName=self.game.interfaceFont2)
```

Error Encountered - When new data was printed to the screen, it overwrote the previous data without clearing the screen. This makes the text unreadable.

Solution - Adding the code to load the image for this screen into the function meant that when the function was called, the background image is blitted to the screen, overwriting any existing text. This solved the problem as when the level buttons were clicked, the background image re-loads.

Review

The leader board screen and all of the information required for it has successfully been programmed into the game. All of the buttons on this screen work as intended and the data is clear and easy to read. The next step from here is to add more questions to the question CSV file and to incorporate the functionality to choose which difficulty questions to ask. After this, all major features of the game will have been implemented therefore, graphics can be added.

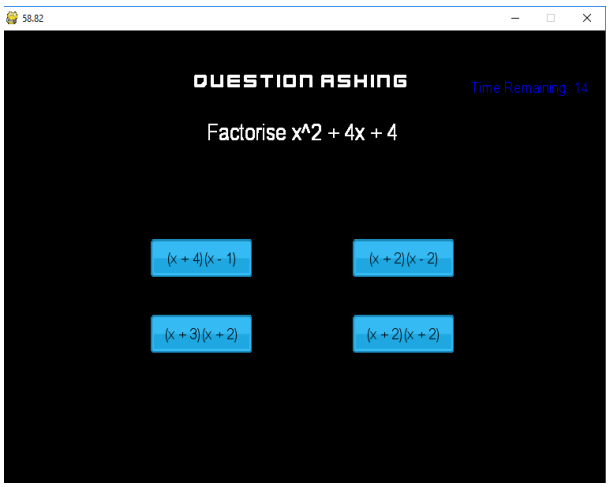
Changing question difficulty

One of the requirements of the program is that the player has the option to choose which difficulty question they will be asked in the level. They can pick from easy, medium and hard with the age ranges being years 7-9, years 10-11 and years 12 and onwards respectively. Before this feature can be added, more question with different difficulty levels needs to be added to the questions csv file.

Adding questions to the CSV file

questionID	questions	correctAnswer	wrongAnswer1	wrongAnswer2	wrongAnswer3	wrongAnswer4	difficulty	level	isMajor
1	2 + 2	4	3	2	6	5	easy	1	FALSE
2	5 + 7	12	13	25	10	15	easy	1	FALSE
3	15 + 17	32	30	42	23	34	easy	1	FALSE
4	25 ÷ 5	5	1	7	11	4	easy	1	FALSE
5	4 x 3	12	11	15	17	8	easy	1	FALSE
6	(2 + 4) x 3	18	19	16	24	21	easy	1	FALSE
7	98 - 7	91	89	87	93	95	easy	1	FALSE
8	15 + 17	32	30	42	23	34	easy	1	FALSE
9	49 ÷ 7	7	6	11	40	9	easy	1	FALSE
10	21 + 9	30	27	28	35	42	easy	1	FALSE
11	14 x 6	104	100	98	110	92	medium	1	FALSE
12	Find x when 40x = 2x + 19	2	1	0	3	4	medium	1	FALSE
13	Find x when 20x - 17x = 24	8	9	11	4	13	medium	1	FALSE
14	Find x when 14x/7 = 28-2x	7	10	9	12	5	medium	1	FALSE
15	Find x when 27x = 3 - (5x - 2x)	10	15	17	9	6	medium	1	FALSE
16	Factorise x^2 + 4x + 4	(x + 2)(x + 2)	(x + 3)(x + 2)	(x + 4)(x - 1)	(x + 2)(x - 2)	(x - 3)(x - 1)	medium	1	FALSE
17	Factorise x^2 + 7x - 8	(x + 8)(x - 1)	(x + 8)(x + 1)	(x + 4)(x - 2)	(x + 4)(x + 3)	(x - 2)(x - 6)	medium	1	FALSE
18	Factorise x^2 -5x + 6	(x - 3)(x - 2)	(x + 3)(x + 2)	(x + 6)(x - 1)	(x + 5)(x + 3)	(x + 6)(x + 5)	medium	1	FALSE
19	Factorise 2x^2 - x - 1	(2x + 1)(x - 1)	(2x + 3)(x + 2)	(x + 1)(x - 1)	(x + 1)(2x - 2)	(2x - 1)(x + 3)	medium	1	FALSE
20	Complete the Square: x^2 + 8x - 5	(x + 4)^2 - 21	(x + 4)^2 - 5	(x - 4)^2 + 21	(x + 8)^2 - 5	(x - 8)^2 + 3	medium	1	FALSE
21	Complete the Square: x^2 + 10x + 12	(x + 5)^2 - 13	(x + 5)^2 + 12	(x - 5)^2 + 13	(x + 10)^2 + 12	(x - 4)^2 + 7	medium	1	FALSE
22	Complete the Square: 2x^2 + 8x - 5	2(x + 2)^2- 13	(x + 4)^2 - 21	2(x + 8)^2 - 5	2(x + 2)^2 + 13	2(x + 4)^2 - 15	hard	1	FALSE
23	Complete the Square: 2x^2 + 12x - 7	2(x + 3)^2- 25	2(x + 3)^2 - 7	2(x + 6)^2 - 25	2(x + 12)^2 - 7	2(x - 6)^2 + 9	hard	1	FALSE
24	Complete the Square: 3x^2 + 24x + 5	3(x + 4)^2 - 43	3(x + 8)^2 - 40	2(x + 12)^2 - 7	3(x - 8)^2 - 17	3(x - 4)^2 - 11	hard	1	FALSE
25	Factorise 4x^2 + 17x + 4	(4x + 1)(x + 4)	(4x + 3)(x + 2)	(4x + 3)(x - 1)	(2x - 1)(2x + 4)	(2x - 3)(2x - 1)	hard	1	FALSE
26	Factorise 3x^2 + 5x + 4	(3x + 2)(x + 2)	(3x + 3)(x + 2)	(3x - 2)(x - 1)	(x + 3)(x - 5)	(3x - 4)(x + 5)	hard	1	FALSE
27	Factorise 6x^2 + 7x + 2	(3x + 2)(2x + 1)	(6x + 3)(x + 2)	(6x + 3)(x - 2)	(3x + 2)(2x - 1)	(3x - 4)(2x + 3)	hard	1	FALSE
28	Find dy/dx when y = 3x^2 +7x	6x+7	4x+9	5x^2+7x+9	3x	17	hard	1	FALSE
29	Find dy/dx when y = 7x^3 - 5x^2+17x -24	21x^2 - 10x + 17	21x^2 + 10x + 24	7x^3 - 5x + 17	3x^2 -10X - 7	5x - 24	hard	1	FALSE
30	Find dy/dx when y = sin(2x) + cos(x)	2cos(2x) - sin(x)	2sin(2x) - sin(x)	cos(2x) + sin(x)	cos(2x) - sin(x)	2sin(4x) + cos(x)	hard	1	FALSE
31	Find dy/dx when y = tan(3x) + 7x^2 + 84x	3sec(3x)^2 + 14x + 84	3sec(3x)^2 + 84x + 14	3cos(3x)^2 + 7x + 14	sin(3x)^2 + 7x + 42	tan(3x)^2 + 14x + 84	hard	1	FALSE

I have added more easy questions which are simple arithmetic calculation style question. The intermediate/medium questions include some arithmetic questions but are more focused on algebraic manipulation to find the solution to x. Factorisation and completing the square question have also been added in at this level. The hard question involves more difficult factorisation and completing the square question, alongside questions based on finding the derivative.



As the code for loading and selecting the questions were initially created with scalability in mind, there is no problem with these questions being loaded into the game. The only issue is the lack of the option to choose which difficulty question to get asked.

Adding buttons to the Settings screen

The player will be able to pick the question difficulty on the settings page. This will be implemented similar to how the buttons were added to the leader board screen. This implies that when a button is clicked, if the current scene is the settings screen, the action taken needs to be different.

```
self.easyButton = Button(self.game, "easy", WIDTH*7/10, HEIGHT*4/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Easy")
self.mediumButton = Button(self.game, "medium", WIDTH*7/10, HEIGHT*5/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Medium")
self.hardButton = Button(self.game, "hard", WIDTH*7/10, HEIGHT*6/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Hard")
```

The three buttons for the difficulties were created inside the 'settingsMenu' function with similar dimensions to the level buttons on the leader board screen.

```
self.game.drawText("Question Difficulty: {}".format(self.game.settingsQuestinDiff.capitalize()), 18, \
                   WHITE, WIDTH*11/16, HEIGHT/4, fontName=self.game.interfaceFont2)
```

The 'drawText' function was used to draw the current question difficulty to the screen above the buttons to make it easier to determine when the difficulty has been changed. The '.capitalize' method was called on the 'settingsQuestionDiff' variable to capitalise the first letter of the word. This improves the aesthetic of the work and makes the settings page appear more sophisticated. The 'settingsQuestionDiff' variable was created in the game file with the intention to store the chosen question difficulty. This variable was initialised with "easy" in the 'new' function. This variable will need to be changed when the buttons on the settings screen are pressed.

```
self.settingsQuestionDiff = "easy"
```

```
if self.currentScene != "settingsMenu" or (self.currentScene == "settingsMenu" and button.tag == "mainMenu") \
    or (self.currentScene == "settingsMenu" and button.tag == "startScreen"):
```

To do this, I first added this code to the 'sceneManager' class's 'update' function. The purpose of this is to execute a different block of code if the buttons on the settings screen for choosing the difficulties are pressed. The 'or' operator is used to exclude the back button and the start screen button on this screen from being interfered by making the usual code to execute when these are pressed.

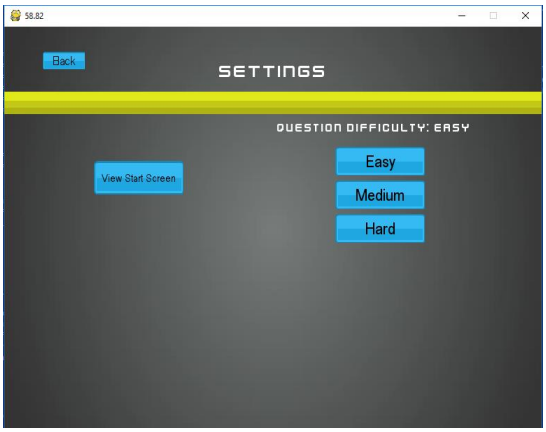
```
else:
    self.game.settingsQuestionDiff = button.tag
    for button in self.game.buttons:
        button.kill()
    self.loadLevel("settingsMenu")
```

The different block of code is in the 'else' statement for this 'if' statement. Here, the 'settingsQuestionDiff' variable created in the game class is changed to store the value of the tag of the clicked button. This will be either "easy", "medium", or "hard", which is the same as the text in the CSV file in the 'difficulty' column. The buttons are then removed from the screen and following this, the 'settingsMenu' function is called. This will re-draw the buttons with the new difficulty drawn to the screen using the 'drawText' function.

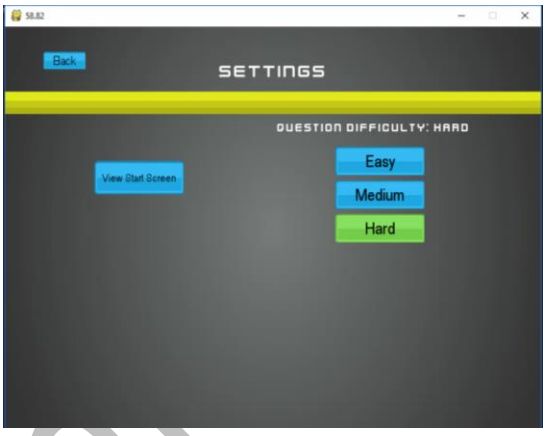
As a result of the 'settings' screen being loaded again, the image for the screen needs to be blitted to the screen. However, the code for doing this is in the block of code which has been avoided. Therefore, the statements to achieve this were moved to the start of the settings function to achieve the same effect.

```
def settingsMenu(self):
    self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)
```

Testing the buttons

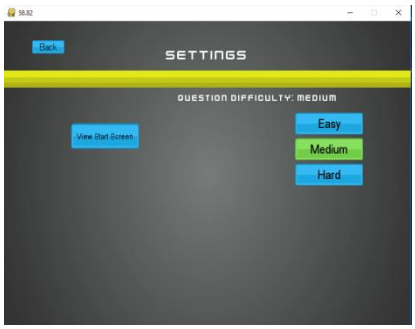


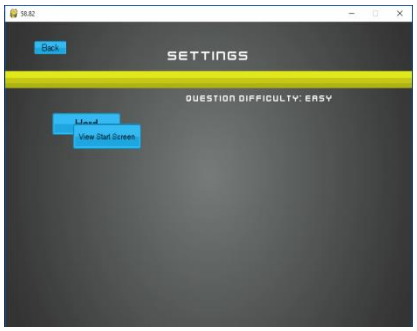
As the 'settingsQuestionDiff' variable was initialized with the value "easy", the player will by default be asked easy questions. This is also shown when the settings screen is launched as the question difficulty is shown as easy. The buttons loaded successfully and worked as expected. They highlighted and when a different difficulty button was pressed, the text above the buttons changed appropriately to reflect this change. When further functions are added to the settings screen, the buttons can be repositioned elsewhere but currently, their position makes them clear and organized.



Error Encountered - First the buttons were loaded in on top of each other. After this, the buttons were not in the expected location.

Solution - These problem are linked to the specified location of the buttons. Therefore, these issues were resolved by re-organizing the buttons to their correct and current positons.

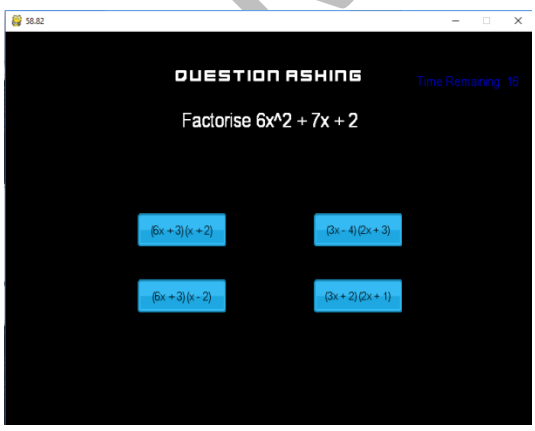




Asking question based on the difficulty

Currently the questions are being asked by loading the 'questionID' for each question into a list and randomly selecting a 'questionID' to ask. To implement the specific difficulty questions, the 'questionID' values in this list should only be those corresponding to the 'easy, 'medium' or 'hard' questions. To do this, after the 'settignsQuestionDiff' variable is set to the tag of the clicked button, the 'questionID' list in the main file is made empty. The question data array is then iterated analysing the 7th index of each element, which stores the difficulty information. If this data is equal to the difficulty chosen in the settings screen, then the 'questionID' value of that question is added to the 'questionID' array. Now when a question is chosen, the choice is only between the questions with the selected difficulty.

```
self.game.questionID = []
for i in range(len(self.game.questionData)):
    if self.game.questionData[i][7] == self.game.settingsQuestionDiff:
        self.game.questionID.append(self.game.questionData[i][0])
```



If the player doesn't choose a difficulty then the 'questionID' list will be empty. To avoid this from happening and causing an error, I added this loop to the 'new' function. This will use the "easy" difficulty, as this is the value of 'settingsQuesionDiff' by default, to add the appropriate 'questionID' values to the list. I tan the game, chose the hard difficulty and collided with a question object inside a level. A hard difficulty question was successfully asked. This also worked for the medium and easy difficulties.

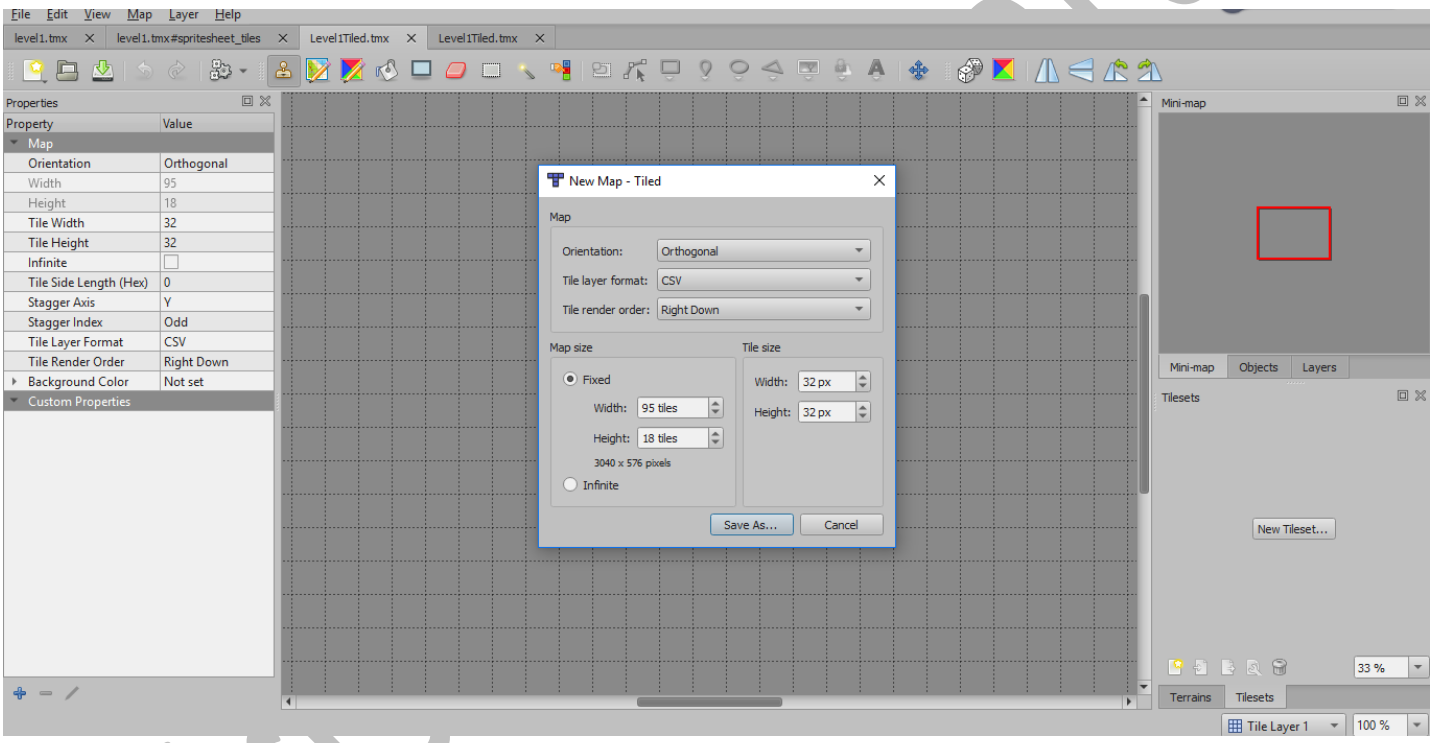
All fundamental features have now been added to the game. Graphics can now be added to improve the look and excitement of the game.

Adding graphics to the game

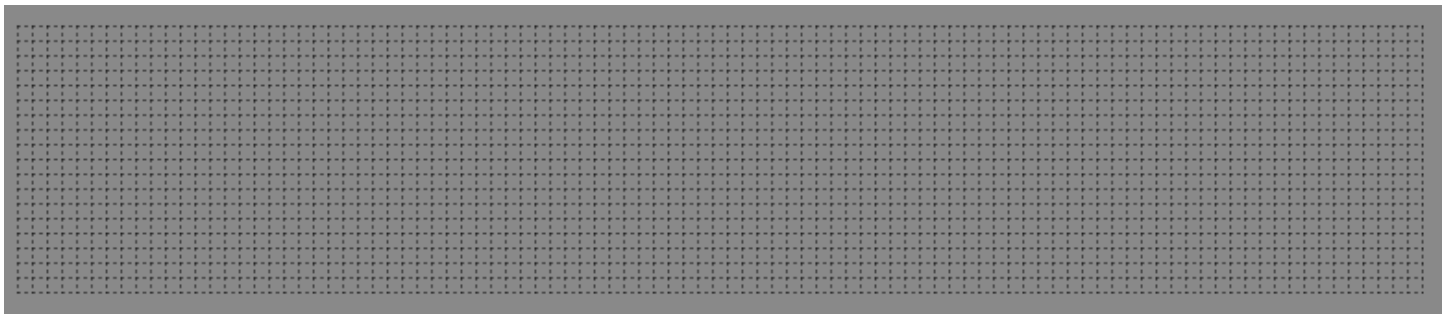
Up to this point I have been using surfaces for the different objects in pygame to get the game fully working before more complicated images are introduced. The plan is to include a scrolling background image and have the images on the platform on this as well. Then, I will use this image as a template in Photoshop to create the track file. After some re-search, I realised that for creating images, it will be preferable to use many spate tiles and place them together. This way, many different levels can be created from only using a small amount of tiles. To achieve this, I will be using the 'Tiled Map Editor' software. 'Tiled' is a 2D level editor with the primary feature to edit tile maps of various forms. This means that individual tile images can be loaded into the program and a level can be created using them. I downloaded the program at this link: https://thorbjorn.itch.io/tiled and used the documentation and this link: https://doc.mapeditor.org/en/stable/.

Creating the file in Tiled

After installing Tiled, I opened it and created a new map by pressing Ctrl + N.

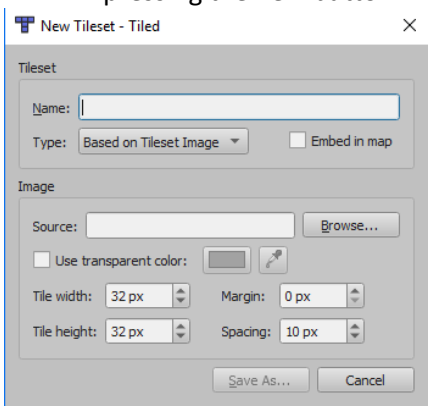


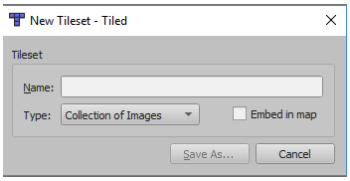
This opened a pop-up window in which there were the options to pick the orientation of the tiles, the layer format and the render order. After this, the size of the map is provided. Currently, the output of this program will only be an image therefore, the text file map will still be used to draw the walls and platforms. As a result of this, I used the dimensions of the text file as the size of the map. Additionally, I changed the tile size from the default 64 pixels by 64 pixels to the tile size used in the game which is 32 pixels by 32 pixels. This map was then saved to the map folder as 'Level1Tiled.tmx', and the new map was opened. It was an empty map on which tiles can now be drawn.



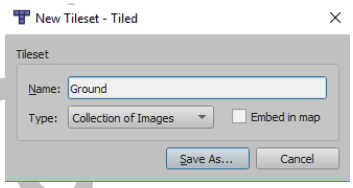
Loading the tile images

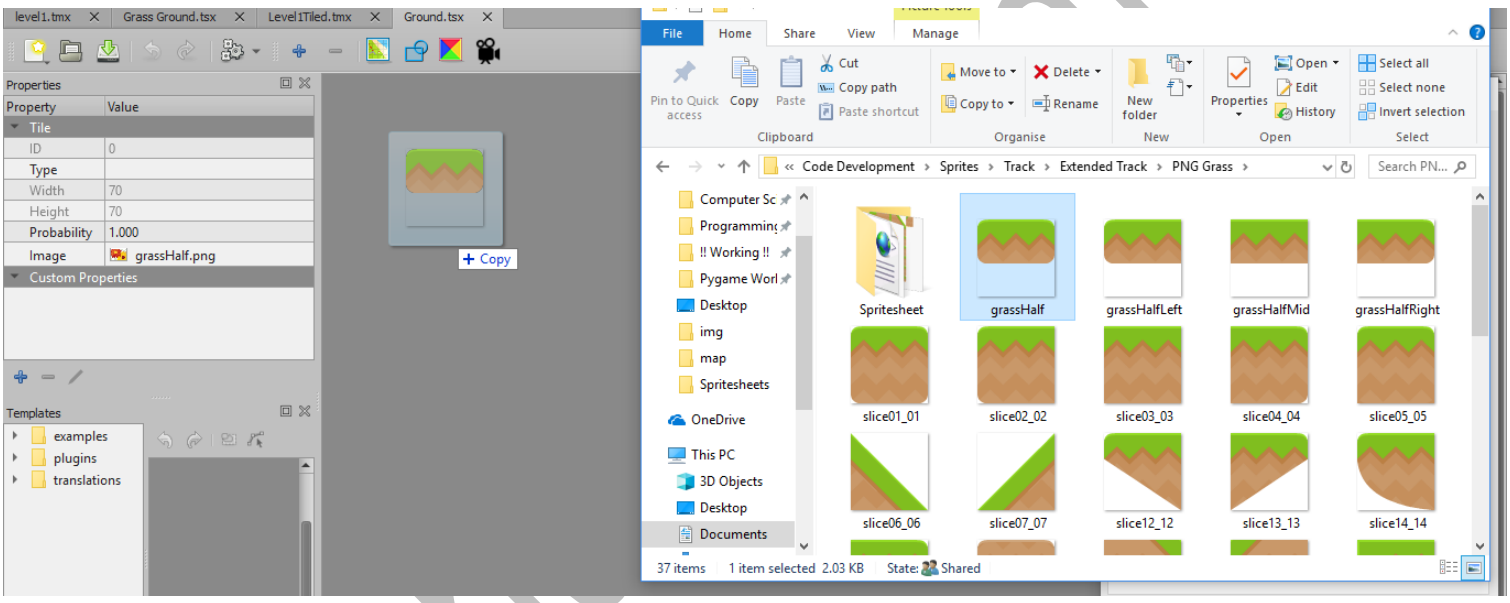
The tile sets are shown in the bottom-right side of the display in the program. A new tileset is loaded by pressing the new button in that region. This open the following pop-up window in which there are two options for the type of tileset: one based on a tileset image, or one created from a collection of images. The tile images that I have area not in the form of a tileset therefore, I chose the collection of images option. This removes the other options apart from the name of the file.



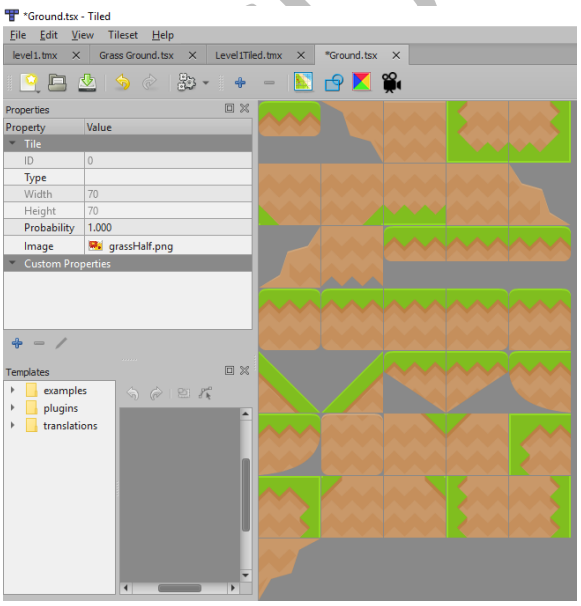


This tileset was going to be a collection of images to use for making the platforms therefore, I called it 'Ground' and saved the file in the Map folder.

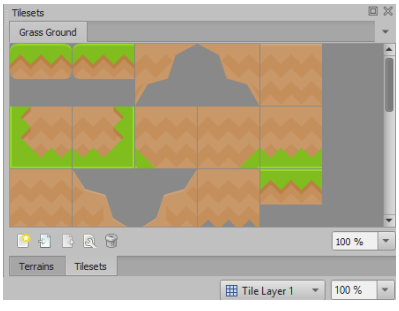




This opened a new file in which the chosen images can be dragged in. I obtained these images of the ground from this website: https://www.kenney.nl/ . In the license of use, the artist has specified that crediting for their intellectual property is optional and not mandatory. I loaded in all of the images and saved the file. It was clear that the program has kept all the images separate and is treating them as separate tiles.

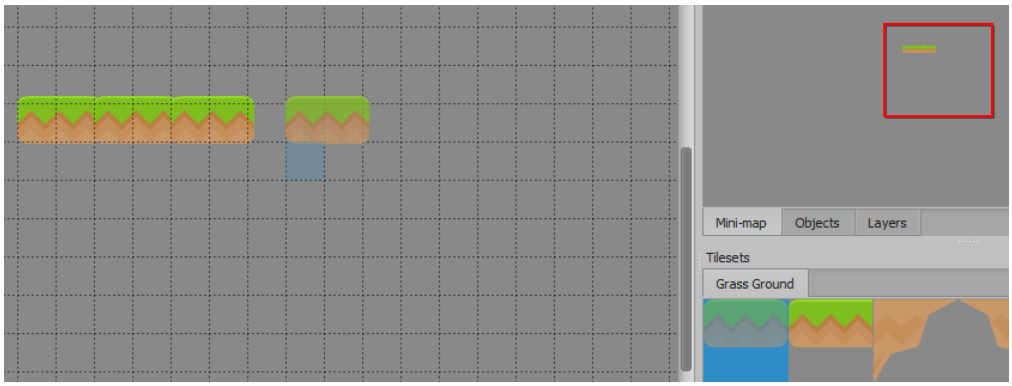


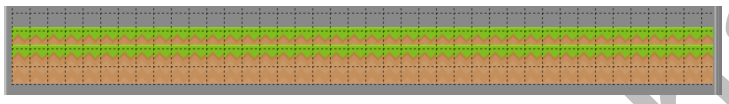
I switched to the main file ('Level1Tiled.tmx') and found that all the images of the created tileset have been correctly loaded into the program as they appeared in the bottom-right region of the screen.

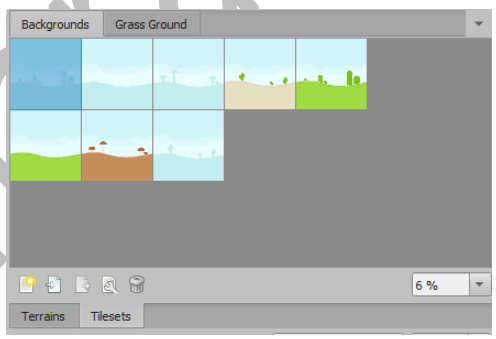


Making the Level

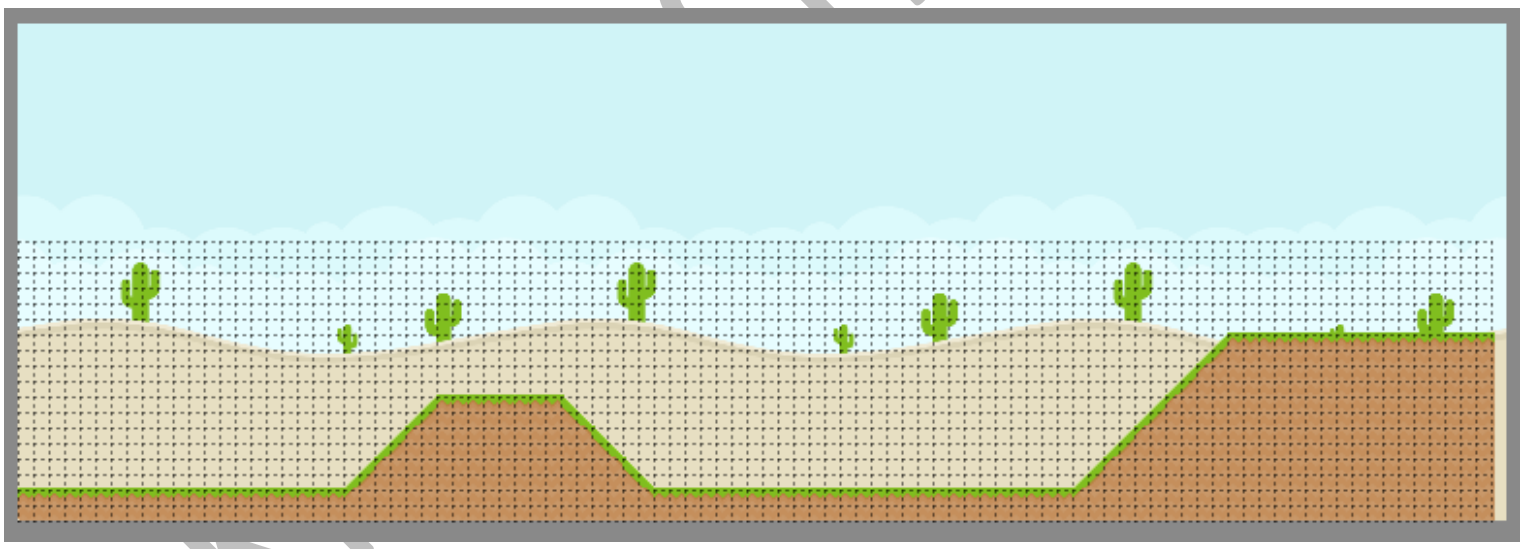
These images can now be used to draw the level. This is done using the stamp brush. This is in the top toolbar and can also be triggered by the hotkey 'B'. Using this tool, I selected a platform image and begin stamping that image onto the map canvas. It was possible to hold down the mouse button to continuously stamp images to the canvas as the mouse is moved. This made it easier and quicker to create the base platform image.



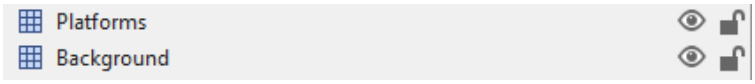




A feature of this 'Tiled Map Editor' program is the different layers that can be used when drawing an image. This allows images to be drawn in a specific order, which is required when adding a background image. I created a new tileset for the background images (which I also obtained from the same website), and loaded in an image into the level which I has drawn.



The background images were larger in height than required but maybe when the image is loaded into pygame, only the parts on the canvas will be displayed. This is also the case for the width as the image is one tile too long of the map. I used 3 of the same image to make the background. Fortunately, they have been designed to fit together and so the transmission between two images is seamless.



To achieve this effect, I first created a new layer, which is available in the top-right region of the program, and called it background. I renamed the current layer to platforms and dragged the backgrounds layer to be below the platforms layer. This means that any platform tiles will be drawn on top of any background tiles if they both exist on the same tile. The layers can be locked and made visible/invisible if needed.

Loading the level into the game

To load this style of map into the game, I first created a new class called 'TiledMap' in the 'sceneManager' class. There is a python library that has been created for the 'Tiled' program called 'pytmx'. I imported this library at the start of the 'sceneManager' file.

```
import pytmx
```

```
class TiledMap():
    def __init__(self, filename, trackfile):
        self.tiledMapFile = pytmx.load_pygame(filename, pixelaplha=True)
        self.width = self.tiledMapFile.width * self.tiledMapFile.tilewidth
        self.height = self.tiledMapFile.height * self.tiledMapFile.tileheight
```

There are two parameters to this class, 'filename' and 'trackfile'. The 'filename' stores the file path to the tiled map file. The 'trackfile' stores the file path to the bitmap image storing the track image. Inside the 'init' function, the 'load_pygame' function from the 'pytmx' library is called to load the tiled map image into the game. This loaded file is stored in a class variable called 'self.tiledMapFile'. The width and height of the map is obtained and stored in appropriate class variables. This is achieved by getting the width of the tile map, and then multiplying by the width of the tile.

The code to load the track image is added to this function. This is the same code that was previously in the 'Map' class and was explained in the 'Adding 4 bit image processing' section.

```
self.trackData = []
self.trackImage = Image.open(trackfile)
self.trackWidth = self.trackImage.size[0]
self.trackHeight = self.trackImage.size[1]
self.trackCount = 0

for col in range(self.trackHeight):
    for row in range(self.trackWidth):
        pixelData = self.trackImage.getpixel((row, col))
        if pixelData == 1:
            self.trackData.append((row, col))
            self.trackCount += 1
```

```
def render(self, surface):
    tileImage = self.tiledMapFile.get_tile_image_by_gid
    for layer in self.tiledMapFile.visible_layers:
        if isinstance(layer, pytmx.TiledTileLayer):
            for x, y, gid, in layer:
                tile = tileImage(gid)
                if tile:
                    surface.blit(tile, (x * self.tiledMapFile.tilewidth, y * self.tiledMapFile.tileheight - TILESIZE))
```

Next, a function called 'render' was created in which the image created in the tiled program is loaded. The '.get_tile_image_by_gid' function is used to get the tile image in the tiled program at a specific tile. As this function is long, it is saved in a shorter 'tileImage' variable. The visible layer in the tiled file is then iterated for every layer. If this layer is an instance of the 'tiled' layer, as there are other layers such as the image layer, then the tile layer is analysed. The x and y coordinates and the 'gid' (which is the ID of the tag) values in this layer are then used to iterate over the layer. The previously shortened and saved function is used to get the image with a corresponding id and save it to a variable called 'tile'. If this function is successfully (i.e the value of the 'tile' variable is not False), then the tile is blitted to the screen with the x and y coordinates being multiplied by the tile width and height. There was an error with the image being 1 tile too low therefore, I used the '- TILESIZE' statement to shift the map image up one tile in the y axis.

Finally, a function called 'makeMap' is created to create to make the map image. In this function, a temporary surface with the width and height of the map image is created. The 'render' function is then called onto this temporary surface and then this surface is returned.

```
def makeMap(self):
    tempSurface = pg.Surface((self.width, self.height))
    self.render(tempSurface)
    return tempSurface
```

```
self.tiledMap = None
self.tiledMapImg = None
self.tiledMapRect = None
```

Inside the 'new' function three new variables are created for the storing the tiled map image and the data associated with it. These variables are initiated with the 'None' variable and will be changed when a level is loaded.

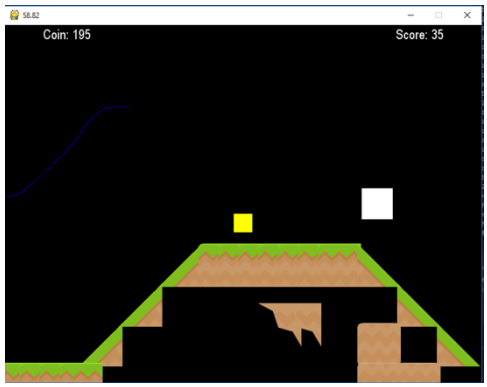
```
self.game.tiledMap = TiledMap(path.join(self.game.mapFolder,'Level1.tmx'), path.join(self.game.mapFolder,'Level1TrackBMP.bmp'))
self.game.tiledMapImg  = self.game.tiledMap.makeMap()
self.game.tiledMapRect = self.game.tiledMapImg.get_rect()
self.game.camera = Camera(self.game.tiledMap.width, self.game.tiledMap.height)
```

Next, in the 'level'1' function, an instance of the 'TiledMap' object is created using the 'level1.tmx' file in the map folder, and the bitmap image in the same folder. The map image is created by calling the 'makeMap' function on the 'TiledMap' object and saved to the 'tiledMapImg' variable. Following this, the 'rect 'object of the map is obtained and stored in the 'tiledMapRect' variable. The dimensions of the tiled map are then used when creating the 'Camera' object.

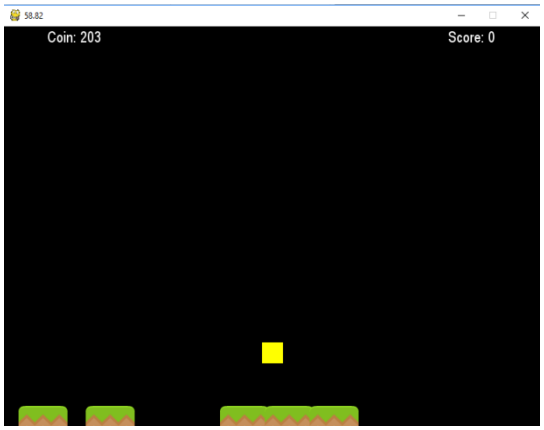
```
if self.sceneMan.currentScene not in MENU_SCREENS:
    self.screen.blit(self.tiledMapImg, self.camera.applyOffsetRect(self.tiledMapRect))
    for sprite in self.allSprites:
        self.screen.blit(sprite.image, self.camera.applyOffset(sprite))
    self.drawText("Score: " + str(self.score), 22, HUD_COLOUR, WIDTH-100, 15)
    self.drawText("Coin: " + str(self.coinAmount), 22, HUD_COLOUR, 100, 15)
```

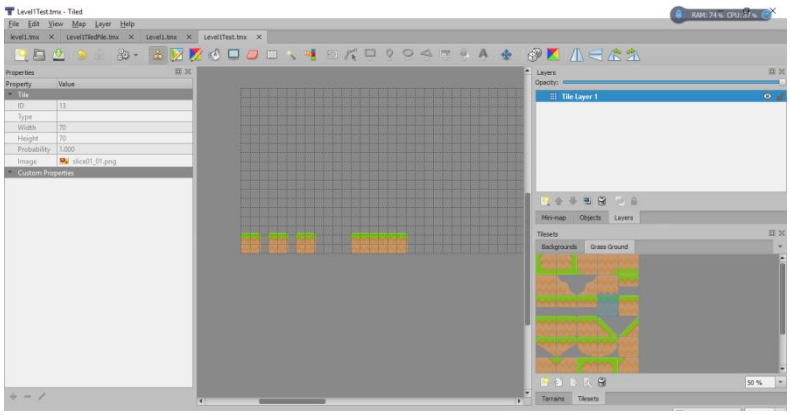
Currently, the screen is being filled black in every frame. This is removed and a statement to blit the tile image to the screen is added to the 'draw' function. The 'applyOffsectRect' function from the 'Camera' class is called to apply the necessary offset to the map image before it is drawn.

Error Encountered - The background image was not being loaded in as expected. Firstly, the background image was not being loaded, and some tile image were not being loaded. From analysing the output of the program, it was clear that the tile image filled in the empty space below the platforms was not being loaded. In the first image, the platforms and walls data from the text file map were being used. I fixed this problem by stopping an instance of the 'Map' object from being created in the level functions.



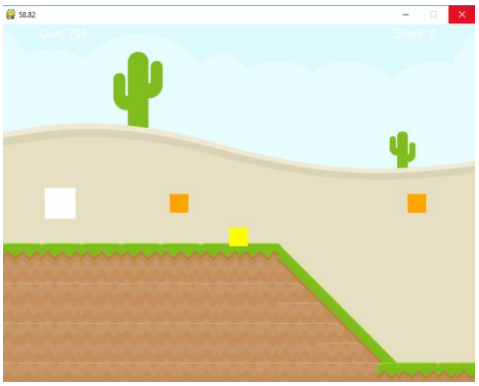
Solution - I tried using a different tile image and it worked and loaded in. The problem must be with the specific tile image. Therefore, I fixed this problem by using correcting working and loading in tile images to make the 'Tiled' map

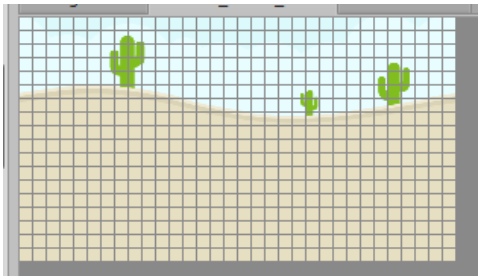




Error Encountered - The background image was still not being correctly loaded. As suspected when creating the level, the increased size of the background image has caused there to be an issue with loading it in. The rest of the game was still functioning as normal as when an question object was collided with, a question was asked as expected.

Solution - I created a cropped version of the background image so that there was Not an y excess image however this did not solve the problem. That is when I realised that every tile needs to be given an image and as the background was one large image, this was not being achieved. I solved this problem by making a tileset using the background image with tile size 32 and spacing between the tiles set to 0 pixels. With the background layer selected, I selected all of the tiles on this in this tileset and added it to the map image.

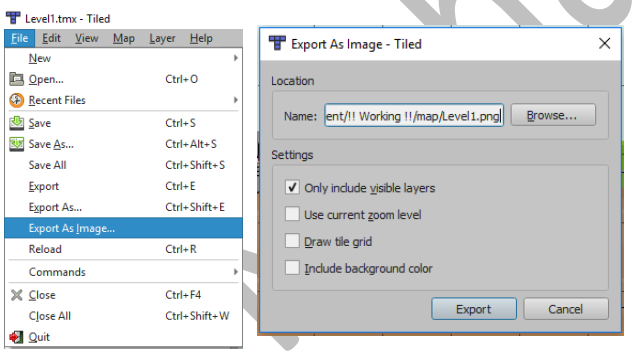




The background has image has successfully been loaded in after that fix.

Making the track file

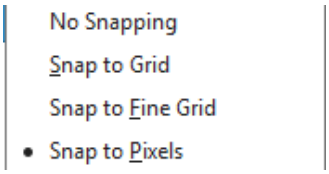
The level image created in the 'Tiled Map Editor' program is now being correctly loaded into the game. Following this stage, the objects now need to be added to the level. This means that the track needs to be created. To do this I exported the map as an image file. This option was available in the file tab and I made sure to deselect the 'Use Current Zoom Level' option so that the created image is the full size.



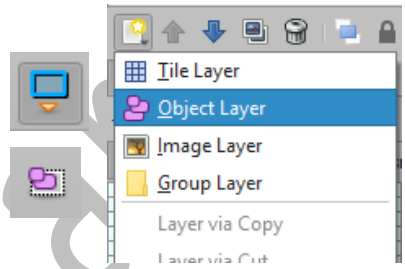
Next I opened up the image in Photoshop in order to use the previously explained process (under the 'Improving track implementation methods' section) to create a track image for the map. I used the opacity settings to essentially use this map image as a template. This process took a long time as the lines needed to be accurate in order to make the graphics of the game appear smooth. I ran the program to check the implementation of the lines and found that as more lines were drawn, the map took longer and longer to load. I decided that this will become a problem for bigger levels as it will take a long time to make the track file and also take a long time to load the level. I researched another method of achieving the same effect using the Tiled program, and discovered the objects feature.

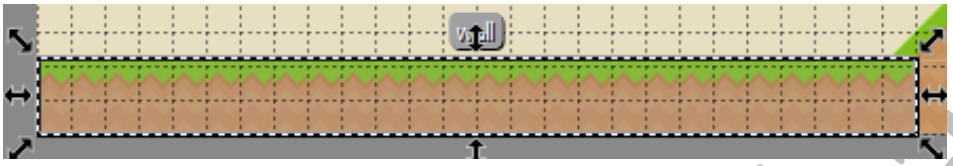
Spawning in objects from the Tiled file

Objects are a layer in Tiled and can be accessed in pygame. The location of the objects can be obtained as well as a property called name. This name property can be used to spawn different game objects in at different locations. This makes the track file surplus to requirement as objects can be created in the tiled file to be track. Similar to how the track drawn using the bitmap image uses individual pixels, the snapping to grid feature can be changed to snapping to individual pixels to re-size and manoeuvre the tiles on a pixel-by -pixel basis.

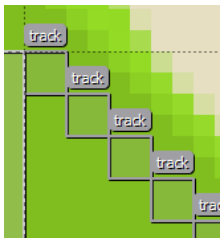


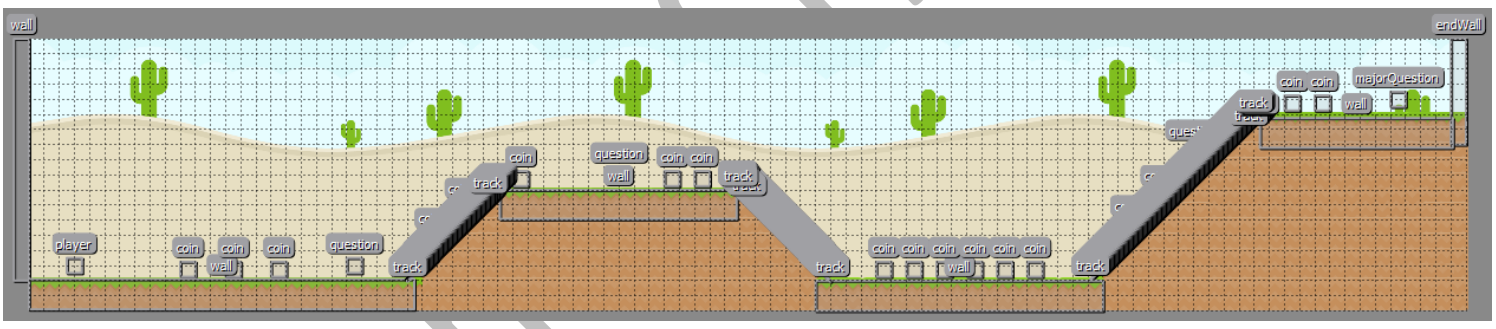
A new object layer was created using the region in the top-right section of the program. Following this, the rectangle tool was used to draw the object onto the screen. After this, the select objects tool was used to move and modify the shape of the object.





I made an object called wall and modified the shape to fit the platforms along the bottom. By making large rectangles for the platforms, the number of collision checks required is reduced, making the game run more efficiently. I created more objects for the other game sprites such as the player, end walls, coins, questions, majorQuestions (this will be explained later) and track. The track objects were created by making square objects of length 2 pixels and placing them diagonally down the slanted platforms. This is the finished map file.

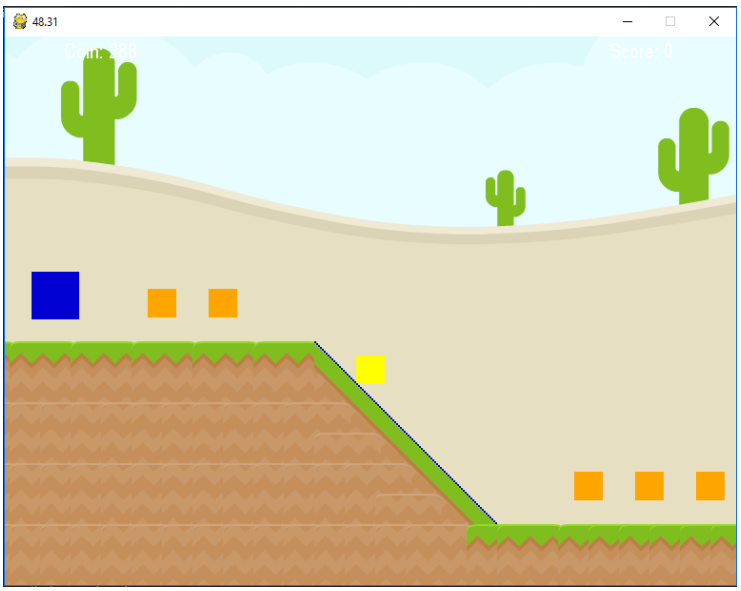


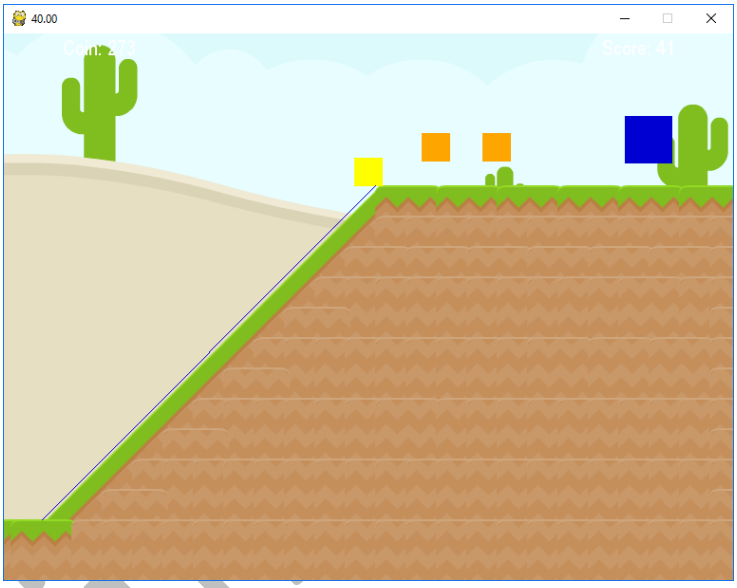



```
for tileObject in self.game.tiledMap.tiledMapFile.objects:
    if tileObject.name == "player":
        self.game.player = Player(self.game, tileObject.x, tileObject.y)
    elif tileObject.name == "wall":
        print("A wall has been added")
        Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled")
    elif tileObject.name == "endWall":
        print("A wall has been added")
        Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled", mode2="end")
    elif tileObject.name == "coin":
        Coin(self.game, tileObject.x, tileObject.y)
    elif tileObject.name == "question":
        Question(self.game, tileObject.x, tileObject.y)
    elif tileObject.name == "majorQuestion":
        Question(self.game, tileObject.x, tileObject.y, major=True)
    elif tileObject.name == "track":
        Platform(self.game, tileObject.x, tileObject.y, tileObject.width, tileObject.height, LIGHT_BLUE, "track", show=False)
```

The objects now need to be spawned into the game. To do this, inside the level function I used a ‘for’ loop to iterate through the objects in the tile map file. The ‘.name’ method was used to access the name attribute of each object and if this is equal to a specific string, the appropriate object is created. The x and y coordinates of the tile object is accessed for the location of the sprite, and the width and height of the tile object can also

be obtained if required. An additional parameter called ‘mode’ was added to the ‘Wall’ object. If this is equal to ‘tiled’ then the object is not added to the ‘allSprites’ group, which means that it will not be drawn to the screen, and a rectangle object is created to represent the object instead of a surface. This is required as the additional of the map graphics means that images for the walls and platforms are no longer required.





I ran the game to test out the spawning of the objects and found that all of the objects have been correctly loaded at their expected locations. The track objects are being shown in blue above. This colouring can be removed as explained for the wall objects.

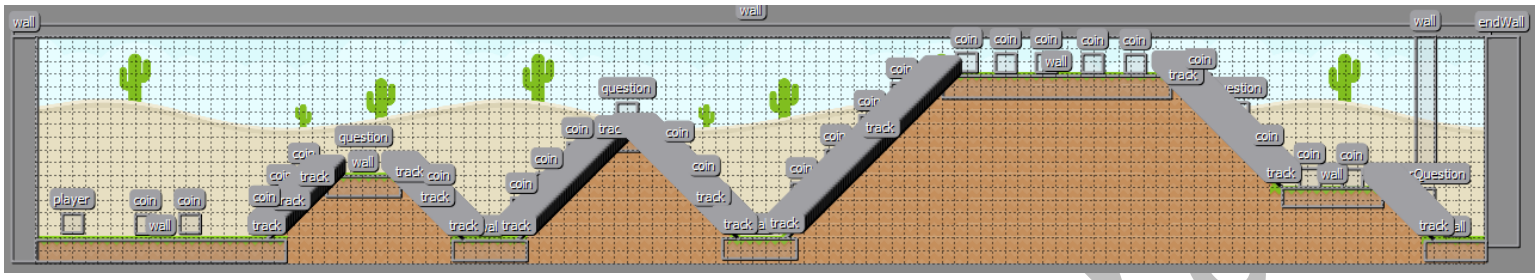
The level has now been successfully created and the remaining levels can be created using the same method.

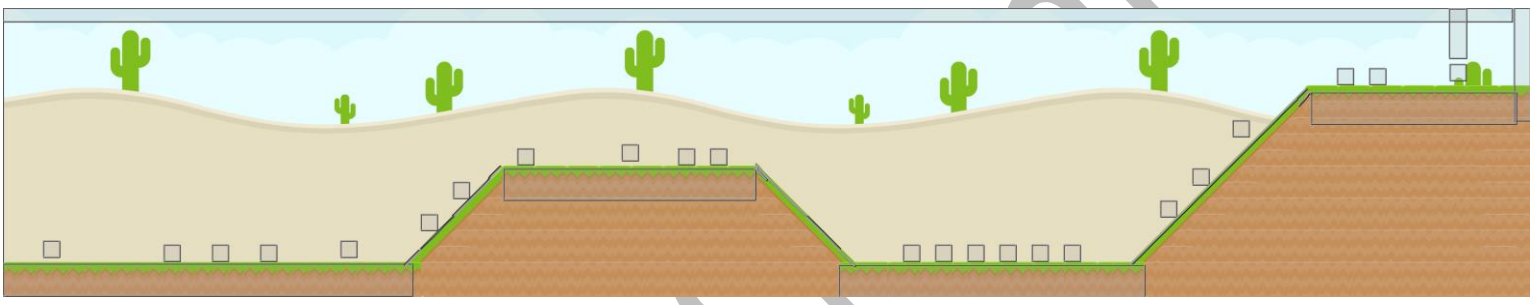
.

Creating the remaining levels

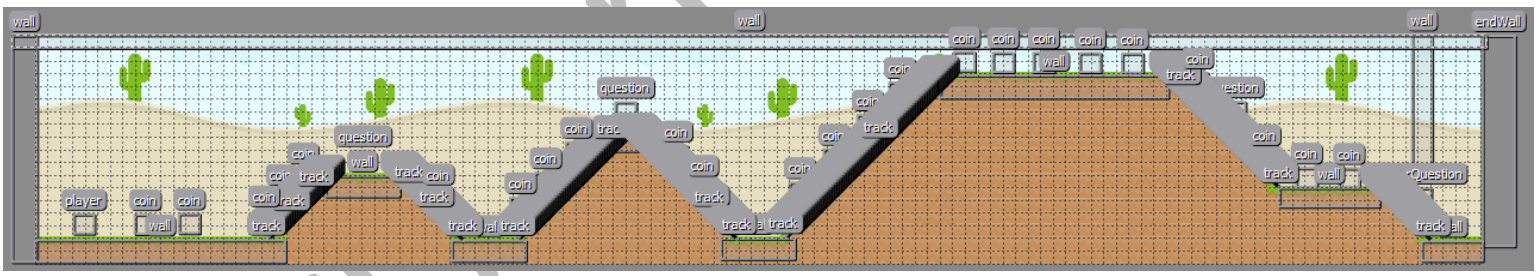
As a method of quickly crating levels for the game has now been implemented, more levels can easily be added. Here are the levels that have been created:

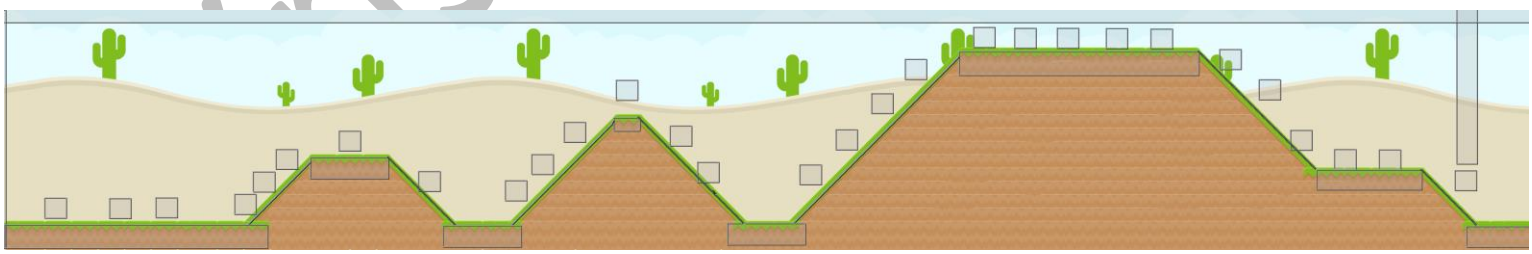
Level 1



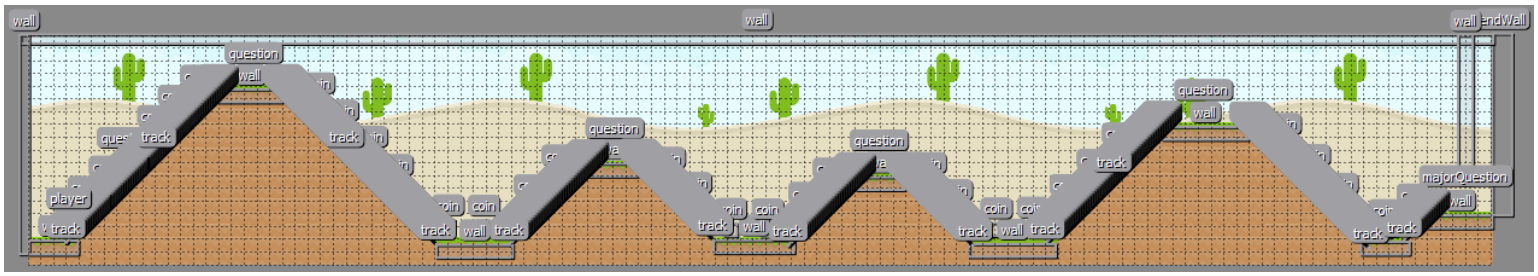


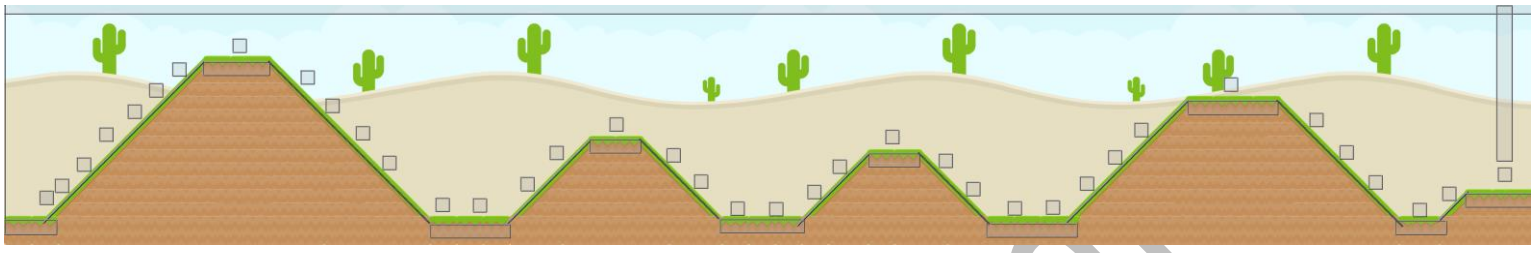
Level 2



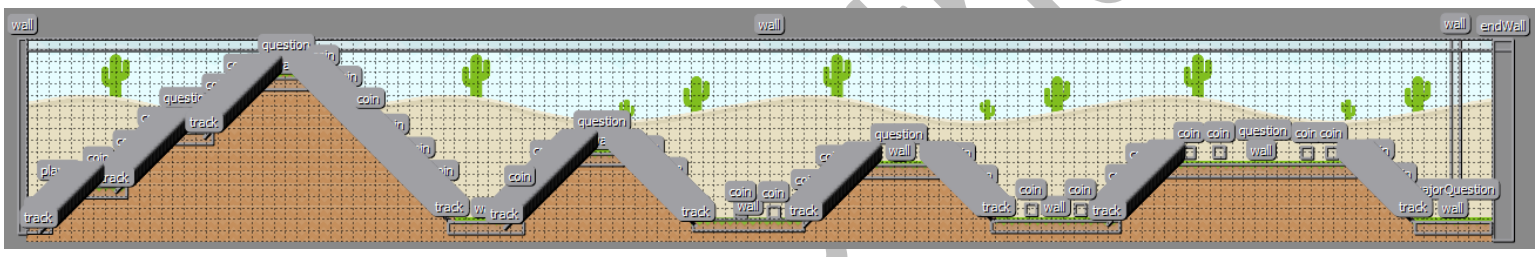


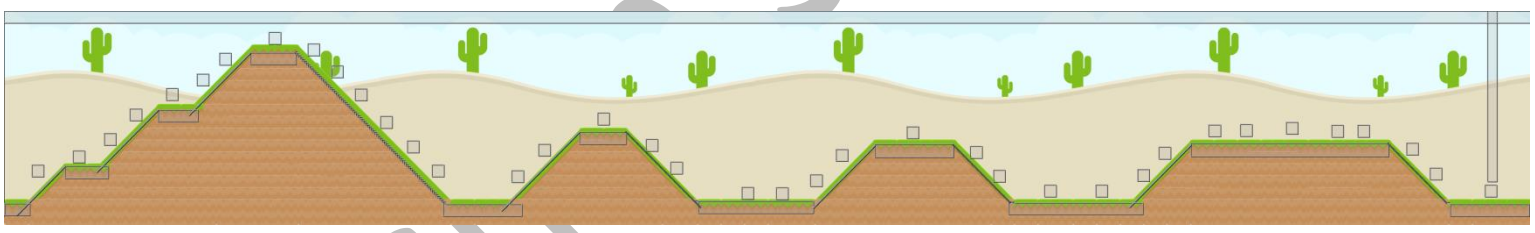
Level 3





Level 4





The first image shown for each level displays the view from the Tiled program including all of the objects and the names of them. From this image it is clear to see the wall objects that have been added around the outside border of the level to restrict the player from leaving the play area.

The second image shows just the graphics of the level with the objects represented by light grey coloured boxes. It is clearer to see the actual design of the level in this view.

To avoid the player from avoid the final question, by jumping over it, I added a wall above it. This means that the player cannot possibly travel past the final question and finish the level without answering this final question. All of the levels correctly loaded into the game and could be played.

Adding Animations

Adding graphics and animations are simple ways of making the game much more aesthetically pleasing and thus more enjoyable to play. There are two items in the game which could be animated and these are the coins and the questions objects.

Animating the coins

For this coins, the animation could be to have them rotating. To do this, first I found some images of coins rotating on the website referenced on page 152. I created a mirrored version of these images to add extra frames to the movement to make the animation smoother and longer. These were the 6 images used for the coin animation. The final image is the same as the first.

When these images are played one after another quickly, it creates a visual effect of the coins rotating. The code to achieve these needs to be implemented in the 'Coin' class so that this applies to every coin object created in the level.

```
self.image = pg.Surface((30,30))
self.image.fill(ORANGE)
self.rect = self.image.get_rect()
self.rect.center = (x, y)
```

I removed all of this code from the 'init' function of the 'Coin' class responsible for creating a surface for the image of the coin. Now, instead of a surface, these coin image need to be blitted to the screen.

However, the images first need to be loaded into the game. I made an array in the 'settings' file containing the filename of each of the coin images.

```
self.coinImages = {}
for img in range(len(COIN_IMAGES)):
    image = pg.image.load(path.join(self.imgFolder, COIN_IMAGES[img])).convert_alpha()
    size = image.get_size()
    self.coinImages[img] = pg.transform.scale(image, (int(size[0]/2), int(size[1]/2)))
```

```
COIN_IMAGES = ["coin1.png",
               "coin2.png",
               "coin3.png",
               "coin4.png",
               "coin5.png",
               "coin6.png"]
```

Next, in the 'loadData' function, I created a dictionary to store the coin images alongside an index, which will be the order in which they need to be blitted. I used a 'for' loop to iterate over the 'COIN_IMAGES' array created in the 'settings' file and load each of the images into the game. The size of each image is obtained using the 'get_size()' function. The image is then added to the dictionary using the current index of the 'for' loop. Before the image is added it is re-sized to make it smaller as the original image was 84 pixels by 84 pixels, which is too large completed to the players previous size of 30 pixels by 30 pixels. The 'pg.transform.scale' function was used to perform this task.

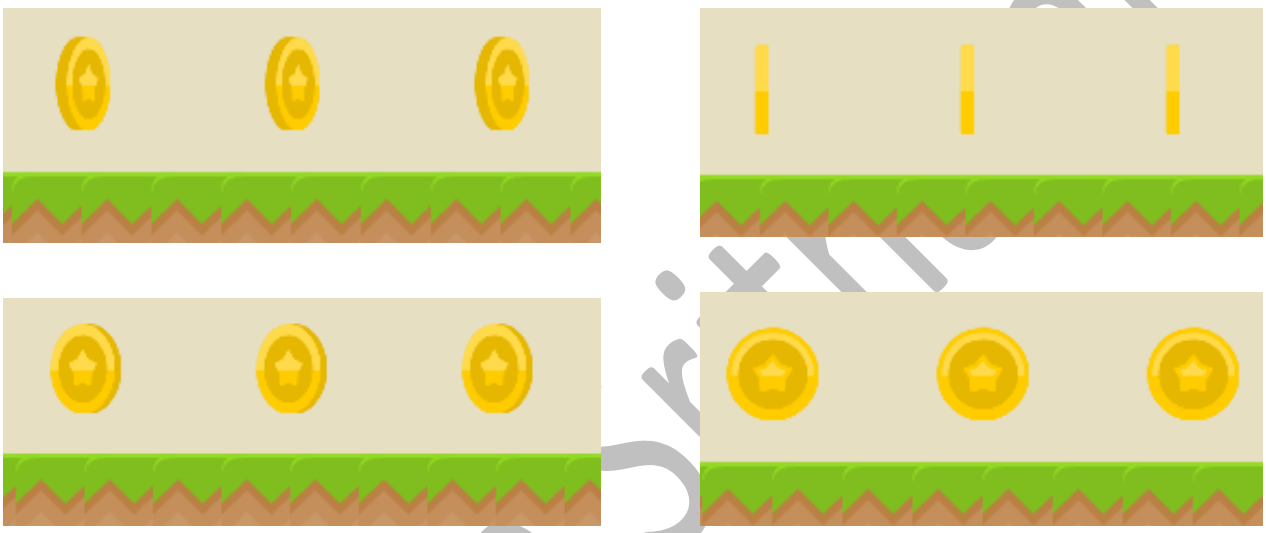
```
def update(self):
    now = pg.time.get_ticks()
    if now - self.lastUpdated > 200:
        self.lastUpdated = now
        self.currentFrame = (self.currentFrame + 1) % len(self.game.coinImages)
        self.image = self.game.coinImages[self.currentFrame]
        self.rect = self.image.get_rect()
        self.rect.center = (self.x, self.y)
```

A new 'update' function in the 'Coin' class was created to calculate which of the coin images to be displayed. The image needs to be changed depending on how long it has been since the previous image was blitted to the screen. The exact value of this time was calculated using trial and error. If the animation seemed to take too long, the time was decreased to change the images quicker and vice-versa. The 'time.get_ticks()' function is used to get the current time. This time minus the time of the previous picture change, which is stored in the 'self.lastUpdated' variable, needs

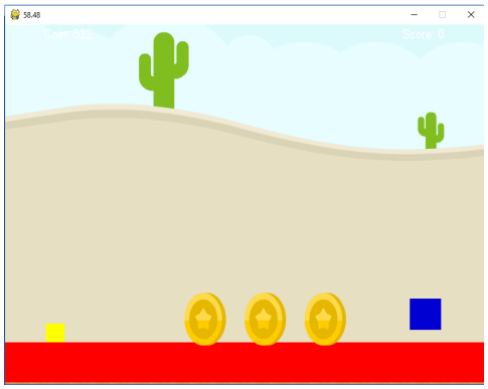
to be greater than 200 for the new coin's image to be changed. This 'self.lastUpdated' variable, alongside a variable called 'self.currentFrame' for storing the current coin image index, is defined in the 'init' function and have both been assigned values of 0. If the time has come to change the coin image, the 'self.lastUpdated' variable is updated to the current time. Following this, the current frame is calculated. The value of the current frame is incremented by one and then, the remainder after division by the number of coin images is calculated. This equation gives the index of the next image to be displayed. The 'image' variable of the coin object is obtained from the 'coinImages' dictionary in the main file using the value of the 'currentFrame' variable. The 'rect' object is calculated and the centre of the image is moved to the original location of the coin.

```
self.currentFrame = 0
self.lastUpdated = 0
```

I loaded the game and launched the level and the coin objects were correctly spawned and were being animated.



Error Encountered - When I ran the game to test if the coins have been spawned, the images for the coins were much larger than anticipated. This increased size is causing the coins to overpower the game screen.



Solution - I used the 'pg.transform.scale' function to change the dimensions of the coin image as they were loaded into the game. This solved the problem as the image being displayed for the coin is now smaller. The red coloured platforms are for testing purposes to detect whether the collisions between them and the player are correct.

Animating the question boxes

I decided to animate the question boxes by making them move up and down in the y axis. I first created an image for the question box using Adobe Photoshop.

Next, I loaded the image into the game using the 'pg.image.load' function. This was performed inside the 'loadData' function and the filename of the question box image was stored in the 'settings' file in the constant 'QUESTION_IMAGE'.

```
QUESTION_IMAGE = "boxQuestion.png"
```

```
self.questionImage = pg.image.load(path.join(self.imgFolder, QUESTION_IMAGE)).convert_alpha()
```

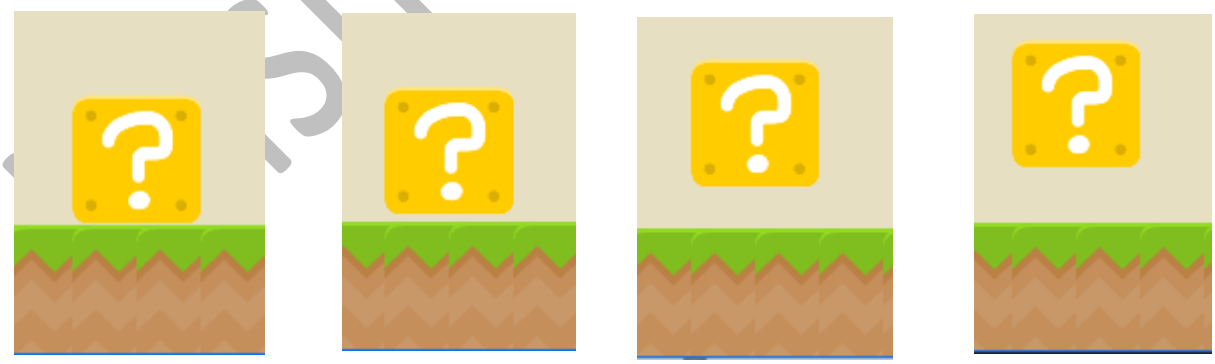
```
self.image = game.questionImage.copy()
self.image = pg.transform.scale(self.image, (64, 64))
size = self.image.get_size()
self.width, self.height = size[0], size[1]
self.rect = self.image.get_rect()
```

In the 'init' function of the 'question' class the code to create a surface for the object is removed. In its place, this code to set the 'image' variable of this object to created question image is added. The 'copy' function is used to set the image to a copy of the loaded in image. This means that if any changes are made to the image, the original image is not affected and can still be used for other objects. The 'pg.transform.scale' function is used to re-scale the image to a smaller size, 64 pixels by 64 pixels. The size of the image is retrieved and saved in appropriate variables. The 'rect' object of the image is then calculated and stored.

```
self.velY = 1
```

An 'update' function is then created in the 'Question' class to animate the object. In this function, the y position of the image is increased by a value for the vertical velocity. This 'velY' variable is created in the 'init' function and initialised with a value of 1. An upwards and downwards movement needs to be implemented. When a certain height above the original position is reached, the velocity of the object is made negative to initiate a downwards movement. Similarly, when a certain height below the original positon is reached, the velocity is made positive. This position is a quarter of the height above and below the original position. Now the question boxes should be animated by moving upwards and downwards by a small but noticeable amount.

```
def update(self):
    self.rect.y += self.velY
    if self.rect.top > self.y - self.height/4:
        self.velY = -1
    if self.rect.bottom < self.y + self.height/4:
        self.velY = 1
```



These images show that the question box is correctly and successfully animating as programmed. This small animation has improved the aesthetics and enjoyment of playing the game as the player will believe that they could possibly avoid the questions by using their in-game driving skills and going under the boxes. This is not possible however, as the objects are not high enough, but it does make the game more enjoyable.

Implementing Major questions

As required by the success criteria of the project, there needs to be a more difficult final question which the player has to get correct in order to complete the level.

Asking major questions

The first step of adding this feature was to edit the CSV file containing the questions to make some questions major questions. The term major question refers to the harder questions asked at the end of the level.

questionID	questions	correctAnswer	wrongAnswer1	wrongAnswer2	wrongAnswer3	wrongAnswer4	difficulty	level	isMajor
3	15 + 17	32	30	42	23	34	easy	1	TRUE
6	(2 + 4) x 3	18	19	16	24	21	easy	1	TRUE
14	Find x when 14x/7 = 28-2x	7	10	9	12	5	medium	1	TRUE
21	Complete the Square: x^2 + 10x + 12	(x + 5)^2 - 13	(x + 5)^2 + 12	(x - 5)^2 + 13	(x + 10)^2 + 12	(x - 4)^2 + 7	medium	1	TRUE
27	Factorise 6x^2 + 7x + 2	(3x + 2)(2x + 1)	(6x + 3)(x + 2)	(6x + 3)(x - 2)	(3x + 2)(2x - 1)	(3x - 4)(2x + 3)	hard	1	TRUE
31	Find dy/dx when y = tan(3x) + 7x^2 + 84x	3sec(3x)^2 + 14x + 84	3sec(3x)^2 + 84x + 14	3cos(3x)^2 + 7x + 14	sin(3x)^2 + 7x + 42	tan(3x)^2 + 14x + 84	hard	1	TRUE

I edited the 'isMajor' column of the file to make this value equal to TRUE for some of the questions from each difficulty. These specific questions now need to be asked for the final question in the levels.

```
self.majorQuestion = False
self.majorQuestionCorrect = True
```

Two varaibles called 'majorQuestion' and 'majorQuestionCorrect' were created in the 'new' function to determine when this type of question needs to be asked. The second variable is used to determine whether the player has answered this question correctly as if not, they have failed the level and an appropriate screen needs to be displayed.

Next, I modified the 'Question' class to add a 'major' parameter with a default value of False. This parameter is then stored as a class variable in the 'init' function.

```
class Question(pg.sprite.Sprite):
    def __init__(self, game, x, y, major=False):
```

```
self.major = major
```

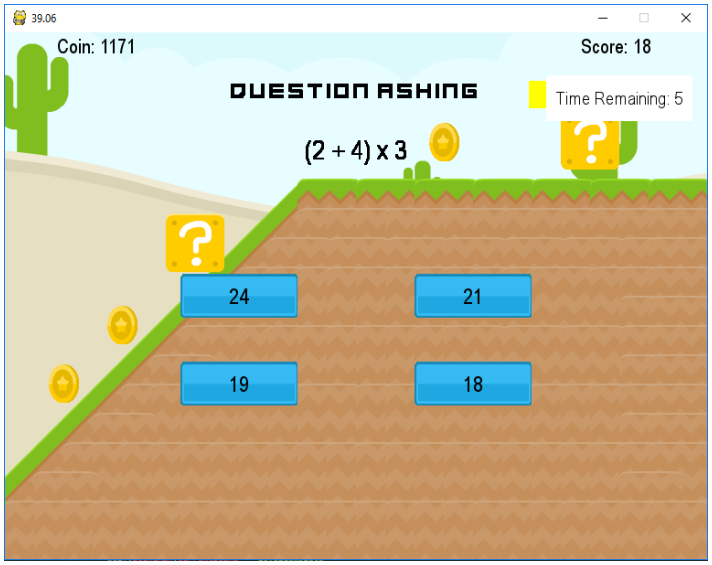
```
if hitQuestion[0].major == True:
    self.majorQuestion = True
```

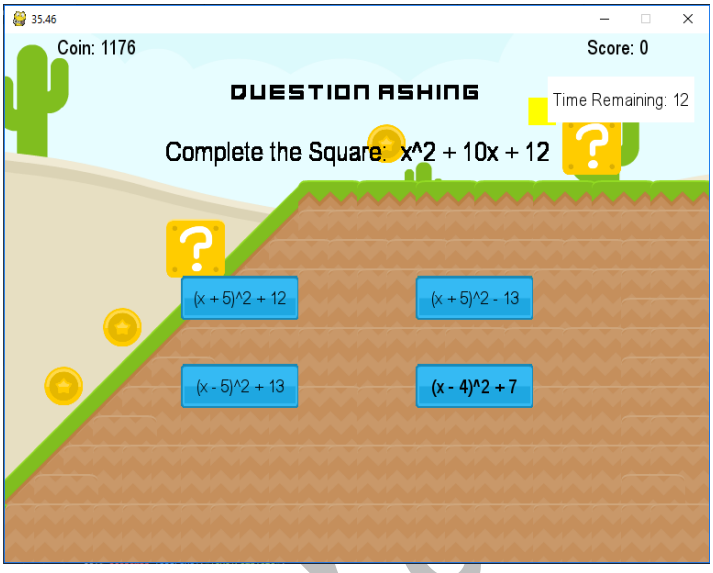
In the collision code between the player and the question box, if the 'major' parameter of the collided question box is equal to True, then the 'self.majorQuestion' variable is set to True as this indicates that the collided question is a major question. When the player collides with the end of the level wall, this 'majorQuestion' variable is reset to False in preparation for the next level's major question.

```
if hitLevelEnd:
    self.totalScore += self.score
    self.majorQuestion = False
```

```
if self.majorQuestion == True:
    self.questionID = []
    for i in range(len(self.questionData)):
        if self.questionData[i][9] == 'TRUE':
            if self.questionData[i][7] == self.settingsQuestionDiff:
                print("This is the Question:", self.questionData[i][1], self.questionData[i][9])
                self.questionID.append(self.questionData[i][0])
```

This block of code has been added to the 'getQuestion' function to execute a different procedure if a major question is needed. If when the 'getQuestion' function is called the 'majorQuestion' variable is True, the 'questionID' array is re-declared as empty. The 'questionData' array is then iterated over checking if the 9th column of the file (which is the 'isMajor' column) is equal to 'TRUE'. If this is the case, the question's difficulty is then checked to see if it matches the chosen difficulty. If so, the 'questionID' value of the question is appended to the 'questionID' array. The remainder of the code in the 'getQuestion' function executes depending on the contents of the 'questionID' array. By changing the contents of this array to meet certain needs, the remainder of the function will work as expected to ask the player a major question.





These images show that the major question is correctly being asked at the end of the level. As shown previously in the spawning objects from Tiled section on page 157, a 'majorQuestion' object has been added to the end of the level and, the code to create an instance of the 'Question' class with the 'major' parameter set to True has been added as well. The second image is when the difficulty is set to medium. The correct medium difficulty major question is being asked to the player.

Failing the level

If the player answers the major question incorrectly they have failed the level and should not be allowed to submit their score to the leader board, even if it is high enough to be in the top ten scores for that level. The 'majorQuestionCorrect' variable can be used to track whether this question has been answered correctly. Be default, this variable is set to True so if an answer is clicked and it is correct, this variable remains True. If an answer is chosen and it is incorrect, if this is a major question, which is checked using the 'self.majorQuestion' variable, then the player has answered the major question incorrectly so the 'majorQuestionCorrect' is set to False. Similarly, when the player runs out of time to answer a question, if the question is a major question then this variable is set to False.

```
else:
    if self.majorQuestion:
        self.majorQuestionCorrect = False
    print("INCORRECT ANSWER CHOSEN: {}".format(self.selectedAns))
```

```
if self.timeRemaining == -1:
    self.timeOut = True
    self.answerClicked = True
    if self.majorQuestion:
        self.majorQuestionCorrect = False
```

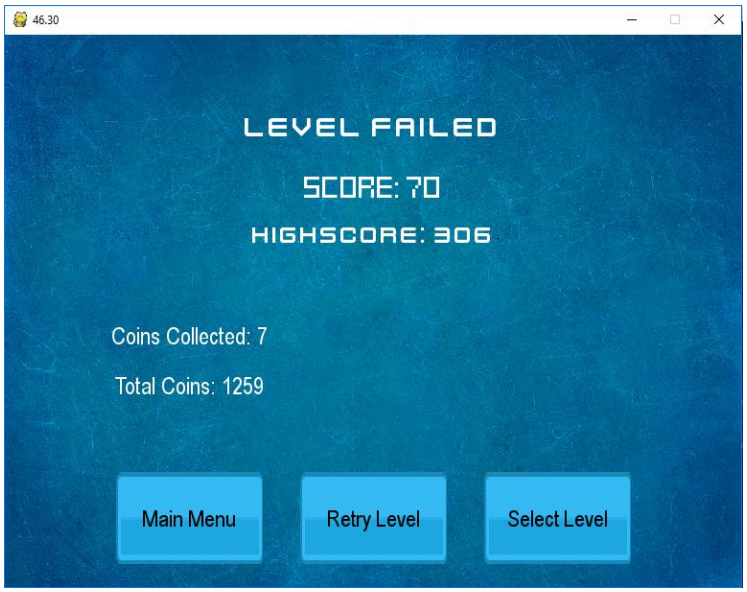
```
if self.game.majorQuestionCorrect:
    self.game.drawText("Level Completed", 30, WHITE, WIDTH/2, HEIGHT/6, fontName=self.game.interfaceFont)
```

In the 'levelComplete' function in the 'sceneManager' class, the 'majorQuestionCorrect' variable is checked and if it is True, the level has been completed so the original code is executed. If it is False, the text 'Level Failed' is displayed to the screen instead and a button for retrying the level is created. The tag of this button is the value of the previous scene. The main menu and level select buttons are still displayed however the input box for the highscore is not displayed as the player has failed the level.

```
else:
    self.game.drawText("Level Failed", 30, WHITE, WIDTH/2, HEIGHT/6, fontName=self.game.interfaceFont)
    self.samelevel = Button(self.game, self.prevScence, WIDTH /2, HEIGHT*7/8, WIDTH/5, HEIGHT/6, YELLOW, LIGHT_BLUE, "Retry Level")
```

```
if self.currentScene == "levelComplete":
    if not self.updatedHighscore and self.game.majorQuestionCorrect:
        self.updateHighscore(self.nextLevelTagNumber-1, self.game.score)
```

Finally, in the 'sceneManager' class's 'update' function the 'updateHighscore' function is called if the highscore has not been updated and the major question was answered correctly. This step is required as otherwise, the program will attempt to update the highscore when the level has been failed.



I ran the program and got the final question incorrect. The level failed screen was correctly displayed alongside the retry button and the other game information. The retry button successfully loaded the same level again.

Review

End of level questions and graphics have now been implemented into the game, greatly enhancing its appearance and enjoyment. Spawning the game objects using 'Tiled' is much more efficient that using bitmap images as this requires extensive image processing to achieve the same effect. This method of creating the track also means that the pixel-wide collision boxes only need to be used for the slopes. This improves the performance of the game as an increased number of these objects had a negative correlation on the frame rate of the game. This is expected as there will be a dramatically increased number of collision checks that need to be performed in every frame.

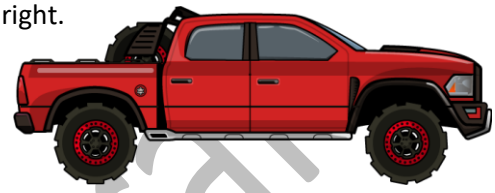
The end of level questions means that the player has an incentive to getting the questions correct as they will not be able to submit the score that they have worked for otherwise. The next and final features to add are the player's image and the shop screen.

Player image and Shop Screen

Up until this point the player's image has been a surface primarily for testing purposes. It is simpler to get all the features of the game fully functioning with a basic image for the player in comparison with a complicated image from the start. This player image links closely with the shop screen as this screen will have the functionality to change this variable inside the 'Player' class.

Setting a new player image

The image that I have decided to use for the player image is shown on the right. First this image needs to be loaded into the game and resize as its original size is 691 pixels by 257 pixels. The code shown below was added to the 'loadData' function and it loads the car image into pygame, gets its dimensions and then rescales it to an appropriate size.



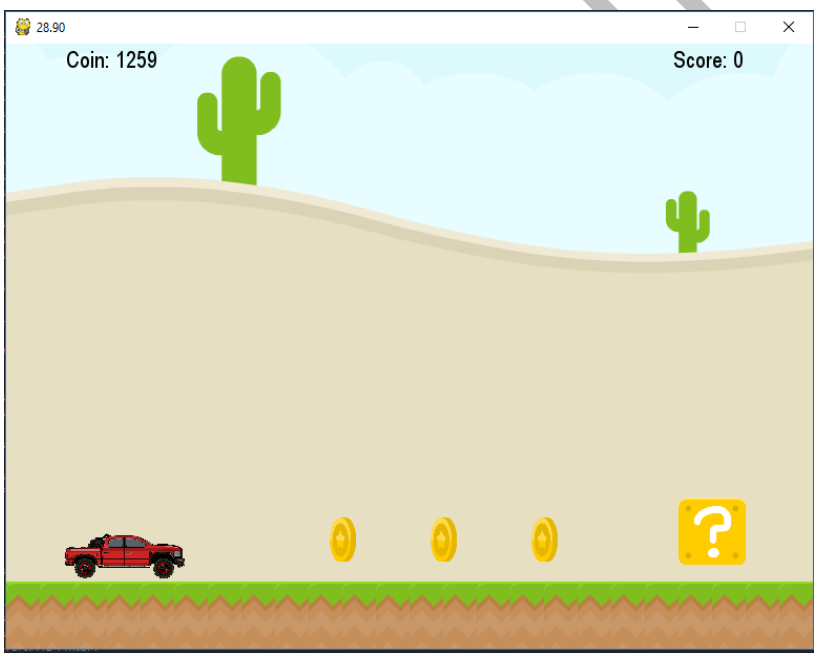
```
self.carImage = pg.image.load(path.join(self.imgFolder, 'car.png')).convert_alpha()
self.carImgSize = self.carImage.get_size()
self.carImage = pg.transform.scale(self.carImage, (int(round(self.carImgSize[0]/6,0)), int(round(self.carImgSize[1]/6,0))))
```

```
self.playerImage = self.carImage
```

A variable called 'playerImage' is then created in the same file and this will be the current image of the player. By default, this variable is set to the car's image.

```
self.image = self.game.playerImage.copy()
self.width = self.image.get_size()[0]
self.height = self.image.get_size()[1]
```

Next, the code to create a surface in the 'Player' class's 'init' function is removed and replaced with the code shown on the left. In this, the player object's image is set to a copy of the image stored in the 'playerImage' variable. The dimensions of the image are obtained and stored in the appropriate class variables to be used in other parts of the class.



The player image has correctly been loaded into the game. The movement of the player still correctly works as previously. Now more vehicles images can be added and stored in the shop, where the player can switch the current vehicle.

Making the shop screen

I first imported another vehicle image using the same previously explained procedure.

```
self.bikeImage = pg.image.load(path.join(self.imgFolder, 'bike.png')).convert_alpha()
self.bikeImgSize = self.bikeImage.get_size()
self.bikeImage = pg.transform.scale(self.bikeImage, (int(round(self.bikeImgSize[0]/10,0)), int(round(self.bikeImgSize[1]/10,0))))
```

Next, in the 'shop' function, I added the code to laod the appropriate background image for the current scene. This is required as the 'shop' function will be re-called when a button is pressed to select a vehicle.

```
def shop(self):
    self.image = pg.transform.scale(self.game.menuImages['shop'], (WIDTH, HEIGHT))
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)
```

```
self.carButton = Button(self.game, "car", WIDTH*2/5, HEIGHT*5/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Car")
self.bikeButton = Button(self.game, "bike", WIDTH*2/5, HEIGHT*8/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Bike")
```

Two buttons for the car and the bike vehicles are then created. They are positioned towards the centre of the screen and have the tags 'car' and 'bike' respectively. I created a variable in the 'Game' class called 'imageType' to store the actual vehicle that is currently chosen for the player's image. The value of the 'imageType' variable is set in the 'loadData' function depending on the 'playerImage' variable.

```
if self.playerImage == self.carImage:
    self.imageType = "car"
elif self.playerImage == self.bikeImage:
    self.imageType = "bike"
```

```
if self.game.imageType == "car":
    currentVehicle = "Car"
if self.game.imageType == "bike":
    currentVehicle = "Bike"
```

Back in the 'shop' function, the 'imageType' variable is checked and a variable called 'currentVehicle' is created and adjusted accordingly.

This variable is used to print the text "Current Vehicle:" to the screen followed by the current vehicle.

```
self.game.drawText("Current Vehicle: {}".format(currentVehicle), 25, WHITE, WIDTH/2, HEIGHT*3/12, fontName=self.game.interfaceFont2)
```

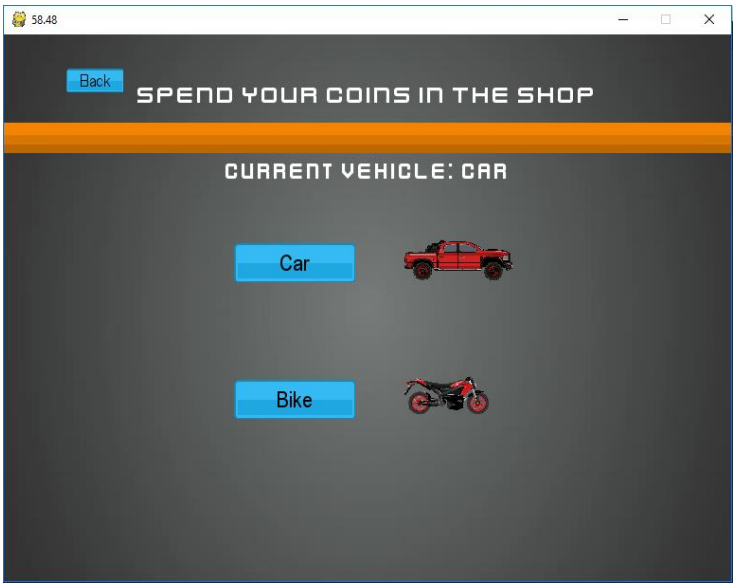
```
self.game.screen.blit(self.game.carImage, (WIDTH*11/20, HEIGHT*9/24))
self.game.screen.blit(self.game.bikeImage, (WIDTH*11/20, HEIGHT*15/24))
```

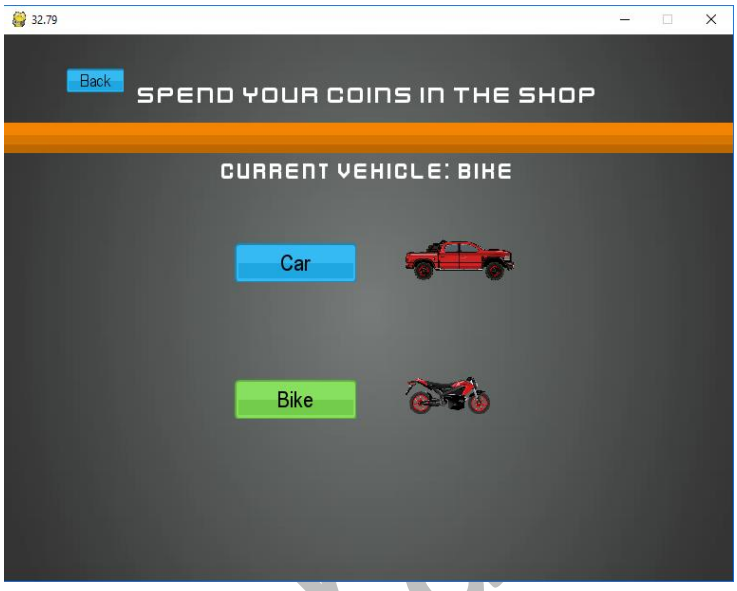
Lastly, the images of the car and bike are blitted to the screen to the right of the buttons so that it is clear to the player which vehicle they are picking.

```
if self.currentScene != "shop" or (self.currentScene == "shop" and button.tag == "mainMenu"):
```

The 'update' function of the 'sceneManager' class is then modified to accept button presses from the buttons on the shop screen. The above 'if' statement is added to make the function execute a different block of code if the current scene is the shop and the button pressed does not have the 'mainMenu' tag. In the 'else' statement, the tag of the button is checked the 'playerImage' variable in the 'Game' class if set to a new image as required. The 'imageType' variable is also reset to the current vehicle type. The buttons on the screen are then cleared and the 'shop' function is called once more. The "Current Vehicle:" text will now change and the player will be able to select a new vehicle if required.

```
else:
    if button.tag == "car":
        self.game.playerImage = self.game.carImage
        self.game.imageType = "car"
    elif button.tag == "bike":
        self.game.playerImage = self.game.bikeImage
        self.game.imageType = "bike"
    for button in self.game.buttons:
        button.kill()
    self.loadLevel("shop")
```





These images of the shop screen show that the screen is working as expected, with the text changing as required when the bike button is selected.

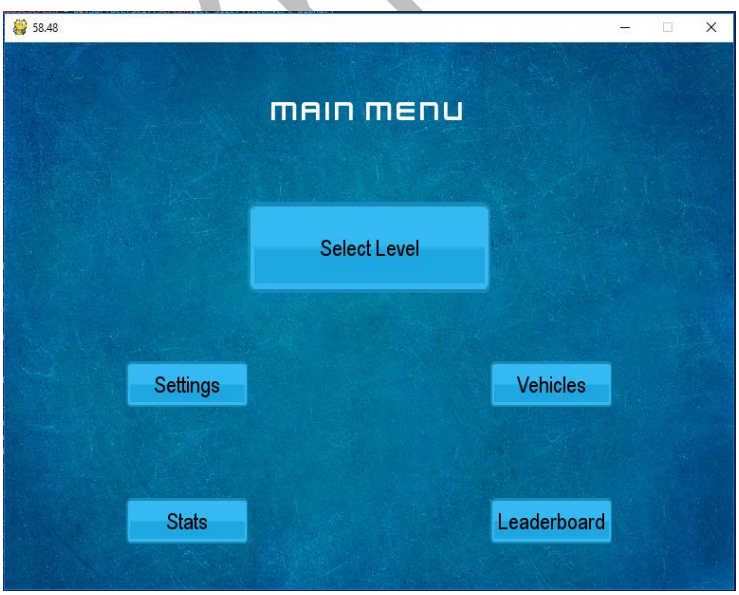
User Feedback for the Shop Screen

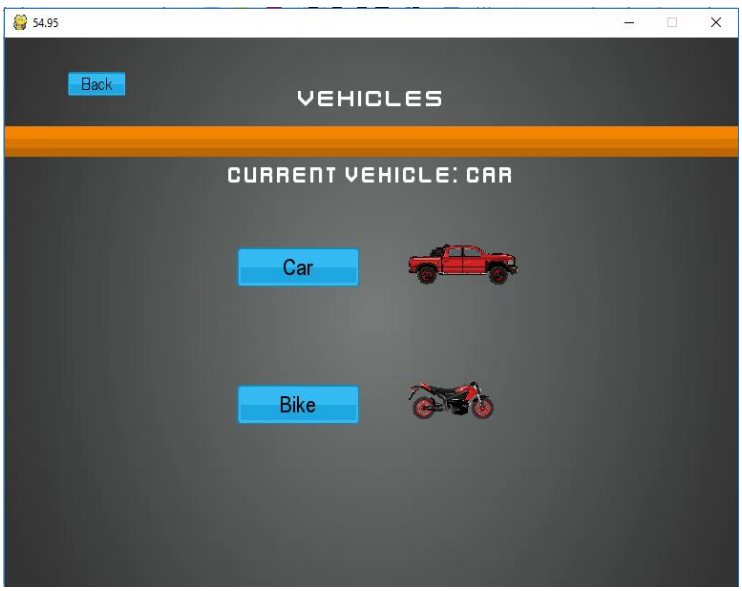
Krishan: What are your thoughts about this shop screen in my game?

Ayub: I really like the graphics, especially how it shows you a picture of the vehicle you are selecting. One improvement I can suggest is to rename the screen to vehicles instead of Shop as there are not any options to use the coins to buy the vehicles.

Krishan: Yes, the intention was to add this feature to unlock the vehicles using the earned coins however due to time issues, I have not been able to add this functionality to the game. You are correct about this misleading of the name of the screen. It is better to call it Vehicles.

Using this feedback, I have changed the name of this screen to vehicles both in the screen itself as well as on the main menu screen.





Adding an Instructions screen

Currently the player is not given any information on how to play the game. They are not aware of the controls nor the aims of the game or how to compete the level. They need to be presented with this information at the start of the level so that they know how to finish it. To do this, I created a new menu screen called 'Instructions'.

A function called 'instructions' was created in the 'SceneManager' class and the image for this screen is set to the same as the image for the 'levelComplete' screen. This image is then blitted to the screen.

```
def instructions(self):
    self.image = pg.transform.scale(self.game.menuImages['levelComplete'], (WIDTH, HEIGHT))
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)
```

```
self.game.drawText("Instructions", 30, WHITE, WIDTH/2, HEIGHT/6, fontName=self.game.interfaceFont2)
self.game.drawText("WASD or Arrow Keys to move", 20, WHITE, WIDTH/2, HEIGHT*3/8, fontName=self.game.interfaceFont2)
self.game.drawText("Space Bar to jump", 20, WHITE, WIDTH/2, HEIGHT*4/8, fontName=self.game.interfaceFont2)
self.game.drawText("Answer the questions and collect the coins!", 20, WHITE, WIDTH/2, HEIGHT*5/8, fontName=self.game.interfaceFont2)
self.game.drawText("Make sure you get the final question", 20, WHITE, WIDTH/2, HEIGHT*23//32, fontName=self.game.interfaceFont2)
self.game.drawText("correct to finish the level!", 20, WHITE, WIDTH/2, HEIGHT*25/32, fontName=self.game.interfaceFont2)
self.game.drawText("Press A Button To Start!", 22, WHITE, WIDTH/2, HEIGHT*7/8, fontName=self.game.interfaceFont2)
```

Next inside the function, a series of 'drawText' are statements are called to print the text instructions to the screen. These state the controls for moving the vehicle, the aim of the player in the level and information about how the final question needs to be answered correctly in order to complete the level.

The player needs to be given time to read this information therefore, I decided to implement a 'press-a-button-to-continue' feature, allowing the player to press a button to let the program know that they want the level to start. A function called 'waitForKey' (which was explained when the 'SceneManager' class was created on page 81) has already been created for this purpose. Before this function is called, the 'display.flip()' function is called to update the game display so that the recently blitted text is visible to the player.

```
pg.display.flip()
self.waitForKey()
```

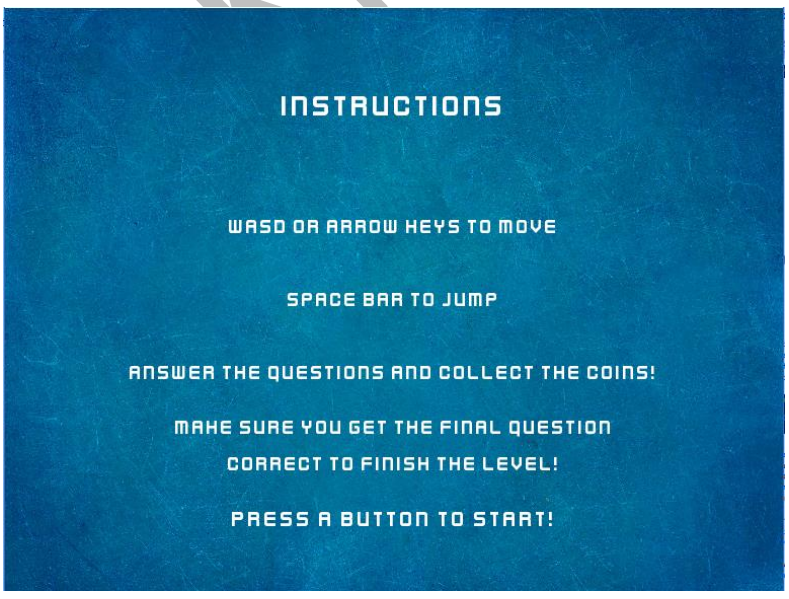
```
self.loadLevel('instructions')
```

This function now needs to be called and this occurs when a level button is pressed. The 'loadLevel' function to load this screen is called after the block of code initialising everything for the start of a new level.

I decided to also add a button to access the instructions screen to the 'settings' screen so inside the 'instructions' function, I added a statement to check the previous scene and if it is 'settingsMenu', the settings screen will be loaded. A button was created on the settings page for the instructions screen.

```
if self.prevScence == "settingsMenu":
    self.currentScene = 'settingsMenu'
    self.loadLevel('settingsMenu')
```

```
self.instructionsButton = Button(self.game, "instructions", WIDTH/4, HEIGHT*4/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Instructions")
```



The instructions screen correctly loads at the start of a level. When a button is pressed, the 'waitForKey' function correctly detects it and causes the level to initiate. This screen can also correctly be accessed from the settings menu. As expected, when a button is pressed, the settings menu is correctly resumed.

Adding Music

The final addition to the game that I have decided to make is the inclusion of game music. This will enhance the atmosphere while playing the game and make it more enjoyable to the user. First I obtained some music tracks to be played whilst in the menu screens, the level and the pause menu. I saved these files inside a new directory called 'snd' and created a variable called 'sndFolder' in the 'loadData' function to access this folder.

```
self.sndFolder = path.join(self.gameFolder, 'snd')
```

```
self.menuMusic = path.join(self.sndFolder, 'backgroundMusic.ogg')
self.pauseMusic1 = path.join(self.sndFolder, 'PauseMusic.ogg')
self.pauseMusic2 = path.join(self.sndFolder, 'PauseMusic2.ogg')
self.levelMusic = path.join(self.sndFolder, 'LevelMusic.ogg')
```

Next, I saved the file locations of the required music files to class variables in the 'loadData' function. To be compatible with use in pygame, I first converted the music files to 'ogg' format.

In the 'new' function, I created three new variables to track when the music were being played. The 'pg.mixer.music.load()' function was called to load the music in required location. The 'pg.mixer.music.play()' function was then called to start playing the music that has been loaded in. The 'loops' parameter of this function was set to '-1' to keep the music constantly looping.

```
pg.mixer.music.load(self.menuMusic)
pg.mixer.music.play(loops=-1)
self.menuMusicStarted = True
self.pauseMusicStarted = False
self.levelMusicStarted = False
```

Next, in the 'update' function, if the level music has not started when the current scene is a level one, the current music is stopped. This is achieved using the 'fadeout' function to slowly stop the music and make it less noticeable. The fadeout duration is 500 miliseconds, which is half of a second. The music for the level is then loaded and played. The 'levelMusicStarted' variable is set to True to indicate that the level music has started to play.

```
if self.sceneMan.currentScene not in MENU_SCREENS:
    if self.levelMusicStarted == False:
        pg.mixer.music.fadeout(500)
        pg.mixer.music.load(self.levelMusic)
        pg.mixer.music.play(loops=-1)
        self.levelMusicStarted = True
```

The code to start the menu music is added to the 'else' statement of the above 'if' statement. This is only required if the 'menuMusicStarte' variable is False. The same procedure is followed to stop the level music and begin the menu music.

```
else:
    if self.menuMusicStarted == False:
        self.levelMusicStarted = False
        pg.mixer.music.fadeout(500)
        self.menuMusicStarted = True
        pg.mixer.music.load(self.menuMusic)
        pg.mixer.music.play(loops=-1)
```

Finally, the pause screen music is added. In the 'draw' function, when the 'paused' variable is True, the music needs to be started if the music has not already been started, and a question is not being asked. As there are two tracks for the pause screen, the 'random'.choice' function is used to randomly select either '0' or '1'. If a '0' is chosen, one track is load into the mixer otherwise, the other track is loaded. Following this, the 'play' function is called to begin playing the music.

```
if self.paused:
    if self.pauseMusicStarted == False and self.askQuestion == False:
        self.pauseMusicStarted = True
        self.levelMusicStarted = False
        option = random.choice([0,1])
        if option == 1:
            pg.mixer.music.load(self.pauseMusic1)
        else:
            pg.mixer.music.load(self.pauseMusic2)
        pg.mixer.music.play(loops=-1)
```

A variable called 'musicOn' was created and this controlled whether the music in the game was played. In all locations where music was played, the state of this variable was checked and the code to play the music was only executed if this was True.

```
self.musicOn
```

An additional track to the level music was added and functionality to change this track in the 'settings' screen was added. Additional buttons have been added to this screen to change the track or switch off the music completely. The music correctly loaded into the game and was played seamlessly throughout the menu screens. The music correctly changed when the level was started, when the pause menu was opened and when it was turned off. The benefits of adding this music into the game and further evidence of it correctly functioning will be provided in the User Feedback section.

Testing

Throughout the development process, I have used testing in each iterative phase to find the problems and issues with the program. I used various debugging methods such as tracking the values of the variables by printing them to the console, and executing the program using a different IDE when the 'input' function was not working. Additionally, all of the errors encountered along this development process were documented and explained. How the solution was derived following the failed tests, as well as the logical understanding of the changes that needed to be made to fix the problem were also thoroughly explained.

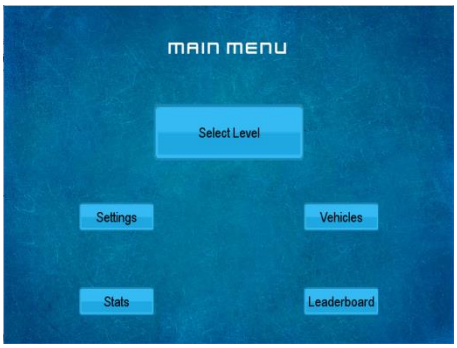
The game has now been completed so the entire program can be tested for versatility, robustness and correct functionality. The testing under the 'Evidence' column in the Success Criteria on pages 39 and 40will be used to guide the testing process. The menu system of the game will first be tested. The test plans created for each user interface screen on pages 48 to 66 will be used to perform this task.

Testing the Menu System

The success criteria being tested are criteria numbers 2, 13 and 14.

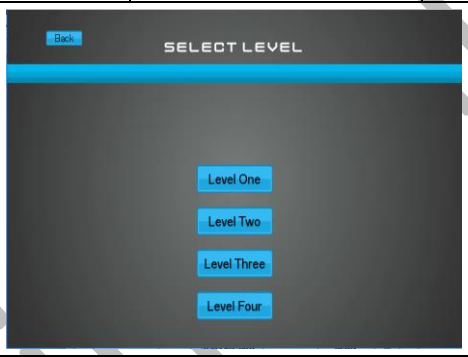
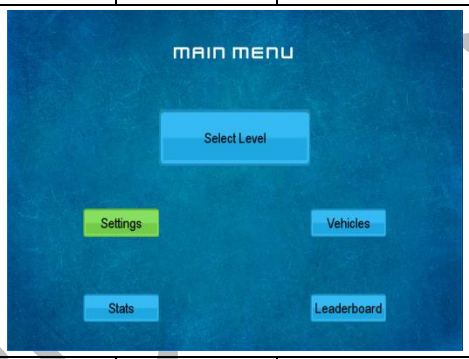
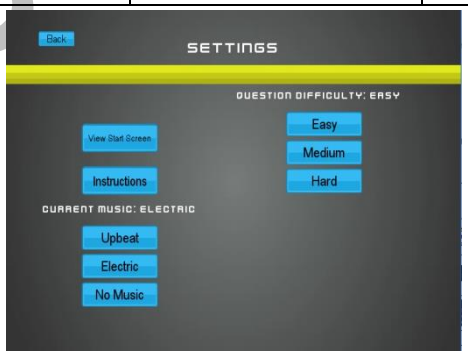
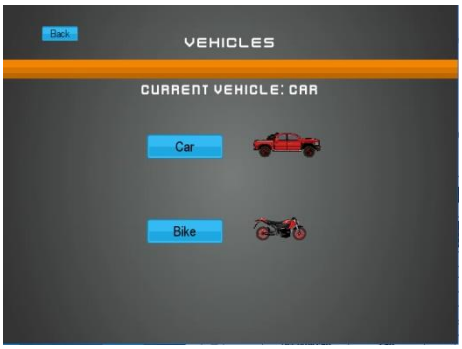
- Create an arrangement of screens which can be linked to form a menu system. All the settings and features should be accessible from here.
- All scores, for levels, mini game and endless mode, are correctly added to their respective leader boards. All leader boards are accessible from the leader board screen via the main menu.
- The game statistics are correct and are accessible via the main menu.
  The game achievements are correctly unlocked as the game progresses.
  Coins can be used to purchase different items in the shop.

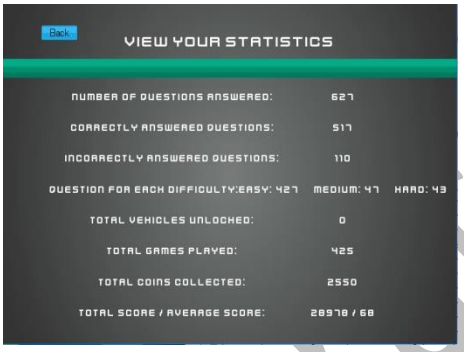
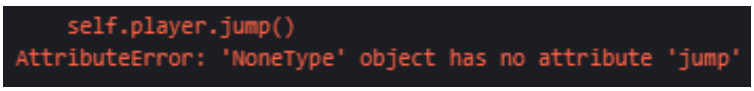
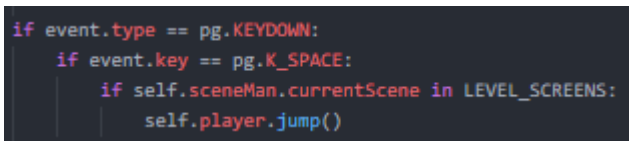
Testing the Start Screen

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
1	Check the response of the Start screen	Valid	The main menu screen should be loaded	Click anywhere inside the game window using the mouse buttons	Expected outcome returned. Test successful
	 				.
2	Check that the Start Screen is correctly functioning	Valid	The main menu screen should be loaded	Press a button on the keyboard whilst on this screen.	Expected outcome returned. Test successful
3	Make sure that when the game window is closed when the cross in the top-right is clicked.	Valid	The game window should close as the 'pg.quit function should be executed.	Click anywhere inside the game window.	Expected outcome returned. Test successful

Testing the Main Menu Screen

The test plan on page 50 was used for these tests. Some of the tests specified in the test plan need to be modified due to those features not being added to the game. The reason for this will be explained in the evaluation.

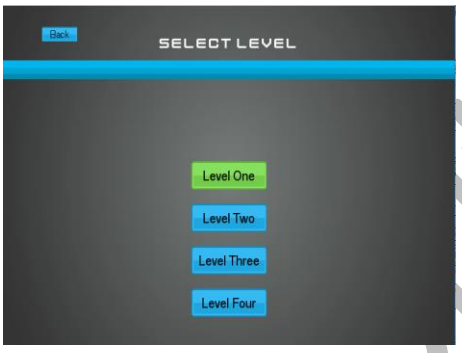
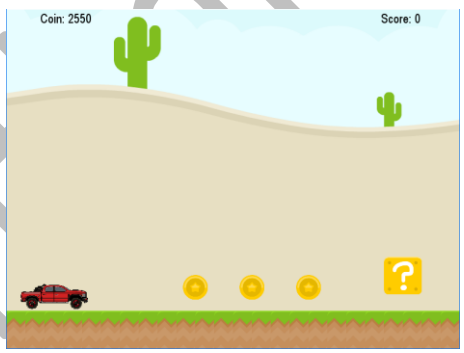
Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
4	Check the functioning of the Level Select button	Valid	The Level Select screen should be loaded	Click on the Level Select button	Expected outcome returned. Test successful
	 				
5	Check the navigation of the Settings button	Valid	The Settings screen should be loaded	Click on the Settings button	Expected outcome returned. Test successful
	 				
6	Check the functionality of the Vehicles button	Valid	The Vehicles screen should be loaded with the Car selected by default	Click on the Vehicles button	Expected outcome returned. Test successful
	 				

7	Check the functioning of the statistics button	Valid	The statistics screen should be loaded and show the player's current statistics.	Click on the statistics button	Expected outcome returned. Test successful
	 				
8	Check the navigation of the Leader Board screen	Valid	The Leader Board screen should be displayed showing the scores for Level 1 by default.	Click on the Leader Board button	Expected outcome returned. Test successful
	 				
9	Check that pressing buttons on the keyboard doesn't interact with the program	Invalid	The game should not respond in any way from the keyboard inputs.	Press keyboard buttons whilst in the main menu screen	When the space bar was pressed and the 'P' button was pressed, there was an error.
	Error Description When the space bar was pressed the game was trying to make the player jump however a player object had not been created.				
	Fixing the Problem I added a statement to check if the current scene is one of the level screens before the 'jump' function is called on the player object.				

The images show that the buttons correct highlight when the mouse cursor hovers over them. When the buttons were clicked at the edges of them, they correctly functioned. These were tests 12 and 13 in the test plan for this screen.

Testing the Level Select Screen

The test plan on page 52 was used for these tests.

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
10	Check the functioning of the Back button	Valid	The main menu screen should be loaded	Click on the Back Button	Expected outcome returned. Test successful
11	Check that the Level buttons correctly load their respective levels	Valid	The level associated to each level should be loaded when pressed	Press each Level button	Expected outcome returned. Test successful
	This shows the Level One button working. Tests were completed for all the other Level buttons as well.  				
12	Test if the buttons work when they are clicked at their edges	Valid Extreme	Game correctly executing the function associated with the buttons	Click on all the buttons at their edges	Expected outcome returned. Test successful
13	Check that pressing any buttons on the keyboard or clicking in places on the screen with no buttons doesn't interact with the program	Invalid	The game should not respond in any way from these inputs.	Press keyboard buttons and click on places on the screen with no buttons, whilst in the Level Select screen	Expected outcome returned. Test successful

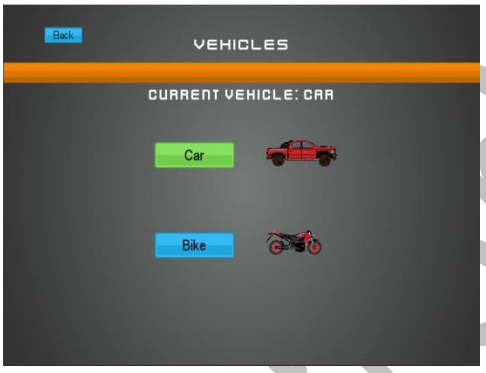
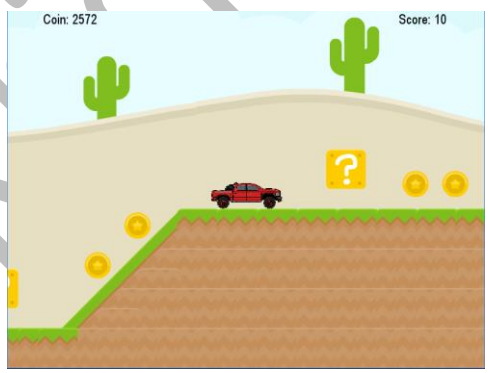
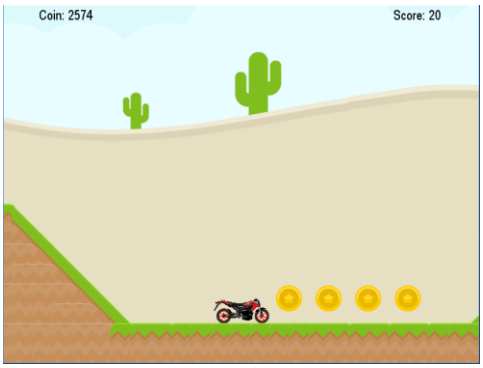
Testing the Settings Screen

The test plan on page 55 was used for these tests.

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
14	Check the functioning of the Back button	Valid	The main menu screen should be loaded	Click on the Back Button	Expected outcome returned. Test successful
15	Check that the View Start Screen button correctly functions	Valid	The level associated to each level should be loaded when pressed	Press each Level button	Expected outcome returned. Test successful
16	Make sure that the music correctly turns off when the No Music button is pressed	Valid	The music that was playing will fade out into silence	Click on the No Music button	Expected outcome returned. Test successful
17	Determine that the music correctly turns on when a track is chosen after the music was previously off.	Valid	The menu music will begin playing again and the level music will change to the chosen track.	First click on the No Music button. Then click on each music track button	Expected outcome returned. Test successful
18	Check that the music is correctly changed when a different music track is chosen	Valid	The music that is played in the level should be the rack which has been picked in the settings screen	Click on each music track and test that the correct music is played in the level	Expected outcome returned. Test successful
19	Check the navigation of the Instructions button.	Valid	The instructions screen should be loaded	Click on the Instructions button	Expected outcome returned. Test successful
20	On the Instructions Screen: Make sure that pressing a button on the keyboard or clicking the mouse returns to the settings screen.	Valid	The settings page will be loaded	Press keyboard buttons and click in various locations inside the game window	Expected outcome returned. Test successful
21	On the Instructions Screen: Attempt to press many buttons at the same time	Valid Extreme	The settings page should be loaded	Press multiple keys on the keyboard simultaneously whilst also clicking the mouse button	Expected outcome returned. Test successful
22	Make sure that all the difficulty buttons correctly function	Valid	Depending on the difficulty chosen, the difficulty of the questions being asked will correctly change	Click on each difficulty button	Expected outcome returned. Test successful
23	Check that all of the button correctly highlight when the mouse cursor is over them	Valid	The buttons should change their colour to their highlight colour when the mouse cursor is over them	Hover the mouse cursor over each button on the screen	Expected outcome returned. Test successful

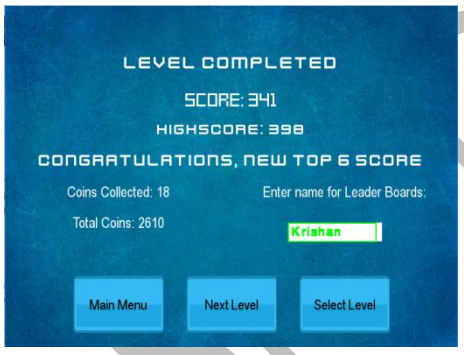
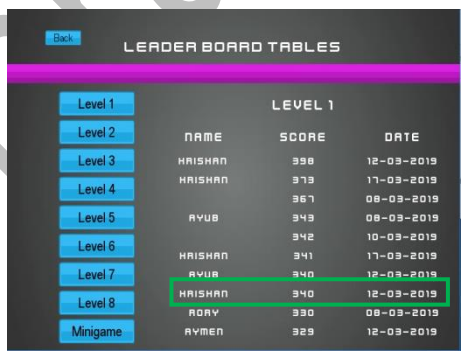
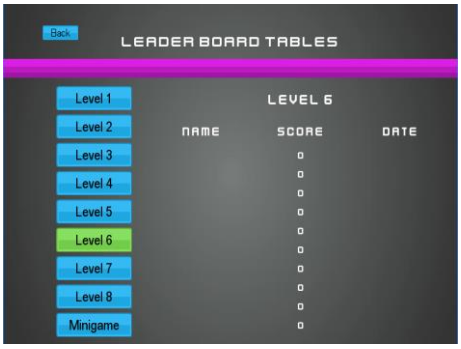
Testing the Vehicles Screen

This page was previously named the Shop screen but due to some features not being added to the game, this screen was changed to be more appropriately called the Vehicles screen. This was explained on page 169. The test plan on page 58 was used for these tests.

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
24	Check the functioning of the Back button	Valid	The main menu screen should be loaded	Click on the Back Button	Expected outcome returned. Test successful
25	Check that the Current Vehicle text changes when a different vehicle is chosen	Valid	The Current Vehicle text displayed on this screen should correctly change depending on the vehicle chosen	Click on the Car and Bike buttons	Expected outcome returned. Test successful
	 				
26	Make sure that the vehicle correctly changes to a Car when the Car button is pressed	Valid	The vehicle displayed in the levels should be a Car	Click on the Car button	Expected outcome returned. Test successful
	 				
27	Check that the vehicle correctly changes to a Bike when the Bike button is pressed	Valid	The vehicle displayed in the levels should be a Bike	Click on the Bike button	Expected outcome returned. (test image shown above) Test successful

Testing the Leader Board Screen

The test plan on page 63 was used for these tests.

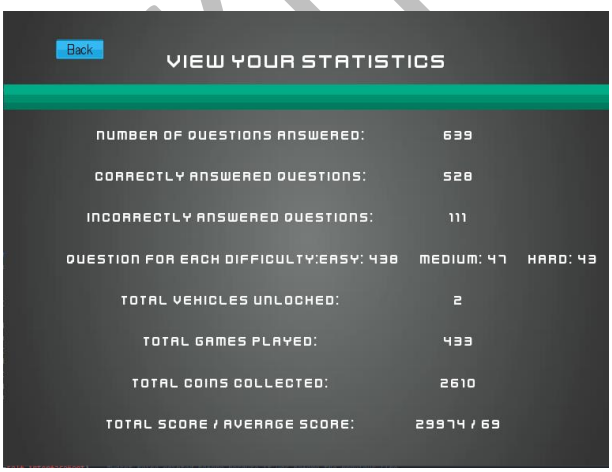
Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
28	Check the functioning of the Back button	Valid	The main menu screen should be loaded	Click on the Back Button	Expected outcome returned. Test successful
29	Check that the leader board information correctly updates	Valid	The newly achieved score will be added to the leader board in the correct location	Play a level and obtain a score in the top ten scores for that level	Expected outcome returned. Test successful
	   The first image shows the initial leader board scores for Level 1. I played this level and achieved a score of 341, which is the sixth highest score on the table. I entered my name into the input box and pressed enter. The third image shows the leader board table for Level 1 after the new score. As shown by the green box, the recently achieved score has been correctly inserted into the scores table in the correct position.				
30	Make sure that the Level 1 scores are displayed by default when this screen is launched	Valid	The level 1 leader board information will be displayed	Click on the Leader Board button from the Main Menu screen	Expected outcome returned. Test successful
31	Check that the Level buttons on the Leader Board screen correctly change the score information to that of the chosen level	Valid	The score data associated with each level should be displayed once it is clicked	Press each Level button in the Leader Board screen	Expected outcome returned. Test successful
	   Even though there are not any levels after 4, I have still created the buttons and files for the scores for those levels. This makes it easier to implement these levels in the future				

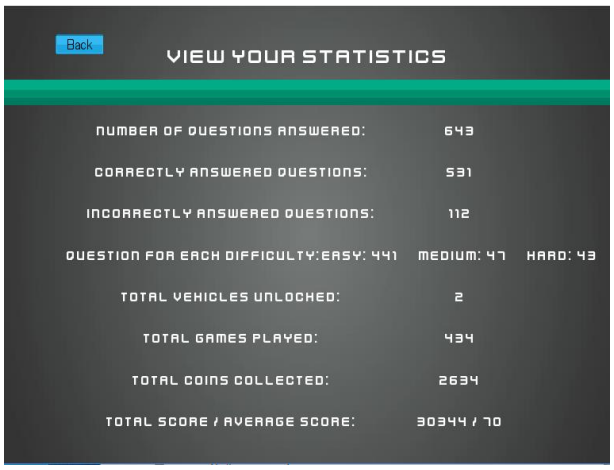
Testing the Statistics Screen

The test plan on page 66 was used for these tests.

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
32	Check the functioning of the Back button	Valid	The main menu screen should be loaded	Click on the Back Button	Expected outcome returned. Test successful
33	Check that the total games played correctly updates after a level is played	Valid	The total games played is incremented by one	Play a level	Expected outcome returned. Test successful
34	Make sure that the total questions answered correctly updates after a level is played	Valid	The total question answered increases by the number of questions answered in the level	Play a level and answer some questions in the level	Expected outcome returned. Test successful
35	Determine whether the correctly answered question information is correctly updated after a level is played	Valid	The values for the correctly and incorrectly answered questions, as well as the values in the question for each difficulty	Play a level and answer some questions correctly	Expected outcome returned. Test successful
36	Check that the total coins collected correctly updates after a level is played	Valid	The value for the coin total will increase depending on the number of coins collected in the played level	Play a level and collect some coins	Expected outcome returned. Test successful
37	Check that the total score and average score correctly updates after a level is played	Valid	The total score will increase, the average score will be recalculated using the new total score and new total games played	Play a level and achieve a score	Expected outcome returned. Test successful

On the left is the statistics screen before these tests were carried out. On the right is the statistics screen after the above tests were carried out. Level 2 was played were a score of 370 was achieved whilst collecting 24 coins and answering 4 questions (3 correct and 1 incorrect). The new statistics information has correctly acknowledged the question data and has increased the total games played by 1, as well as the total coins collected by 24. This information shows that the statistics screen is operating as expected.





Testing the Levels

The testing for levels will be conducted in accordance with the tasks set out by the success criteria. After these, any additional added features will be individually tested. The success criteria being tested are criteria numbers 1, 3, 4, 5, 8 and10.

Success Criteria 1

- Load the questions from the CSV file and store the required data.
  (i.e. store the questions with the difficulty selected)

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
38	Check that the questions have been correctly loaded into the game	Valid	All the questions in the CSV file will be loaded into the game and asked to the user player during the level.	Make a debug message to print the loaded version of the CSV file.	Expected outcome returned. Test successful

```
Question Data
[1, '2 + 2', '4', '3', '2', '6', '5', 'easy', 1, 'FALSE']
[2, '5 + 7', '12', '13', '25', '10', '15', 'easy', 1, 'FALSE']
[3, '15 + 17', '32', '30', '42', '23', '34', 'easy', 1, 'TRUE']
[4, '25 ◆ 5', '5', '1', '7', '11', '4', 'easy', 1, 'FALSE']
[5, '4 x 3', '12', '11', '15', '17', '8', 'easy', 1, 'FALSE']
[6, '(2 + 4) x 3', '18', '19', '16', '24', '21', 'easy', 1, 'TRUE']
[7, '98 - 7', '91', '89', '87', '93', '95', 'easy', 1, 'FALSE']
[8, '15 + 17', '32', '30', '42', '23', '34', 'easy', 1, 'FALSE']
[9, '49 ◆ 7', '7', '6', '11', '40', '9', 'easy', 1, 'FALSE']
[10, '21 + 9', '30', '27', '28', '35', '42', 'easy', 1, 'FALSE']
[11, '14 x 6', '104', '100', '98', '110', '92', 'medium', 1, 'FALSE']
[12, 'Find x when 40x = 2x + 19', '2', '1', '0', '3', '4', 'medium', 1, 'FALSE']
[13, 'Find x when 20x - 17x = 24', '8', '9', '11', '4', '13', 'medium', 1, 'FALSE']
[14, 'Find x when 14x/7 = 28-2x ', '7', '10', '9', '12', '5', 'medium', 1, 'TRUE']
[15, 'Find x when 27x = 3 - (5x - 2x)  ', '10', '15', '17', '9', '6', 'medium', 1, 'FALSE']
[16, 'Factorise x^2 + 4x + 4', '(x + 2)(x + 2)', '(x + 3)(x + 2)', '(x + 4)(x - 1)', '(x + 2)(x - 2)', '(x - 3)(x - 1)', 'medium', 1, 'FALSE']
[17, 'Factorise x^2 + 7x - 8', '(x + 8)(x - 1)', '(x + 8)(x + 1)', '(x + 4)(x - 2)', '(x + 4)(x + 3)', '(x - 2)(x - 6)', 'medium', 1, 'FALSE']
[18, 'Factorise x^2 -5x + 6', '(x - 3)(x - 2)', '(x + 3)(x + 2)', '(x + 6)(x - 1)', '(x + 5)(x + 3)', '(x + 6)(x + 5)', 'medium', 1, 'FALSE']
[19, 'Factorise 2x^2 - x - 1', '(2x + 1)(x - 1)', '(2x + 3)(x + 2)', '(x + 1)(x - 1)', '(x + 1)(2x - 2)', '(2x - 1)(x + 3)', 'medium', 1, 'FALSE']
[20, 'Complete the Square:  x^2 + 8x - 5', '(x + 4)^2 - 21', '(x + 4)^2 - 5 ', '(x - 4)^2 + 21', '(x + 8)^2 - 5', '(x - 8)^2 + 3', 'medium', 1, 'FALSE']
[21, 'Complete the Square:  x^2 + 10x + 12', '(x + 5)^2 - 13', '(x + 5)^2 + 12', '(x - 5)^2 + 13', '(x + 10)^2 + 12', '(x - 4)^2 + 7', 'medium', 1, 'TRUE']
[22, 'Complete the Square:  2x^2 + 8x - 5', '2(x + 2)^2- 13', '(x + 4)^2 - 21', '2(x + 8)^2 - 5', '2(x + 2)^2 + 13', '2(x + 4)^2 - 15', 'hard', 1, 'FALSE']
[23, 'Complete the Square:  2x^2 + 12x - 7', '2(x + 3)^2- 25', '2(x + 3)^2 - 7', '2(x + 6)^2 - 25', '2(x + 12)^2 - 7', '2(x - 6)^2 + 9', 'hard', 1, 'FALSE']
[24, 'Complete the Square:  3x^2 + 24x + 5', '3(x + 4)^2 - 43', '3(x + 8)^2 - 40', '2(x + 12)^2 - 7', '3(x - 8)^2 - 17', '3(x - 4)^2 - 11', 'hard', 1, 'FALSE']
[25, 'Factorise 4x^2 + 17x + 4', '(4x + 1)(x + 4)', '(4x + 3)(x + 2)', '(4x + 3)(x - 1)', '(2x - 1)(2x + 4)', '(2x - 3)(2x - 1)', 'hard', 1, 'FALSE']
[26, 'Factorise 3x^2 + 5x + 4', '(3x + 2)(x + 2)', '(3x + 3)(x + 2)', '(3x - 2)(x - 1)', '(x + 3)(x - 5)', '(3x - 4)(x + 5)', 'hard', 1, 'FALSE']
[27, 'Factorise 6x^2 + 7x + 2', '(3x + 2)(2x + 1)', '(6x + 3)(x + 2)', '(6x + 3)(x - 2)', '(3x + 2)(2x - 1)', '(3x - 4)(2x + 3)', 'hard', 1, 'TRUE']
[28, 'Find dy/dx when y = 3x^2 +7x', '6x+7', '4x+9', '5x^2+7x+9', '3x', '17', 'hard', 1, 'FALSE']
[29, 'Find dy/dx when y = 7x^3 - 5x^2+17x -24', '21x^2 - 10x + 17', '21x^2 + 10x + 24', '7x^3 - 5x + 17', '3x^2 -10X - 7', '5x - 24', 'hard', 1, 'FALSE']
[30, 'Find dy/dx when y = sin(2x) + cos(x)', '2cos(2x) - sin(x)', '2sin(2x) - sin(x)', 'cos(2x) + sin(x)', 'cos(2x) - sin(x)', '2sin(4x) + cos(x)', 'hard', 1, 'FALSE']
[31, 'Find dy/dx when y = tan(3x) + 7x^2 + 84x', '3sec(3x)^2 + 14x + 84 ', '3sec(3x)^2 + 84x + 14 ', '3cos(3x)^2 + 7x + 14 ', 'sin(3x)^2 + 7x + 42 ', 'tan(3x)^2 + 14x + 84 ', 'hard', 1, 'TRUE']
[32, '3 x 3 x 3 ', '27', '18', '24', '29', '31', 'easy', 1, 'FALSE']
[33, 'Find x when 8x - 2x = 3', '2', '1', '5', '4', '3', 'easy', 1, 'FALSE']
[34, '60 ◆ 12 ', '5', '4', '6', '10', '8', 'easy', 1, 'FALSE']
[35, '85 ◆ 5 ', '17', '15', '12', '22', '9', 'easy', 1, 'FALSE']
[36, '5 x 5 x 2', '50', '110', '15', '12', '25', 'easy', 1, 'FALSE']
[37, '4 x 4 x 4', '64', '16', '44', '50', '20', 'easy', 1, 'TRUE']
[38, 'Find x when (10x + 5x) / 3 = 25', '5', '6', '15', '3', '22', 'easy', 1, 'TRUE']
[39, '7 x 8 x 2', '4 x 4 x 7', '7 x 8 x 4', '7 x 1 x 2 x 3', '4 x 4 x 4 x 2 ', '7 x 5 x 1 x 2', 'medium', 1, 'TRUE']
[40, '64 is equal to ', '4 x 4 x (2 + 2)', '(13 + 1) x 4 ', '(20 - 3) x 4 ', '(34 + 2) x 2', '(3 x 4 x 5) + 3', 'medium', 1, 'TRUE']
[41, '71 is equal to', '(7 x 5 x 2) + 1', '(6 + 1) x (5 + 5)', '(13 x 3) - 3 ', '(6 x 5) + (6 x 7)', '(7 x 8) + 18', 'medium', 1, 'TRUE']
```

The CSV file used for the questions was shown on page 148 under the heading "Changing question difficulty" and "Adding questions to the CSV file". I used a 'print' statement to print each line of the loaded CSV file to the console. The output shows that the CSV file has been correctly loaded into the game.

```
self.questionData.append(temp)
print(temp)
```

I added a message to print the 'QuestionID' values which have been chosen before a question from these values is randomly selected. I played a level using Easy difficulty

```
print("These are the QuestionID: {}".format(self.questionID))
```

```
These are the QuestionID: [3, 6, 37, 38]
```

The image above shows the values for the QuestionID's for a major question and the image below shows the QuestionID values for all easy questions. These values match up with the values in the CSV file, showing that this aspect of the program is correctly functioning.

```
These are the QuestionID: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 32, 33, 34, 35, 36, 37, 38]
```

Success Criteria 3

- Level is loaded and the player's car and in-game items are spawned.

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
39	Check that the car object is correctly loaded into the game	Valid	The car object will be spawned into the level in the correct location	Print the coordinates of the player and play a level.	Expected outcome returned. Test successful

I added 'print' statements to the block of code creating instances of game object classes inside the 'level1' function to print the coordinates where these objects are spawned.

```
if tileObject.name == "player":
    self.game.player = Player(self.game, tileObject.x, tileObject.y)
    print("Player X Coordinate: {}".format(tileObject.x))
    print("Player Y Coordinate: {}".format(tileObject.y))
```

Name	player
Type	
Visible	☑
X	80.00
Y	464.00
Width	32.00
Height	32.00

When this level was played in the game, the log message to the console correctly printed and showed that the player object was being spawned in at the coordinate (80, 464). I analysed the 'Tiled' file for this level and found that the Player object was places at the coordinates (80, 464). This shows that the game is correctly spawning the player object in the required location.

```
Player X Coordinate: 80.0
Player Y Coordinate: 464.0
```

40	Make sure that all in-game objects are loaded in their correct positions	Valid	All in-game objects will be correctly loaded in their respective coordinates	Make a log of the coordinates of all the in-game objects to the console. Play a level to generate this information.	Expected outcome returned. Test successful

```
print("Coin X Coordinate: {} Coin Y Coordinate {}".format(tileObject.x, tileObject.y))
print("Question X Coordinate: {} Question Y Coordinate {}".format(tileObject.x, tileObject.y))
print("Major Question X Coordinate: {} Major Question Y Coordinate {}".format(tileObject.x, tileObject.y))
```

Similarly, I added several 'print' statements to display the coordinates of the spawned coins and question items in the first level. I chose not to check that wall and track objects as it was clear from the image of the map and interaction between the player and the map that they were being correctly spawned. The message printed to the console shows the coordinates of all the coins and question items in the level. I checked these values against the coordinates shown in the Tiled program and found that they all matched. Below, the coordinate of one object of each type of are shown.

```
Coin X Coordinate: 320.0 Coin Y Coordinate 472.0
Question X Coordinate: 672.0 Question Y Coordinate 464.0
Coin X Coordinate: 512.0 Coin Y Coordinate 472.0
Coin X Coordinate: 416.0 Coin Y Coordinate 472.0
Coin X Coordinate: 832.0 Coin Y Coordinate 411.0
Coin X Coordinate: 896.0 Coin Y Coordinate 347.0
Coin X Coordinate: 1024.0 Coin Y Coordinate 280.0
Question X Coordinate: 1232.0 Question Y Coordinate 272.0
Coin X Coordinate: 1344.0 Coin Y Coordinate 280.0
Coin X Coordinate: 1408.0 Coin Y Coordinate 280.0
Coin X Coordinate: 1792.0 Coin Y Coordinate 472.0
Coin X Coordinate: 1856.0 Coin Y Coordinate 472.0
Coin X Coordinate: 1920.0 Coin Y Coordinate 472.0
Coin X Coordinate: 1984.0 Coin Y Coordinate 472.0
Coin X Coordinate: 2048.0 Coin Y Coordinate 472.0
Coin X Coordinate: 2112.0 Coin Y Coordinate 472.0
Coin X Coordinate: 2304.0 Coin Y Coordinate 384.0
Coin X Coordinate: 2368.0 Coin Y Coordinate 320.0
Coin X Coordinate: 2656.0 Coin Y Coordinate 120.0
Coin X Coordinate: 2720.0 Coin Y Coordinate 120.0
Question X Coordinate: 2448.0 Question Y Coordinate 224.0
Major Question X Coordinate: 2880.0 Major Question Y Coordinate 112.0
```

Coin Object

X	320.00
Y	472.00

Question Object

X	1,232.00
Y	272.00

Major Question Object

X	2,880.00
Y	112.00

Success Criteria 4

- Player provides input to control the car. The appropriate action is performed on the car. The new position of the car is rendered to the screen.

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
41	Check that the provided player inputs manoeuvres the player object as expected	Valid	The car object's position will change in accordance to the inputs provided by the player	Press the "A", "S", "D" keys and "Left", "Down" and "Right" arrow keys inside a level. Also press the Space Bar to check if the player jumps	Expected outcome returned. Test successful

I added several 'print' statements to print messages to the console when the keyboard inputs were pressed. The messages for the movement keys were added to the 'Player' class's 'update' function and for the Space bar, the message was added in the 'events' function.

```
if event.key == pg.K_SPACE:
    if self.sceneMan.currentScene in LEVEL_SCREENS:
        self.player.jump()
        print("Pressed Space Bar")
```

```
if keys[pg.K_RIGHT]:
    print("Pressed Right Arrow Key")
else:
    print("Pressed D Key")
```

```
if keys[pg.K_DOWN]:
    print("Pressed Down Arrow Key")
else:
    print("Pressed S Key")
```

```
if keys[pg.K_LEFT]:
    print("Pressed Left Arrow Key")
else:
    print("Pressed A Key")
```

I ran the game and played a level. Upon analysing the console logs, I found that the expected messages for the key presses had been correctly printed. This means that the program is correctly registering all of the movement inputs provided by the player. Additionally, the game was responding to the inputs provided and was moving the player object in the direction provided. A new image of the car was drawn in the new location, meaning that as required by this success criteria, the appropriate action is being performed on the car. Additionally, the new position of the car is being rendered to the screen in the form of a new frame.

```
Pressed D Key
Pressed Space Bar
```

```
Pressed S Key
Pressed A Key
```

```
Pressed Right Arrow Key
Pressed Left Arrow Key
```

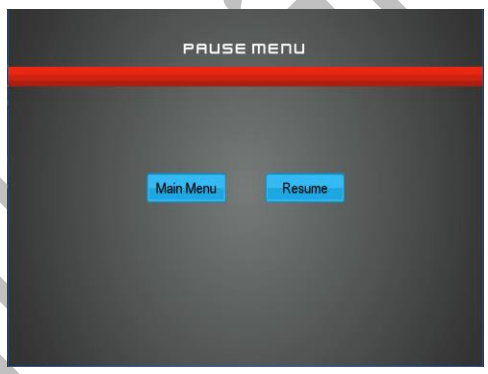
```
Pressed Left Arrow Key
Pressed Down Arrow Key
```

42	Press multiple valid, control buttons at the same time	Valid Extreme	The key that was pressed the last will be the motion which is applied to the car. This is because there will be a small delay between the key presses.	Press multiple, valid control keys at the same time	Expected outcome returned. Test successful
43	Make sure that pressing any other keyboard buttons doesn't interact with the program in any way.	Invalid	The game has been programmed to only look for certain key pressed. Other key presses will be detected by the 'pg.events.get' function however, the game will not acknowledge these.	Whilst in a level, press buttons which have not been assigned a control for the player	Expected outcome returned. Test successful
44	Press multiple invalid buttons at the same time	Invalid Extreme	The program will not interpret these key presses and these key presses will simply be ignored.	Press multiple invalid keys at the same time	Expected outcome returned. Test successful

Success Criteria 5

- Temporarily pause the game and ask the player a question at specific stages in a level.
- Initiate a countdown depending on the difficulty of the question.
- Wait for the player's response and determine their result.
- Resume the game (tests for each bullet point are on separate pages)

<table>
<tr><th>Test no.</th><th>Test Description</th><th>Test Type</th><th>Expected Outcome</th><th>Test Data</th><th>Actual Outcome</th></tr>
<tr><td rowspan="2">45</td><td>Check that the player can pause the level by pressing the pause button</td><td>Valid</td><td>The pause screen will be displayed once the pause button is pressed</td><td>Press the “P” button</td><td>Expected outcome returned.

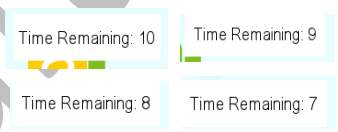
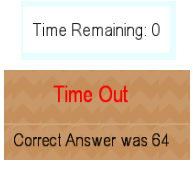
Test successful</td></tr>
<tr><td colspan="5">
<pre>print("P button pressed -- Paused")</pre>
<pre>P button pressed -- Paused</pre>


When the “P” button is pressed, the game correctly displays the Pause menu. A print statement was added to see if the key press Is detected. The message on the console shows that this was successful. The level can be resumed by pressing the “P” button again, or by pressing the resume button.
</td></tr>
<tr><td rowspan="2">46</td><td>Make sure that when the player collides with a Question object, the game pauses to ask a question</td><td>Valid</td><td>The game will pause to ask the question</td><td>Play a level and collide with a Question object.</td><td>Expected outcome returned.

Test successful</td></tr>
<tr><td colspan="5">
<pre>if hitQuestion:
    print("Collided with Question Object")</pre>
<pre>Collided with Question Object</pre>

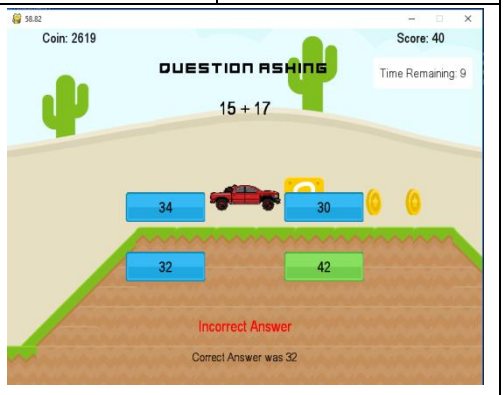

I added a statement to print when there was a collision between the player and a question object. This statement was printed to the console when I collide with a question object in a level, showing that this is functioning as expected. The screenshot shows that the rest of the game is paused when a question is asked.
</td></tr>
<tr><td>47</td><td>Check that a question is correctly asked to the player after colliding with a Question object</td><td>Valid</td><td>When the player collides with a Question object, they will be asked a question</td><td>Play a level and collide with a Question object.</td><td>Expected outcome returned.

Test successful</td></tr>
</table>

48	Check that a timer is initiated and shown on the screen once the player has been asked the question	Valid	The timer will be ready and shown to the player as soon as the question is proposed	Play a level and collide with a Question object.	Expected outcome returned. Test successful
	This shows that the timer that is displayed on the question surface. The black text on the white background specifically for the timer make the time easier to see and read. 				
49	Make sure that the timer is correctly counting down while the player has not answered the question	Valid	The timer will begin counting down as soon as the question has been asked and one second has passed.	Play a level and collide with a Question object.	Expected outcome returned. Test successful
	These images show that the timer is correctly counting down whilst the player has not answered the question. 				
50	Determine whether the timer counts down from the correct time for each difficulty question	Valid	The timer will begin at a different starting time for each difficulty. The harder the difficulty, the more time will be given.	Play a level and collide with a Question object. Change the question difficulty and repeat	Expected outcome returned. Test successful
	These images show the timer at the start of each question for each difficulty type. The orders of the difficulties are easy, medium and hard respectively. The expected times are 10, 15 and 20 seconds respectively and that is what is present in the game. 				
51	Check that the timer stops counting after the player has answered the question	Valid	When the timer reaches zero it should stop counting down and continue to display zero until the question ends.	Play a level and collide with a Question object.	Expected outcome returned. Test successful
	This image shows that the timer correctly displays zero once the timer has reached this. It doesn't continue to count into negative numbers. This number is displayed for a further 1500 milliseconds (as programmed) and then the timer is removed from the screen. 				
52	Make sure that the question is failed if the user runs out of time	Valid	When the timer reaches zero, a message stating that the time has will be displayed. The question should then end as the player has run out of time.	Play a level, collide with a Question object and let the timer count down to zero	Expected outcome returned. Test successful
	These images show that when the timer reaches zero, the "Time Out" message is correctly displayed at the bottom of the screen. The correct answer is also shown to the player. After 1500 milliseconds, this message, the question and the timer is correctly removed from the screen and the player regains control of the car. 				

53	Check that the player is able to respond to the asked question	Valid	After the question has been asked to the player, they will be able to choose one of the answers displayed on the screen.	Play a level, collide with a Question object and click on an answer	Expected outcome returned. When a button is clicked, the action is correctly recognised. Test successful
54	Confirm that the player's chosen answer is correctly interpreted.	Valid	The game should correctly map the questions shown on the screen to the answers they present.	Play a level, collide with a Question object and click on an answer	Expected outcome returned. Test successful

	`print("Selected Answer: {}".format(self.game.selectedAns))` I added a 'print' statement to print the answer that is chosen by the player. I then played the game and answered a question correctly (shown for text number 55). The console message shows that the answer was detected with tag 2 (which means that it is the correct answer). 50 is shown as the selected answer. This is chosen button therefore, the game correctly recognised the player's response
	`button.tag 2`
	`Selected Answer: 50`
	`CORRECT ANSWER CHOSEN: 50`				

55	Make sure that the correct message is displayed for the answer that is picked.	Valid	If the correct answer is picked, "Correct Answer" will be displayed. Otherwise, if an incorrect answer is chosen "Incorrect Answer", with the correct answer will be shown	Play a level, collide with a Question object and click on a correct and then incorrect answer	Expected outcome returned. Test successful
	As shown by the images, when a correct answer is selected, the text "Correct Answer" is displayed in green towards the bottom of the screen. Similarly, when an incorrect answer is chosen, the text "Incorrect Answer" in red, alongside the correct answer, is displayed	 			
56	Check that the question cannot be avoided by pressing keys on the keyboard or clicking on parts of the screen with no buttons	Invalid	The game should not respond in any way to any inputs other than when one of the four answers is clicked	Press keyboard buttons and click on places on the screen with no buttons while a question is being asked	When the "P" button was pressed the pause menu loaded. This should not happen.
	<u>Error Description</u> As the code to pause the game to ask the question uses the same implementation used for the pause menu, when the "P "button was pressed, the question stopped being asked. This is because the "paused" variable got set to False however, there was no answer in the "selectedAns" variable. This also made the button on the pause screen not function correctly.				

	<u>Fixing the Problem</u>
	I added a statement in the 'events' function to check if question is being asked before loading the pause screen. If a question is being asked, the 'askQuestion' variable will be True, so the code to load the pause menu is only executed when this variable is False. This solved the problem and made the pause menu buttons work correctly again.
	`if event.key == pg.K_p:`
	`  if self.sceneMan.currentScene in LEVEL_SCREENS:`
	`  if not self.askQuestion:`				

57	Check that the game correctly pauses for 1500 milliseconds after the player has answered the question	Valid	The game will pause for 1500 milliseconds (1.5 seconds) after the player has answered the question	Play a level, collide with a Question object and click on an answer	Expected outcome returned. Test successful
58	Make sure that the game is correctly resumed once the question has been asked	Valid	After the delay allowing the player to read the on-screen message, the player will regain control of the car.	Play a level, collide with a Question object and click on an answer	Expected outcome returned. Test successful
59	Check that the vehicle object can be correctly controlled once the game has resumed	Valid	The vehicle object should be fully controllable as the 'paused' variable will no longer be True	Play a level, collide with a Question object and click on an answer. Then attempt to move the vehicle	Expected outcome returned. Test successful
60	Make sure that during the delay, pressing any buttons on the keyboard and clicking on the screen doesn't cause an error.	Invalid	The game should not respond in any way to any inputs during the delay. This is because these inputs wouldn't be checked and processed during this time. Despite this, these inputs will be ignored even once the delay finishes.	Play a level, collide with a Question object click on an answer. Then during the delay, press keyboard buttons and click on places on the screen.	Expected outcome returned. Test successful

Success Criteria 8

- The score for the level is determined from how quickly the level is completed. Score will increase if coins on the level are collected.

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
61	Determine that the score gained for answering a question depends on the time taken to answer it.	Valid	The dependency does exist therefore, the quicker a question is answered, the higher the score for that question is.	Play a level, collide with a Question object and click on a correct answer.	Expected outcome returned. Test successful

```
print("The time take to answer that question was: ", self.timeTaken)
print("The random number chosen is: {}".format(multiplier))
print("The score given for this question is: {}".format(scoreIncrease))
```

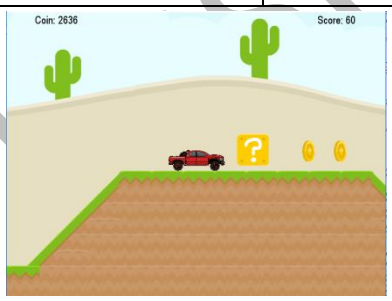
I added three 'print' statements to the 'calculateScore' function to show the time taken to answer the question, the random multiplier chosen for the question and, the score given for that question. As explained in the Calculating the Score section on page 121, a random number is chosen to be used in the calculation of the score to make it more random and varied. This reduces the likelihood of two players achieving the same score on a level, thus making the game more competitive.

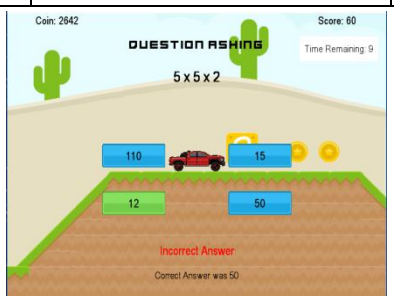
I played the game and answered two questions. The messages in the console show that a higher score is given for the question which was answered in 2 seconds, compared to the question answered in 3 seconds. However, the random multiplier chosen was the same for both questions. Despite this, the number chosen for the multiplier is more likely to be a lower number due to the abundance of them as shown on page 121. Additionally, if a longer time is taken to answer this question, this value will be more dominant in the score calculation and will bring the score down.

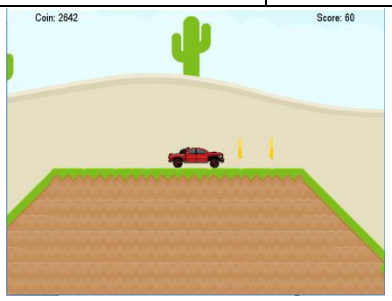
```
The time take to answer that question was:  3
The random number chosen is: 8
The score given for this question is: 56
```

```
The time take to answer that question was:  2
The random number chosen is: 8
The score given for this question is: 64
```

62	Check that the score is not increased if a question is answered incorrectly.	Valid	The score should only be increased if the question is answered correctly. Therefore, the score should not be increased for an incorrect answer	Play a level, collide with a Question object and click on an incorrect answer.	Expected outcome returned. Test successful

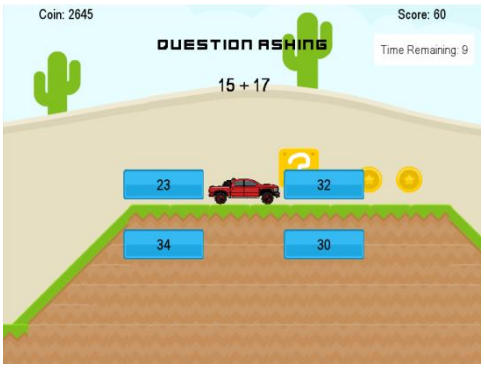






These images show the score before, during and after the question asking phase. The score before the question was 60. After answrerin answering the question incorrectly, the score remained at 60, showing that the game has correctly identified the incorrect answer and not increased the score for it.

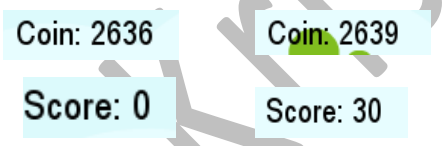
63	Check that score is not increased if a question is not answered because the time has finished	Valid	The score should not be increased	Play a level, collide with a Question object in a Level and let the timer count down to zero	Expected outcome returned. Test successful

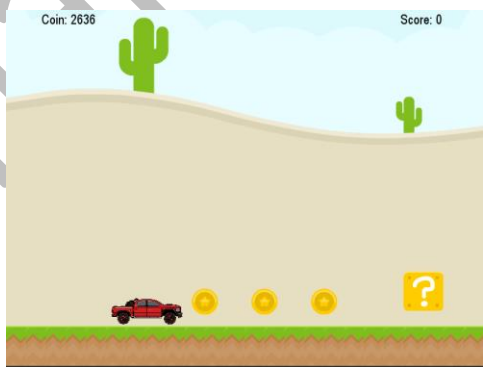


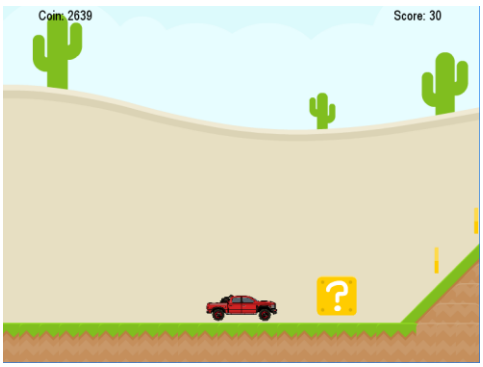
The score before the question was asked is 60. The timer the counted down to zero and after the "Time Out" message, the score remained at 60. This shows that the game has successfully not increased the score when a question has not been answered because of the allocated time finishing.

64	Make sure that the score increases when coins are collected	Valid	The score should increase by a pre-programmed 10 every time a coin is collected	Play a level and collide with coin objects.	Expected outcome returned. Test successful

These images show the start level 1. At the start of the level, there were a total of 2336 coins and a score of 0.







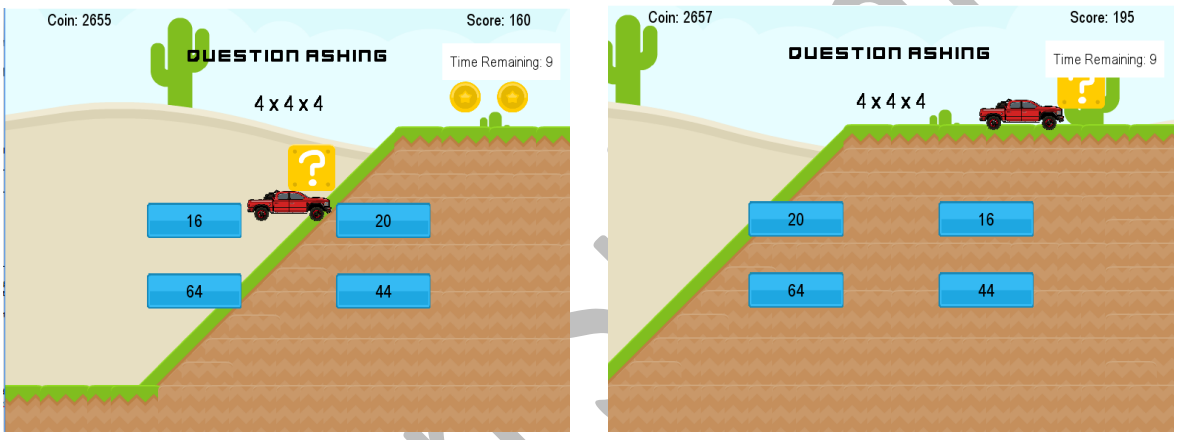
After collecting the three coins shown on the screen, the coin total correctly increased to 2639 and the score increased to 30. This shows that the score increases when coins are collected.

Success Criteria 10

1. Question asked will be random and the choice of answers will be random. (i.e the correct answer will not always be A)

Test no.	Test Description	Test Type	Expected Outcome	Test Data	Actual Outcome
65	Make sure that even if the same question is asked, the answers given are not in the same order	Valid	Even if the same question is chosen in the same level (which could happen due to the random process), the order of the answers will be randomised again so should be in different places.	Play the level and collide with Question objects	Expected outcome returned. Most of the time the order of the answers are different. Test successful

As the process of generating the order of the answers is random, there is a small probability of the answers being in the same order as the previous time.



As shown by these images, when the same question was asked again, the order of the answers was different. On top of this, as there are four incorrect answers in the questions CSV file, there is also likely to be a different answer in the selection. The possible inclusion of a different answer each time will make it less likely for the player to simply memorise the answer to each question because of its position.

66	Check that the questions are chosen at random	Valid	A random question is selected from all of the questions of the chosen difficulty each time a question is needed.	Play a level and collide with a Question object.	Expected outcome returned. Test successful

```
print("These are the QuestionID: {}".format(self.questionID))
print("This is the chosen QuestionID: {}".format(questionNumber))
```

I added two 'print statements to display the value of the questions ID's that are chosen from and also the question ID which is chosen.

```
These are the QuestionID: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 32, 33, 34, 35, 36, 37, 38]
This is the chosen QuestionID: 8
```

```
These are the QuestionID: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 32, 33, 34, 35, 36, 37, 38]
This is the chosen QuestionID: 9
```

The output showed that for the easy questions, the QuestionID reset each time and but the choice of question was different. The inclusion of the major questions meant that the QuestionID array is re-calculated each time a question is needed therefore, any previous removals from this array (such as the removal of question IDs of asked questions) will be ignored. This is why the same questions are being asked again.

67	Check that the answers for the questions are chosen at random	Valid	The correct answer, alongside 3 of the 4 possible incorrect answers will be chosen in a random order.	Play a level and collide with a Question object.	Expected outcome returned. Test successful

```
print("Chosen answers: ", chosenAns)
random.shuffle(chosenAns)
print("Chosen answers after randomising order: ", chosenAns)
```

```
Chosen answers:  [2, 4, 5, 3]
Chosen answers after randomising order:  [3, 5, 2, 4]
```

```
Chosen answers:  [2, 6, 4, 3]
Chosen answers after randomising order:  [6, 2, 4, 3]
```

I added two 'print' statements to display the contents of the 'chosenAns' array before and after the order has been randomised. The output to the console shows that the 'random.shuffle' function is correctly working as the order of the array has been changed as required. This means that the order the answers are displayed to the screen is also changed.

68	Make sure that an error is not caused by no questions being available for selection.	Valid	As the 'questionID' array is re-calculated each time a question is asked, there will always be the maximum number of questions to choose from.	Play a level and collide with a Question object. Check that no error is caused during this process.	Expected outcome returned. Test successful

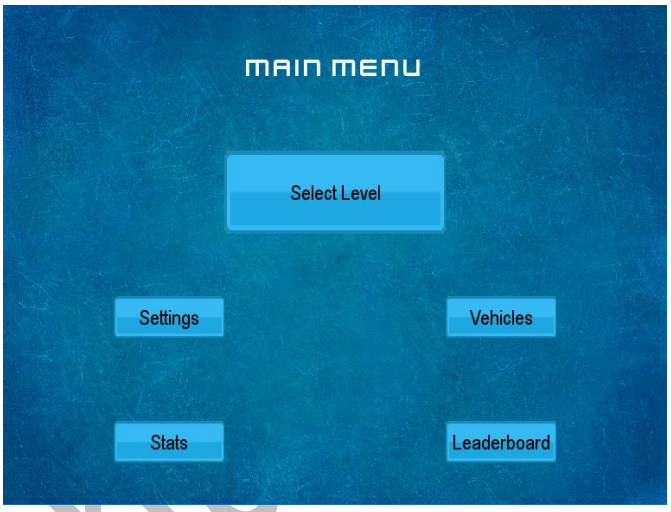
Before the major question functionality was added, the 'remove' list operation was being used to remove the question ID value from the 'questionID' array once the question has been asked. This meant that eventually, there will be no question ID values left in the array. When this inevitably happened, an error was caused.

```
    raise IndexError('Cannot choose from an empty sequence') from None
IndexError: Cannot choose from an empty sequence
```

The error was due to the attempt to extract data from an empty array. This problem was solved by recalculating the 'questionID' array for each question, meaning that the 'questionID' array will always be at its maximum capacity.

Testing the Usability Features

As mentioned on pages 66, the final game needs to include certain usability features to allow the game to be used by all possible users. These usability features include:



1. Having large buttons to navigate the game screens

As shown in the testing for the menu pages and by the screen shot on the right, the buttons on the menu pages are very large and thus easy to press. On the main menu screen, the button which will be clicked the most, the "Select Level" button, is the largest and so easier to press. This allows seamless manoeuver ability through the menu screens for the user.

2. There is large text depicting the purpose of each button

For each button, there is text showing the purpose of the button. This text usually represents the screen that will be loaded as a result of clicking the button. All of the text is at the maximum text size for the size of the button. This larger size makes them easier to read and interpret.

3. On screen text changes to reflect changes

When the user is selecting the question difficulty and choosing a vehicle, there is text on the screen showing the currently chosen values. This text will be updated if the associated value changes. This makes it easier for the user to identify when the buttons are pressed as the text shown will change appropriately.

4. Contrasting colours are used

The text colour for all of the buttons is black and the default "solid" colour for each button is blue. There is a strong contrast between theses colours therefore, it will be easier for the user to read the text. This is because they will need to strain less to make out the text against the background colour.

Additionally, the colour for when the cursor is over the mouse (the "highlighted" colour) is light green and the "solid" colour is blue. Once again, the contrast between these colours makes it easier for the user to identify when the mouse cursor is over a button.

5. Button colour changes when cursor is over the button

As shown by the pictures above, when the mouse cursor is over the button's area, the colour of the button changes to the "highlighted" colour. This assists users in identifying the interactive region for the button. Also, it will be especially helpful for the visually impaired as otherwise, it will be difficult to accurately locate the mouse cursor in the game

6. Bright, vivid colours are used

As shown by the pictures on this page and the screenshots during the testing, the colours used on all the game screens and during the level are bright, vivid and clear. This makes it easier for the user to differentiate objects on the screen whilst also making these objects easier to see.

7. Strong black border around buttons

There is not a black border around the button edges however there is a clear contrast between the background colour of the screen and the colour of the button. This makes it easier to identify the edges of the buttons for visually impaired users.

8. Sharper characters and images

When the text surface is rendered in the 'drawText' function, the optional anti-aliasing parameter is used. This means that as required, anti-alkalising is used for all text drawn in the game. This makes the text sharper and more detailed for users with vision difficulties.

```
textSurface = font.render(text, True, colour)
```

9. Images are used beside buttons

Images are used besides buttons to make it simpler for the user to recognise and understand the purpose of the button. This is used in the 'Vehicles' screen for the 'Car' and 'Bike' buttons.

10. Music is played throughout the game

Music tracks have been added for while the user is in the menu screen, the level and in the pause menu. The addition of this audio enhances the user's experience whilst navigating these screens. However, if the user doesn't want to listen to music, there is an option in the 'settings' screen to turn off the music. This fades out the currently playing music and disables all sources of music. This feature to provide the user the option to disable music means that the game attracts a larger and wider audience.

Testing Review

All functionality of the program has now been thoroughly tested. At each major stage during the testing, invalid data and extreme invalid data has been used to test and verify the robustness and versatility of the program. Most of the time, the game had already been programmed to avoid the invalid data however, in the cases where an error was caused, the error was correctly diagnosed, explained and fixed. All the success criteria which have been fully met have been tested. The partially and not met success criteria will be explained in the evaluation. Finally, usability testing has been performed to identify all the beneficial features of the game to allow it to be played by all users.

Evaluation

The game has now been completed and fully and thoroughly tested. The success of the project in terms of whether the aims that were set out were achieved will be analysed in this section. First, the success criteria will be evaluated to examine which criteria were fully, partially or not met, and their appropriate reasons.

Meeting the Success Criteria

The purpose of the success criteria was to set out a series of aims to structure the development of the game. The criteria towards the end are more desirable features than essential ones. This means that not meeting these criteria doesn't have a huge effect on the game compared to if one of the essential, earlier criteria were not met.

1. Load the questions from the CSV file and store the required data. (i.e. store the questions with the difficulty selected)

This success criteria has been fully met. Test Number: 38

A method to load the CSV file containing the question data has successfully and correctly been implemented into the game. The data inside this CSV file is extracted and stored locally in the game. This local file is then accessed each time a question is required. The local file is stored as a two-dimensional array. This allows the columns of the CSV file to be easily accessed using their indexes.

Test number 38 showed the contents of the CSV file after it has been imported into the game. This means that, as required by the success criteria, the questions have been loaded and the required data has been stored. This test also shows that the question numbers of the questions with the difficulty selected are chosen and stored in a separate array. This fulfils the second part of the success criteria.

2. Create an arrangement of screens which can be linked to form a menu system. All the settings and features should be accessible from here.

This success criteria has been fully met. Test Numbers: 1 to 37

A system for creating and managing the menu screens has been implemented. This is achieved through the Scene Manager class. This class holds all the functions for all the screens in the game. It detects the clicks of buttons and loads the appropriate screen. It records the previous screens so that they can be resumed if required, for example when the pause screen is pressed. Buttons with the same purpose, i.e. buttons which lead to the same game screen, have been given the same tag. This allows the Scene Manager class to simply call the function responsible for the required screen instead of having to spend unnecessary computation time and resources figuring out the screen required using another method.

Test numbers 1 through 37 have tested the whole Menu system. Each screen in the game has been fully tested using the test plans created for them in the User Interface Design section on page 48 to 66. The buttons on all screens have been tested and are fully functional. This evidence shows that criteria has been fully met.

As shown by the tests, all of the game screens are accessible from the main menu screen. This shows that all the screens have been successfully linked together. Additionally, there is a back button on each screen to navigate the user to the previous screen. This shows that the player is able to access all the settings and

game features as they can navigate to the main menu screen, and then navigate to the required screen from there. Hence, the second part of this success criteria has been fulfilled.

3. Level is loaded and the player's car and in-game items are spawned.

This success criteria has been fully met. Test Numbers: 11, 39 to 40

As explained for Success Criteria 2, the Scene Manger object is responsible for managing the game screens, including the level screens. As shown by test number 11, the levels are correctly loaded when a level button inside the Level Select screen is pressed.

Classes have been used for different game sprites. Instances of these classes are then created when the level is loaded. As shown by test number 39, an instance of the player class is correctly created and the image for the vehicle is correctly loaded in the required location. Test number 40 shows that all other in-game sprites, such as the coins and question boxes, are also correctly loaded in their required positons around the map.

Hence this success criteria has been fully fulfilled.

4. Player provides input to control the car. The appropriate action is performed on the car. The new position of the car is rendered to the screen.

This success criteria has been fully met. Test Numbers: 41 to 44

The player input detection takes place inside the Player class. This is also where the new positon of the vehicle is calculated. The position vector of the vehicle is updated, meaning that when the 'draw' function in the main game loop is called, the vehicle will be drawn to the screen in the new location.

Test number 41 shows that the inputs provided by the user are being recorded and correctly interpreted. The print statements used in this test were in the same block of code which changes the position of the vehicle. Therefore, as those statements were correctly printed to the console, the vehicle's position has also been updated. Confirmation that the vehicle moves as expected has been shown in other tests, such as test number 62. This has also been shown throughout the development in the section 'Testing the controls' under the heading 'Creating the car' on page 86 to 90. Hence this success criteria has been fully met.

5. Ask the player a question:
   11. Temporarily pause the game and ask the player a question at specific stages in a level.
   12. Initiate a countdown depending on the difficulty of the question.
   13. Wait for the player's response and determine their result.
   14. Resume the game

This success criteria has been fully met. Test Numbers: 45 to 60

The aim of this project is to improve the player's mental calculations speeds by asking them maths questions whilst playing a game. This shows that asking questions is a major component of the project and this aim has been fully met.

The game is correctly paused before asking a question. The current state of the level is saved in order to be resumed after the question has been answered. This is achieved by stopping the vehicle's position from being updated while paused. This is shown by test numbers 45 to 47. Test number 45 shows that game can be paused by pressing the 'P' button. Test number 46 shows that the game correctly pauses when the

vehicle collides with a question object. A question is then randomly chosen from the locally stored CSV questions file. This is shown by test number 47, where a question has correctly been asked after the game for colliding with a question object.

Once the question and answer choice have been proposed, a countdown timer is initiated. Test number 48 shows that the timer is correctly initialised and shown on the screen. The time which the timer counts down from should be the pre-determined time for the difficulty of the question. Test umber 50 shows that this feature of the timer correctly works. The timer is expected to countdown to zero, i.e. deduct one from the timer every second. This is shown to be working by test number 49. Test number 51 shows that the timer correctly stops counting after the question has been answered or the timer reaches zero. When the timer reaches zero, the question should be failed. When this happens, the game is expected to display an appropriate message to the screen. This is shown to be working by test number 52.

Next, the game waits for the player to respond to the question. As previously mentioned, if the player runs out of time, a 'Time Out' message is displayed and the question is failed. Test number 53 shows that the player is correctly given the option to choose an answer to the question. Four button objects will be created for the answers and the player needs to click on one. Test number 54 shows that the button which the user clicks is correctly interpreted by the game. After the response has been given, the game needs to determine whether the correct answer has been chosen. Test number 55 shows that the appropriate message is displayed to the screen in both possible situations (the correct or incorrect answer is chosen). Test number 56 shows that the question correctly cannot be avoided by exploiting a bug in the program. Such a bug did exist but, as shown in test number 56, it has been fixed.

After the correct/incorrect message is displayed to the screen, there is a delay of 1.5 seconds to allow the player to read the message. Test number 57 shows that this delay correctly happens. After this delay, the game is resumed and the player is able to re-gain control of the vehicle. Test number 58 shows that the game correctly resumes, and test number 59 shows that the player can successfully re-control the vehicle. During the delay, no inputs from the player are be taken there so the delay cannot be interrupted. This is shown by test number 60, where keyboard inputs and mouse clicks were made during the delay but there was no response from the game as expected.

Therefore, all the tests from 45 to 60 shows that this success criteria has been successfully fulfilled.

6. The time to answer the final question in the level should decrease depending on the tine it took to answer the minor questions.

This success criteria has been partially met. Reason: Time limitation

What has been included in the game is that the final question in a level needs to be answered in order of the level to be completed. Additionally, there is a column in the questions CSV file to identify which questions to ask for the final question. The questions with TRUE in this column are selected when the last question in a level is chosen. This process was explained in the Implementing Major questions section, where it was also shown to be correctly functioning.

What was required to be included in the game by this success criteria was for the time given to answer the final major question to depend on the time taken to answer the previous questions in the level. The reason why this feature was not incorporated into the program was due to time limitations. When compared to other components of the game, such as the level complete screen and the leader board page, this feature didn't seem essential and so was put off until a later notice.

How the Success Criteria could be addressed in further development

If I had more time to further develop this program, this feature can be easily implemented. This is because there already exists a method for timing the time taken by the player to answer the questions. This time is used to calculate the score awarded for the particular question. Inside each level, these times taken can be added an array. Then all the times in the array can be added to together and deducted from a set value for the major question. The player will then have the calculated length of time to answer the final question.

This calculated time needs to be validated by making sure that it is greater than zero, as negative time is not possible, and that a reasonable time is given even if multiple previous questions were answered incorrectly. This is because the final question is already intended to be more difficult than the questions throughout the level. Therefore, making the player answer a more difficult question in a shorter time will possibly cause them stress and make the user stop playing the game. On the other hand, this increased pressure could improve the player's mental calculation speeds over time in order for them to answer in time. The score awarded for this question can then be determining the remaining time, similar to the other questions.

7. When the car is in the air, it will rotate. If the car lands with the roof on the track, it has crashed. The level is failed.

This success criteria has been partially met. Reason: Time limitation

After adding the question boxes and before implementing the scores, I decided to attempt to rotate the player image. I didn't manage to reach a working solution due to time constraints. I had to move on to developing the other more important components of the game. Given more time, I would be able to develop the code that will be explained below, to reach a working solution.

Attempted solution

```
def rotateCenter(self, image, angle):
    orig_rect = image.get_rect()
    rotatedImage = pg.transform.rotate(image, angle)#rotates iamge by a particuar angle
    rotatedImageRect = orig_rect.copy()#keeps the original iamge
    rotatedImageRect.center = rotatedImage.get_rect().center#re-ceneters the transformed image
    #makes a new surface using the transformed image
    rotatedImage = rotatedImage.subsurface(rotatedImageRect).copy()
    return rotatedImage#returns the rotated image
```

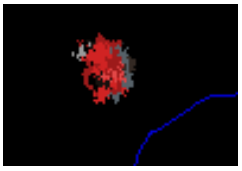
This is the code that I used to attempt this feature. My aim was to tilt the vehicle's image when it was contacted against a slope. The magnitude of the rotation would be the same as the gradient of the slop to ensure that the wheels of the vehicle are against the ground. I created a function to rotate an image whilst maintaining the centre of the image. The angle to be rotated by was a parameter to this function.

In order to calculate the angle to rotate by, I decided to use vector manipulation. When the player object collides with a platform object, the vector to the play is calculated by subtracting the position vector of the player from the position vector of the bottom-left corner of the game window. This direction vector was

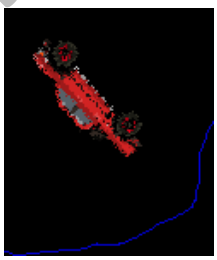
```
if hitPlatform:
    self.player.pos.y = hitPlatform[0].rect.y - self.player.height / 2
    self.player.vel.y = 0

    self.vecToPlayer = self.player.pos - vec(0, HEIGHT)
    self.rotate = self.vecToPlayer.angle_to(vec(1, 0))
    if self.prevRotate != round(self.rotate,2):
        self.player.image = self.rotateCenter(self.player.image, self.rotate)
        self.prevRotate = round(self.rotate,2)
```

then used alongside the horizontal vector $\begin{pmatrix}1\\0\end{pmatrix}$to calculate the angle between these. This will be the angle to rotate the image by. An 'if' statement was then used to limit how often the rotation will occur. Following this, the 'rotateCenter' function was called and the returned image was set as the new image for the player.

When I ran this code, the output was not as expected. This code was attempted before graphics were added so the background of the level screen is black. The vehicle feel under gravity as normal and when it made initial contact with the platform directly below where it was falling, it did rotate as expected. However, the image started getting distorted and parts of the image were being removed.



This effect was magnified when the vehicle was moved. The further collisions between the platform and therefore more rotations completely changed the vehicle's image and made it unrecognisable.



I suspect that this is because of the 'rect' object of the image was staying constant whilst the image itself was being rotated. Also, rotations were being calculated every time a collision happened and due to the platform objects being 1 pixel wide, this was being called move times than necessary. Additionally, the increased distorting of the image may have been caused by the program rotating an already distorted image. On top of this, there must have been a logical mistake in the calculation of the angle to rotate.

How the Success Criteria could be addressed in further development

Given more time, I would have returned to this code and attempted to solve the existing problems. A possible solution is rotating a copy of the image instead of the previous image. This would reduce the distortion caused to the image. Moreover, a separate surface will be created in section of the vehicle's image towards the top. Then collision checks will happen between this surface and the platform and track objects. If there is a collision, then the vehicle's has overturned and the roof has made contact with the ground. As required by the success criteria, this would mean that the level is failed. To implement this, the level complete function can be called when this happens.

Additionally, when the vehicle is in the air, it will need to rotate if the forwards or backwards buttons are pressed. To do this, the motion of the vehicle first needs to be decreased. This is because currently the speed of the vehicle will make rotation in the air difficult to notice. After this, checks can be made to see if the forward and backward buttons are pressed while the vehicle is above ground. This can be achieved by checking if the vehicle's velocity is greater than zero when there is no collision with a platform. If so, the vehicle's image should be rotated, with the direction of rotation being clockwise if the forward button is pressed, and anti-clockwise if the backwards button is pressed. When landing back on the ground, the collision checks will the vehicle's roof will resume to check that the player has landed the vehicle on its wheels.

Adding this feature will make the game more difficulty however, this added skill will make the game more enjoyable. This is because the player would be required to actively control the vehicle throughout the level and therefore increasing their sense of satisfaction when completing the level. In the current game, no skill is required by the player to collect the coins and collided with the question objects. The difficulty that will be added by the rotation will ensure that each time a level is played, the experience that a player has is different and so more enjoyable.

8. The score for the level is determined from how quickly the level is completed. Score will increase if coins on the level are collected.

This success criteria has been fully met. Test Numbers: 61 to 64

Score is awarded when a question has been correctly answered. The amount of score to reward the player for each question is calculated from the time taken by the player to answer the question. This is shown to be correctly working by test number 61. In this test, a longer time was taken to answer the same difficulty question and as expected, the score awarded for that question was less than the score awarded for a question that was answered more quickly.

Score should not be rewarded if the player has incorrectly answered the question. This is shown to be working by test number 62. Similarly, test number 63 shows that the game correctly doesn't increase the score if the player runs out of time on a question. When the player collides with a coin object the score should be increased by a set amount. Test number 64 showed that when the player collided with a coin object, the score increased by the pre-determined value (10), each time as expected.

Hence this success criteria has been fully fulfilled.

9. The next level will be unlocked only when the previous level has been completed.

This success criteria has not been met. Reason: Time limitation

This feature is a desirable one and was therefore intended to be implemented towards the end of the development of the game. Due to time limitations, I was unable to return to adding this feature. Even if this feature was implemented, it would only be appropriate is there a multiple user system. The reason why this success criteria will only be beneficial to a multi-user program is because when there is only one user, once the player has unlocked all of the levels, there is no longer a need for this feature.

This would require a system to log in and log out of accounts, as well has having to manage the accounts themselves. The data that will need to be stored about the accounts includes the name of the player, the total amount of coins they have, the vehicles that they have unlocked, their total score, the levels that they have unlocked and the settings they have currently chosen. On top of this, each user's password will need to be securely saved. Before storing the password, it could be hashed using a hashing algorithm, meaning that even if an unauthorised personal accesses the file, they will be unable to reverse engineer the hash to get the password.

How the Success Criteria could be addressed in further development

In further development, in order to add this level locked feature, each level buttons can be given an additional parameter called locked. This parameter will be specific to the level buttons. For all other buttons, this parameter could be set as 'None' by default, but will need to be provided when level buttons are created. This parameter will hold a Boolean value (i.e. True or False), with True meaning the level is locked and False implying that it is not locked. If a level is locked, when it is clicked, the game should open the level but instead display a message such as "This level is currently locked". The level buttons which are unlocked will function as normal buttons.

The locked status of each level button will need to be stored so that it is saved whilst the game is closed. This can be achieved by using a CSV file or as used in the game, a pickle dictionary. This data will then be read in and the level buttons' locked parameter will depend on this data. If a multi-user system is developed, this locked level data will need to be created for each user.

10. Question asked will be random and the choice of answers will be random. (i.e the correct answer will not always be A)

This success criteria has been fully met. Test Numbers: 65 to 68

A question is asked when the vehicle collides with a question object. When this event occurs, a random question is selected from the locally stored two-dimensional array. Test number 66 shows that this requirement of the program correctly functions. There are four incorrect answers for each question. Three of these need to be randomly selected and proposed to the player alongside the correct answer. Test number 67 shows that the game correctly, randomly chooses the incorrect answers for the question.

Even if the same question is chosen again in the same level, the proposed answers should be in a different order. This prevents the player from simplify memorizing the position of the correct answer. Test number 65 shows that the answers to the same question are in a different order the second time as expected. This also implies that the correct answer will not be in the same location for each question. Hence this success criteria has been fully met.

11. Mini game for answering questions correctly functions:
    - Questions are only asked within the time period
    - The player's result is determined at the end of the question
    - Player is rewarded coins depending on the number of their correct answers.
    - Incorrect answers will decrease time remaining
    - Score is calculated depending on the difficulty and number of questions answered

This success criteria has been partially met. Reason: Time limitation

What has been included in the game is the question asking functionality, as required by bullet points 2 and 5 of the success criteria. Questions can be asked to the user by making the 'askQuestion' variable True. Additionally, the current game is able to calculate whether the player has answered the question correctly or not and from there, determine the score to award the player. Moreover, the top ten score achieved in the mini game can be view from the leader board screen. This means that when the mini game is added, the functionality from loading the top ten scores achieved on it in the leader board screen already exists. Due to time limitations, I was unable to fully add this mode into the game.

How the Success Criteria could be addressed in further development

Given more time, I would have been able to fully implement this game mode into the program. To do this, the first stage will be to code a timer with the starting time set at 90 seconds (which is the time that had been determined or the mini game). A timer function has already been created in the game for use during the questions. Therefore, this same function can be used for the mini game by setting the 'timeRemaining' variable to 90. This can be used for the solution as the timer will not be required for the questions asked in the mini game. If however two timers were needed at the same time, a timer class could be created.

After this, the 'askQuestion' and 'paused' variables will be made True. This will cause the game to ask a question, as the state of these variables is checked in every frame. The question asking functionality is independent of the current level so no alterations to this code will need to be made for the mini game. After the player has answered the question, there will be a delay for them to read the displayed message. In this game mode, the delay duration can be reduced to 0.75 seconds to maximise the time given to the player to answer the questions. This can be implemented by checking the current scene before the delay and if it is 'minigame', the delay will be shorter.

During the question, the timer will need to be constantly updated. This can be implemented by repeatedly calling the mini game function, with code to update the timer inside this function. Inside this function, the 'askQuestion' and 'paused' variables will also be set to True. This will means that as soon as the delay for answering a question finishes, the 'minigame' function will be called again and therefore, a new question will be answered. If the player answers a question incorrectly, the timer needs to be decreased. This can be achieved by decreasing the 'timeRemaining' variable be a specific amount. Inside this function, the time on the timer will be checked before setting the 'askQuestion' and 'paused' variable to True. When the timer reaches zero, these variables will no longer be set to True.

Throughout the mini game, a record of the number of questions answered will be kept. This will then be used at the end of the mode to calculate the score to award the player. The number of coins to reward the player will then be calculated using the score. Next, the top ten scores achieved on the mini game will be examined and if the achieved score is in the top ten, the player will be presented with an input box to enter their name. This information will be shown on a separate screen, such as the 'levelComplete' screen. On this screen there will be buttons to replay the mini game, and go to the main menu.

12. Endless mode of the game correctly functions:

- Questions are regularly asked
- Time for major question adjusted according to minor question answer times.
- Fuel is awarded for correct answering of major question
- Mode ends when player runs out of fuel
- Score is calculated depending on distance travelled

This success criteria has not been met. Reason: Time limitation

Similar to the mini game mode, this game mode has not implemented into the game due to time limitations. Also, this game mode and the mini game mode were non-essential features of the game, as explained under the heading 'Key features of the proposed solution' on page 32. This means that not implementing these features doesn't have an impact on the rest of the game. However decomposing this problem now reveals that there could have been more complications in programming this mode even if it was attempted.

How the Success Criteria could be addressed in further development

Following the bullet points set out of for this mode, the questions could have been regularly asked by spawning question objects at a regular distance. If this method was proving to be difficult, for example it was difficult to identify the correct locations to spawn the questions due to the uneven ground surface, there could be a question every 10 seconds. This can be programmed by having a timer on the screen counting down and when it reaches zero, a question will be asked. Alternatively, instead of time, a question could be asked after a specific distance is passed in the level. This would require the position vector of the player object to be used to actively calculate the distance the vehicle has travelled.

The second bullet point can be achieved by recording the time taken to answer the minor questions in the game. The total time taken to answer all of the previous minor questions will be deducted from the major questions time. After a major question is correctly answered, this process will reset and only times taken to answer the succeeding questions will be stored.

Fuel level can be added to the vehicle as an attribute that will only be used whilst in the endless mode. This fuel variable will hold a value that will decrease as the vehicle moves. To perform this, the vehicles position will be analysed and after it travels a set distance, the fuel value will be decreased by a pre-determined value. The fuel level will be displayed to the screen in the form of a bar. There will be green, amber and red

areas on this bar and a rectangle or arrow indicating the current value of the fuel. The fuel value will be increased when a major question is answered correctly.

When the fuel reaches zero, the game should be ended. This can be programmed by repeatedly checking the fuel variable whilst the endless game mode is running, and stopping the game mode when it reaches zero.

The total distance travelled by the vehicle can be calculated using its initial position and the final position vector. The magnitude of this vector will be the distance travelled. This value will be used alongside the number of questions answered to calculate the score. Similar to the mini game and the levels, if the achieved score is in the top ten, an input box object will be created for the player to enter their name for the leader board. Coins will not be rewarded as they will be placed around the map for the player to collect.

The main problem with this game mode is how the map will be loaded. The requirement of an endless mode is for the level to be continuous so if a map is created manually using the method used for creating the levels, no matter how long this map is there is an end. Therefore, it is not continuous and cannot be used. The other solution is to have multiple small segments of the map and randomly load them in one after another. This would require all of the current objects on the screen, excluding the player object, to be deleted and then re-created in their new locations. This could affect the performance of the game due to the increased level of computation. If this is the case, then the game engine (pygame) itself could be a limitation.

13. All scores, for levels, mini game and endless mode, are correctly added to their respective leader boards. All leader boards are accessible from the leader board screen via the main menu.

This success criteria has been fully met. Test Numbers: 28 to 31

The leader board screen is correctly functioning so s required by this success criteria, the scores for all of the level and other game modes are correctly being shown. Currently, due to the mini game and endless mode not being included, their pages on the leader board show zero for all the scores. This is also the case for the levels which have not yet been added however when they are, the scores will seamlessly be updated to the leader board page as this code is independent of the levels.

Test number 28 shows that the back button on this screen is correctly functioning. Test number 29 shows that the score information shown on this screen correctly updates when the player achieves a score in the top ten for that level. Test number 30 shows that when the leader board screen is loaded, the scores for level 1 are displayed by default as expected. Moreover, test number 31 shows that the scores displayed on the screen correctly change when a different level or mini game button is pressed.

Hence this success criteria has been fully met.

14. The game statistics are correct and are accessible via the main menu.
The game achievements are correctly unlocked as the game progresses.
Coins can be used to purchase different items in the shop.

This success criteria has been partially met. Test Numbers: 7, 32 to 37 Reason: Time limitation

The statistics screen has been fully implemented into the game. What has not been achieved is the achievement screen. A shop screen, which was later renamed to vehicles screen, was also created however the functionality to purchase different vehicles using coins was not added. The reason for this is because of time limitations. I was unable to return to implementing this feature when there were other more important

components of the game that needed to be developed. Despite the lack of purchasing in the vehicles screen, the player is still able to correctly change their current vehicle here.

As required by the first section of this success criteria, the statistics screen is correctly functioning and is able to be accessed from the main menu screen. Test number 7 shows that the statistics button on the main menu screen correctly loads the statistics screen. Test number 32 shows that the back button on the statistics screen correctly takes the user to the main menu screen. Moreover, test numbers 33 through 37 show that all of the data shown on the statistics screen correctly updates when the appropriate changes are made by the player in game. For example, test number 33 shows that the total games value correctly increases when the user plays a level.

How the Success Criteria could be addressed in further development

The intention of the achievements screen was to provide the player with goals to reach throughout their progression through the game. When the player reaches these goals, they will be awarded with coins. In this screen, the user would have been able to view their progress on each of these goals. Therefore, the data that will need to be stored for each achievement includes: the name of the achievement, the value required to complete the achievement, the current value of the achievement and the coins awarded for completing the achievement. If a multi-user system is added to the game, this achievements information will need to be stored for each user.

The achievements screen itself can be created using the 'Scene Manager' class. The data about the achievements will be used to display the information about the current achievements on this screen. There will be a progress bar for each goal showing how far the player is from completing it. Buttons can be created for completed achievements and when these are pressed, the coins will be rewarded. After the coins have been given, the achievement will colour to inform the user that it has been completed.

For the final part of this success criteria, the purchasing using coins feature needs to be added. If I had more time, the method that I would have used to implement this would have been as follows. First, there will be multiple vehicles on the vehicles screen. Each of these will have their name, image, value and locked status. The name and image of each vehicle will be displayed on the screen. The value will be the amount of coins required to unlock the vehicle. The locked status will be a variable holding a Boolean value. This variable will be used to determine whether the particular vehicle has already been purchased.

There will be a button for each vehicle that can be clicked to purchase it. When this button is clicked, the first check is if the vehicle is locked (i.e. it has not been purchased yet). If this is False, the coin total and the value of the vehicle will be examined. If the player has enough coins to complete the purchase, then the locked status variable will be changed to False. Once a vehicle has been purchased, when its button is now pressed, it should change the current vehicle to that instead of re-purchasing it. On top of vehicles, there will be themes, backgrounds and features such as car horn and headlights that can be purchased. When these items are purchased, variables in the map loading or player object creating process will be changed to incorporate these choices.

15. The game is entertaining for the player.

This success criteria has been fully met.

In my opinion this success criteria has been fully met. The inclusion of graphics and music to the game has improved the user experience as there are now more elements of the game for the user to interact with. More evidence for the accomplishment of this success criteria will be provided in the User Feedback section on page 205.

Usability features incorporated in program

The purpose of usability features being incorporated into a program is to increase the size of possible users for the program. Specific features, which if were not present will not allow certain people use the program, are include to widen the user base. A list of these features which will be added to the game was explained on page 66. In the Testing sections, tests were also performed to test the usability features. I will now judge the success and effectiveness of each of these usability features.

1. Buttons

This usability feature requirement has been fully met.

The requirements for the buttons used throughout the game, but especially in the menu system, was for them to be easy to press, have easily identifiable purposes, to highlight when the mouse is over them and, to have a bright and vibrant colour scheme. Also, in consideration for user with vision difficulties, a strong black border should be used on the buttons to assist in conveying the interactive region of the buttons.

Test number 1 shows that large, and therefore easy to press, buttons has been used on all the menu screens. Buttons which will be pressed more often are even larger, as explained in this test. Test number 2 shows that the text is used on the buttons to depict the purpose of each of them. Additionally, test number 9 shows that images are displayed beside some buttons to further show the purpose of them. Test number 5 shows that the buttons correctly change colour when the mouse cursor is over them. This makes it easier to identify the position of the mouse.

Finally, test number 6 shows that bright and vivid colours have successfully been used for the buttons to make it easier for the user to differentiate between these objects and others on the screen. A lack border has not been included in the buttons as I believe that it will overwhelm the otherwise simple user interface. However, in its place, as shown by test number 4 and 7, contrasting colours have been used between the button's solid (i.e. normal, un-highlighted) and highlighted colour. This graciously compensates for the black borders as it achieves the same effect, which is to convey the interactive region for the buttons. Therefore, this usability feature has been fully fulfilled.

2. Text

This usability feature requirement has been fully met.

The requirements for the text used throughout the game were for it to be large, have contrasting colours and to use anti-aliasing. All these were intended to make the text easier to read.

Test number 2 shows that, as required, there is large, easily readable text inside each button. The colour of this text is black, which greatly contrasts with the images of the buttons. Moreover, as shown by test number 8, anti-aliasing has been used when creating the text surfaces inside the 'drawText' function. As all text elements used in the game have been drawn using the 'drawText' function, this means that the text characters in the game are sharper and more detailed than ordinary text. This feature aids users with visual difficulties to use my program. Therefore, this usability feature has been fully met.

Additionally, instructions have been added to the game to inform them of how to play the game. This instructions screen is displayed at the start of each level and can also be accessed via the settings screen. The purpose of this screen is to make sure that all users of the program are completely aware of the aim of the game, so that no user is at a disadvantage.

3. Graphics and music

This usability feature requirement has been fully met.

On top of improving the overall aesthetic of the game, graphic aid users with vision difficulties as they make it easier to identify objects compared to text. The requirement for the music and sound effects used in the game is for it to provide assistance to partially sighted people when using the game.

Test number 10 shows that music has correctly been implemented throughout the game. There is a choice for the music to be played during the level and the user is able to select this in the settings screen. Moreover, the option to disable game music entirely has also been incorporated to appeal to all users. Users with hearing difficulties will be able to enjoy the graphics of the program and will not be at any sort of disadvantage in the game. This is because the added music is solely for an atmosphere enhancing purpose and provides no assistance when answering the questions in the levels. A possible improvement to the game for further aiding users who are partially sighted, is to add more sound effects. This will be further explained below. Despite this, this usability feature requirement has been fulfilled.

Additional usability features in further development

On top of the already implemented usability features, additional features could be included in further development of the program to increasingly widen its user base. One such feature, as mentioned above, is to add more music and sound effects. Currently, no sound effects are being used in the game. These could be implemented to play when a button is clicked, a certain screen is select and when coins are collected. Adding this feature will further enhance the atmosphere of the game whilst also providing more assistance to partially-sighted users. This is because they will be able to associate certain sounds with events in the game.

Another improvement is the addition of different colour themes. This was initially designed to be an option in the shop screen but was not added due to time limitations. The colour themes will include the colours of the buttons and the background images of the menu screens. This option will allow the user to choose whichever theme they like, with reasons for this choice being due to the chosen theme being the darkest to reduce eye strain. Moreover, the option to change the images of the maps could also be added to reflect the changes to the menu screens inside the levels.

Animations can also be added to the game. This will further enhance the user atmosphere whilst also improving the aesthetic look of the game. On top of this, animations will aid users with vision difficulties to identify the current screen as they may be able to link a specific animation to it. Finally, the size of the buttons can all be increased to fully make use of all the available space on the game window. This will increase the interactive region of the buttons, thus making them easier to press.

These additional usability features will help to broaden the user base for my program to allow more people to improve their mental arithmetic calculation speed.

User Feedback

Name	Age group	Comments	Improvements
Harini	Years 10-11	Brilliant game! I loved answering the questions and collecting the coins in the levels. The statistics screen is a great way to monitor my performance in the game and I found myself on it a lot. It's nice that the date it also saved in the leader board. The addition of music is great as well as the options for the choice of vehicles and question difficulty.	The leader board system is an enjoyable addition however, it can be demoralising and intimidating for users who have not achieved the highest scores. If they repeatedly get scores in the bottom half of the table they may stop playing the game. A possible solution to this is to have the leader board reset on a time basis.
Ayub	Years 12-13	This is a very enjoyable game! I love playing through each of the levels and achieving a high score so that I am at the top of each of leader board for each level. The simple but smooth graphics work well with the flow and design of the game. The music is a good touch, especially the pause and menu music. Answering the questions quickly and watching the score be calculated is so satisfying!	The designs for the levels were pretty similar and there was no real challenge in obtaining the coins. Adding different style levels and enemies in order to make getting the coins and navigating through the level more of a challenge will make the game even more fun! Also, instead of a blank name in the leader board when one is not provided, have Anonymous (this was added to the code).
Rory	Years 12-13	Awesome game! I was startled by the hard questions but beyond the surface, there were achievable in the given time. The player controls were smooth and fluent during the game and the brake button was a great addition. Although the designs of the levels are repetitive, they are nice to look at. Leader board is great as it creates good competition.	Sometimes the vehicle blocks the question. Maybe add a background around the question to avoid this. Also, add the sound of the clock for the timer when a question is asked. The increased tension will introduce urgency and also make the game more thrilling! The questions did start repeating often. Maybe get the computer to randomly generate questions instead of hard coding them.
Matthew	Years 10-11	I love playing the levels and using the different vehicles. The questions were fun to answer and certain ones really made me think. The options in the game were decently varied, sufficient for this game. The leader board is great. I played so many levels to make sure that I was at the top for each level!	I found myself often restarting the level to achieve a highscore. A restart button or hotkey could be added to make this process easier. More variety in the music and the transition between the music tracks could be improved as they seemed sudden. I want more themes and vehicles, especially a boat!
John	Years 7-9	I had a lot of fun playing this! It was difficult for a new user to understand the concepts of the game but after the initial learning curve and muscle memory for quickly answering the questions, I started to really enjoy the game. I tried hard difficulty but I have not learned how to answer those questions yet so the difficulty changer is a very good option.	A tutorial could be included to introduce a new user to all the features of the program. This will be useful as I was unaware of the wide range of options in the settings for some time. Some of the questions were a bit difficult so a bit more time could be given for them. Also, the difficulty and music choices could save when the game is closed.
Mr Lemma (stakeholder)	Teacher	Wow! This is the first time I am seeing the game and I'm impressed. The questions are well spaced out and are of appropriate difficulty for the stated difficulty in my opinion. I think that the time allocated is sufficient. I really like how questions can easily be added to the CSV file and thus the game.	Image based questions could be included to increase the variety of questions which could be asked. In addition to this, a system for monitoring the performance of students could be added so I can monitor each student's progress. Overall the game is excellent and I look forward to using it with my students!

Maintenance issues

Throughout development, I have tried to keep my program as easily maintainable in the future as possible. Here are the beneficial features of the program which aid future maintenance.

This has been achieved by thoroughly commented code, as shown in the 'Final Code' section on pages 212 to 248, and consistent camel case naming system used for the variables in the program. This naming structure reduces the possibilities of confusion with variable names occurring in later development. Furthermore the 'Development' section of this document provides a detailed and thorough explanation for each component of the program, so if the program needs to be modified by others in the future, this coursework could be used as documentation for how the program functions. The program consists of separate files and many classes and functions. This means that there are lots of independent blocks of code, making it easier to isolate errors that occur in the future, as we as making it simpler to add more features to the program. In addition, this style of programming allows the program to execute more efficiently whilst also improving the readability of the code for others.

There are some issues with the current program which could make future development of the program more difficult that necessary.

One such issue is the method used to run the game. Currently, to execute the game, the python interpreter and all the imported modules (e.g. pygame, csv, pytmx etc.) need to be correctly installed on the system. If all of the required libraries are not present in their expected locations, the program will not run. Furthermore, the data used for the game are also required in specific locations. The images, map data, question csv file and music used in the game are expected in their specific folders and if they are not present, the game will not run. I have reduced to likelihood of an error for the file data occurring by scanning for the file path to the current directory in the game instead of hard coding it. Despite this, if a file is missing due to a problem occurring in transportation then this error will occur. A solution to this problem is to make an executable version of the program. When the executable is made, the libraries and files required for the program are compiled into the executable file, thus allowing the program to be run on any system without requiring python or pygame to be installed.

Another issue is the forward and backwards compatibility of the software used. Python 3.6 and pygame version 1.9.4 has been used on the system used to develop the program. This source https://www.pygame.org/wiki/PythonVersions states that pygame 1.9 is compatible with python versions 2.4, 2.5, 2.6 and3.1 and onwards. Any version of python or pygame that is released in the future is likely to be backwards compatible unless there is a major change (i.e. Python 4.0). In this case, the syntax could be changed, similar to how parentheses were added for the print statements for the change between Python 2.4 and Python 3.0. In this case, the whole program will need to be fixed to work as previously. This problem can however ben avoided by converting the game to an executable file. As mentioned above, this file will be independent of python and the libraries used and so forwards and backwards compatibility of the software used will not be a concern.

Distribution of the software and future updates to the program are also issues. This is because the program will need to run on many different operating systems and computer architectures. Likewise, when the updates are created, they will need to be made specifically for each operating system. Similar to the previous two issues, the solution to this problem is to create an executable version of the program. The executable will be able to run on all windows machines. Mac users will be able to run the executable file using the 'Boot Camp' software. https://support.apple.com/en-gb/boot-camp This is an official Apple software which lets Mac users run a virtual windows machine. Python and pygame have Mac and Unix (for Linux operating systems) versions so theses could be installed on the user's machine. However, this would

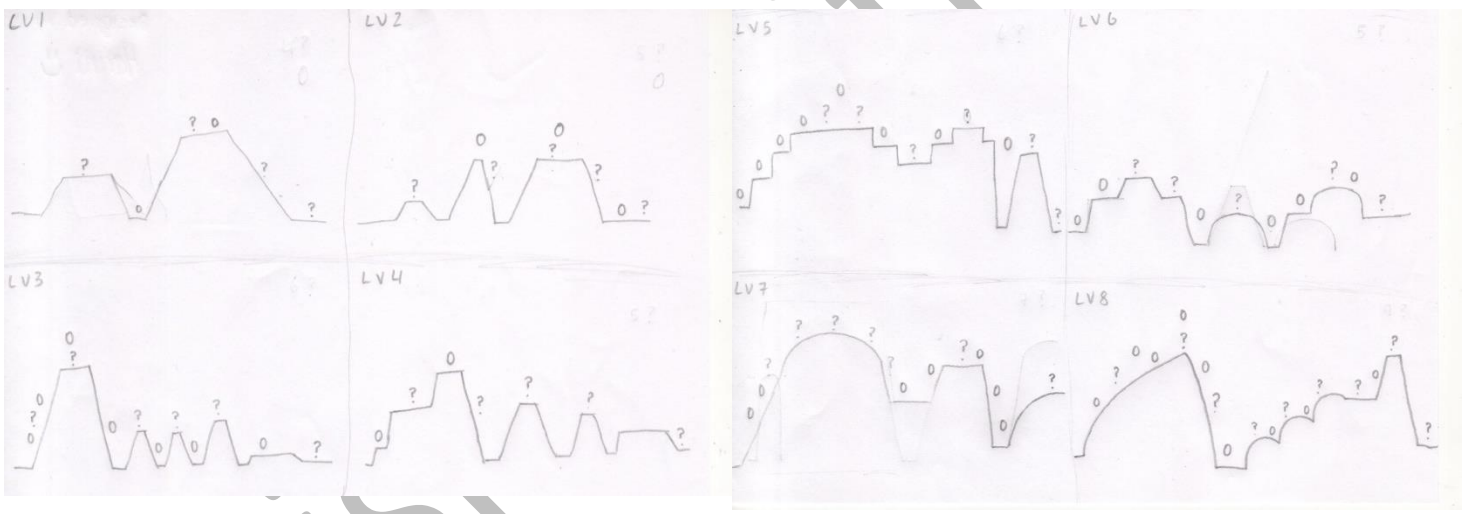
require the source code to be directly distributed to the user in each update. This is a problem as the user will be able to access and modify the source code and potentially introduce an error to the program.

Future updates will consist of bug fixes and feature adding. However currently, there is no method for users providing feedback to the developer about certain bugs or potential features that could be included. This issue can be solved by getting the program to record bugs when they are encountered (such as if the program crashes), and send it to the developer via email. Alternatively, an online form could be created and sent alongside the executable file to the users. Then when an error is encountered, the users could fill out the form to inform the developer. The problem with this solution is that there is no guarantee all errors will be informed so both these solutions could be implemented so that ongoing feedback can be received from the users for features that could be added to the game in further updates.

Limitations of the solution

There are a few limitations in my program which could be addressed in further development. I have explained possible solutions to these limitations as well.

One such limitation is how only four levels have been added. As mentioned when evaluating the success criteria, time limitations were the cause for not adding all of the levels and game modes. How these will be implemented in further development has been discussed and I have also drawn the designs for all the levels.



I used this template when creating levels 1 through 4, and planned on using it for the rest. I have intentionally made the method of adding levels to the game easy, meaning that the remainder of the levels could effortlessly be added at a later stage. This process involves: adding a level function to the 'Scene Manager' class, drawing the level in 'Tiled' and then copying the code in a previous level function but changing the filename for the map file. I realised that the code used defined in each level function was virtually the same apart from the filename. I was planning on making a general function to load each level, with the level number and map file being parameters to this function, however due to time limitations I was unable to do this. To create this function, the existing code in a level function can be copied and the level number and filename parameters can be used for calling the 'loadHighscore' function and to load the map image respectively.

Another limitation to the program is the graphics and audio elements used. Pygame is slower and less efficient that other game engine modules due to the ways that it fundamentally works. It affects performance by having to unnecessarily re-draw each frame and it is not able to handle high quality images. This means that high quality graphics with smooth movement cannot be achieved using pygame in further development. If this was a requirement, and it will be as the user base grows, other python game engines

will need to be explored. Examples of these modules include 'Pyglet' and 'Kivy'. If this transition is required, the pygame commands used in the program for drawing to the screen ('blit'), creating surfaces and etcetera, could be replaced with suitable 'Pyglet' or 'Kivy' functions. Alternatively, instead of switching to another python module, which will inevitably have its own drawbacks, the initial plan could be followed and the game could be developed using Unity. The plan of the project was to complete two stages of development, one using python and pygame and the second using unity however I was unable to complete the latter due to time limitations. Given more time, or in future development, the existing python code could be used alongside the many features of Unity to create the game. Unity will also provide more advanced audio options for the in-game music and effects.

Further limitations include the questions that can be asked during the levels and in the mini game. Currently, the questions are hardcoded into a CSV file, imported, searched and then asked. In the 'User Feedback' section, users informed that the same question was being asked often. This allowed them to simply remember the answer, the actual value and not just its position, for subsequent questions. I did program a method to remove the previously asked questions however after adding the major questions, the possible question numbers are re-calculated each time. This means that any previous removal of asked question is lost. A potential solution to this limitation in the future is to make the program less random in order to make it appear more random. This can be achieved by similarly removing asked question numbers or making it more unlikely to choose those questions. Alternatively, the same effect can be achieved by increasing the number of questions in the csv file. This would mean that mathematically, it will be less likely for the same question to be asked.

An additional limitation with the questions is the variety of them. There are only a few styles of questions which can be quickly calculated mentally whilst also being text-based. As requested by the stakeholder in the 'User Feedback' section, image-based questions could be added in future development to have a different style of question. These questions will also tend to users who more easily understand visual elements compared to text, thus making the game fairer for all users. This feature can be added by linking an image's filename in the CSV file and then loading and displaying the image when the question is asked. Possible question topics that could be explored using images include: angles in polygons, trigonometry, perimeter and area calculations and, circle theorem.

Finally, the program is limited to a PC or laptop running Windows. This is due to the libraries that have been used and they could not be available for other operating systems. As mentioned previously, a solution to this is to make an executable file however, this will not allow the game to be run on a mobile device. I re-searched methods of compiling pygame files into APK files however even if this was possible, the controls cannot be easily mapped to touch screens areas or buttons. To achieve multi-platform support, as initially planned at the start of the project, the game will need to be further developed using Unity. The 'Kivy' python module does provide multi-platform support however, it has been designed to be used for creating user interface elements and so will not be beneficial when creating the actual game. Once the game has been fully developed using Unity, the same project can be built to multiple different operating systems. Another benefit that this provides is that updates can be created in one file and then converted to the required operating systems. Moreover, the game created using Unity can have superior graphics, better audio and, a more sophisticated user interface (menu system), whilst also running more efficient due to the more advanced game engine.

Potential Improvements

Some potential improvements to the program have already been explained in the 'Maintenance Issues' and 'Limitations of the solution' sections. Briefly, these include:

- Making an executable version of the program
- Recording bugs/errors in the game
- Asking more questions with different styles
- Making the questions more random
- Adding multi-platform support using Unity

There are more potential improvements which have been suggested in the 'User Feedback' section.

- Make the coins in the level more difficult to obtain and add in enemies

  This would make it more challenging for the user to user manoeuvre around the level and thus make the game feel less repetitive. By increasing the difficulty of completing the levels, the sense of satisfaction for completing the levels is also increased. Therefore, doing this will make the game more enjoyable for the users. The enemies could be programmed to have complex movement, meaning that they detect the position of the player and move accordingly. A health bar could be added to the player and when an enemy collides with the player, the health decreases. The level/mode will also be failed if the health reaches zero. Alternatively, instead of enemies, the track could be made more difficult by adding gaps and loops. If the player lands the vehicle on its roof or falls into the gaps, the level/mode will be failed.

- Add sound effects for the question timer

  This will increase the tension present when a question is being answered. As explained in the 'User Feedback' section, the ticking noise will introduce urgency into the player and get them to answer the question more quickly.

- Add a restart button or hotkey

  When this key is pressed, the current level will restart. This will be useful when a highscore is being attempted as this button will replace pausing the game, pressing the main menu button, pressing the select level button and then choosing the required level.

- Improve the music transitions

  Fade out have already been used in the game for when the music changes. To further make this transition smoother, fade in commands can also be added. This will gradually increase the volume of the playing track at the beginning of it. Additionally, the fade in/out period could be increased from 0.5 seconds to make the transition between the music tracks smoother and so more unnoticeable.

- Add more vehicles

  Additional vehicles will provide players with more ways to enjoy the game. They will provide the user encouragement to play through the game and earn the coins in order to spend them to buy more vehicles. Moreover, providing a large-variety of vehicles will catch the eye of some users who do not have a specific favourite but will want to unlock everything.

- Include a tutorial of all the game's features

  The purpose of this tutorial will be to introduce new users to all the game screens and specific features. This is required as unless shown, many users will be unaware of the existence of the options in the settings screen. Although mentioned on the instruction, from testing my game on multiple users, I found that often people didn't read this. This could be due to it having too much text so I could try reducing the text on this screen or formatting the existing text differently. Due to people not reading the instructions, a tutorial would be required to make sure that all users are at the same understanding of the aim and controls of the game. This tutorial will only be forced to the user the very first time they launch the game. After that, it will be accessible from the settings screen.

- Add a background to the questions

  The question was sometimes getting blocked by the vehicle depending on its position in the level. This makes the question unreadable. A potential improvement that will solve this problem whilst also improving the aesthetic look of the game is to have a coloured background for the question text. This will ensure that the question text is visible and readable for the user. On top of this, the correct/incorrect/time out message is also sometimes blocked or made hard to read by the map image that is behind it. This problem can similarly be approached by using a coloured background for the text.

- Save music and question difficulty choices

  The music and difficulty choice for questions do have default values however any changes that are made to these options are completely lost when the game is closed. An improvement will be to save all the user's chosen settings and the next time the game is launched, the saved user settings should be used to set the game options. This can be achieved by storing this data in a pickle dictionary and analysing it at the start of the game.

- Multi-user system

  As explained when evaluating success criteria number 9, a multi-user system could be added to the game to allow different users to use the program without losing their game progress information. To create this system, a login and logout system needs to be implemented. After this, each piece of data that is currently stored externally (i.e. in pickle dictionary files) will need to be created for each new user. This would require suitable file management to make sure that all the files are in their appropriate locations. Next, the username and associated passwords need to be securely stored. A hashing function could be used when storing password to increase security.

- Monitoring student performance for teachers in a multi-user system

  Once a multi-use system has been implemented, a feature for teachers could be added to allow them to monitor their students' progress through the game. Teachers should be able to view the leader boards, which in a multi-user system will have to be network based to allow the scores from all users of the game to be saved. Additionally, a feature of the teachers selecting which questions they want their students to answer across the game (i.e. in all game modes) could be added.

- Making a mobile version of the game

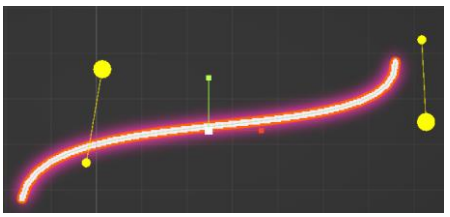
  As explained during the 'Analysis' section, the plan was to make a mobile version of the game using Unity. This would have allowed my game to be played on many more devices that my target audience are more likely to have. To achieve this, the game can be developed using Unity as previously planned.

Overall Comments

The project has managed to meet all of the essential features and most of the desirable/extra features identified in the 'Key features of the Proposed Solution' section on page 32. The created game successfully fulfils the essential requirements of asking mathematic questions that are appropriate to the user, providing a vehicle for the user to manoeuvre across a track, completing the level when the player reaches the end of the track, correcting mapping controls to vehicle actions and, creating a fully functional menu systems linking all game screens. Additionally, the created game also successfully fulfils the desirable features to include a statistics and leader board page to enhance the player's experience, to have the vehicle in the background when the game is paused for a question, to have collectable items (coins) in the game, to have different vehicles to choose from and, to have a choice for the music played during the game. Any sources used throughout this document have been fully cited and credited at the used locations.

There are a few benefits with the method used to program the game. An example of this is how levels can be easily added to the game. As explained throughout the 'Development' section, Tiled is a third-part software that has been used to draw and create the levels. It has a friendly user interface whilst also being advanced, allowing the creation of any imaginable level by anybody. Additionally, object oriented programming and many classes, functions and independent blocks of code have been used to develop this program. This means that its will be easier to improve, debug and manage the program in the future. This style of programing has allowed the use of global variables to be avoided whilst also making the code more efficient overall. Furthermore, the code has been fully commented and documented to aid future development.

In further development, in priority to all the improvements which have already been described and explained in the 'Maintenance Issues', 'Limitations of the solution' and 'Potential Improvements' sections, I will definitely like to develop the game using Unity. This is because this software will provide many additional features and downloadable assets which can be used to aid in development. One such asset is called the 'Bezier Curve Editor' (https://bit.ly/2WZVSCb) and this will allow smooth, editable curves to be created for the track in the levels. Also, physics attributes, such as rolling down slopes, can be easily added to the vehicles. In addition to this, procedural map generation in Unity can be used to create the endless game mode. Moreover, Unity provides many graphical options and features. An example of this is the glowing line shown on the right. Once the game has been developed in Unity, as explained previously, it will be much more manageable for distributing to different devices and for updates. Therefore, it will be appropriate to incorporate the other improvements to the actual game once this stage is reached. Developing the program further using this method will also provide a wider variety of potential improvements and features to be available.



Using the feedback from the user, it is clear that this program has achieved its aim of improving the mental calculation speeds of its users. I analysed the students who I provided the game to test and found that most of them were repeatedly playing the levels to increase their score and get to the top of the leader board. Additionally, on subsequent goes of attempting each level, from my observations, they were getting faster at answering the questions. This could be partially due to them learning the muscle memory to move to the answers quicker however it will be greatly dependant on them calculating the answer to the question. I told the users that they will be rewarded a higher score if they answer the questions faster and I found that this further improved the speed at which they answered the questions.

Taking all of this into consideration makes it clear that this project has fully achieved its aim.

Final Code

The game is now finished and has been evaluated using the success criteria and user feedback. No further changes will be made to the code as the testing procedure has now finished. Therefore, the final commented code can now be shown. Each script, class and function has been shown separately to improve the readability of the code.

Main.py script

Importing the modules

```
import pygame as pg #pygame, OS, random and csv modules are imported
from os import path
import random
import csv
from settings import *#Three other game files are imported
from sprites import *
from management import *

vec = pg.math.Vector2#vector function stored as 'vec'
```

Game Class:

'init' function

```
class Game:#Game class is created
    def __init__(self):#game is initialised
        pg.init()#pygame module is initalised
        pg.mixer.init()#Sound and music functionality is initialised
        self.screen = pg.display.set_mode((WIDTH, HEIGHT))#Game window is created
        pg.display.set_caption(TITLE)#Sets the title of the window
        self.clock = pg.time.Clock()#starts the internal clock to sync the game
        self.running = True
        self.loadData()#LoadData funtion is called
```

'loadStatsData' function

```
def loadstatsData(self):#Loads statistics data
    statsFile = open("stats.pickle", "rb")#statistics file is loaded
    self.statsData = pickle.load(statsFile)
    statsFile.close()
    self.coinAmount = self.statsData["coinTotal"]#statistics data is read and stored locally
    self.gamesPlayed = self.statsData["gamesPlayed"]
    self.questAnswered = self.statsData["questAnswered"]
    self.correctAnswerQuesEasy = self.statsData["correctAnswerQuesEasy"]
    self.correctAnswerQuesMed = self.statsData["correctAnswerQuesMed"]
    self.correctAnswerQuesHard = self.statsData["correctAnswerQuesHard"]
    self.vehicleUnlock = self.statsData["vehicleUnlock"]
    self.coinSpent = self.statsData["coinSpent"]
    self.totalScore = self.statsData["totalScore"]
```

‘loadData’ function

```
def loadData(self):#all external files, images and music are loaded in
    self.gameFolder = path.dirname(__file__)#path to the current directory is obtained
    self.imgFolder = path.join(self.gameFolder, 'img')#path to the image, map and music directories are obtained
    self.mapFolder = path.join(self.gameFolder, 'map')
    self.sndFolder = path.join(self.gameFolder, 'snd')

    self.interfaceFont = path.join(self.imgFolder, 'Future.ttf')#fonts are loaded in
    self.interfaceFont2 = path.join(self.imgFolder, 'FutureNarrow.ttf')
    self.buttonFont = path.join(self.imgFolder, 'PixelSquare.ttf')

    self.menuButtonSolid = pg.image.load(path.join(self.imgFolder, 'blue_button01.png')).convert_alpha()#Menu button images are loaded in
    self.menuButtonHighlight = pg.image.load(path.join(self.imgFolder, 'green_button01.png')).convert_alpha()

    self.menuImages = {}#menu background images are loaded in
    self.menuImages["startScreen"] =  pg.image.load(path.join(self.imgFolder, 'grey_background.jpg')).convert_alpha()
    self.menuImages["mainMenu"] =  pg.image.load(path.join(self.imgFolder, 'blue_background.jpg')).convert_alpha()
    self.menuImages["settingsMenu"] =  pg.image.load(path.join(self.imgFolder, 'Grey_yellow_background.jpg')).convert_alpha()
    self.menuImages["levelSelect"] =  pg.image.load(path.join(self.imgFolder, 'Grey_blue_background.jpg')).convert_alpha()
    self.menuImages["stats"] =  pg.image.load(path.join(self.imgFolder, 'Grey_green_background.jpg')).convert_alpha()
    self.menuImages["leaderboard"] =      pg.image.load(path.join(self.imgFolder, 'Grey_violet_background.jpg')).convert_alpha()
    self.menuImages["shop"] =  pg.image.load(path.join(self.imgFolder, 'Grey_orange_background.jpg')).convert_alpha()
    self.menuImages["pause"] =  pg.image.load(path.join(self.imgFolder, 'Grey_red_background.jpg')).convert_alpha()
    self.menuImages["levelComplete"] =  pg.image.load(path.join(self.imgFolder, 'blue_background.jpg')).convert_alpha()

    self.pauseScreen = pg.Surface(self.screen.get_size()).convert_alpha()#pause screen surface is created
    self.pauseScreenImage = pg.transform.scale(self.menuImages['pause'], (WIDTH, HEIGHT))

    self.pauseScreen = pg.Surface(self.screen.get_size()).convert_alpha()#pause screen surface is created
    self.pauseScreenImage = pg.transform.scale(self.menuImages['pause'], (WIDTH, HEIGHT))
    self.pauseIMGWhite = pg.image.load(path.join(self.imgFolder, 'pauseWhite.png')).convert_alpha()
    self.pauseIMGBlack = pg.image.load(path.join(self.imgFolder, 'pauseBlack.png')).convert_alpha()

    self.pauseIMGWhite = pg.image.load(path.join(self.imgFolder, 'pauseWhite.png')).convert_alpha()
    self.pauseIMGBlack = pg.image.load(path.join(self.imgFolder, 'pauseBlack.png')).convert_alpha()


    self.bikeImage = pg.image.load(path.join(self.imgFolder, 'bike.png')).convert_alpha()#bike image loaded in
    self.bikeImgSize = self.bikeImage.get_size()
    self.bikeImage = pg.transform.scale(self.bikeImage, (int(round(self.bikeImgSize[0]/10,0)), int(round(self.bikeImgSize[1]/10,0))))#image scaled down

    self.playerImage = self.carImage#default vehicle set to car

    if self.playerImage == self.carImage:
        self.imageType = "car"
    elif self.playerImage == self.bikeImage:
        self.imageType = "bike"

    self.loadQuestions()#questions are loaded
    self.questionSurface = pg.Surface(self.screen.get_size()).convert_alpha()#question surface is created
    self.loadstatsData()#statistics data is loaded

    self.menuMusic = path.join(self.sndFolder, 'backgroundMusic.ogg')#game music is loaded in
    self.pauseMusic1 = path.join(self.sndFolder, 'PauseMusic.ogg')
    self.pauseMusic2 = path.join(self.sndFolder, 'PauseMusic2.ogg')
    self.levelMusic1 = path.join(self.sndFolder, 'LevelMusic.ogg')
    self.levelMusic2 = path.join(self.sndFolder, 'LevelMusic2.ogg')
    self.levelMusic = self.levelMusic2
    self.currentLevelMusic = "electric"
    self.musicOn = True

    self.coinImages = {}#coin images are loaded in
    for img in range(len(COIN_IMAGES)):
        image = pg.image.load(path.join(self.imgFolder, COIN_IMAGES[img])).convert_alpha()
        size = image.get_size()
        self.coinImages[img] = pg.transform.scale(image, (int(size[0]/2), int(size[1]/2)))

    self.questionImage = pg.image.load(path.join(self.imgFolder, QUESTION_IMAGE)).convert_alpha()#question box image loaded in
```

'loadQuestions' function

```
def loadQuestions(self):#questions are loaded in
    questionCSV = path.join(self.gameFolder, 'questions.csv')#opens CSV file
    with open(questionCSV, 'r') as questionFile:
        reader = csv.reader(questionFile)#stores CSV data locally
        next(questionFile)#skips top column containing titles
        self.questionData = []
        self.questionID = []
        for line in reader:#loops through CSV file and seperates data
            temp = []
            questionID = int(line[0])
            question = str(line[1])
            cAns = (line[2])
            wAns1, wAns2, wAns3, wAns4 = (line[3]),(line[4]),(line[5]),(line[6])
            diff = str(line[7])
            level = int(line[8])
            isMaj = str(line[9])
            self.questionID.append(questionID)
            temp.extend((questionID,question,cAns,wAns1,wAns2,wAns3,wAns4,diff,level,isMaj))
            self.questionData.append(temp)#data stored in this 2D array
    questionFile.close()#file is closed to avoid altering
```

'drawText' function

```
def drawText(self, text, size, colour, x, y, surf=None, align=None, fontName=None):#draws text to screen
    if surf == None:#sets text surface if not provided
        surf = self.screen
    if fontName == None:#sets text font if not provided
        font = pg.font.SysFont('arial', size)
    else:
        font = pg.font.Font(fontName, size)

    textSurface = font.render(text, True, colour)#renders text to surface
    textRect = textSurface.get_rect()

    if align == None:
        textRect.center = (int(x),int(y))#aligns the text to the center

    surf.blit(textSurface, textRect)#blits text to the screen
```

'new' function

```
def new(self):#initialises game variables
    self.allSprites = pg.sprite.LayeredUpdates()#All sprite groups are created and blitted in a specific order
    self.buttons = pg.sprite.Group()
    self.platforms = pg.sprite.Group()
    self.walls = pg.sprite.Group()
    self.rectangles = pg.sprite.Group()
    self.questionItems = pg.sprite.Group()
    self.endWalls = pg.sprite.Group()
    self.coins = pg.sprite.Group()
    self.userInputBox = pg.sprite.Group()

    self.map = None#map object variables are created
    self.tiledMap = None
    self.tiledMapImg = None
    self.tiledMapRect = None

    self.camera = None#camera object variable is created
    self.sceneMan = sceneManager(self)#an instance of the SceneManager class is created
    self.player = None#player object cariable is created

    self.paused = False#pause screen variables are created
    self.pauseScreenPrinted = False
    self.askQuestion = False#question screen variables are created
    self.questionScreenPrinted = False

    self.score = 0#score is initialised as zero
    self.delay = False

    self.lastCountdownTime = 0
    self.settingsQuestionDiff = "easy"#question difficulty set to 'easy' by default
    self.questionID = []#contains chosen question numbers
    for i in range(len(self.questionData)):
        if self.questionData[i][7] == self.settingsQuestionDiff:
            if self.questionData[i][9] == 'FALSE':#checks chosen question is not a 'major' question
                self.questionID.append(self.questionData[i][0])#question number adeded to 'questionID' array
    self.majorQuestion = False
    self.majorQuestionCorrect = True

    if self.musicOn:
        pg.mixer.music.load(self.menuMusic)#loads menu music
        pg.mixer.music.play(loops=-1)#plays menu music
    self.menuMusicStarted = True#music variables are changed appropriately
    self.pauseMusicStarted = False
    self.levelMusicStarted = False

    self.sceneMan.loadLevel('startScreen')#start screen is looded
    self.run()#once start screen is exited, 'run' function is called
```

'run' function

```
def run(self):#main game loop
    self.playing = True
    while self.playing:#keeps calling 'events', 'update' and 'draw' function until game is closed
        self.dt = self.clock.tick(FPS) / 1000 #get the time of the previous frame in seconds
        self.events()
        self.update()
        self.draw()
```

'events' function

```
def events(self):#Gets user inputs
    for event in pg.event.get():#Loops throguh detected user inputs
        if event.type == pg.QUIT:#check for closing the window
            self.updateStatsData()#statistics data is updated
            if self.playing:
                self.playing = False#sets these variables to FALSE to close the game
                self.running = False

        for box in self.userInputBox:
            box.inputKeys(event)#'inputKeys' function on 'InputBox' object is called

        if event.type == pg.KEYDOWN:#is a key is pressed
            if event.key == pg.K_SPACE:#if the space bar is pressed
                if self.sceneMan.currentScene in LEVEL_SCREENS:
                    self.player.jump()#call the player object's jump' function

            if event.key == pg.K_p:#if the 'P' button is pressed
                if self.sceneMan.currentScene in LEVEL_SCREENS:
                    if not self.askQuestion:
                        self.paused = not self.paused#pause the game screen
                        if self.paused == False:#if unpauusing
                            for button in self.buttons:
                                button.kill()
                            self.pauseScreenPrinted = False
                            if self.musicOn:#stop the previously playing music
                                pg.mixer.music.fadeout(500)
                                self.pauseMusicStarted = False
                                self.levelMusicStarted = False
```

‘update’ function

```
def update(self):#Calculates and makes the changes to the current frame
    self.sceneMan.update()#call 'update' function on SceneManager object
    if not self.paused:#only updates if game is not paused

        self.allSprites.update()#call 'update' function on al sprites obejcts in 'allSprites' group
        self.userInputBox.update()#updates Input Box object

        if self.sceneMan.currentScene not in MENU_SCREENS:#following code only executed if in level screens

            if self.musicOn and self.levelMusicStarted == False:#starts level music if not already
                pg.mixer.music.fadeout(500)
                pg.mixer.music.load(self.levelMusic)
                pg.mixer.music.play(loops=-1)
                self.levelMusicStarted = True
                self.menuMusicStarted = False

            self.camera.update(self.player)#updates the Camera object using the Player object's position

            hitPlatform = pg.sprite.spritecollide(self.player, self.platforms, False)#checks if collided with platform
            if hitPlatform:#if collided
                self.player.pos.y = hitPlatform[0].rect.y - self.player.height / 2#move player to top of platform
                self.player.vel.y = 0#stop player moving down

            hitQuestion = pg.sprite.spritecollide(self.player, self.questionItems, False)#checks if collided with question object
                self.player.vel.y = 0#stop player moving down

            hitQuestion = pg.sprite.spritecollide(self.player, self.questionItems, False)#checks if collided with question object
                if hitQuestion:#if still collided
                    self.askQuestion = True#ask a question
                    self.paused = True
                    if hitQuestion[0].major == True:#if collided question is a major question
                        self.majorQuestion = True#set this variable to TRUE

            hitCoin =  pg.sprite.spritecollide(self.player, self.coins, True)#checks if collided with coin object
            if hitCoin:#if collided
                self.coinAmount += 1#increase coin total by one
                self.sceneMan.coinCollected += 1#increase coin collected by one
                self.calculateScore("coin")#increase the level's score

            hitLevelEnd = pg.sprite.spritecollide(self.player, self.endWalls, True)#checks if collided with endWall object
            if hitLevelEnd:#if collided
                self.totalScore += self.score#update 'totalScore' variable with achieved score
                self.majorQuestion = False#reset this variable to FALSE

                self.sceneMan.secondPrevScence = self.sceneMan.prevScence#set scene variables
                self.sceneMan.prevScence = self.sceneMan.currentScene
                self.sceneMan.currentScene = "levelComplete"

                self.sceneMan.showPlayer = False#hide the player object on the screen
                self.sceneMan.loadLevel("levelComplete")#Launch the LevelCompelte screen

        else:#if current scene not in level screens
            if self.musicOn == False:#if user has turned off music
                pg.mixer.music.fadeout(500)#stop music
                self.menuMusicStarted = False

            if self.musicOn == True and self.menuMusicStarted == False:#if menu music hasn't started
                self.levelMusicStarted = False
                pg.mixer.music.fadeout(500)#stop level music
                self.menuMusicStarted = True
                pg.mixer.music.load(self.menuMusic)#load menu music
                pg.mixer.music.play(loops=-1)#play menu music
```

'getQuestion' function

```
def getQuestion(self):#Obtains a question from 2D array
    self.answerCorrect = False#initialises correct, clicked and chosen answer variables
    self.answerClicked = False
    self.selectedAns = None

    self.startTime = pg.time.get_ticks()#starts question timer

    if self.majorQuestion == True:#if a major question is needed
        self.questionID = []#questions numbers are re-calculated using chosen difficulty
        for i in range(len(self.questionData)):
            if self.questionData[i][9] == 'TRUE':
                if self.questionData[i][7] == self.settingsQuestionDiff:
                    self.questionID.append(self.questionData[i][0])
    else:#if a major question is not required
        self.questionID = []#all questions of the chosen difficulty are added to this array
        for i in range(len(self.questionData)):
            if self.questionData[i][7] == self.settingsQuestionDiff:
                self.questionID.append(self.questionData[i][0])

    questionNumber = random.choice(self.questionID)#a random question number is selected
    wrongAns = [3,4,5,6]#indexs of incorrect answers
    chosenAns = [2,]#indexs of answers which will be displayed

    self.questionID.remove(questionNumber)#question number removed from 'questionID' array
    self.questionNumberIndex = questionNumber - 1#index of chosen question is stored
    correctAns = self.questionData[self.questionNumberIndex][2]#correct answer is stored

    for i in range(3):#three random incorrect answer indexs are chosen
        randomIndex = random.choice(wrongAns)
        wrongAns.remove(randomIndex)
        chosenAns.append(randomIndex)#chosen indexs are added to 'chosenAns' array

    random.shuffle(chosenAns)#the order of the indexs in 'chosenAns' is shuffled

    self.drawText(str(self.questionData[self.questionNumberIndex][1]), 30, HUD_COLOUR, WIDTH/2,\
                  HEIGHT*2/9, surf=self.questionSurface)#the question is printed to the screen

    answer1 = Button(self, str(chosenAns[0]), WIDTH/3, HEIGHT/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, \
                     str(self.questionData[self.questionNumberIndex][chosenAns[0]]))#The four answers buttons are created
    answer2 = Button(self, str(chosenAns[1]), WIDTH*2/3, HEIGHT/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE,\
                     str(self.questionData[self.questionNumberIndex][chosenAns[1]]))#They will be added to the allSprites group
    answer3 = Button(self, str(chosenAns[2]), WIDTH/3, HEIGHT*2/3, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE,\
                     str(self.questionData[self.questionNumberIndex][chosenAns[2]]))#and therefore be drawn to the screen
    answer4 = Button(self, str(chosenAns[3]), WIDTH*2/3, HEIGHT*2/3, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE,\
                     str(self.questionData[self.questionNumberIndex][chosenAns[3]]))
```

'getTimeAllowe' function

```
def getTimeAllowed(self):#calculates the time allowed for question
    if self.questionData[self.questionNumberIndex][7] == "easy":#checks the difficulty of the chosen question
        self.questionDiff = "easy"#stores the difficulty for use when calculating statistics
        self.timeAllowed = TIME_FOR_EASY_Q#'timeAllowed' variable is set to pre-determined value for this difficulty questions
    if self.questionData[self.questionNumberIndex][7] == "medium":
        self.questionDiff = "medium"
        self.timeAllowed = TIME_FOR_MEDIUM_Q
    if self.questionData[self.questionNumberIndex][7] == "hard":
        self.questionDiff = "hard"
        self.timeAllowed = TIME_FOR_HARD_Q

    self.timeRemaining = self.timeAllowed#timer allowed value copied to 'timeRemaining' variable
    self.timeOut = False#question over variable set to FALSE
```

'calculateScore' function

```
def calculateScore(self, type):#calculates the score

    if type == "answer":#when a question has been answered
        self.timeTaken = int(round((self.endTime - self.startTime) / 1000, 0))#time taken for the question is calculated in seconds
        multiplier = random.choice([2,2,2,2,3,3,3,3,3,4,4,4,4,4,4,5,5,5,5,5,5,6,6,6,6,6,7,7,7,7,8,8,8,8,9,9,10,15])
        #random mulitplier integer is chosen
        scoreIncrease = int(round((self.timeAllowed - self.timeTaken) * multiplier, 0))
        #score for the question is calcLated and rounded to nearest integer
        self.score += scoreIncrease#calculated score is added to 'score' variable

    elif type == "coin":#when a coin has been collected
        self.score += 10#score increased by 10
```

'countdownTimer' function

```
def countdownTimer(self):#updates the countdown timer for question
    now = pg.time.get_ticks()#gets the current time

    if now - self.lastCountdownTime > 1000 and self.timeRemaining >= 0:#if one seconds has passed and time still remaining
        self.questionSurface.fill(WHITE, (WIDTH*7/8-80, HEIGHT/8-25, 160, 50))#fill area for timer on screen with white
        self.lastCountdownTime = now#set last updated time to current time

        self.drawText("Time Remaining: {}".format(self.timeRemaining), 20, HUD_COLOUR, WIDTH*7/8, HEIGHT/8, surf=self.questionSurface)
        #Blit the 'Time Remaining' text and the time remaining value to screen
        self.timeRemaining -= 1#reduced remaining time by one

    if self.timeRemaining == -1:#when the time reaches -1
        self.timeOut = True#time is finish so this variable is set to TRUE
        self.answerClicked = True
        if self.majorQuestion:
            self.majorQuestionCorrect = False
```

‘draw’ function

```
def draw(self):#Draws the updated frame to the screen
    pg.display.set_caption("{:.2f}".format(self.clock.get_fps()))
    #sets the current frame rate as the game window's caption

    if self.paused:#if the game is paused
        if self.musicOn and self.pauseMusicStarted == False and self.askQuestion == False:#if question isn't being asked
            self.pauseMusicStarted = True#if the pause music if not being played
            self.levelMusicStarted = False
            option = random.choice([0,1])#randomly choose a pause screen music
            if option == 1:
                pg.mixer.music.load(self.pauseMusic1)
            else:
                pg.mixer.music.load(self.pauseMusic2)
            pg.mixer.music.play(loops=-1)#play the chosen music

        if self.askQuestion:#if a question needs to be asked
            if self.questionScreenPrinted == False:#if the question surface has not been displayed
                self.questionSurface.fill(0)#clear the surface then blit the text 'Question asking' to the surface
                self.drawText("Question asking", 25, HUD_COLOUR, WIDTH/2, HEIGHT*1/9, surf=self.questionSurface,\
                              fontName=self.interfaceFont)
                self.getQuestion()#get a random question
                self.getTimeAllowed()#get the time allowed for the question
                self.questionScreenPrinted = True

            for button in self.buttons:#update the buttons on the question surface
                button.draw(self.questionSurface)

            if self.answerClicked:#if an answer is clicked
                self.questAnswered += 1#increases this variable for statistics data

                self.endTime = pg.time.get_ticks()#end the question timer

                if self.answerCorrect:#if correct answer has been chosen
                    if self.questionDiff == "easy":#increase appropriate question answered variable
                        self.correctAnswerQuesEasy += 1
                    elif self.questionDiff == "medium":
                        self.correctAnswerQuesMed += 1
                    elif self.questionDiff == "hard":
                        self.correctAnswerQuesHard += 1

                    self.calculateScore("answer")#calulate the score for the answer
                    self.drawText("Correct Answer", 35, GREEN, WIDTH/2, HEIGHT*5/6, surf=self.questionSurface)
                    #display 'Correct Answer' to the question surface

                elif self.timeOut:#if the player has run out of time
                    self.drawText("Time Out", 25, RED, WIDTH/2, HEIGHT*5/6, surf=self.questionSurface)
                    self.drawText("Correct Answer was {}".format(self.questionData[self.questionNumberIndex][2]), 20,\
                                  HUD_COLOUR, WIDTH/2, HEIGHT*5/6 + 50, surf=self.questionSurface)
                    #'Time Out' and the correct answer is displayed to the question surface

                    for button in self.buttons:#the buttons on the question surface are removed
                        button.kill()
```

```
            else:#if chosen answer is incorrect
                if self.majorQuestion:#if answered question was a 'major' question
                    self.majorQuestionCorrect = False#this variable is set to FALSE

                self.drawText("Incorrect Answer", 25, RED, WIDTH/2, HEIGHT*5/6, surf=self.questionSurface)
                self.drawText("Correct Answer was {}".format(self.questionData[self.questionNumberIndex][2]), 20,\
                              HUD_COLOUR, WIDTH/2, HEIGHT*5/6 + 50, surf=self.questionSurface)
                #'Incorrect Answer' and the correct answer is displayed to the question surface

            self.screen.blit(self.questionSurface, self.questionSurface.get_rect())
            pg.display.flip()#question surface is blitted to screen and then screen is updated

            pg.time.delay(1500)#game is paused for 1.5 seconds

            self.askQuestion = False
            self.questionScreenPrinted = False
            self.paused = False#game is unpuased

        else:#if an answer has not been clicked yet
            self.countdownTimer()#this function is called to update the remaining time

        self.screen.blit(self.questionSurface, self.questionSurface.get_rect())

    else:#if a question doens't need to be asked
        if self.pauseScreenPrinted == False:
            self.sceneMan.loadLevel('pause')#Loads the pause screen
            self.pauseScreenPrinted = True

        for button in self.buttons:#draws buttons on pause screen
            button.draw(self.pauseScreen)

        self.screen.blit(self.pauseScreen, self.pauseScreen.get_rect())#blits pause screen to the game screen

else:#if the game is not paused

    for button in self.buttons:
        button.draw(self.screen)#draw the buttons
    for box in self.userInputBox:
        box.draw(self.screen)#draw the 'InputBox' object

    if self.sceneMan.currentScene not in MENU_SCREENS:#if not in menu screens

        self.screen.blit(self.tiledMapImg, self.camera.applyOffsetRect(self.tiledMapRect))
        #blit the map image to the screen

        for sprite in self.allSprites:
            self.screen.blit(sprite.image, self.camera.applyOffset(sprite))
            #draw all game sprites in new, offset-applied position

        self.drawText("Score: " + str(self.score), 22, HUD_COLOUR, WIDTH-100, 15)#draw Score and Coin total values
        self.drawText("Coin: " + str(self.coinAmount), 22, HUD_COLOUR, 100, 15)# to the screen

pg.display.flip()#update the display
```

Procedural code

This is the code that was explained on page 70 which creates an instance of the game class in order to run the game.

```
498 g = Game()#create and instance of the 'Game' class
499
500 while g.running:#whilst 'running' variable is TRUE
501     g.new()#call the 'new' function
502
503 pg.quit()#when while loop has stopped, call 'quit' function to unitialse and exit pygame
```

Sprites.py script

Importing the modules

```
import pygame as pg#pygame and settings file are imported
from settings import *

vec = pg.math.Vector2#vector function stored as 'vec'
```

Player Class:

'init' function

```
class Player(pg.sprite.Sprite):#Player class is created and inherits a pygame sptite class

    def __init__(self, game, x, y):#instance of game and postion coordinates needed to create object

        self._layer = PLAYER_LAYER#Layer variable set to layer value for player
        self.groups = game.allSprites#added to 'allSprites' group
        pg.sprite.Sprite.__init__(self, self.groups)#'init' function on inherited class called

        self.game = game
        self.image = self.game.playerImage.copy()#copy of player image obtained and stored
        self.width = self.image.get_size()[0]#dimension of image are calculated
        self.height = self.image.get_size()[1]
        self.rect = self.image.get_rect()
        self.rect.center = (x,y)#player image moved to provided coordiantes

        self.pos = vec(x,y)#position, velocity and acceleration vectors created for player
        self.vel = vec(0,0)
        self.acc = vec(0,0)

        self.direction = "forward"#direction set to forward by default
        self.braking = False#'braking' variable initialsed as false
```

'collideWithWalls' function

```
def collideWithWalls(self, dir):#checks collision between player and wall objects

    hitWall = pg.sprite.spritecollide(self.game.player, self.game.walls, False)#checks for collision with a wall
    if hitWall:#if colided
        if dir == "x":#if checking horizontally
            if self.game.player.vel.x > 0:#going to the right  [hit left of wall]
                self.game.player.pos.x = hitWall[0].rect.left - self.game.player.width / 2#move player to left of wall
            elif self.game.player.vel.x < 0:#going to the left [hit right of wall]
                self.game.player.pos.x = hitWall[0].rect.right + self.game.player.width / 2#mvoe player to right of wall
            self.game.player.vel.x = 0#set horizontal velocity to zero
            self.rect.centerx = self.pos.x

        if dir == "y":#if checking vertically
            if self.game.player.vel.y > 0:#going down [hit top of wall]
                self.game.player.pos.y = hitWall[0].rect.top - self.rect.height / 2#move player to top of wall
            elif self.game.player.vel.y < 0:#going up [hit bottom of wall]
                self.game.player.pos.y = hitWall[0].rect.bottom + self.game.player.height / 2#move player to bottom of wall
            self.game.player.vel.y = 0#set vertical velocity to zero
            self.rect.centery = self.pos.y
```

'jump' function

```
def jump(self):#jump function
    self.vel.y = -20#player's y velcoity change to create jumping effect
```

'update' function

```
def update(self):#calcualtes the new position of the player
    if self.game.sceneMan.showPlayer == True:#new posiion only calcualted if player needs to be showed

        self.acc = vec(0, PLAYER_GRAV)#apply the force of gravity to the player
        keys = pg.key.get_pressed()#get the pressed keys in the last frame

        if keys[pg.K_RIGHT] or keys[pg.K_d]:#forward motion
            if self.direction == "forward" or self.direction == "stopped":#checks if already in that motion
                if self.braking == False:#only moves forward if not also braking
                    self.acc.x = PLAYER_ACC#sets the player's acceleration
                    self.direction = "forward"#sets the player's direction

        if keys[pg.K_LEFT] or keys[pg.K_a]:#backwards motion
            if self.direction == "reverse" or self.direction == "stopped":
                if self.braking == False:
                    self.acc.x = -PLAYER_ACC
                    self.direction = "reverse"

        if abs(self.vel.x) < 0.5:#sets the direction to stopped if x velocity is low
            self.direction = "stopped"

        if keys[pg.K_DOWN] or keys[pg.K_s]:#braking motion
            if self.direction != "stopped":
                if self.vel.x > 0:
                    self.acc.x = -BRAKE_ACC#applies braking accleration in appropriate direction
                else:
                    self.acc.x = BRAKE_ACC
            self.braking = True
        else:
            self.braking = False

        self.acc += self.vel * PLAYER_FRICTION#SUVAT equations are used to calculate
        self.vel += self.acc#the player's new postion
        self.pos += self.vel + 0.5 * self.acc

        self.rect.centerx = self.pos.x#collisions with the walls are checked
        self.collideWithWalls('x')#in the x direction
        self.rect.centery = self.pos.y
        self.collideWithWalls('y')#and the y direction
```

Platform Class

```
class Platform(pg.sprite.Sprite):#Player can stand on top of these objects
    def __init__(self, game, x, y, width, height, colour, mode="platform", show=True):
        self._layer = PLATFORM_LAYER#layer variable set to layer value for platform
        self.mode = mode

        if show == True:#displays platforms to the screen
            self.groups = game.allSprites, game.platforms#by adding to allSprites group
            pg.sprite.Sprite.__init__(self, self.groups)
            self.image = pg.Surface((width, height))#surface is created for the visible object
            self.image.fill(colour)#surface filled with provided colour
            self.rect = self.image.get_rect()

        elif show == False:#if not going to be shown
            self.groups =  game.platforms#object not added to allSprites group
            pg.sprite.Sprite.__init__(self, self.groups)
            self.rect = pg.Rect(x, y, width, height)#rectangle is created for the platform

        if self.mode == "platform":#the positioal coordiantes of the object are changed
            self.rect.x, self.rect.y = x * TILESIZE, y * TILESIZE#depending on the mode
        elif self.mode == "track":
            self.rect.x, self.rect.y = x, y
```

Wall Class

```
class Wall(pg.sprite.Sprite):#Player can't move through these
    #similar to Platform but needs to be added to a different group for collision detection
    def __init__(self, game, x, y, colour=None, altColour=None, mode=None, mode2=None, width=None, height=None):
        self._layer = WALL_LAYER#layer variable set to layer value for wall
        if mode == "tiled":#for maps craeted using 'Tiled'
            if mode2 == "end":
                self.groups = game.endWalls#add to enWalls group if one
            else:
                self.groups = game.walls#add all other walls to walls group
            pg.sprite.Sprite.__init__(self, self.groups)
            self.rect = pg.Rect(x, y, width, height)#rectangle object craeted for wall
            self.rect.x, self.rect.y = x, y#position of rectangle set to provided coordiantes
```

Rectangle Class

```
class Rectangle(pg.sprite.Sprite):#Used for filling in the area under the track
    #when using tracks made frmo bitmpa images
    def __init__(self, game, x, y, width, height, colour):
        self._layer = RECTANGLE_LAYER#Layer variable set to layer value for rectangle
        self.groups = game.allSprites, game.rectangles
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((width, height))#visible surface is created
        self.image.fill(colour)#surface filled in with provided colour
        self.rect = self.image.get_rect()
        self.rect.x, self.rect.y = x, y#moved to provided position coordinates
```

Track Class

```
class Track(pg.sprite.Sprite):#used for creating small track objects for text file maps
    #and bitmap images. #This code was added to the Wall class for 'Tiled' maps
    def __init__(self, game, x, y, width, height, colour):
        self._layer = TRACK_LAYER#Layer variable set to layer value for track
        self.groups = game.allSprites, game.platforms
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((width, height))#a visible surface is created
        self.image.fill(colour)#surface filled with provided colour
        self.rect = self.image.get_rect()
        self.rect.x, self.rect.y = x , y#object moved to provided coordinates
```

WallFile Class

```
class WallFile(pg.sprite.Sprite):#Used by bitmap images to add wall objects one pixel wide
    #Diifferent to Wall class by not mulitpling the poositon coordinates by TILESIZE
    #This class is no longer used due to the 'mode' parameter of the Wall class
    def __init__(self, game, x, y, width, height, colour):
        self.groups = game.allSprites, game.walls
        pg.sprite.Sprite.__init__(self, self.groups)
        self.image = pg.Surface((width, height))
        self.image.fill(colour)
        self.rect = self.image.get_rect()
        self.rect.x, self.rect.y = x, y
```

Question Class:

'init' function

```
class Question(pg.sprite.Sprite):#Creates Question objects
    def __init__(self, game, x, y, major=False):
        self._layer = QUESTION_LAYER#Layer variable set to layer value for question
        self.groups = game.allSprites, game.questionItems
        pg.sprite.Sprite.__init__(self, self.groups)

        self.image = game.questionImage.copy()#takes a copy of the question box image
        self.image = pg.transform.scale(self.image, (64, 64))#resizes the image
        size = self.image.get_size()
        self.width, self.height = size[0], size[1]#stores the dimensions of the image
        self.rect = self.image.get_rect()

        self.x, self.y = x, y
        self.rect.center = (self.x, self.y)#sets the position of the question box

        self.velY = 1#initialises velocity and 'major' variables
        self.major = major
```

'update' function

```
def update(self):#ainimates the question box
    self.rect.y += self.velY#updates the question box's y position
    if self.rect.top > self.y - self.height/4:#moves the box downwards when it surpases a set height
        self.velY = -1
    if self.rect.bottom < self.y + self.height/4:#moves the box upwards when it travels down
        self.velY = 1#beyond a set height
```

Coin Class:

'init' function

```
class Coin(pg.sprite.Sprite):#Creates Coin objects
    def __init__(self, game, x, y):
        self._layer = COIN_LAYER#Layer variable set to layer value for coin
        self.groups = game.allSprites, game.coins
        pg.sprite.Sprite.__init__(self, self.groups)
        self.game = game
        self.x = x#stores the coordinates in class variables
        self.y = y

        self.currentFrame = 0#initalises the currentFrame and lastUpdated variables
        self.lastUpdated = 0
```

'update' function

```
def update(self):#animates the coin object
    now = pg.time.get_ticks()#get the current time
    if now - self.lastUpdated > 200:#updates the coin's position every 0.2 seconds
        self.lastUpdated = now
        self.currentFrame = (self.currentFrame + 1) % len(self.game.coinImages)#calculates the new coin image
        self.image = self.game.coinImages[self.currentFrame]#gets the new coin image
        self.rect = self.image.get_rect()
        self.rect.center = (self.x, self.y)#repositions the new image
```

InputBox Class:

'init' function

```
class InputBox(pg.sprite.Sprite):#Creates an input box object
    def __init__(self, game, x, y, width, height, text='', textSize=32, fontName=None):
        self._layer = INPUT_BOX_LAYER#layer variable set to layer value for input box
        self.groups = game.userInputBox
        pg.sprite.Sprite.__init__(self, self.groups)
        self.game = game
        self.rect = pg.Rect(x, y, width, height)#creats a rectangle object for the input box
        self.colour = RED#set the default colour to red
        self.text = text
        if fontName == None:#sets the font name if not provided
        self.colour = RED#set the default colour to red
        self.text = text
        if fontName == None:#sets the font name if not provided
        self.textSurf = self.font.render(text, True, self.colour)#creates the text surface
        self.boxActive = False#initilaises the 'boxActive' variable
```

'update' function

```
def update(self):#change the width of the box if string length increases beyond box width
    newWidth = max(150, self.textSurf.get_width()+10)#calculate new box width
    self.rect.width = newWidth#set new width
```

'draw' function

```
def draw(self, surf):#draws the text to the box
    surf.blit(self.textSurf, (self.rect.x+5, self.rect.y+5))#blits the text to the screen
    pg.draw.rect(surf, self.colour, self.rect, 2)#draws the input box rectangle
```

'inputKeys' function

```
def inputKeys(self, event):#gets the keys typed in the input box
    if event.type == pg.MOUSEBUTTONDOWN:
        if self.rect.collidepoint(event.pos):#chesk if the mouse has been clicked inside the input box
            self.boxActive = not self.boxActive#if so reverse the state of 'boxActive'
        else:#if clicked outside the box
            self.boxActive = False#make 'boxActive' FALSE
        if self.boxActive:#change the colour of the box depedning on 'boxActive'
            self.colour = GREEN
        else:
            self.colour = RED

    if event.type == pg.KEYDOWN:
        if self.boxActive:#only add inputted keys if input box is active

            if event.key == pg.K_RETURN:#if return key is pressed
                if self.game.sceneMan.currentScene == 'levelComplete':#if current scene is 'levelComplete'
                    self.game.sceneMan.updateHighscore(self.game.sceneMan.nextLevelTagNumber-1, self.game.score)
                    #update the Highscore - the provided name is used inside that function
                self.text = ''#reset the text variable

            elif event.key == pg.K_BACKSPACE:#if backspce button is clicked
                self.text = self.text[:-1]#remove the previously entered character

            else:#for any other key input
                self.text += event.unicode#add the key of the text variable

            self.game.screen.fill(WHITE, (WIDTH*5/8, HEIGHT*7/11, self.rect.width+10, 32))
            #clear the input box to remove the previous text
            self.textSurf = self.font.render(self.text, True, self.colour)#re-render the text Surface
```

Button Class:

'init' function

```
class Button(pg.sprite.Sprite):#Creates button objects
    def __init__(self, game, tag, x, y, width, height, solidColour, highlightColour, text, textSize=None, \
                 solidButtonImage=None, highlightButtonImage=None):
        self._layer = BUTTON_LAYER#Layer variable set to layer value for button
        self.groups = game.buttons
        pg.sprite.Sprite.__init__(self, self.groups)

        self.game = game#stores the parameters as class variables
        self.tag = tag
        self.width = width
        self.height = height
        self.solidColour = solidColour
        self.highlightColour = highlightColour
        self.text = text

        if textSize == None:#sets the text size variable if not provided
            self.textSize = int(width/(len(text)*0.45))
            if self.textSize > 24:
                self.textSize = 24
        else:
            self.textSize = textSize

        self.pos = vec(x,y)#positon vector is created
        self.rectDimensions = (self.pos[0]-self.width/2, self.pos[1]-self.height/2, self.width, self.height)
        #dimensions for the button rectangle are created and stored in a class variable

        if solidButtonImage == None:#sets the image of the button if not provided
            self.solidImage = pg.transform.scale(self.game.menuButtonSolid,(int(self.width), int(self.height)))
            self.highlightImage = pg.transform.scale(self.game.menuButtonHighlight,(int(self.width), int(self.height)))
        else:
            self.solidImage = pg.transform.scale(solidButtonImage,(int(self.width), int(self.height)))
            self.highlightImage = pg.transform.scale(highlightButtonImage,(int(self.width), int(self.height)))

        self.image = self.solidImage#image set by default to solid image
        self.rect = self.image.get_rect()
        self.rect.centerx = x#button image is positioned to provided coordinates
        self.rect.centery = y

        self.clicked = False#initialises 'clicked', 'colchange' and 'first' variables
        self.colchange = False
        self.first = True
```

'highlight' function

```
def highlight(self):#changes the colour of button if mouse if over it
    mousePos = pg.mouse.get_pos()#gets the current mouse position

    if self.pos[0] - self.width/2  <= mousePos[0] <= self.pos[0] + self.width/2 and \
        self.pos[1] - self.height/2 <= mousePos[1] <= self.pos[1] + self.height/2:
        #checks if mouse cursor is within the button's boundaries
        if self.image != self.highlightImage:#changes button image to highlighted image
            self.image = self.highlightImage#if not already that
            self.colchange = True#if the image has been changed, this variable is set to TRUE

        if pg.mouse.get_pressed()[0] == 1:#if clicked inside the button
            self.clicked = True#this variable is set to TRUE

    else:#if mouse cursor not in button's boundary
        if self.image != self.solidImage:#change the image to the sold image
            self.image = self.solidImage#if not alreay that image
            self.colchange = True
```

'update' function

```
def update(self):#update the button image
    self.highlight()#by calling 'highlight' function
    if self.clicked:#when a buton is clicked
        for button in self.game.buttons:
            if button != self:#apart from the clicked button
                button.kill()#delete all other current buttons
```

'draw' function

```
def draw(self, surface):#draw the new button image to the screen
    if self.colchange == True or self.first == True:#only draw button if colour has changed
        #or its the first time
        surface.blit(self.image,self.rect)#blit the button image to the screen

        if self.first:#if its the first time
            self.first = False#change this variable to FALSE

        self.game.drawText(self.text, self.textSize, BLACK, self.pos[0], self.pos[1], surf=surface)
        #draw the button text on the already blitted button image
        self.colchange = False#set this variable to FALSE as the button has now been blitted
```

Management.py script

Importing the modules

```
import pygame as pg#pygame, OS, PIL, pickle, datetime and pytmx modules are imported
from os import path
from PIL import Image
import pickle
import datetime
import pytmx

from settings import *#Two other game files are imported
from sprites import *
```

'Map' class

```
class Map:#Map class for maps created using textfile and bitmap images
    def __init__(self, filename, trackfile, bitSize=4):
        self.data = []#empty array is created for the map data
        with open(filename, 'rt') as file:#map file is opened in read mode
            for line in file:
                self.data.append(line.strip())#each line in the text file is added to array
        file.close()#file is closed to avoid accidental change

        self.tilewidth = len(self.data[0])#variables for the size of the map are created
        self.tileheight = len(self.data)
        self.width = self.tilewidth * TILESIZE
        self.height = self.tileheight * TILESIZE

        self.trackData = []#empty arrays are created to store track, wall, fill and question data
        self.wallData = []
        self.fillData = []
        self.questionData = []

        self.trackImage = Image.open(trackfile)#Image function from PIL library is used to open
        self.trackWidth = self.trackImage.size[0]#the bitmap image
        self.trackHeight = self.trackImage.size[1]#width and height of the bitmap image is stored

        for col in range(self.trackHeight):#each pixel in the bitmap image is analsyed
            for row in range(self.trackWidth):
                pixelData = self.trackImage.getpixel((row, col))#the data of the current pixel is obtained

                if bitSize == 1:#the bit size is checked as the value of the same colour will be different
                    if pixelData == 1:#if the pixel is black
                        self.trackData.append((row, col))#add the coodintaes of the pixel to the trackData array

                else:
                    if pixelData == 0:#if the pixel is black
                        self.trackData.append((row, col))

                if bitSize == 4:
                    if pixelData == 5:#if the pixel correlates to a wall
                        self.wallData.append((row, col))#add the pixel coordinates to the wallData array

                    if pixelData == 12:#if the pixel correlates to a fill in rectangle
                        self.fillData.append((row, col))#add the pixel coordinates to the fillData array

                    if pixelData == 2:#if the pixel correlates to a question
                        self.questionData.append((row, col))#add the pixel coordinates to the questionData array
```

'TiledMap' class:

'init' function

```
class TiledMap():#Creates maps using 'Tiled' files
    def __init__(self, filename):
        self.tiledMapFile = pytmx.load_pygame(filename, pixelaplha=True)#loads the map using pytmx library

        self.width = self.tiledMapFile.width * self.tiledMapFile.tilewidth#gets dimensions of map
        self.height = self.tiledMapFile.height * self.tiledMapFile.tileheight
```

'render' function

```
def render(self, surface):#gets tile image and displays it
    tileImage = self.tiledMapFile.get_tile_image_by_gid#function to get tile image is stored in a shortened variable

    for layer in self.tiledMapFile.visible_layers:#all layers in file are iterated
        if isinstance(layer, pytmx.TiledTileLayer):#if the layer is a tile layer

            for x, y, gid, in layer:#each tile on the image is iterated
                tile = tileImage(gid)#the image of the tile is obtained

                if tile:#if there is an image, it is blitted to the screen in the appropriate position
                    surface.blit(tile, (x * self.tiledMapFile.tilewidth- TILESIZE, y * self.tiledMapFile.tileheight - TILESIZE))
```

'makeMap' function

```
def makeMap(self):#map image is created using the 'render' function
    tempSurface = pg.Surface((self.width, self.height))
    self.render(tempSurface)
    return tempSurface
```

'Camera' class:

'init' function

```
class Camera:#Moves the camera depending on the player's position
    def __init__(self, width, height):
        self.camera = pg.Rect(0, 0, width, height)#rectangle is created depedning on the size of the map
        self.width = width - TILESIZE
        self.height = height
```

'applyOffset' function

```
def applyOffset(self, item):#used to calculate the new position of an object depending on the new
    return item.rect.move(self.camera.topleft)#position of the player
```

'applyOffsetRect' function

```
def applyOffsetRect(self, rect):#used to calculate the new position of the rect attribute of an
    return rect.move(self.camera.topleft)#object depending on the position of the player
```

'update' function

```
def update(self, target):#calculates the new position of the camera
    x = -target.rect.centerx + int(WIDTH/2)#the x and y coordinates of the camera on the map is adjusted
    y = -target.rect.centery + int(HEIGHT/2)

    #These four conditions are used to limit scrolling towards the sides of the map
    x = min(0, x)#left side
    y = min(0, y)#top side
    x = max(-(self.width - WIDTH), x)#right side
    y = max(-(self.height - HEIGHT), y)#bottom side

    self.camera = pg.Rect(x, y, self.width, self.height)#a new rect object is created using the camera's new position
```

‘sceneManager’ class

‘init’ function

```
class sceneManager():#Used to manage the scenes in the program
    def __init__(self, game):#an instance of the game is provided to create this class
        self.game = game#provided parameter stored as a class variable

        self.currentScene = 'startScreen'#currentScene set to 'startScreen' by default

        self.first = True#first, previous scene, second previous scene and updateed highscore
        self.prevScence = None#variables are initilaised
        self.secondPrevScence = None
        self.updatedHighscore = True
        self.leaderboardLoadLevel = "level1Score"#the default level in the leaderboard table set to level one
```

‘loadLevel’ function

```
def loadLevel(self, level=None):#this function is used to load level functions
    self.level = level
    if self.level == "settingsMenu":#the provided paramter is checked against set values
        self.settingsMenu()#if it matches, the appropriate function is called
    if self.level == "mainMenu":
        self.mainMenu()
    if self.level == "levelSelect":
        self.levelSelect()
    if self.level == "stats":
        self.stats()
    if self.level == "leaderboard":
        self.leaderboard()
    if self.level == "shop":
        self.shop()
    if self.level == "startScreen":
        self.startScreen()
    if self.level == "gameOverScreen":
        self.gameOverScreen()
    if self.level == "level1":
        self.level1()
    if self.level == "level2":
        self.level2()
    if self.level == "level3":
        self.level3()
    if self.level == "level4":
        self.level4()
    if self.level == "levelComplete":
        self.levelComplete()
    if self.level == "pause":
        self.pause()
    if self.level == "instructions":
        self.instructions()
```

‘mainMenu function

```
def mainMenu(self):#function for the main menu screen
    if self.image != pg.transform.scale(self.game.menuImages["mainMenu"], (WIDTH, HEIGHT)):#background image is loaded and re-sized
        image = pg.transform.scale(self.game.menuImages["mainMenu"], (WIDTH, HEIGHT))
        rect = image.get_rect()
        self.game.screen.blit(image, rect)

    self.game.drawText("Main Menu", 30, WHITE, WIDTH/2, HEIGHT*1/8, fontName=self.game.interfaceFont)#title is displayed

    #buttons for each game screen are created
    self.levelSelectButton = Button(self.game, "levelSelect", WIDTH*1/2, HEIGHT*3/8, WIDTH/3, HEIGHT/6, YELLOW, LIGHT_BLUE, "Select Level")
    self.settingsMenuButton = Button(self.game, "settingsMenu", WIDTH/4, HEIGHT*5/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Settings")
    self.shopButton = Button(self.game, "shop", WIDTH*3/4, HEIGHT*5/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Vehicles")
    self.leaderboardButton = Button(self.game, "stats", WIDTH/4, HEIGHT*7/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Stats")
    self.statsButton = Button(self.game, "leaderboard", WIDTH*3/4, HEIGHT*7/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Leaderboard")
```

‘settings’ function

```
def settingsMenu(self):#function for the settings screen
    self.image = pg.transform.scale(self.game.menuImages['settingsMenu'], (WIDTH, HEIGHT))#background image is obtained and re-szied
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)

    self.game.drawText("Settings", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)#title is blitted to the screen

    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/9, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)#back button is created

    #button to view the start screen and access the instructions page are created
    self.startScreenButton = Button(self.game, "startScreen", WIDTH/4, HEIGHT*3/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "View Start Screen")
    self.instructionsButton = Button(self.game, "instructions", WIDTH/4, HEIGHT*4/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Instructions")

    self.game.drawText("Question Difficulty: {}".format(self.game.settingsQuestionDiff.capitalize()), 18, \
                       WHITE, WIDTH*11/16, HEIGHT/4, fontName=self.game.interfaceFont2)#Question difficulty text is drawn with appropriate value

    self.easyButton = Button(self.game, "easy", WIDTH*7/10, HEIGHT*4/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Easy")#difficulty buttons are created
    self.mediumButton = Button(self.game, "medium", WIDTH*7/10, HEIGHT*5/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Medium")
    self.hardButton = Button(self.game, "hard", WIDTH*7/10, HEIGHT*6/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Hard")

    self.funkyMusicButton = Button(self.game, "upbeat", WIDTH/4, HEIGHT*8/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Upbeat")#music buttons are created
    self.electricMusicButton = Button(self.game, "electric", WIDTH/4, HEIGHT*9/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Electric")
    self.noMusicButton = Button(self.game, "noMusic", WIDTH/4, HEIGHT*10/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "No Music")

    if self.game.currentLevelMusic == "electric":#chosen music is analysed to calculate the text to display
        music = "electric"
    elif self.game.currentLevelMusic == "upbeat":
        music = "upbeat"
    else:
        music = "no music"

    #displays the currently chosen music or choice of no music
    self.game.drawText("Current Music: {}".format(music.capitalize()), 18, WHITE, WIDTH/4, HEIGHT*7/12, fontName=self.game.interfaceFont2)
```

‘levelSelect’ function

```
def levelSelect(self):#function for Level Select screen
    self.game.drawText("Select Level", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)
    #title is blitted to the screen
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)
    #back button is created

    #a button for each level is created
    self.level1Button = Button(self.game, "level1", WIDTH*1/2, HEIGHT*1/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Level One")
    self.level2Button = Button(self.game, "level2", WIDTH*1/2, HEIGHT*5/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Level Two")
    self.level3Button = Button(self.game, "level3", WIDTH*1/2, HEIGHT*3/4, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Level Three")
    self.level4Button = Button(self.game, "level4", WIDTH*1/2, HEIGHT*7/8, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Level Four")
```

'level1' function

```
def level1(self):#Loads all the objects required for Level one
    self.showPlayer = True#sets this variable to TRUE to show the player on the screen

    self.game.tiledMap = TiledMap(path.join(self.game.mapFolder,'Level1.tmx'))
    #An instance of the TiledMap class is created using the tiled map for the first level
    self.game.tiledMapImg  = self.game.tiledMap.makeMap()#makeMap function is called to create the map image
    self.game.tiledMapRect = self.game.tiledMapImg.get_rect()

    self.game.camera = Camera(self.game.tiledMap.width, self.game.tiledMap.height)
    #an instance of the camera object is created using the dimensions of the map
    self.loadHighscore(1)#the highscore of the first Levl is loaded

    for tileObject in self.game.tiledMap.tiledMapFile.objects:#each object in the map is iterated

        if tileObject.name == "player":#if the name attribute of the tile is player
            self.game.player = Player(self.game, tileObject.x, tileObject.y)#a player object is spawned at that location

        elif tileObject.name == "wall":#if the name attribute of the tile is wall
            Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled")
            #a wall object is spawned at that location

        elif tileObject.name == "endWall":#if the name attribute of the tile is endWall
            Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled", mode2="end")
            #an endWall object is spawned

        elif tileObject.name == "coin":#if the name attribute of the tile is coin
            Coin(self.game, tileObject.x, tileObject.y)#a coin object is spawned

        elif tileObject.name == "question":#if the name attribute of the tile is question
            Question(self.game, tileObject.x, tileObject.y)#a question object is spawned

        elif tileObject.name == "majorQuestion":#if the name attribute of the tile is majorQuestion
            Question(self.game, tileObject.x, tileObject.y, major=True)#a major question object is spawned

        elif tileObject.name == "track":#if the name attribute of the tile is track
            Platform(self.game, tileObject.x, tileObject.y, tileObject.width, tileObject.height, LIGHT_BLUE, "track", show=False)
            #a platform object with mode='track' is spawned at that location
```

'level2' function

```
def level2(self):#similar to level one, all objects required for level two are loaded
    self.showPlayer = True
    self.game.tiledMap = TiledMap(path.join(self.game.mapFolder,'Level2.tmx'))
    self.game.tiledMapImg  = self.game.tiledMap.makeMap()
    self.game.tiledMapRect = self.game.tiledMapImg.get_rect()
    self.game.camera = Camera(self.game.tiledMap.width, self.game.tiledMap.height)
    self.loadHighscore(2)

    for tileObject in self.game.tiledMap.tiledMapFile.objects:
        if tileObject.name == "player":
            self.game.player = Player(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "wall":
            Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled")
        elif tileObject.name == "endWall":
            Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled", mode2="end")
        elif tileObject.name == "coin":
            Coin(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "question":
            Question(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "majorQuestion":
            Question(self.game, tileObject.x, tileObject.y, major=True)
        elif tileObject.name == "track":
            Platform(self.game, tileObject.x, tileObject.y, tileObject.width, tileObject.height, LIGHT_BLUE, "track", show=False)
```

'level3' function

```
def level3(self):#similar to level one, all objects required for level three are loaded
    self.showPlayer = True
    self.game.tiledMap = TiledMap(path.join(self.game.mapFolder,'Level3.tmx'))
    self.game.tiledMapImg  = self.game.tiledMap.makeMap()
    self.game.tiledMapRect = self.game.tiledMapImg.get_rect()
    self.game.camera = Camera(self.game.tiledMap.width, self.game.tiledMap.height)
    self.loadHighscore(3)

    for tileObject in self.game.tiledMap.tiledMapFile.objects:
        if tileObject.name == "player":
            self.game.player = Player(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "wall":
            Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled")
        elif tileObject.name == "endWall":
            Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled", mode2="end")
        elif tileObject.name == "coin":
            Coin(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "question":
            Question(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "majorQuestion":
            Question(self.game, tileObject.x, tileObject.y, major=True)
        elif tileObject.name == "track":
            Platform(self.game, tileObject.x, tileObject.y, tileObject.width, tileObject.height, LIGHT_BLUE, "track", show=False)
```

'level4' function

```
def level4(self):#similar to level one, all objects required for level four are loaded
    self.showPlayer = True
    self.game.tiledMap = TiledMap(path.join(self.game.mapFolder,'Level4.tmx'))
    self.game.tiledMapImg  = self.game.tiledMap.makeMap()
    self.game.tiledMapRect = self.game.tiledMapImg.get_rect()
    self.game.camera = Camera(self.game.tiledMap.width, self.game.tiledMap.height)
    self.loadHighscore(4)

    for tileObject in self.game.tiledMap.tiledMapFile.objects:
        if tileObject.name == "player":
            self.game.player = Player(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "wall":
            Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled")
        elif tileObject.name == "endWall":
            Wall(self.game, tileObject.x, tileObject.y, colour=ORANGE, width=tileObject.width, height=tileObject.height, mode="tiled", mode2="end")
        elif tileObject.name == "coin":
            Coin(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "question":
            Question(self.game, tileObject.x, tileObject.y)
        elif tileObject.name == "majorQuestion":
            Question(self.game, tileObject.x, tileObject.y, major=True)
        elif tileObject.name == "track":
            Platform(self.game, tileObject.x, tileObject.y, tileObject.width, tileObject.height, LIGHT_BLUE, "track", show=False)
```

‘shop’ function

```
def shop(self):#function for the Vehicles screen
    self.image = pg.transform.scale(self.game.menuImages['shop'], (WIDTH, HEIGHT))#background image is obtained and re-sized
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)

    self.game.drawText("Vehicles", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)#title is drawn
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)#back button is created

    self.carButton = Button(self.game, "car", WIDTH*2/5, HEIGHT*5/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Car")#buttons for the vehicles
    self.bikeButton = Button(self.game, "bike", WIDTH*2/5, HEIGHT*8/12, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Bike")#are created

    self.game.screen.blit(self.game.carImage, (WIDTH*11/20, HEIGHT*9/24))#images of the vehicles are loaded in beside
    self.game.screen.blit(self.game.bikeImage, (WIDTH*11/20, HEIGHT*15/24))#the corresponding buttons

    if self.game.imageType == "car":#the currently chosen vehcile is calculated
        currentVehicle = "Car"
    if self.game.imageType == "bike":
        currentVehicle = "Bike"

    #the chosen vehicle is displayed to the screen
    self.game.drawText("Current Vehicle: {}".format(currentVehicle), 25, WHITE, WIDTH/2, HEIGHT*3/12, fontName=self.game.interfaceFont2)
```

‘instructions’ function

```
def instructions(self):#function for the instructions screen
    self.image = pg.transform.scale(self.game.menuImages['levelComplete'], (WIDTH, HEIGHT))#background image is obtained and re-sized
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)

    #instruction text is printed to the screen
    self.game.drawText("Instructions", 30, WHITE, WIDTH/2, HEIGHT/6, fontName=self.game.interfaceFont2)
    self.game.drawText("WASD keys or Arrow Keys to move", 20, WHITE, WIDTH/2, HEIGHT*3/8, fontName=self.game.interfaceFont2)
    self.game.drawText("Space Bar to jump", 20, WHITE, WIDTH/2, HEIGHT*4/8, fontName=self.game.interfaceFont2)
    self.game.drawText("Answer the questions and collect the coins!", 20, WHITE, WIDTH/2, HEIGHT*5/8, fontName=self.game.interfaceFont2)
    self.game.drawText("Make sure you get the final question", 20, WHITE, WIDTH/2, HEIGHT*23//32, fontName=self.game.interfaceFont2)
    self.game.drawText("correct to finish the level!", 20, WHITE, WIDTH/2, HEIGHT*25/32, fontName=self.game.interfaceFont2)
    self.game.drawText("Press A Button To Start!", 22, WHITE, WIDTH/2, HEIGHT*7/8, fontName=self.game.interfaceFont2)

    pg.display.flip()#the display is updated
    self.waitForKey()#the waitForKey function is called to detect an input from the user to change to the screen

    if self.prevScence == "settingsMenu":#if this screen was opened from the setings screen, re-load the settings screen
        self.currentScene = 'settingsMenu'
        self.loadLevel('settingsMenu')
```

‘pause’ function

```
def pause(self):#function for the pause screen - the main menu and resume buttons are created
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*3/8, HEIGHT*1/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Main Menu")
    self.resumeButton = Button(self.game, "resume", WIDTH*5/8, HEIGHT*1/2, WIDTH/6, HEIGHT/12, YELLOW, LIGHT_BLUE, "Resume")
```

'startScreen' function

```
def startScreen(self):#function for the start screen
    self.image = pg.transform.scale(self.game.menuImages["startScreen"], (WIDTH, HEIGHT))#background image is retrieved and re-sized
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)

    #information text is displayed to the screen
    self.game.drawText("Welcome to the", 32, WHITE, WIDTH/2, HEIGHT*1/4, fontName= self.game.interfaceFont2)
    self.game.drawText("Mental Maths Game", 32, WHITE, WIDTH/2, HEIGHT*3/8, fontName= self.game.interfaceFont2)
    self.game.drawText("Press A Button To Continue!", 22, WHITE, WIDTH/2, HEIGHT*3/4, fontName= self.game.interfaceFont2)

    pg.display.flip()#display is updated so blitted text is visble

    self.waitForKey()#waitForKey function called to not change the screen until the user wants to

    self.loadLevel('mainMenu')#load the main menu once the user has provided an input
```

'waitFor'Key' function

```
def waitForKey(self, key = True, click = True):#function for pausing the screen until an input is provided
    #key and click paramters determine if pressing a key or clicking the screen causes the screen to change
    pg.event.wait()#waits for an input event to occur

    waiting = True
    while waiting:#keeps looping until 'waiting' variable is set to FALSE
        self.game.clock.tick(FPS)#pygame clock continues

        for event in pg.event.get():#in all the recorded events

            if event.type == pg.QUIT:#if the game window is closed
                pg.quit()#exit the game

            if event.type == pg.KEYUP and key:#if the provided key parameter is true and a key is pressed
                waiting = False#this variable is set to FALSE - ending the while loop

            if event.type == pg.MOUSEBUTTONUP and click:#if the provided click parameter is true and the mouse is clicked
                waiting = False#this variable is set to FALSE - ending the while loop
```

'loadHighscore' function

```
def loadHighscore(self, level):#function for loading the highscore of a provided level
    self.updatedHighscore = False#this variable is initilaised as FALSE

    fileNameList = ["l","e","v","e","l",".","p","i","c","k","l","e"]#a list of the characters of the filename is created
    fileNameList.insert(5, str(level))#the provided level parameter is inserted into the fifth position in the list
    self.levelFileName = "".join(fileNameList)#the list of charaters is joinded into a string

    highscoreFile = open(self.levelFileName, "rb")#the highscore pickle file for the required level is opened in read mode
    self.highscoreDict = pickle.load(highscoreFile)#the pickle dictionary is saved in a class variable
    highscoreFile.close()#the file is closed to avoid accidental alteration

    self.highscore = self.highscoreDict[1][0]#the highscore for the current level is saved to a class variable
```

‘updateHighscore’ function

```
def updateHighscore(self, level, score):#function for updating the highscore data for the current level
    self.updatedHighscore = True#sets this variable to true as the highscore will be updated

    if self.inputBox == None:#if a name was not provided in the input box
        self.scoreName = "Anonymous"#the name is saved as anonymous
    else:
        self.scoreName = self.inputBox.text#the provided text is saved for the name of the player

    currentDate = datetime.datetime.now()#the current date is obtained
    self.scoreDate = currentDate.strftime("%d-%m-%Y")#the data is formated to a date-month-year form

    index = 11
    for j in range(10):#its checked if the achieved score is in the top ten for the current level
        if score > self.highscoreDict[9-j][0]:
            index = 9-j

    addedScore = True#controls the intial input of the scores
    if index != 11:
        for i in range(10-index):#loops thorugh the values that need to change position

            if addedScore:
                tempScore = self.highscoreDict[index+i][0]#the values stored in that positon are saved to temporary variables
                tempName = self.highscoreDict[index+i][1]
                tempDate = self.highscoreDict[index+i][2]

                self.highscoreDict[index][0] = score#the data achieved in the current level is saved to this location
                self.highscoreDict[index][1] = self.scoreName
                self.highscoreDict[index][2] = self.scoreDate
                addedScore = False#as the current data has been added, this variable is changed to FALSE

            else:
                tempScore1 = self.highscoreDict[index+i][0]#the data in this location is stored in another set of temporary variable
                tempName1 = self.highscoreDict[index+i][1]
                tempDate1 = self.highscoreDict[index+i][2]

                self.highscoreDict[index+i][0] = tempScore#this first temporary variable data is written to this location
                self.highscoreDict[index+i][1] = tempName
                self.highscoreDict[index+i][2] = tempDate

                tempScore = tempScore1#the second set of temporary variables are copied to the first set
                tempName = tempName1
                tempDate = tempDate1

    highscoreWriteFile = open(self.levelFileName,"wb")#the highscore file for the current level is opened in write mode
    pickle.dump(self.highscoreDict, highscoreWriteFile)#the pickle file is overwritten with the updated dictionary
    highscoreWriteFile.close()#the updated file is closed
```

‘levelComplete’ function

```
def levelComplete(self):#function for the level complete screen
    self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))#the background image is obtained and re-sized
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)

    #Level information is displayed to the screen
    self.game.drawText("Score: {}".format(str(self.game.score)), 25, WHITE, WIDTH/2, HEIGHT*3/11, fontName=self.game.buttonFont)
    self.game.drawText("Highscore: {}".format(str(self.highscoreDict[0][0])), 25, WHITE, WIDTH/2, HEIGHT*4/11, fontName=self.game.interfaceFont)
    self.game.drawText("Coins Collected: {}".format(self.coinCollected), 25, WHITE, WIDTH/4, HEIGHT*6/11, fontName=None)
    self.game.drawText("Total Coins: {}".format(self.game.coinAmount), 25, WHITE, WIDTH/4, HEIGHT*7/11, fontName=None)

    if self.game.majorQuestionCorrect:#if the final major question has been answered correctly
        self.game.drawText("Level Completed", 30, WHITE, WIDTH/2, HEIGHT/6, fontName=self.game.interfaceFont)#'Level Completed' is displayed

        self.nextLevelTagNumber = int(self.prevScence[-1]) + 1#the integer value of the next level is calculated
        levelList = list(self.prevScence)#the previous level scene name is split into characters
        levelList[-1] = str(self.nextLevelTagNumber)#the final integer in the level name is replaced with the newly calculated integer
        nextLevel = "".join(levelList)#the charaters are joined into a string

        index = 11#index variable created to analyse highscore data
        for i in range(10):#checks if the achieved score is in the top ten for that level
            if self.game.score > self.highscoreDict[9-i][0]:
                index = 9-i

        if index != 11:#if the score was in the top ten
            if index == 0:#if the score achieved was the highest score achieved for that level
                #this text is displayed
                self.game.drawText("Congratulations, new High Score", 28, WHITE, WIDTH/2, HEIGHT*5/11, fontName=self.game.interfaceFont)

            else:#otherwise, this text is displayed
                #the position of the achieved score in the leaderboard table is included
                self.game.drawText("Congratulations, new top {} Score".format(index+1), 28, WHITE, WIDTH/2, HEIGHT*5/11, fontName=self.game.interfaceFont)
            else:#otherwise, this text is displayed
                #the position of the achieved score in the leaderboard table is included
                self.game.drawText("Congratulations, new top {} Score".format(index+1), 28, WHITE, WIDTH/2, HEIGHT*5/11, fontName=self.game.interfaceFont)
            self.inputBox = InputBox(self.game, WIDTH*5/8, HEIGHT*7/11, 150, 32, text='')#an input box object is created

        else:#if score achieved is not in the top ten
            self.inputBox = InputBox(self.game, WIDTH*5/8, HEIGHT*7/11, 150, 32, text='')#an input box object is created

        else:#if score achieved is not in the top ten
            self.nextLevel = Button(self.game, nextLevel, WIDTH /2, HEIGHT*7/8, WIDTH/5, HEIGHT/6, YELLOW, LIGHT_BLUE, "Next Level")

    else:#if the final question has not been answered correctly
        self.game.drawText("Level Failed", 30, WHITE, WIDTH/2, HEIGHT/6, fontName=self.game.interfaceFont)#this text is displayed and a retry button
        self.samelevel = Button(self.game, self.prevScence, WIDTH /2, HEIGHT*7/8, WIDTH/5, HEIGHT/6, YELLOW, LIGHT_BLUE, "Retry Level")#is created

        #buttons to go to the main menu and level select screen are created
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/4, HEIGHT*7/8, WIDTH/5, HEIGHT/6, YELLOW, LIGHT_BLUE, "Main Menu")
    self.levelSelectButton = Button(self.game, "levelSelect", WIDTH*3/4, HEIGHT*7/8, WIDTH/5, HEIGHT/6, YELLOW, LIGHT_BLUE, "Select Level")
```

‘stats’ function

```
def stats(self):#function for the Statistics screen
    self.game.drawText("View your statistics", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)#Title is blitted
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)#back button is created

    #the total correctly answered questions is calculated
    correctAnswerQuest = self.game.correctAnswerQuesEasy + self.game.correctAnswerQuesMed + self.game.correctAnswerQuesHard
    incorrectAnswerQuest = self.game.questAnswered - correctAnswerQuest#total incorrectly answered questions is calculated
    averageScore = int(round(self.game.totalScore / self.game.gamesPlayed,0))#average score is calculated and rounded to the nearest integer

    #total number of answered questions is displayed
    self.game.drawText("Number of Questions Answered:", 18, WHITE, WIDTH*3/8, HEIGHT*3/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(self.game.questAnswered), 18, WHITE, WIDTH*3/4, HEIGHT*3/11, fontName=self.game.interfaceFont2)

    #Number of correctly answered questions is displayed
    self.game.drawText("Correctly Answered Questions:", 18, WHITE, WIDTH*3/8, HEIGHT*4/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(correctAnswerQuest), 18, WHITE, WIDTH*3/4, HEIGHT*4/11, fontName=self.game.interfaceFont2)

    #Number of incorrectly answered questions is displayed
    self.game.drawText("Incorrectly Answered Questions:", 18, WHITE, WIDTH*3/8, HEIGHT*5/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(incorrectAnswerQuest), 18, WHITE, WIDTH*3/4, HEIGHT*5/11, fontName=self.game.interfaceFont2)

    #Number of correctly answered questions for each difficulty is displayed
    self.game.drawText("Question for each Difficulty:", 18, WHITE, WIDTH*5/16, HEIGHT*6/11, fontName=self.game.interfaceFont2)
    self.game.drawText("Easy: {}     Medium: {}     Hard: {}".format(self.game.correctAnswerQuesEasy, self.game.correctAnswerQuesMed,\
                                self.game.correctAnswerQuesHard), 18, WHITE, WIDTH*3/4, HEIGHT*6/11, fontName=self.game.interfaceFont2)

    #Number of total vehicles unlocked is displayed
    self.game.drawText("Total Vehicles Unlocked:", 18, WHITE, WIDTH*3/8, HEIGHT*7/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(self.game.vehicleUnlock), 18, WHITE, WIDTH*3/4, HEIGHT*7/11, fontName=self.game.interfaceFont2)

    #Value for total games played is displayed
    self.game.drawText("Total Games Played:", 18, WHITE, WIDTH*3/8, HEIGHT*8/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(self.game.gamesPlayed), 18, WHITE, WIDTH*3/4, HEIGHT*8/11, fontName=self.game.interfaceFont2)

    #Amount of total coins collected is displayed
    self.game.drawText("Total Coins Collected:", 18, WHITE, WIDTH*3/8, HEIGHT*9/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{}".format(self.game.coinAmount), 18, WHITE, WIDTH*3/4, HEIGHT*9/11, fontName=self.game.interfaceFont2)

    #Total score and average score is displayed
    self.game.drawText("Total Score / Average Score:", 18, WHITE, WIDTH*3/8, HEIGHT*10/11, fontName=self.game.interfaceFont2)
    self.game.drawText("{} / {}".format(self.game.totalScore, averageScore), 18, WHITE, WIDTH*3/4, HEIGHT*10/11, fontName=self.game.interfaceFont2)
```

‘leaderboard’ function

```
def leaderboard(self):
    self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))#background image is obtained and re-sized
    rect = self.image.get_rect()
    self.game.screen.blit(self.image, rect)
    self.game.drawText("Leader Board Tables", 25, WHITE, WIDTH/2, HEIGHT*1/9, fontName=self.game.interfaceFont)#title is displayed
    self.mainMenuButton = Button(self.game, "mainMenu", WIDTH*1/8, HEIGHT*1/12, 60, 25, YELLOW, LIGHT_BLUE, "Back", 18)
    #back button is created

    #buttons for each level are created
    self.level1Score = Button(self.game, "level1Score", WIDTH/5, HEIGHT*3/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 1")
    self.level2Score = Button(self.game, "level2Score", WIDTH/5, HEIGHT*4/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 2")
    self.level3Score = Button(self.game, "level3Score", WIDTH/5, HEIGHT*5/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 3")
    self.level4Score = Button(self.game, "level4Score", WIDTH/5, HEIGHT*6/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 4")

    #these levels have not yet been programmed but their scores will seemlessly be shown once these levels are added
    self.level5Score = Button(self.game, "level5Score", WIDTH/5, HEIGHT*7/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 5")
    self.level6Score = Button(self.game, "level6Score", WIDTH/5, HEIGHT*8/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 6")
    self.level7Score = Button(self.game, "level7Score", WIDTH/5, HEIGHT*9/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 7")
    self.level8Score = Button(self.game, "level8Score", WIDTH/5, HEIGHT*10/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Level 8")
    self.level0Score = Button(self.game, "level0Score", WIDTH/5, HEIGHT*11/12+15, WIDTH/6, HEIGHT/14, YELLOW, LIGHT_BLUE, "Minigame")

    fileNameList = ["l","e","v","e","l",".","p","i","c","k","l","e"]#template of pickle dictionary file names
    levelNumber = self.leaderboardLoadLevel[:-5][-1]#extracts the level number from the button tags
    fileNameList.insert(5, str(levelNumber))#adds the integer to the list
    self.leaderboardLevelFileName = "".join(fileNameList)#joins the characters in the list to form a string

    scoreFile = open(self.leaderboardLevelFileName, "rb")#opens the pickle dictionary containing the scores in read mode
    self.leaderboardDict = pickle.load(scoreFile)#loads the dictionary into a class dictionary
    scoreFile.close()#file is closed

    if levelNumber == 0:#level number zero is will be used for the mini game feature if added
        #Minigame will be displayed instead of 0 in this case
        self.game.drawText("Mini Game", 25, WHITE, WIDTH*21/32, HEIGHT*9/32, fontName=self.game.interfaceFont2)
    else:#for all other levels
        #The name of the current level is displayed to the screen
        self.game.drawText("Level {}".format(str(levelNumber)), 25, WHITE, WIDTH*21/32, HEIGHT*9/32, fontName=self.game.interfaceFont2)

    self.game.drawText("Name", 22, WHITE, WIDTH*7/16, HEIGHT*3/8, fontName=self.game.interfaceFont2)#Name, Score and Date column titles
    self.game.drawText("Score", 22, WHITE, WIDTH*21/32, HEIGHT*3/8, fontName=self.game.interfaceFont2)#are created
    self.game.drawText("Date", 22, WHITE, WIDTH*7/8, HEIGHT*3/8, fontName=self.game.interfaceFont2)

    for i in range(10):#ten loops are made to iterate thorugh the dictionary and display each collection of data on a new row
        self.game.drawText(str(self.leaderboardDict[i][0]), 18, WHITE, WIDTH*21/32, HEIGHT*(8+i)/18, fontName=self.game.interfaceFont2)
        self.game.drawText(str(self.leaderboardDict[i][1]), 18, WHITE, WIDTH*7/16, HEIGHT*(8+i)/18, fontName=self.game.interfaceFont2)
        self.game.drawText(str(self.leaderboardDict[i][2]), 18, WHITE, WIDTH*7/8, HEIGHT*(8+i)/18, fontName=self.game.interfaceFont2)
```

‘update’ function

```
def update(self):#update function is used to check button presses and calculate the appropriate action
    self.game.buttons.update()#all the buttons are updated

    for button in self.game.buttons:#for all the buttons
        if button.clicked == True:#if a buttons 'clicked' attribute is TRUE

            #if the button is not on the leaderboard screen
            if self.currentScene != "leaderboard" or (self.currentScene == "leaderboard" and button.tag == "mainMenu"):

                #if the button is not on the settings screen
                if self.currentScene != "settingsMenu" or (self.currentScene == "settingsMenu" and button.tag == "mainMenu") \
                    or (self.currentScene == "settingsMenu" and button.tag == "startScreen") \
                    or (self.currentScene == "settingsMenu" and button.tag == "instructions"):

                    #if the button is not on the vehicles screen
                    if self.currentScene != "shop" or (self.currentScene == "shop" and button.tag == "mainMenu"):

                        #if the button is on the level complete screen
                        if self.currentScene == "levelComplete":

                            #if the highscore has not been updated and the final major question has been correctly answered
                            if not self.updatedHighscore and self.game.majorQuestionCorrect:
                                self.updateHighscore(self.nextLevelTagNumber-1, self.game.score)#call the updateHighscore function

                        #if the button is not an answer for a question
                        if button.tag not in ("2", "3", "4", "5", "6"):

                            #the scene variables are set
                            self.secondPrevScence = self.prevScence
                            self.prevScence = self.currentScene
                            self.currentScene = button.tag#the current scene is set to the tag of the button

                            #if the chosen button's screen is not in the specified list
                            if button.tag not in ('level1', 'level2', 'level3','level4', 'pause', 'resume', 'levelComplete', 'instructions'):

                                #the background image of the new screen is retrieved and reized
                                self.image = pg.transform.scale(self.game.menuImages[self.currentScene], (WIDTH, HEIGHT))
                                rect = self.image.get_rect()
                                self.game.screen.blit(self.image, rect)#the background image is blitted to the screen

                                for inputBox in self.game.userInputBox:#an inputBox objects are removed
                                    inputBox.kill()

                            elif button.tag in (LEVEL_SCREENS):#if a level button is pressed

                                #all the current walls, endwalls, platforms, rectangles, questions and coin objects are deleted
                                for wall in self.game.walls:
                                    wall.kill()
                                for endWall in self.game.endWalls:
                                    endWall.kill()
                                for platform in self.game.platforms:
                                    platform.kill()
                                for rectangle in self.game.rectangles:
                                    rectangle.kill()
                                for questions in self.game.questionItems:
                                    questions.kill()
                                for coin in self.game.coins:
                                    coin.kill()

                                if self.game.player != None:#if a player object exists, that too is deleted
                                    self.game.player.kill()

                                self.loadHighscore(int(self.currentScene[-1]))#the highscore of the new level is loaded

                                self.game.score = 0#the score, coins collected and major question variables are reset to 0 and TRUE
                                self.coinCollected = 0
                                self.game.majorQuestionCorrect = True

                                self.game.gamesPlayed += 1#this variable for the statistics data is updated

                                self.loadLevel('instructions')#the instructions screen is laoded for the start of each level
```

```
                #if the button is not an answer button for a question, the tag of the button is checked and
                #the appropriate function is called using the loadLevel function
                if button.tag == "startScreen":
                    self.loadLevel('startScreen')
                if button.tag == "levelSelect":
                    self.loadLevel('levelSelect')
                if button.tag == "shop":
                    self.loadLevel('shop')

                if button.tag == 'mainMenu':
                    if self.game.paused:#if the main mneu button in the pause screen is pressed
                        self.game.paused = False#reverse the state of the paused variable
                        for button in self.game.buttons:#remove all of the buttons on the screen
                            button.kill()
                    self.loadLevel('mainMenu')#then load the main menu function

                if button.tag == "settingsMenu":
                    self.loadLevel('settingsMenu')
                if button.tag == 'leaderboard':
                    self.loadLevel('leaderboard')
                if button.tag == 'stats':
                    self.loadLevel('stats')
                if button.tag == 'level1':
                    self.loadLevel('level1')
                if button.tag == 'level2':
                    self.loadLevel('level2')
                if button.tag == 'level3':
                    self.loadLevel('level3')
                if button.tag == 'level4':
                    self.loadLevel('level4')
                if button.tag == "levelComplete":
                    self.loadLevel('levelComplete')
                if button.tag == 'pause':
                    self.loadLevel('pause')
                if button.tag == 'instructions':
                    self.loadLevel('instructions')

                if button.tag == 'resume':#the resume button is only available on the pause screen
                    self.game.paused = False#reverse the state of the paused variable
                    for button in self.game.buttons:#remove all of the buttons on the screen
                        button.kill()

                    pg.mixer.music.fadeout(500)#fadeout the playing music over 0.5 seconds
                    self.game.pauseMusicStarted = False#change the music controlling variables appropriately
                    self.game.pauseScreenPrinted = False

                    self.currentScene = self.prevScence#revert to the scene before pausing

            else:#the button clicked if for an answer to a question
                self.game.answerClicked = True#set the answerClicked variable to TRUE

                #selected answer is obtained from the 2D arry using the chosen index
                self.game.selectedAns = self.game.questionData[self.game.questionNumberIndex][int(button.tag)]

                if button.tag == "2":#if the button with tag 2 is chosen, the correct answer has been selected
                    self.game.answerCorrect = True#therefore this variable is set to TRUE
```

```
                        else:#otherwise the incorrect answer has been selected so
                            self.game.answerCorrect = False#this variable is set to FALSE

                        for button in self.game.buttons:#all the current buttons are removed
                            button.kill()

                    button.kill()#the clicked button is deleted

                else:#the button clicked is on the Vehicles screen

                    if button.tag == "car":#depending on the vehicle chosen
                        self.game.playerImage = self.game.carImage#change the player's image to the chosen vehicle
                        self.game.imageType = "car"#change the value of this variable to the chosen vehicle

                    elif button.tag == "bike":
                        self.game.playerImage = self.game.bikeImage
                        self.game.imageType = "bike"

                    for button in self.game.buttons:#remove all of the current buttons
                        button.kill()

                    self.loadLevel("shop")#reload the vehicles screen

            else:#the button clicked os on the settings screen

                #if a button linked to music is pressed
                if button.tag == "electric" or button.tag == "upbeat" or button.tag == "noMusic":

                    if button.tag == "electric":#depedning on the music chosen
                        self.game.currentLevelMusic = "electric"#change this variable to the chosen music
                        self.game.levelMusic = self.game.levelMusic2#change this variable to the file for the chosen track
                        self.game.musicOn = True#set this variable to TRUE

                    elif button.tag == "upbeat":
                        self.game.currentLevelMusic = "upbeat"
                        self.game.levelMusic = self.game.levelMusic1
                        self.game.musicOn = True

                    elif button.tag == "noMusic":#if the no music button is pressed
                        self.game.currentLevelMusic = "noMusic"#change this variable to noMusic
                        self.game.musicOn = False#set thisvariable to FALSE

                else:#if a button for the difficulty is pressed

                    self.game.settingsQuestionDiff = button.tag#set the chosen difficulty to the required variable

                    self.game.questionID = []#recalculate the questionID numbers for the questions which can be asked
                    for i in range(len(self.game.questionData)):
                        if self.game.questionData[i][7] == self.game.settingsQuestionDiff:
                            if self.game.questionData[i][9] == 'FALSE':
                                self.game.questionID.append(self.game.questionData[i][0])

                    for button in self.game.buttons:#remove all of the current buttons
                        button.kill()

                    self.loadLevel("settingsMenu")#Load the settings screen

        else:#if the button clicked is on the leaderboard screen

            self.leaderboardLoadLevel = button.tag#set this variable to the clicked button

            for button in self.game.buttons:#remove all of the current buttons
                button.kill()

            self.loadLevel("leaderboard")#Load the leaderboard function

    #after looping through all of the buttons
    if self.currentScene  not in LEVEL_SCREENS:#if the current scene is not in the level screens
        self.showPlayer = False#set this variable to FALSE as the player does not need to be shown

    else:#otherwise, the current scene is in a level so
        self.showPlayer = True#set this variable to TRUE to show the player on the screen
```

Pickle Creater.py script

Importing the module

```
import pickle
```

‘statistics function

```
def statisticsRead():#read the statistics data
    exportDict = open("stats.pickle", "rb")
    dict = pickle.load(exportDict)
    exportDict.close()
    print(dict)#prints the statistics data
```

‘statisticsRead’ function

```
def statistics():#reset the statistcs data
    statsDict = {"gamesPlayed":0, "questAnswered":0, "correctAnswerQuesEasy":0, "correctAnswerQuesMed":0, \
                 "correctAnswerQuesHard":0, "vehicleUnlock":0, "coinTotal":0, "coinSpent":0, "totalScore":0}
    exportDict = open("stats.pickle", "wb")
    pickle.dump(statsDict, exportDict)
    exportDict.close()
```

‘createScoreDict’ function

```
def createScoreDict(level):#create an empty pickle scores file for a new level
    levelDict = {0:[0, "",""], 1:[0, "",""], 2:[0, "",""], 3:[0, "",""], 4:[0, "",""], 5:[0, "",""], \
                 6:[0, "",""], 7:[0, "",""], 8:[0, "",""], 9:[0, "",""]}

    fileNameList = ["l","e","v","e","l",".","p","i","c","k","l","e"]
    fileNameList.insert(5, str(level))#provided level paramter inserted into list
    pickleFile = "".join(fileNameList)

    exportDict = open(pickleFile, "wb")#created string file name used to open file
    pickle.dump(levelDict, exportDict)
    exportDict.close()
```

‘loadLevel’ function

```
def loadLevel(level):
    fileNameList = ["l","e","v","e","l",".","p","i","c","k","l","e"]
    fileNameList.insert(5, str(level))
    pickleFile = "".join(fileNameList)
    exportDict = open(pickleFile, "rb")
    dict = pickle.load(exportDict)
    print(dict)#prints the laoded level's score data
```

Settings.py script

```python
import pygame as pg#pygame module is imported

TITLE = "Mental Maths Game"#title of the game is set

WIDTH = 768#dimensions of the game window are set
HEIGHT = 576

FPS = 60#frame rate of the game is set

TILESIZE = 32#size of the tiles in pixels is set
TILESIZE_TRACK = 32

GRIDWIDTH = WIDTH / TILESIZE#width and height of the grid
GRIDHEIGHT = HEIGHT / TILESIZE#containing tiles is calculated

TIME_FOR_EASY_Q = 10#time for each difficulty of question
TIME_FOR_MEDIUM_Q = 15#is set
TIME_FOR_HARD_Q = 20

#the list of scene names of the menu screens is defined
MENU_SCREENS = ["settingsMenu",
                "mainMenu",
                "levelSelect",
                "stats",
                "leaderboard",
                "shop",
                "startScreen",
                "gameOverScreen",
                "pause",
                "levelComplete",
                "levelFailed",
                "instructions"]

#the list of scene names of the level scenes is defined
LEVEL_SCREENS = ["level1",
                 "level2",
                 "level3",
                 "level4",
                 "level5",
                 "level6",
                 "level7",
                 "level8",
                 "minigame"]

#Player properties
PLAYER_ACC = 0.65
BRAKE_ACC = 0.4
PLAYER_FRICTION = -0.08
PLAYER_GRAV = 0.8

#Layers - these are the orders in which the sprites will be rendered
PLAYER_LAYER = 2#Larger number means rendered on top
PLATFORM_LAYER = 1#of smaller numbered objects
WALL_LAYER = 1
TRACK_LAYER = 1
RECTANGLE_LAYER = 2
QUESTION_LAYER = 3
COIN_LAYER = 3
RECTANGLE_LAYER = 2
QUESTION_LAYER = 3
COIN_LAYER = 3
#colours - RGB values of the colours used in the game are defined
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
YELLOW = (255, 255, 0)
ORANGE = (255, 165, 0)
LIGHT_RED = (210, 0, 0)
LIGHT_GREEN = (0, 150, 0)
LIGHT_BLUE = (0,0,210)

HUD_COLOUR = BLACK#the HUD colour is set ot black by default

#Images

#all six coin images are stored in an array
COIN_IMAGES = ["coin1.png",
               "coin2.png",
               "coin3.png",
               "coin4.png",
               "coin5.png",
               "coin6.png"]
QUESTION_IMAGE = "boxQuestion.png"
```