# Contents

[Contents 1](#_Toc102589109)

[Analysis 3](#_Toc102589110)

[Problem Identification: (Name: CS Saga) 3](#_Toc102589111)

[Stakeholders 4](#_Toc102589112)

[Identification of the target platform 4](#_Toc102589113)

[Explanation of user needs 4](#_Toc102589114)

[Meeting with Stakeholders 5](#_Toc102589115)

[Detailed research into existing similar products 7](#_Toc102589116)

[Features of the proposed solution 10](#_Toc102589117)

[Initial concept of my solution considering this research: 10](#_Toc102589118)

[Limitations 11](#_Toc102589119)

[Computational Methods and Approach 11](#_Toc102589120)

[Identification and explanation of computational method 11](#_Toc102589121)

[Success Criteria 13](#_Toc102589122)

[Design 16](#_Toc102589123)

[Structure of the Solution 16](#_Toc102589124)

[An overview of the structure of the solution 16](#_Toc102589125)

[Problem Decomposition 16](#_Toc102589126)

[Explanation of each of the modules, methods, procedures, and functions required 16](#_Toc102589127)

[A justification of the approach taken 23](#_Toc102589128)

[Key Variables and Data Structures 26](#_Toc102589129)

[Identification of the key classes and attributes 26](#_Toc102589130)

[Explanation and justification of key variables & data structures 31](#_Toc102589131)

[Algorithms 33](#_Toc102589132)

[Usability Features 50](#_Toc102589133)

[Explanation and justification of the design and the user interface 50](#_Toc102589134)

[Stakeholder Input 55](#_Toc102589135)

[Description and justification of the usability features 56](#_Toc102589136)

[Description and justification of the validation required 57](#_Toc102589137)

[Identification and justification of test data to be used during post-development 58](#_Toc102589138)

[Development 62](#_Toc102589139)

[Platform Access 62](#_Toc102589140)

[Version 1: Home Page 62](#_Toc102589141)

[Version 2: Credentials Page 69](#_Toc102589142)

[Version 3: Rebranding 77](#_Toc102589143)

[Level Generation and Rendering: 82](#_Toc102589144)

[Version 1: Creating Levels 82](#_Toc102589145)

[Version 2: Loading Levels 87](#_Toc102589146)

[Level Gameplay and Mechanics 92](#_Toc102589147)

[Version 1: Player 92](#_Toc102589148)

[Version 2: Animations 97](#_Toc102589149)

[Version 3: Camera movement, Text, and Death Blocks 103](#_Toc102589150)

[Version 4: Enemy AI with animations 108](#_Toc102589151)

[Quiz 116](#_Toc102589152)

[Version 1: Question screen + introduction of AutoBox class + Option handling 116](#_Toc102589153)

[Version 2: Feedback 124](#_Toc102589154)

[Version 3: Selecting questions and Transitions 125](#_Toc102589155)

[Version 4: Summary statistics 125](#_Toc102589156)

[Main Menu 125](#_Toc102589157)

[Version 1: Topic boxes 125](#_Toc102589158)

[Version 2: Leader boards and Instructions 126](#_Toc102589159)

[Version 3: Points Graph 126](#_Toc102589160)

[Evaluation 127](#_Toc102589161)

[Meeting the Success Criteria 127](#_Toc102589162)

[Limitations 127](#_Toc102589163)

[Maintenance 127](#_Toc102589164)

Adjacency list for linked topics

>>> https://www.ocr.org.uk/Images/514655-programming-project-set-a-high.pdf

>>> https://www.ocr.org.uk/Images/514657-programming-project-set-a-mid.pdf

# Analysis

## Problem Identification: (Name: CS Saga)

A common phenomenon among teenagers transitioning onto A-level courses is that the material they engage with is more formal in tone than the resources they used at GCSE. Whereas before, students were able to attend school and not undertake challenging study, now the new specification of A-levels expect deep thinking and higher quality academic work. A significant number of students find this transition traumatic and wish they could study for these courses using more accessible and engaging resources.

I would like to create a piece of educational revision software that presents A-level Computer Science (CS) in the form of a platform game. Students could play this game, and study at the same time - it would present challenges that the player could meet, while also requiring the player to input strong knowledge of CS. The game will feature a theme consisting of various types of enemies in different themes. Each level will correspond to a topic in A-level Computer Science, where the player demonstrates knowledge to defeat an enemy. Upon defeating all the enemies, the user’s performance for that level will be stored, and questions where performance was lacking will be prioritised next time to help strengthen their knowledge and understanding.

### Stakeholders

The clients for this software will be KS4 students with an interest in CS, and a casual interest in gaming. The target audience will predominantly be KS4 students who lack confidence as independent students, or who find it difficult to establish a balance between gaming and studying due to their lack of engagement and focus during lesson time. My aim is to provide an insight into A-level Computer Science in order to provide a smoother transition into the course.

I have chosen one student to represent this user base, and one teacher to aid me in developing the questions and progression of the game. Dan Phillips, a 16-year-old student and avid gamer who will be studying computer science as an A-level, will represent the target audience of this software. Another stakeholder Alister Waring, an experienced teacher with 10 years of teaching Computer Science at high school level, who will provide opinions and criticism regarding the style, format, difficulty of the questions, the pace, and the tone, due to his experience and expertise in the field of education.

### Identification of the target platform

Hardware

A computer capable of running software, with standard Input/output peripherals. The software will use a mouse to select screen items and a keyboard to handle movement and text input. A computer with a minimum 2 GHz processor is required, as this software will not make intense demands on the processor but will need to run Pygame smoothly. In addition, a monitor is required to display the game interface.

Software

**Windows/Linux/MacOS** is required, to support Python3 and Pygame (the game engine library used to provide a game development environment).

**Python Interpreter/IDE** – the code will be written in Python, so a Python interpreter is needed.

**Pygame for Python** – this game engine library allows manipulation of sprites and input/output from the keyboard and will be utilised to run the game window.

**Matplotlib for Python** – this library enables a visual representation of data. -> recommend use of repl.it on systems who don’t have this installed by default or may be unsupported.

**Data store format (json)** - This will be used to sore a large majority of information such as user and level information because it is simple and easy to navigate with its resemblance to Python’s dictionaries and is commonly used between diverse electronic data exchanges.

### Explanation of user needs

The user will a require software that allows them to learn computer science knowledge without requiring them to explain, program, or describe in detail, to keep it simple and not overwhelm them. Because the user is likely to struggle with maintaining engagement and focusing on lessons, this software should provide a way for them to learn knowledge and facts about Computer Science by incorporating them into a game with a goal for them to work towards at each level. It will tell the user whether they are correct or incorrect after an attempt, and if they are incorrect, it will explain why they were incorrect, allowing them to gradually develop fundamental knowledge of computer science. The software should be simple and engaging to use, so that the game's core mechanics are quick and easy to grasp. The gameplay should be challenging but playable in order to not discourage the user, and users will require individual profiles in order to save their progress and compare results with other system players as a form of healthy competition and maintain incentives to keep pushing themselves.

### Meeting with Stakeholders

A thorough investigation is identified and carried out. There are concise summary sections of key points discovered after each interview and existing solutions are analysed well; the ideas to be carried through to the user requirements are clearly explained and justified.

I have decided to interview my two stakeholders in an attempt to gain some information as to what they expect and want from my final product.

**Dan Phillips**

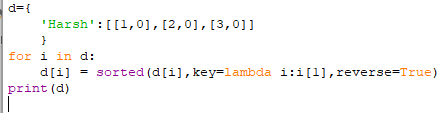
I asked some questions to Dan in attempt to gain an insight into his thoughts.

1. **Have you used any platforms previously to study for Computer Science? And if so, which ones?**

*I have not studied Computer Science prior to A-levels and so I have little experience with platforms specifically focused on CS revision. However, I mainly used Seneca Learning to revise for my GCSE subjects along with other websites such as BBC Bitesize and Anki, although I would get bored and go back to gaming not too long after starting revision.*

1. **Can you recall on any distinct features that you used whilst using these programs?**

*Anki was a straightforward and simple to use program that easily enabled me to create short and concise questions with their respective answers. Along with Anki, I particularly liked using Seneca to target my weak points through its use of strength-based learning algorithm which helped me pinpoint sub-sections of topics in which I wasn’t too confident in*. -> implement efficient sorting algorithm to present weakest – strongest questions based on the order



1. **I recall that you enjoy gaming, so what are some of your favourite genres and features of video games?**

*Two of my favourite genres include platformers and first-person games. I like platformers when I want to relax and have an easy-going gaming experience such as Mario and limbo. FPS games, such as the Witcher 3 and Karlson, provide a sense of enjoyment by overcoming a series of challenges, with a variety of difficulty. Some of the features that I particularly enjoy from these two, is the objective of defeating enemies to collect rewards, which can then be used to improve the player’s equipment or stats. The progressive difficulty of these games is what helps maintain my engagement, I find myself in anticipation and ecstasy for the new adventures that are awaiting me.*

1. **How much time do you spend on gaming and studying?**

*I typically find myself spending more time on gaming than studying. I have recently seen a dip in my grades at school because of this, but I just find studying too boring.*

1. **Have you tried educational games before?**

*Well, I’ve tried looking for games for my age group, but they’re too childish and don’t manage to maintain my attention which often leads me going back to video games such as the ones I talked about before.*

1. **If you were to make an educational game, how would you make it more engaging?**

*A depth and lore to the game would help keep me entertained, along with a series of challenges, so that it isn’t too easy to complete and is also able to help me with my understanding of the subject. I would implement a clear objective at each level, so the user knows exactly what to do with a goal in mind.*

**Alister Waring**

I then proceeded, to ask some questions to Mr Waring to gain a perspective from an experienced computer science teacher.

1. **What resources do you think have been the most effective when teaching computer science?**

*I have mainly been using ‘Craig n Dave’ as the main platform because the videos cover every point within the specification, with appropriate depth for the curriculum without going well beyond what the students need to know. I have realised that other resources, such as CSUK, have led students to revising some unnecessary content. On the other hand, Craig n Dave uses supervised learning based around exam focus. It’s the goldilocks of computing resources, not too much nor too little information, Craig n Dave tasks build knowledge and experience around typical exam questions, all their slides and classroom resources really resemble exam style questions, giving precise and concise definitions, providing students with exam like experience. I have also realised that Seneca Learning seems to have been quite useful mainly because of its visual interface which motivates students because they find it more enjoyable than other methods, but I’ve realised how it doesn’t always present the kids with challenging questions, because they can easily just look back on the information provided and copy the answer without thinking about possible answers and applying their learnt knowledge.*

1. **What are some of the difficulties you have noticed whilst teaching Computer Science?**

*There has always been a large proportion of students who felt disconnected from the content, due to the lack of understanding, attention and motivation when studying this subject. There have been misconceptions about the basics of certain topics*.

1. **What potential solutions do you think would be useful to overcome these difficulties?**

*Like Craig N Dave, it would be great if the students could have an interactive platform which maintains the students’ engagement and attention whilst also allowing them to develop their fundamental knowledge without overflowing their brain with unnecessary information about computer science.*

1. **What features would you propose be added within an interactive educational platform to increase its effectiveness?**

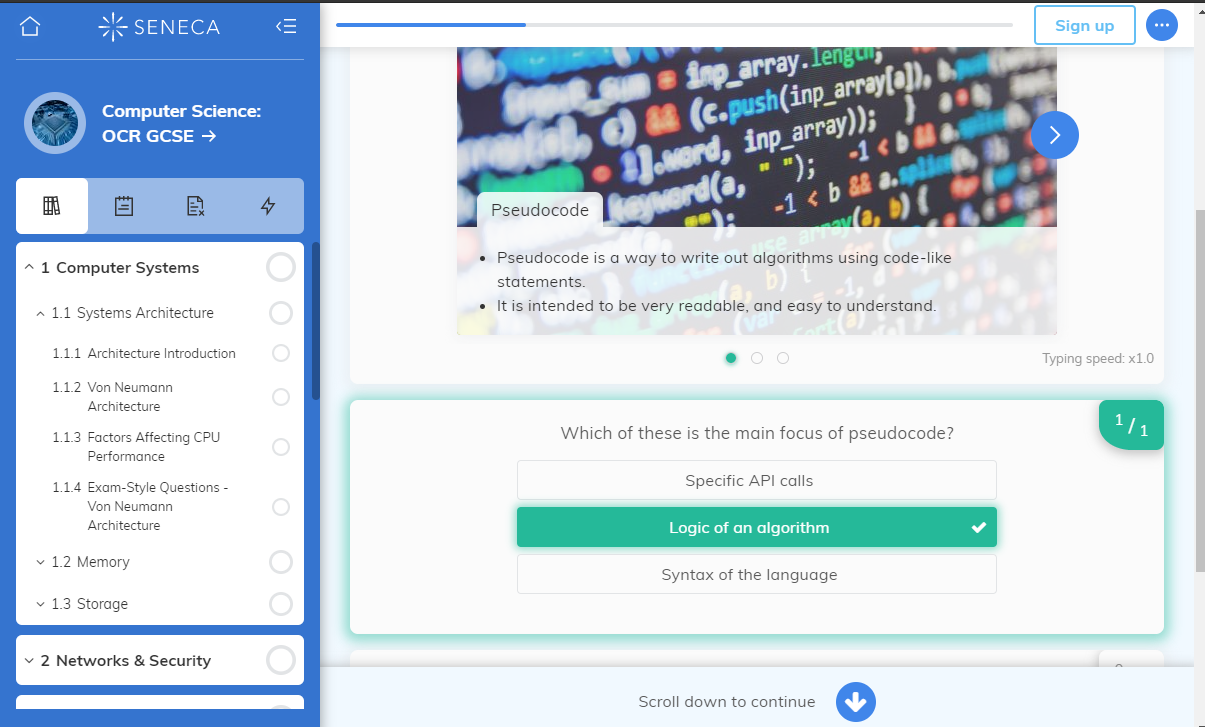
*I would like the answers to be exam-style, resembling short 1–2-mark exam questions, don’t add in 9 markers, or extended questions. It should require basic knowledge that is sufficient for these styles of questions. Secondly, when the program informs the users whether the question was right or wrong, if they get it wrong, provide them with effective feedback. For example, if there was a question such as: “what does RAM stand for?” and the student answers: “random access monitor” the program should display some informative statement like*

*“RAM is a volatile computer hardware” which might nudge them towards the correct answer.*

### Detailed research into existing similar products

After conducting a thorough interview with my two stakeholders, I have picked 3 of the most suitable software which contain elements that my stakeholders requested for, which I will use as inspiration for my final product.

**Seneca Learning:**



**Overview**:

Seneca learning is a commonly used educational software used by a variety of age groups to aid in revision and learning, offering a multitude of courses to choose from. Seneca has been designed alongside a team of neuroscientists to implement a series of efficient learning strategies such as active recall and visual memory cues. Each individual user profile is able to enrol in several different courses for difference specifications and different key stages in which their progress is tracked. From the data gathered after completing a sub module, a complex algorithm is used to target the user’s weaknesses and focus on those questions upon revisiting the sub module. Other features include a display of leader boards for each individual course.

**Parts that can be applied to my solution:**

I will be using several features of Seneca in the educational part of my software. Due to scope constraints, I will be focusing mainly on the application of an overall leader board system within the game in order to increase engagement and encourage healthy competition between users. The scoring will simply be added to the player’s tally upon correctly answering a question which will be given for the respective topic/module**. Iterative development: I will also implement the tracking of data over the history of the user’s account which will enable teachers such as Mr Waring to monitor his students’ progress.**

When re-attempting a module in Seneca, it often targets the weaknesses from the previous (if existing) attempts, and so I will also implement a ‘strength-based learning’ algorithm when picking questions to ensure the most suitable questions are picked for optimal learning experience. This will mainly take place in the background, hence from the user’s perspective it will not be overly explicit that this is being used.

Although Seneca provides a variety of question types and styles. I will not be implementing the learning style approach as the main purpose of my software is to quiz and help students consolidate their knowledge. Before implementing this, I discuss with my stakeholders as to what style of questions they will find most useful from Seneca to ensure that this software will be of use and beneficial to them.

**Captain Coffer 2D:**

**Overview:**



Captain Coffer 2D is a retro arcade platformer starring a real, immortal treasure of a hero in which it traverses through several different maps defeating enemies whilst solving puzzle-orientated stages under timed constraints. The user is able to move around using their keyboards with the addition of controller support, with the aim being to collect coins which are used to defeat monsters which can also be used to activate switches and doors.

**Parts that can be applied to my solution:**

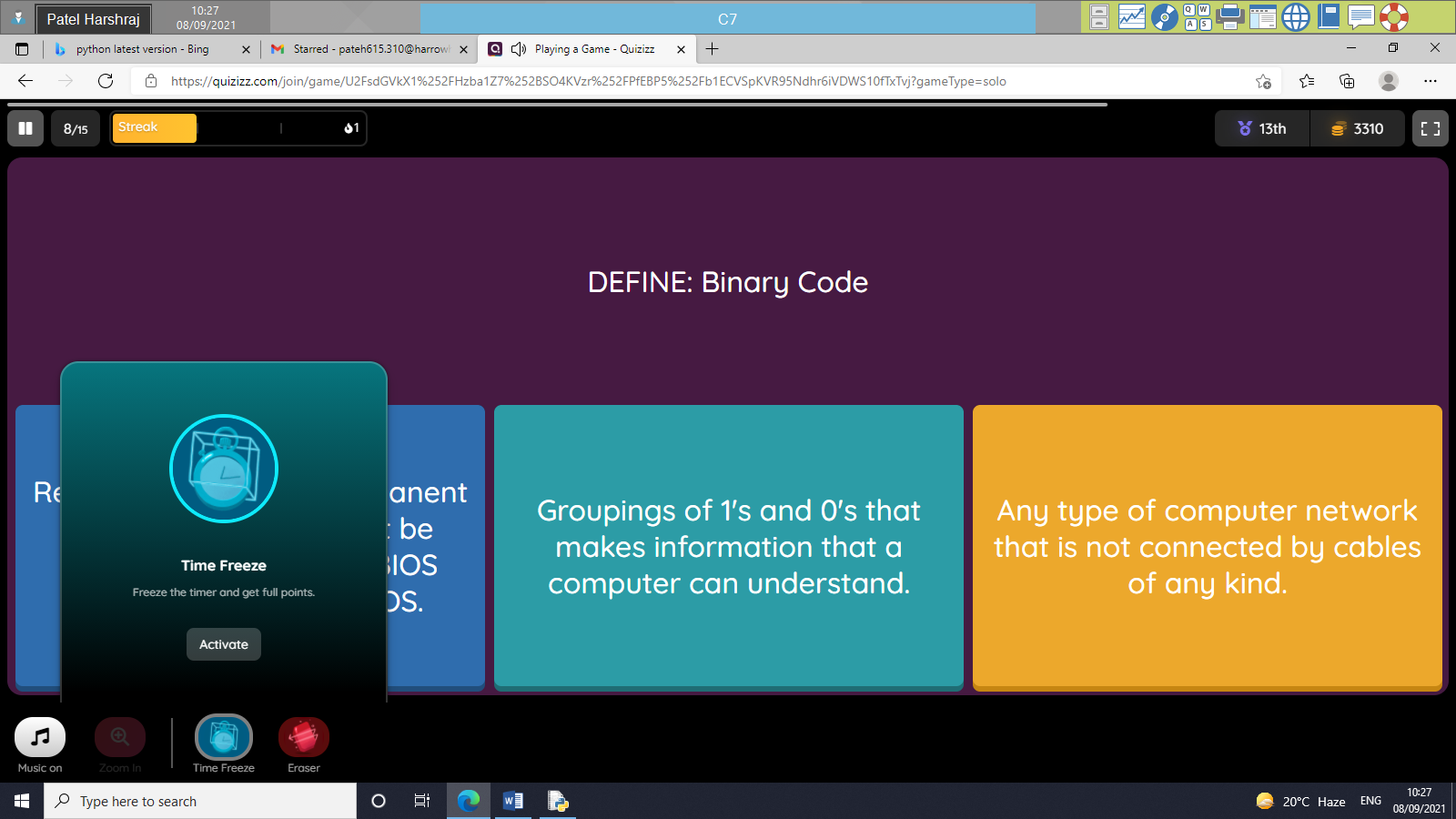
From this game, I would specifically like to implement the map designs as it is quite simplistic which makes it easy to understand and navigate. The simple character movement will also be implemented such as moving up, down, left, and right along with the appearance of enemies on the map which in my case will act as an obstacle to be overcome by answering the questions that pop up.

Reviews: <https://store.steampowered.com/app/1563720/Captain_Coffer_2D/>

**Quizziz:**

Quizizz is an internet quizzing program that empowers users to play, make and offer tests with different players. This free stage has many appraisals on an assortment of subjects composed and shared by different users. Unique access codes additionally permit groups of users to play the quiz simultaneously and is specially designed to be student-paced. Each member will see a question from a collection of questions and the alternative options for those specific questions will likewise show up on the screen. They should answer it by choosing at least one of the options and understudies will be stamped dependent on their speed and precision if the clock for the test is turned on. In case this is disabled, they can focus on getting the correct answer.

Additional features consist of a streak and score display along with occasional power ups that the user can use, aimed to maintain user engagement. These power-ups each provide unique affects to aid the user.



**Parts that can be applied to my solution:**

Quizziz’s use of concise questions to test users, is something that I will implement within my game as it’s similar to Dan’s prior experience with Anki, which other users are likely to also benefit from because of its straightforward user interface.

I would also like to implement power ups because they may prove valuable to enhance user motivation. However, with the addition of power ups, there is also the possibility of these being distracting to users, so I may not use these in order to balance my software. In addition to this, Quizziz provides a summary of how the user performed in a question after they have answered all the questions. On the contrary, evidence suggests that revealing the correct answer immediately after an attempt helps in storing that information in the long-term memory, hence I will adjust the feedback feature slightly so that an incorrect attempt gives the user an immediate response as to why it was incorrect. A competitive feature: there is a multiplier to the points gained based on how fast they complete the level. -> Use for dan Phillips interview -> add in iterative development after Dan finds it too easy.

## Features of the proposed solution

### Initial concept of my solution considering this research:

My solution will be an application that, when started, will display the user with the option to login or signup (with the data stored in a database). On the menu screen, a series of specification strands from the OCR AS level Computer Science specification will be displayed where each strand will display the user’s current accuracy percentage and clearance time to gain a brief idea into their strengths and weaknesses in the specification.

To make the learning engaging, I will apply platformer elements from Captain Coffer 2D, where the user can move around in a map and engage in melee and ranged combat with enemies. Upon defeating an enemy, the user will be prompted with a question with options, and after selecting an option, the user will be informed whether they were correct or incorrect, followed by appropriate feedback. Completing the topics will earn the student points, which will allow them to move up the leader boards. In addition, I will utilise Quizziz’s simplistic form of presenting questions, so the user isn’t overwhelmed in the question interface along with its use of vivid colours to distinctly tell the user if they got a question right or wrong.

### Limitations

One of my biggest limitations is the limited sample size of users. Despite choosing a representative stakeholder for my target audience, it is unlikely that their preferences and opinions will reflect those of every user. Therefore, with a limited number of testers/stakeholders coupled with their limited knowledge in the psychology of learning, it might be difficult to adapt the game to scaffold the learning of individuals in the wider audience.

Furthermore, as my game will require simplistic responses, along with the aim of mainly developing fundamental knowledge of each topic, the users will be limited to a certain style of short questions, inhibiting them from developing their comprehensive ability of answering extended exam style computer science questions. Although the users’ core understanding of each topic will improve, it may not develop their ability of answering questions under exam conditions.

## Computational Methods and Approach

### Identification and explanation of computational method

**Why the problem is suited to a computation approach**

For a variety of reasons, a computational approach can be used to implement a solution to the problem. The built platform will make use of several external libraries and modules, such as Pygame, to generate a graphical user interface (GUI). The platform will use a multitude of algorithms to generate features targeted at improving interaction. A strength-based algorithm, for example, will analyse the user's performance history to select the best collection of questions to effectively cement the understanding of concepts in the player's long-term memory. Another element that lends itself to a computational approach is the solution's reliance on libraries such as json, which allows for the management of data stores of json format in order to increase the fluidity and understanding of file handling throughout the software. A leader board system will be used where sorting methods for 2-dimensional arrays will be introduced as part of the scoreboards functionality sorting algorithm to sort the scores and act as a factor to encourage the users to perform better and provide a more immersive experience with the introduction of competition between other users along with trying to beat their personal scores.

**Thinking Abstractly**

The platform will include the implementation of a global scoreboard, which will be dedicated to ranking the top 10 performing users. These scoreboards will only include the information that is required, such as a user's visual position, username, and the points used to sort the users. Details such as their best question streak and the number of questions answered correctly are irrelevant and will be omitted from the scoreboard's initial display. Additionally, abstraction will be applied to the actual quiz window to ensure that only the most relevant information is displayed to the user. The user's performance history and current statistics for that question are essential, whereas a display of their progress in other specification strands/ chapters is unrelated and may be distracting.

Another example of abstraction is with the Computer Science content itself; the questions are designed to reinforce declarative knowledge retrieval and address common misconceptions, and abstraction is used to ensure that only the key information required for the A-level scope is included in the platform. For example, one of the disadvantages of Solid-State Drives is their limited read/write cycles, this is because of the deterioration of the oxide layer from which electrons pass to signal 1s or 0s. However, such degree of knowledge is not required to be known by the specification hence I will choose to ignore including excessive information as described above.

Several areas of the software will be abstracted during gameplay, such as damage and health numbers, movement speeds, and many other properties that the user does not need to see or be overwhelmed with while playing. Only important information, such as the current weapon, the user's health bar, and the health bars of the enemies, will be displayed because this is the only relevant data the user will require while playing.

Abstraction is also used when storing information about a level. Instead of storing all of the information such as the name, location, and type of tile in the map, when saving a map, it is exported as a 2D array which consists of sub-arrays which will contain elements that are integers. These integers are used as filenames to load images in the main game and indexing of the array is used to calculate the coordinates of each tile by using tiles’ x and y dimensions. This reduces the amount of data needed to be stored as algorithms will be put in place to decompress and analyse this level data, to make saving

**Thinking Concurrently**

Concurrent thinking will be used to improve the platform's performance in a few situations in this project. A good example is the updating of files during and after a quiz. Rather than updating the external file itself, a hash-map/dictionary data structure containing the student learning statistics is updated throughout the quiz. Furthermore, while the player is moving, enemy AI will be handling collisions and movement at the same time. Additional related tasks are completed concurrently as the game progresses.

**Thinking Ahead**

In terms of user inputs, the platform will only accept mouse and text inputs. Usernames and passwords will be accepted as text inputs and entered using a standard keyboard. The platform will be navigable by the user by pressing buttons. A mouse and keyboard are required input peripherals. The inputs are processed, and the only device required to relay this information back to the user is a monitor that displays the platform window. The display of a leader board, user progress on a graph, and the game level with quiz window incorporated with a question, and options are examples of outputs.

When generating game levels, it needs to be ensured that each level has the same number of enemies so that the same number of questions are asked on each level in order to fairly distribute points on each level. If there were to be a different number of enemies on a certain level, it is likely to give less points which may discourage the user from trying to attempt that level and act as a potential deterrent in picking certain topics.

Furthermore, the solution well accommodates the concept of reusability, which is important given the scope of the project and the development time available. Because the primary text inputs are the username and password (in both sign in and sign up), the function that takes the input could be generalised so that it could be reused to gain all the inputs when the appropriate set of arguments is passed during the function invocation. Aside from displaying various messages, it is critical that the password characters be displayed as asterisks and the username be displayed as the corresponding Unicode character. Creating reusable methods will also help development by reducing the need to write repetitively. An example can include buttons and textboxes being grouped. Because my software is likely to use these quite frequently, it would be very difficult individually manage large clusters of buttons and textboxes. Therefore, using a class for each and then placing them in a group can allow me to utilise polymorphism (since both will have common methods such as drawing, and mouse click checking) efficiently.

Furthermore, previously developed program components could be reused to simplify the development process. As part of developing the scoreboard component, for example, the platform requires sorting algorithms to be applied to sort a two-dimensional array. The password must also be strong enough to be checked against a set of password criteria, and because I have already developed these aspects of the platform in a previous project, I can reuse those functions to reduce development time. Because the previous projects were also written in Python, no changes to the subroutines are required to resolve any compatibility issues.

In addition, this platform will be managing many external files, caching is an important part of the planning process. To improve the efficiency of reading/writing to files, I've chosen to read the contents into a suitable data structure, where I can perform additional operations. When the data required to answer the questions is fetched, as in the strength-based algorithm, I will transfer the topic data into a dictionary and perform the subsequent computations against that data structure. Another example is that once a level has been completed or halted, I have chosen to update the relevant files at the end by storing any changes in variables, arrays, or dictionaries on order to avoid constantly opening, writing, and closing files.

**Thinking Procedurally**

I will also use divide and conquer to decompose the proposed solution into smaller sub-problems which are more manageable to solve making the development process easier as most sub-problems can be solved using a modular and/or object-oriented approach by implementing the lowest-level modules and integrating them into a logical set of components. These methods enable reusability throughout the code preventing code duplication and making it easier to test and debug certain features in isolation. Due to the constant presence of various computational techniques needed for my proposed solution, it makes the problem suited to a computational approach.

### Success Criteria

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strand** | **Requirement** | | **Evidence** | |
| Platform Access  and Main Menu | 1 | A main page - Provide the user with options:  - Sign up  - login  - Also validates credentials. | A screenshot of:  - Login/Sign Up fields  - Restricting access to unregistered users/incorrect login details.  - Denying invalid registration (username exists and/or their password is too weak) |
| 2 | Main menu should have the option to visit the user profile and access each level for each topic within the game.  - Sign out in the game menu. | - A screenshot of the different sections of the software the user can navigate to  - Sign out option in the game menu. |
| User Account and Data store | 3 | Users’ accuracy levels should be stored as a float. | - Screenshots of different user profile pictures with their number of points and accuracies |
| 4 | Store each user’s best time for each topic and update this if they perform better in a session | - A screenshot of all the best times being stored for each level. |
| 5 | Users’ points progress should be stored in a suitable format to allow tracking of previous attempts. | - Screenshot showing the ability to click on an option to show user progress graph. |
| Game-  Question preparation and display | 6 | Select questions categorised in red, amber, green determined by their accuracy in each question and select a certain quota for each difficulty | - A screenshot of an array showing different difficulty questions, relative to the user, being picked |
| 8 | The question should be displayed on a screen big enough to give room for the question to be loaded, along with the answer options. | - A screenshot of the question screen |
| 9 | If the user picks an incorrect option for a question, they shouldn’t gain any points, and it should then take them to a feedback screen explaining why they may have been wrong | - A screenshot of the feedback screen |
| Game -  Player gameplay, enemy AI, and Points | 10 | The user will be allowed to use a keyboard for navigation around the map. The user should be restricted to the confines of the level to prevent falling off the edge | - A video of player trying to move off screen but being prevented |
| 11 | During combat: User should have health reduced based on enemy’s damage. Enemy should have health reduced based on user’s current weapon’s damage. | - A video of enemy attacking player and health being reduced. Also containing user attacking enemy with a bow and sword dealing different amounts of damage. |
| 12 | Enemy entities should be bound to move within a certain radius | - A video of enemy entities only moving within a certain radius assigned to them. (doesn’t have to be equal for every enemy) |
| Game -  Level generation | 13 | There should be a software that allows levels to be created or edited and then save this data in another file which can be read in the main game file. | - Side by side screenshot of level generation software with a level and its abstract numbered format. |
| 14 | Implement tile collision with the user to make sure they don’t go through tiles. | - Video of level loading, with player in different scenarios such as jumping, falling and moving sideways but being restricted by tiles/blocks that are meant to be checked for collision |
| Leader boards | 15 | Leader board system which will rank all the current users based on their score. | - A screenshot of the leader board of all registered users. |

# Design

## Structure of the Solution

### An overview of the structure of the solution

The following is a refined version of the stepwise-refinement diagram

## Problem Decomposition

### Explanation of each of the modules, methods, procedures, and functions required

#### Platform Access

The platform access will be the first display that the users will be prompted. Here they will choose to either sign up, log in. This page’s design doesn’t need to be too complex, but just enough to look neat and professional as it will act as the medium for the user to access the login/sign up pages. The interface will consist of a series of boxes that the user can click which will then redirect them to the appropriate screen, with the option to go back if they wish.

**Procedure: main**

This procedure will be responsible for creating two boxes: login and signup drawing the two options on the screen and handling any user clicks to check which of the two boxes have been clicked.

**Procedure: update\_details**

**When a user logs in, if any changes have been made to the questions’ data such as adding in a question, this function should automate the process of adding in existing users’ data into all files to ensure there is no inconsistenty or missing information saving time of manually adding in this information.**

**Function: input\_information**

This function will be responsible for showing and updating the ‘username’, ‘password’ and ‘continue’ boxes for both the login and sign up stages. It will take in a parameter which will determine what state is being accessed, ‘login’ or ‘sign up’, according to this it will execute the appropriate sections of code, if the ‘continue’ button is clicked.

For both pages, if a successful attempt was made, this should return True, otherwise it should be False (in the case the user decided to go back). In the login state, if it returns True then it should progress the user onto the next stage which is the menu. If the user is in the sign up stage, and a successful sign up attempt was made, it should inform them that the newly entered details are now registered in the databse.

**Function: check\_details**

This function checks whether the details entered are valid or not, this is determined by the state the user is in. If they are in the login state, it should ensure that the username passed in is already existing by invoking the ‘presence\_in\_file’ function and that the password in the data store matches the one the user’s input. If they are in the sign up state they should ensure that there are no already existing usernames of the username passed in. If this check is passed, the procedure ‘add\_info’ is invoked.

**Procedure: add\_info**

Invoked once a new user has registered, taking in username and password as parameters and adds the data to the user data file along with setting the value for each topic (1.1, 1.2, 1.3, 1.4, 2) to and empty array which will contain the points the user obtains after each attempt at a topic. This will also iterate through each file that corresponds to a topic, go to each question in that topic, and add in the user’s information for that specific question in the format [right: integer, wrong: integer, accuracy: float] to record the user’s attempts and ability for each question.

**Function: presence\_in\_file**

Return True if a given key exists in a dictionary/hash-map (since this is the format json files store data in), otherwise returns False

**Function: validate\_character**

**This function is invoked every time the user presses a key whilst typing in the username or password box. This contains the criteria for all allowed characters because keys such as return, backspace, delete and other keys that aren’t symbols or characters should not be allowed. This will also add in another check to ensure that the string’s length is less than 15 to make sure it doesn’t exceed the boundary of 15 characters. If the checks are passed, then it returns True in which case the character is added to the appropriate field.**

**Function: validate\_username**

This function is used to return a tuple (a, b) where a = an error in length and b = error in username.

The boolean value ‘a’ will be set to True if the username passed in is outside the constraints of

4<= length <= 15. The boolean value of ‘b’ will be set to True if invoking ‘presence\_in\_file’ for the username returns True, as this means that there is already an existing user with that username in the data store.

**Function: validate\_password**

Will ensure that the password contains between 4 to 15 characters. **iterative development: a minimum of: one uppercase, lowercase, and number with the option to also add in special characters**

**Sign Up:**

EVALUATION: Once these details have been input, there will be a hashing algorithm which will encrypt the entered text. This way the data store will not store the actual (username: password), instead a (username:hashed\_password) combination. Upon completion, the user will be redirected back to the main menu, whilst the data store adds in these new user details. -> for security issues on a larger scale.

#### Main Menu

**Procedure: show\_menu**

Responsible for drawing all the topic/levels along with the user’s performance in them, and drawing the button which leads to the user’s progress graph. This will be achieved by creating an array which will contain ‘AutoBox objects of all topics, the leaderboard, and the username box. Events such as mouse clicks will also be handled to check which boxes have been clicked which will determine what parts of the code will be executed next.

**Procedure: show\_graph (In iterative development, change this so that a picture is saved and then blitted)**

This function is invoked when the user clicks on their username/points box on the top left should use a suitable Python module to use the user’s points progress and plot a graph which is then displayed to the screen for the user to monitor how well they peroformed at each level.

**Procedure: show\_leaderboards**

This is invoked when the ‘leaderbords’ button has beeen clicked. This procedure extracts the username and their points, storing them in a 2D aray, whereafter a bubble sort is used to sort all the users. After the sorting, the users and points are displayed on the screen in a suitable, and clean format.

**Function: get\_accuracy**

This function will be used to calcualte the average accuracy of a user for a certain topic. This is done by calculating the mean of all the accuracies and rounding the answer to 1 decimal place.

**Procedure: show\_instructions**

**This procedure is responsible for displaying key information about how the game works, such as user movement, how to identify an enemy, and other on screen data/text.**

**Function: check\_return**

This is used to check whether the back button has been pressed. Upon clicking the ‘back’ button, the user should be taken back to the portal access page as it indicates that they have logged out.

**Function: get\_topic\_boxes**

This function creates instances of a class ‘AutoBox’ which will be used to store the coordinates, size, and contents (text) of the levels/topics, these are all stored in an array and returned for them to be added into the group of boxes in the main function which will contain all the boxes on this screen.

**Function: update\_topic\_boxes**

**Edits the topic boxes such that after the user attempts a level, and their number of points changes, the boxes’ text which displays the points should also be changed so that it is up to date.**

**Function: get\_topic\_number**

This returns the specification topic number to which the topic’s title corresponds to:

Eg 1.1 == Systems Architecture, 1.2 == Software and Software development and so on. This used when querying the question files because they are stored in the specification number format to reduce name size and group questions more easily.

#### Level Gameplay

**Procedure: play\_level**

This is the main sub-routine for the level gameplay section of the code which will handle various sorts of events. It is responsible for displaying the tiles of the current level, the player, and all the enemies. There will also be text on screen which will show how many points the user has collected so far in the current session, along with how much time has elapsed. This loop will also handle the triggering of showing questions and updating any changed data to users’ data store and all the questions’ data store too. At the end this will be responsible for invoking the ‘show\_summary’ which will show the summary statistics upon level comlpetion or user death. This will also be responsible for triggering the ‘update\_data’ at the very end to save changes.

**Function: load\_level**

This is responsible for reading the data from the level file(s) and storing them in a 2D array. This will consist of numbers which will represent empy spaces (-1) or images/tiles (integers >= 0) in each sub-array of the 2D array. Each sub-array points to a row, and each element in each sub-array points to a coloumn to generate a grid of values.

**Function: get\_time\_units**

**This should be used to add units to the time. If the time contained only seconds, then it returns ‘seconds’. If there’s minutes as well it should return ‘minutes’ and if there are hours (this is highly unlikely to happen, but just in case) it should return ‘hours’. Returning the units based on the highest unit of time.**

**Function: convert\_time\_format**

This takes in the time (in seconds) and converts into an hh:mm:ss format which is used when displaying the timer in the level gameplay and also when showing the best times in the main menu for each topic.

**Function: get\_questions**

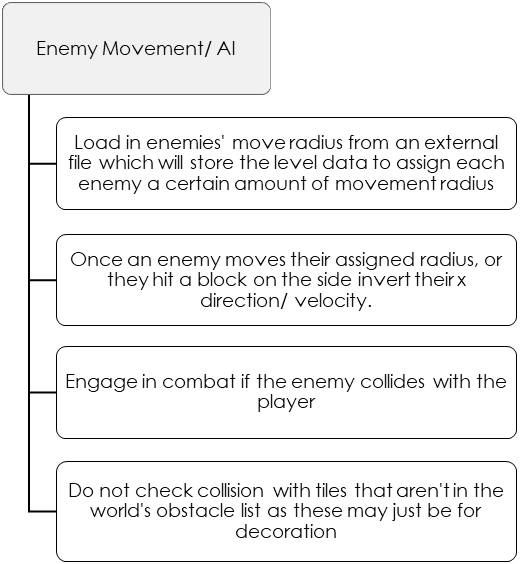
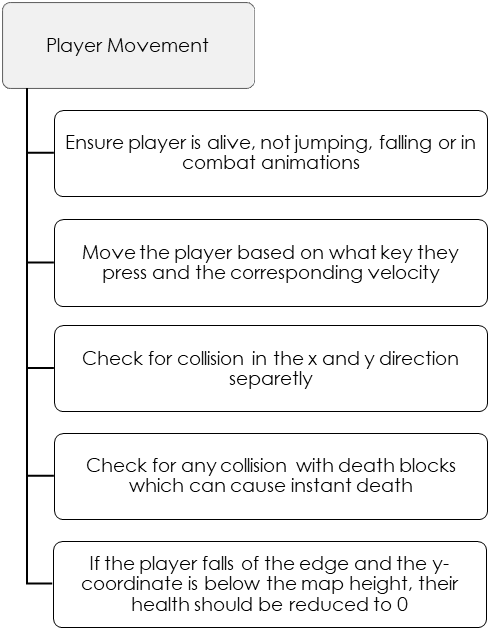
This function extracts a set amount of random questions from the question bank and returns them in the form of an array. Because the questions will be stored in a json file, these questions will be used as ‘keys’ when accessing information related to question data. A strength based algorithm is used to filter questions into red, amber and green categories where each category has a certain quota that needs to be filled. Add in iterative development: use of carry-overs if a quota hasn’t been filled + shuffling questions to introduce randomness

**Procedure: show\_summary**

This function shows a concise summary of key statsitics after the current session is finished.

**Procedure: update\_data**

This procedure adds on points to the user’s account along with ammending any changes made to the user’s statsiticis for each question which will then be written to an external file that holds the questions for the selected topic.



The above is a brief decomposition of how the Player and Enemy movement will occur. Although almost all the code which involves the player and enemy’s movement or interaction will be handled between classes, the code where some of the key methods will be called and movement conditions are determined must be found in this section of the code.

#### Question Display

**Function: start\_question**

This function is responsible for deciding what to display, this is between the question page or the feedback page. Initially, the question and the possible options for that question to the user are displayed as shown in the design above. Since this is the main function for this section it will also handle checking clicks on the option boxes and change the colours of the boxes to display which one was correct (represented by a green colour) and which ones were incorrect (represented by a red colour). This then moves the player onto the feedback screen if they got the question wrong, otherwise send them straight to back to the level gameplay.

**Function: get\_boxes**

This function creates instances of a class ‘AutoBox’ which will be used to store the coordinates, size, and contents (text) of questions, options, and the feedback. After doing this an array will be created to allow these boxes to be handled more cleanly in the StartQuestion function as they will all behave in a similar way, since polymorphism through overloading will take place for each of the boxes.

**ADD IN ITERATIVE DEVELOPMENT: DISPLAY THE USER’S ACCURACY FOR THE QUESTION AND IF THEY GOT IT RIGHT OR WRONG IN TEXT ON THE BOTTOM OF THE QUESTION BOX OR SOMETHING.**

#### Other General Functions and Procedures

The functions listed above are the critical ones, and they will change as the project progresses, as these are merely outlines. The subroutines listed below are accessed and used at various points throughout the program.

**Function: check\_return**

This is used to check whether the back button has been pressed. Upon clicking the ‘back’ button, the user should be taken back to the previous page. If this occurs in the main menu, then they are to be taken back to the beginning of the platform access screen, not the login screen (which was the previous page)

**Function: read\_json**

Reads and returns the data of a json file in the form of a dictionary.

**Function: write\_json**

Writes a given dictionary into a json file.

**Function: bubble\_sort**

Used to sort 2D arrays form smallest value to the largest value.

**Procedure: delete\_user**

Invokes ‘delete\_json\_key’ procedure which is used to remove all occurrences of a certain user across all the question files and users file to maintain referential integrity.

**Procedure: delete\_json\_key**

This goes through every question in every topic and deletes all data about the specified user. Also goes to the users file and removes the record of the specified user.

Because the platform contains numerous objects, it will make use of a variety of classes and methods. Classes will be created to represent entities such as boxes, players, and enemies, as well as text entry fields. These are discussed in greater detail in Key Variables and Structures.

### A justification of the approach taken

The platform will feature a one-page login portal to address the problem of other users being able to brute force their way into someone else’s account. They would only have to enter one existing registered username and continue entering passwords until they find a successful match to potentially gain access to someone else’s account. Using a one-page sign in makes it so that anyone who tries to get in will need to try a variety of different combinations of usernames and passwords to get unauthorised access, which is far more difficult and time consuming and likely to deter such actions from taking place.

The platform's design will be consistent, with pages adhering to the overall house style, using the same base template for backgrounds after platform access and text boxes. The font consideration is also important in ensuring consistency; I chose the 'Sans' font family based on its legibility, media, family, and typeface size. Other considerations include the use of a consistent set of colours to maintain the initial aesthetic appeal and the inclusion of a traditional menu for easy navigation.

Furthermore, certain features were chosen because their inclusion feels natural given their presence on similar platforms. This can be in the form of using username and password criteria, to including quotas as part of the strength-based algorithm. Furthermore, it is critical that all the questions for a specific quiz be generated at once, because generating them individually (after each question's attempt) may contradict the principles of the strength-based algorithm and introduce unnecessary complexities if they are resolved in real time. This approach also maximises the reusability of multiple aspects of the solution because certain classes and procedures, such as the 'Display' class and 'draw text' method, are regularly used throughout the project.

Moreover, storing all the information about tiles in a one-dimensional array instead of its original two-dimensional format by utilising tile classes for separate tiles allows for more efficient processing times because all the calculations are only performed once (Storing the images and calculating where the co-ordinates are). Thus, after all the information being stored in a one-dimensional array, each time the tiles need to be drawn, this can be done in a single iteration where the required information is already loaded and stored. This makes the overall gaming experience smoother and less intensive on the hardware making it more accessible by a wider range of computers.

Another factor to consider is loop optimisation, which is accomplished by avoiding processor-intensive tasks such as calculations and file access from being performed within counter and condition-controlled loops constantly. The updating of user performance and subsequent question/ topic progress calculation is performed only when the user comes back from playing a level in ‘show\_menu’, where the contents of the each of the boxes (topic boxes, and username + points box) are updated to reflect any changes in the user’s progress. This is far more practical and less processor intensive than having to get the user and question progress at each iteration and attempting to update all the boxes constantly, when doing so would only harm the performance of the software and user experience.

Updating the contents of the json files which contain the question information and users’ information after every attempt also necessitates careful consideration. This is because doing so would result in the degradation in the overall performance of the software and may even make it unresponsive at times because too many events are occurring at once. Alternatively, by deciding to use a dictionary to extract the json files not only does it make the reading/writing process easier (because Python dictionaries can be directly written to json files without any extra formatting required using the json module), but it also allows an efficient data structure to be used to store and query statistics during level-play and finally saving the contents to an external json file once the summary statistics have been displayed.

As mentioned above, I have decided to use a json file format to store the data about the users and the questions, but I have also made this my main data structure that will be used when loading data from external files. This is because constantly using indexes of lists can get very confusing, however using a dictionary when possible enables very easy, intuitive, and efficient manipulation of data due to their use of key-value pairs via their Hash-Map like feature which also results in faster O(1) access times, which can be more efficient that carrying out a search in an array.

To sort data, the solution will employ a bubble-sort algorithm. Because it has the same space and time complexities as an insertion sort, this algorithm is appropriate. This code already employs the more efficient, condition-controlled outer loop method, which increases the average-case time complexity.

**The solution also includes an automatic fill-in data sub-program that adds in any missing data about users into the questions’ data store which makes it easier to add/remove or modify questions for each topic as I do not have to manually enter a new value for all existing users’ statistics for every new question added.**

To further increase the maintainability of the code, I have modularised by approach to a satisfactory level using self-contained modules that don’t depend on other modules. This modular design further simplifies the development, testing, and maintenance of the code.

Furthermore, I have chosen to go with the components: ‘1.1’, ‘1.2’, ‘1.3’,’1.4’, and ’2’ from the AS Computer Science Specification. I decided to exclude 1.5 which is about Ethics and Laws; whilst discussing discussing with Dan, he didn’t seem very interested in topic 1.5 and I believe that it may prove to be a topic that not many users decide to select, thus leading to an inefficient use of development time when creating levels and questions. Furthermore, I decided to envelop all of comp 2 into one topic because a majority of it is programming and trying to come up with a sufficient number of questions for certain topics may be really difficult considering the main style of my game is multiple choice questions.

## Key Variables and Data Structures

### Identification of the key classes and attributes

#### Class: Display

This class is used to create a Pygame surface where objects, shapes, text, and images can be drawn.

Key attributes:

- size - stores the width and height of the surface

- big\_font, medlarge\_font, medium\_font, small\_font - used to render font to a certain size

- back\_x, back\_y - the coordinates of where the back button is located

- screen - a Pygame.Surface object which is where everything will be drawn on.

- background - this can be an image or a colour

- back\_image - the image for the back button

- back\_rect - stores the rectangle of the back button

**draw\_back()** (procedure) draws the back button. If a position is passed in then its drawn there, otherwise its drawn at the coordinates using the pre-made class attributes (back\_x and back\_y).

**refresh()** (procedure) draws the background and invokes the ‘draw\_back()’ procedure**.**

**check\_return()** (function) checks if the user has clicked the mouse button and if the position of the mouse is overlapping with the ‘back\_rect’ attribute, it returns True, else False.

**draw\_text()** (procedure) takes in a certain text, position, and size which is then drawn at the position specified using the font to render the text and then draw it on the class attribute ‘screen’.

draw\_back()

refresh()

check\_return()

draw\_text()

size

big\_font

medlarge\_font

medium\_font

small\_font

back\_x

back\_y

screen

background

back\_image

back\_rect

Display

#### Class: Textbox

show()

set\_properties()

check\_click()

Inherit fonts sizes from Display class

background\_colour

border\_colour

text\_colour

text\_colour

padding

size

text

Textbox

Key attributes:

- background\_color - this is the default background colour

- border\_colour - this is the default border colour

**- hover\_colour - this is the colour that the background of the textbox changes to if there is hovering**

- text\_colour - this is the text colour

**- border\_hover - the colour of the border changes to this when its being** hovered

- padding - this is used to create a padding around the text

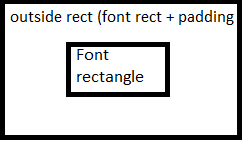
- size - the size of the textbox

**create\_rect()** (procedure) this creates a rectangle for where the text will be drawn, and where the properties of the outside rectangle which takes into account the font size and padding.

**show()** (procedure) draws all the information of the textbox on a specified surface.

**set\_properties()** (procedure) used to change the values of background, border colour, and hover\_colour

**check\_click()** (function) checks if the mouse position overlaps with the textbox and that the left mousebutton has been clicked



#### Class: AutoBox

This is similar to the Textbox, but it’s used in the main menu and the quiz window of the software due to its different functionality and use case.

show

add\_text

update\_text

**check\_hover**

check\_click

Key attributes:

- x, y - pixel coordinates of where the box is to be located

- obj\_type - this stores what type of a box it is. Used in other parts of the code

- surface - this is the local surface on which the text will be drawn on

- text - the actual string text that is to be drawn on the box

- text\_rect - gets the rectangle of the rendered text in the font

- check\_collision - this is a boolean value that is used to signal whether a collision with this button should be checked or not.

**show()** (procedure) this is used to blit the box on the specified surface. This is similar to the Textbox show, but because the text to be displayed is drawn on a separate surface which is then loaded on to the specified surface, it requires a different chain of logic.

**add\_text()** (procedure) this is an automated algorithm that splits text by spaces and attempts to draw text onto a text surface (which is scaled down from the local surface to create a padding). If the word cannot fit, it and the following words are drawn further down in the text surface. The text surface is then drawn onto the local surface, which is then drawn onto the main specified surface.

**update\_text()** (procedure) this is used when a change has been made to the text contents of the box, triggering the “add\_text” method to draw the updated text.

**check\_hover() (procedure) ensures that the “obj\_type” is not a question or feedback because these should not be checked for clicks or hovers.**

**check\_click()** (function) checks if “check\_collision” is true and the “obj\_type” is not a question before checking for mouse collision, because collision should not occur for the question box. If these cases are met, then the collision is checked along with checking if the left mousebutton was pressed to register a valid click.

AutoBox(Textbox)

Inherits from Textbox

x,y

obj\_type

surface

text

text\_rect

check\_collision

#### Class: Tile

This class is used to store data about each tile with an in-built show method which reduces overall code size in the main sub-program

Tile

Key attributes:

- rect – this stores the x, y, width, and height in the form of a Pygame.Rect object

- mask – this is the pixel outline of the tile image

- image – this is the actual png file/ data of the tile image

**draw ()** (procedure) this draws the tile on the specified surface using the “rect” attribute which stored the important metadata for the tile.

draw

rect

mask

image

#### class: Entity

This is a template for the player and enemy as there are many similarities between the two.

move

animation\_handling

update\_action

update

get\_animations

draw

check\_collision

all\_animations

combat\_animations

animations

health

obj\_type

direction

x\_vel

y\_vel

Key attributes:

- all\_animations – an array of strings which contains which animations the entity will execute.

- combat\_animations – the series of animations where the entity is engaging in combat

- animations – a 2D array where each array contains a series of images to show the animation.

- health – this tracks the player’s current health

- obj\_type – an identifier to see what type of entity it is

- direction – used to identify which direction the user is facing

- x\_vel, y\_vel – the movement velocities for the entity

**move()** (procedure) calculates the proposed movement of the entity and checks for collision in the x and y direction separately. if a collision occurs, the movement in that axis is cancelled.

**animation\_handling()** (procedure) this is responsible for showing different parts of an animation.

**update()** (procedure) checks what state the entity is in and invokes ‘update\_action’.

**update\_action()** (procedure) changes what animation the entity is in

**get\_animations()** (procedure) adds a one dimensional array to the class array ‘animations’ for each animation in “all animations”

**draw()** (procedure) draws the entity based on where they currently are on the map.

**check\_collision()** (function) checks for mask (pixel perfect) collision between two entities or an entity and a tile object.

Entity

#### class: Enemy

This will inherit from the Entity class but have some extra methods and overload some too.

Key attributes:

- move\_radius – this is the only unique attribute for the Enemy class as this determines how many blocks the enemy will be moving

**update()** (procedure) checks if the enemy is alive, and checks for collision with the player to invoke ‘start\_attack’. This also checks for collision whilst the enemy is in the attack animation using the Entity class’s “check\_collision” method.

**AI()** (procedure) This controls the enemy movement. After the enemy travels a certain number of blocks, they change direction and walk the same number of blocks again.

**draw()** (procedure) draws the entity based on where they currently are on the map.

**start\_attack()** checks for collision with the player and makes sure that the enemy isn’t in the air. This then changes the enemy’s animation to the attack animation and sets the waiting timer back to 100.

start\_attack

AI

draw

update

move\_radius

Enemy

#### class: World

This class stores all the information related to the world such as all the tiles it contains and the background.

Key attributes:

- tile\_list – this stores an array of Tile objects

**process\_data()** (procedure) loops through an array that stores the level information and uses values stored in the array to generate images and Tile objects which are appended to tile\_list.

**draw()** (procedure) iterates through each tile in “tile\_list” and draws it on a specified purpose using the information from each Tile object.

process\_data

draw

World

tile\_list

### Explanation and justification of key variables & data structures

|  |  |  |
| --- | --- | --- |
| Name | Data Type | How it is used |
| username | A variable which holds a string format of the current user’s username. | This is used through out the code to access information about the user and modify any changes too. |
| WIDTH and HEIGHT | Attributes which are Constants stored in the ‘Display’ class that store the dimensions of the Pygame display surface | Used to determine coordinates of objects, allowing them to be aligned and centred. |
| topics\_num | Dictionary which maps topic names to their specification numbers | Used when trying to access question data because each topic’s questions are stored in a file based on the specification number |
| box\_group | Array used to store Textbox or AutoBox objects | Used throughout the code when handling multiple boxes so that each box can be accessed by iterating through the array. |
| GREEN, RED, … | Multiple tuples that store RGB values | Allows for easier and consistent access of colours throughout the entire software |
| TILE\_DIMENSION\_X and TILE\_DIMENSION\_Y | Attributes of Display class. These store the pixel width and height of one individual tile of the map | This will be used in several parts of the code to determine the WIDTH and HEIGHT of the display along with the calculation of the coordinates for each tile whilst the data is being processed |
| mask | An attribute for tile, and entity classes to store the pixel perfect data about each image. This is a Pygame.Mask object | Is key when performing collision detection to ensure precise and accurate collisions occur. |
| run | A boolean value which is used to check if a certain block of code is to be kept running | This is used in almost every condition controlled loop within the game such as the platform access, main menu, and game menu. When the user wishes to go back (or dies in the game level) this run variable is set to True. |
| question\_data | A dictionary data structure which each question and its information for a certain topic. | This is used constantly to access/modify important important information such as user’s right,wrong and accuracy ratings for each question, feedback for each question and the possible options |
| **and atttack\_conditions** | **Variables storing boolean values move\_conditions** | **Used to check if move and attack conditions can be carried out. Ie if the player is in the air they should not be allowed to do either. Or if they are in a combat animation, they should not be allowed to move.** |
| animations | A 2D array which holds a series of images of all the animations for an entity | Used when updating the current animation for each entity |
| rect | A Pygame.Rect object which holds the (x, y, width and height) for almost all classes. | This is used when drawing or modfiying coordinates due to its versatility and variety of methods available. |
| collisions | A variable which stores a boolean value. Holds the number of collisions that occur between the player and an enemy. | Used to distribute damage to both entities. |
| remove | Another boolean variable which indicates when an entity is to be removed (deleted from memory) | This is triggered when an entities’ death animation finishes fully to trigger their removal |
| health | A variable which holds integer values about an entity’s health | Used to calculate how strong/weak the enemy is and check if they are alive. |
| show\_question | A variable which holds boolean values | Used to select a question and trigger the quiz window after an enemy is killed. |

## Algorithms

***anything in red and not beginning with # == use for iterative development to fix errors***

|  |  |
| --- | --- |
| Platform Access | |
| main (Procedure) | **Invoke update\_details to update user details in all topic files** # fill in any missing information  instantiate Textboxes for login and sign-up options # navigation  Start loop  Refresh screen # fill in with a colour or with a background image  Check for events such as mouse clicks (for textboxes) and exiting  Show **and** **check hover** for the textboxes  Invoke input\_information() depending on which textbox was clicked  Invoke show\_menu(…) based on input\_information’s return value  Update display |
| **update\_details (Procedure)** | **For each USER in users.json**  **For each TOPIC in topics**  **For each QUESTION in path TOPIC:**  **If USER not in QUESTION: QUESTION[USER] = [0, 0, 0]**  **# if a user’s data doesn’t exist for a specific question, set their right, wrong and accuracy to 0.** |
| input\_information (Function) | Username text = ‘’  Password text = ‘’  Create username Textbox as username\_box  Create password Textbox as password\_box  Create continue Textbox  Password\_click = Username\_click = False  Incorrect\_details = display\_string\_length = display\_password\_text = False  Start loop  Refresh window  Update username\_box’s text to Username\_text  Update password\_box’s text to Password\_text    Check for mouse clicks  IF mouse has been clicked and clicked on username\_box:  Username\_click = True  ELIF mouse has been clicked and clicked on password\_box:  Password\_click = True    Check for key presses  Unicode = Get the Unicode of the key  **If validate\_character(Unicode) == False: skip the below code for the key presses**  IF username\_click:  IF key == delete: delete the last letter from Username\_text  Else: Username\_text += Unicode  ELIF password\_click:  IF key == delete: delete the last letter from Username\_text  Else: Password\_text += Unicode  IF incorrect\_details and state == ‘login’ :  Draw text on window to show incorrect username or password  IF incorrect\_details and state == ‘sign up :  Draw text on window to show username is already taken  IF display\_string\_length:  Draw text on window to show the username or password isn’t long enough  IF continue button was clicked:  valid\_username = validate\_username(Username text)  valid\_password = validate\_password(Password text)  IF not valid\_password and state == ‘sign up’:  display\_password\_text = True  ELIF not valid\_username and state == ‘sign up’:  incorrect\_details = True  else: ## if the username and password met the criteria ##  correct\_details = check\_details(Username text, Password text, state)  IF correct\_details and state == ‘sign up’:  invoke add\_info(username, password)  return username  ELIF correct\_details and state == ‘login’:  return username  update display |
| check\_details (function) | Data = Extract user details  Username\_data = check presence in file Data for username text  IF state == ‘login’:  IF there is no Username\_data: return **False** as no users by this username were found  IF username\_data[‘password’] == password text: return **True**  Elif state == ‘sign up’:  IF there is Username\_data: return **False** # the username is already taken  Else: return **True** |
| add\_info(procedure) | Data = read\_json for users.json  Data[username\_text] = {‘password’: ‘ ’, ‘points’:[ ], ‘1.1’: 0, ‘1.2’: 0, ‘1.3’: 0, ‘1.4’, ‘2’: 0}  For each FILE in topics  Questions = read\_json for FILE  For each question in FILE  Questions[username\_text] = [0,0,0]  write\_json to FILE -> Questions  write\_json to users.json -> Data |
| presence\_in\_file(function) | Data = read\_json for users.json  IF username\_text in Data: return **True**  Else: return **False** |
| **Validate\_character(function)** | **valid\_character = regex expression matching the character to valid characters**  **return valid\_character # False if the character was valid, otherwise False** |
| validate\_username(function) | return 4 <= username\_text <= 15 |
| Validate\_password(function) | Regex expression to check if password length is between 4 and 15 characters, and that there is at least one uppercase. |

|  |  |
| --- | --- |
| Main Menu | |
| show\_menu (Procedure) | Topics = get\_topic\_boxes(…)  Username\_box = AutoBox(…)  Leaderboard\_box = AutoBox(…)  Instructions\_box = AutoBox(…)  Topic\_boxes = [AutoBox(…) for each topic in Topics]  Leaderboards = graph = **instructions =** False  Start loop  Refresh window  check for mouse clicks  clicked = [topic.check\_click(…) for each topic in topic\_boxes]  IF clicked.obj\_type == 'topic':  topic number = get\_topic\_number(...)  play\_level(...)  ELIF clicked.obj\_type == 'leaderbord':  leaderboards = True  ELIF clicked.obj\_type == 'username box':  graph = True  **elif clicked.obj\_type == 'instructions':**  **instructions = True**  IF the back button was pressed:  return  IF leaderboards == True:  show\_leaderboard(...)  ELIF graph == True:  show\_graph(...)  **elif instructions == True:**  **show\_instructions(...)**    for topic in topic\_boxes:  **topic.check\_hover()**  update display |
| show\_graph(procedure) | All\_users = read\_json for users.json  User\_data = all\_users[username]  Points\_array = User\_data[‘points’]  Points = sum(Points\_array)  Sessions = list(1 to length of Points\_array)  Use matplotlib to plot and show a graph of Points on the y axis and Sessions on the x axis. |
| show\_leaderboards | All\_users = read\_json for users.json  Users\_points = []  For each USER in all\_users:  Points = sum(User[‘points’])  Users\_points.append([User,Points])  Top\_ten = invoke bubble\_sort2D(…)  For USER, POINTS in top\_ten:  Render USER and POINTS onto the main screen |
| get\_accuracy(function) | Total\_attempted = 0  Accuracy = 0  Question\_data = read\_json for question  For QUESTION in Question\_data  stats = Get user stats for QUESTION -> [right, wrong, accuracy]  total\_attempted += 1  accuracy += stats[2]  final\_accuracy = (accuracy/total\_attempted)\*100  return final\_accuracy |
| **show\_instructions(procedure)** | **Get the image file which contains all the instructions**  **Scale the image size to fir the window**  **Draw the image on the screen at (0,0)** |
| check\_return(function) | Check mouse collision with back button  IF a collision has occurred:  Take user back to platform access  Else:  Continue as normal |
| get\_topic\_boxes(function) | topic\_boxes = []  row\_1 = list of topic names that are in the first row  row\_2 = list of topic names that are in teh second row  for TOPIC in row\_1  topic\_number = invoke get\_topic\_number(...) for TOPIC  question\_data = read\_json for Questions/{TOPIC}.json  accuracy = invoke get\_accuracy(...) for TOPIC  create AutoBox(...) for TOPIC -> append to topic\_boxes  for TOPIC in row\_2  topic\_number = invoke get\_topic\_number(...) for TOPIC  question\_data = read\_json for Questions/{TOPIC}.json  accuracy = invoke get\_accuracy(...) for TOPIC  create AutoBox(...) for TOPIC -> append to topic\_boxes  return topic\_boxes |
| get\_topic\_number(function) | num = get value of key TOPIC from topic\_num dictionary  return num |
| Level Gameplay | |
| play\_level(procedure) | TILE\_TYPES = store a list of all the images for the current level  background = load the background image  window = create a Display object  img\_dict = store the tile name as key and its image as the value  world = instantiate World object  questions, question\_data = invoke get\_questions(...)  player, death\_blocks, enemies, portals = invoke world.process\_data(...)  arrows = []  points = 0  timer = 0  streak = 0  max\_streak = 0  time1 = get current time  total\_right = 0  total\_wrong = 0  Start loop  invoke player.check\_alive(...)  **move\_conditions = player is not attacking and is alive**  **attack\_conditions = player is not already attacking**  refresh window  draw background on window  event handling  IF user decided to exit:  return  IF user presses movement keys **and move\_conditions are met**:  IF key is 'a':  move the player to the left  IF key is 'd':  move the player to the right    IF user presses jump key **and move\_conditions are met**:  set player.jumping to True  IF user presses attack key **and attack\_conditions are met**:  IF the current weapon is a sword:  set player.sword\_attack to True  IF the current weapon is a bow:  set player.bow\_attack to True  set a shooting cooldown timer to roughly 1500 milliseconds    IF the user presses number keys:  IF the key is 1:  set the current weapon to Sword  IF the key is 2:  set the current weapon to Bow    IF mousebutton is pressed:  invoke window.check\_return(...)    IF player attacked and it was a bow:  add a Projectile object to arrows  invoke draw, and update move methods for player, enemies, amd arrows  death\_blocks.update(…) -> pass in player to check for collision  portal\_collision = invoke portal.update(…) -> pass in player to check for collision  IF player collides with the portal **and all questions are answered**:  run = False # don't run the loop anymore  portal\_enter = True    check player collision with alive enemies  IF a collision occurs:  question = pop a question from the questions stack  result, time\_taken = invoke start\_question(...)  right,wrong,accuracy = get users current stats for question  IF result == True: # IF the user got the question right  right += 1  streak += 1  total\_right += 1  else:  wrong += 1  update max\_streak  streak = 0  total\_wrong += 1  accuracy = (right/wrong)\*100 # percentage  update user's new stats in question\_data dictionary  timer += add the time taken to answer the question    current\_time = get current time  IF time1 - current\_time >= 1000: # IF 1000 milliseconds have passed  timer += 1  time1 = current\_time  update the display  total\_accuracy = (total\_right/total\_wrong)\*100  invoke show\_summary(...)  invoke update\_data(...) |
| Load\_level(function) | Level\_data = read\_json for current level  array = Use csv.reader to split each number by ‘,’ and each row to form a 2D array  return array |
| **get\_time\_units(function)** | **split the time from ':' to get (hh,mm,ss)**  **highest\_unit = 'seconds'**  **IF there were minutes:**  **highets\_unit = 'minutes'**  **IF there were hours:**  **highest\_units = 'hours'**  **return highest\_units** |
| get\_questions(function) | question\_data = read\_json for current topic  invoke bubble\_sort2D on question\_data  green = filter questions where user’s accuracy IF >= 75%  amber = filter questions where user’s accuracy is 50% <= accuracy < 0.75%  red = filter questions where user’s accuracy is < 50%  select 4 random questions from each list  return [green + amber + red], question\_data |
| show\_summary(procedure) | timer = invoke convert\_time\_format(...)  accuracy = accuracy+'%'  **units = invoke get\_time\_units(...)**  Start loop  refresh window  event handling  IF user clicked a mousebutton:  invoke window.check\_return(...)  IF they clicked on the back button:  return  IF they user decided to close the game:  terminate the running software and close out of Python    invoke window.draw\_text(...) for each of the statstics:  ['Right','Wrong','Accuracy','Best Streak','Points Gained','Time Survived']  invoke window.draw\_text(...) for each of the values:  [right, wrong, accuracy, streak, points, timer]  update display |
| update\_data(procedure) | invoke write\_json -> question\_data to path: Questions/{current level}.json  user\_info = invoke read\_json on users.json  current\_best = get users current time for the topic  **not considering accuracy: need to ensure 100% accuracy is achieved**  IF the player entered the portal:  IF the new timer is lower than the current\_best or current\_best == 0:  set the current\_best to new\_timer  user\_info[username][current level] = current\_best  IF at least a question was answered:  append points to user\_info[username]['points']  invoke write\_json -> user\_info to users.json |
| Question Display |  |
| start\_question(function) | correct\_answer = first option of current question's options  main\_group = invoke get\_boxes(...)  feedback\_box = instantiate AutoBox(...)  main\_continue = instantiate Textbox(...)  feedback\_continue = instantiate Textbox(...)  invoke create\_rect(...) for main and feedback continue  time1 = get current time  timer = 0  result = None  options\_screen = True  feedback\_screen = False  Start loop  refresh window  event handling  IF the user wants to quit and they havent picked an option yet:  sys.exit()  IF the user presses a mousebutton:  IF the user clicks on the main\_continue box:  don't display the options screen anymore  options\_screen = False  IF the user clicks on the feedback\_continue box:  return result, timer  IF the user clicks on one of the options and check\_click:  result = (clicked box's text == correct\_answer)  check\_click = False  for BOX in main\_group:  IF **BOX is an option and** BOX's text == correct\_answer:  set the background colour to 'correct\_colour'  set the border colour to 'correct\_colour'  ELIF ***Box is an option*** ***and*** Box's text != correct\_answer:  set the background colour to 'incorrect\_colour'  set the border colour to 'incorrect\_colour'  BOX.check\_collision = False  IF options\_screen:  for each BOX in main\_group:  show BOX  **check hover for BOX**  show main\_continue  ELIF feedback\_screen:  show feedback  show feedback\_continue  **check hover for feedback\_continue**  current\_time = get current time  IF current\_time - time1 >= 1000:  timer += 1  time1 = current\_time  invoke window.draw\_text(...) to show time elapsed  update display |
| get\_boxes(function) | text = current question  w, h = width of the window, an appropriate arbitray number # width and height of question box  options = get all options for current question from question\_data  question\_box = Instantiate AutoBox(...)  x\_padding = 30  x1 = x\_padding # x coordinate offset for boxes on the left of the center  x2 = (w//2) + x\_padding # x coordinate offset for boxes on the right of the center  option\_width = (w//2) - (2\*x\_padding) # the width of the box with remaining space  option\_height = (window.HEIGHT - question\_box.height) \* (arbritary number <= 0.5)  options = shuffle the order of options  option\_1 = instantiate AutoBox(...) -> (x,y): (x1, y)  option\_2 = instantiate AutoBox(...) -> (x,y): (x2, y)  option\_3 = instantiate AutoBox(...) -> (x,y): (x1,option\_1.y + padding)  option\_4 = instantiate AutoBox(...) -> (x,y): (x2, option\_2.y + padding)  main\_group = [option\_1, option\_2, option\_3, option\_4]  return main\_group |
| General Functions and Procedures |  |
| convert\_time\_format | h, r = divmod(time, 3600) # calculate how many hours, and the remainder is the number of minutes (in seconds) m, s = divmod(r, 60) # convert the remaining seconds into minutes, and the remainder is the amount of seconds return f'{h}:{m}:{s}' # return the time in hh:mm:ss format |
| read\_json | details = path provided  open details in read mode as file:  return json.load(file) |
| write\_json(procedure) | file\_path = path provided  open file\_path in write mode as file:  file.seek(0) # go to the top of the file  json.dump(data, file, indent=4) |
| bubble\_sort2D(Function) | swapped = True  Loop while swapped == True:  swapped = False  for each ELEMENT in Array[:-1]:  IF ELEMENT[1] < Next ELEMENT[1]:  swap ELEMENT and Next Element  swapped = True  return Array |
| delete\_user(procedure) | For each TOPIC in Questions:  invoke delete\_json\_key(...) in TOPIC for username  invoke delete\_json\_key(...) in users.json for username |
| delete\_json\_key(procedure) | data = read\_json for details\_path  for KEY in data:  IF depth == 1:  for KEY2 in data[KEY]:  IF KEY2 == username:  delete KEY2 from data  ELIF depth == 1:  IF KEY == username:  delete KEY from data  invoke write\_json(...) -> data to details\_path |
| Class Display’s methods |  |
| draw\_back\_button (procedure) | screen.blit(self.back\_image, (self.BACK\_X, self.BACK\_Y) |
| refresh (procedure) | Fill self.screen with the background colour  IF show back button:  self.draw\_back\_button(...) |
| check\_return (function) | IF self.back\_button\_rect collides with mouse\_pos:  return 1  return 0 |
| draw\_text (procedure) | center\_x, center\_y = center  text = self.font\_size.render(text)  text\_rect = get text rectangle  x,y = pos  IF center\_x:  x = (screen width – text\_rect width)//2  IF center\_y:  y = (screen height – text\_rect height)//2  self.screen.blit(text, (x,y)) |
| Class Textbox’s methods |  |
| create\_rect(procedure) | padding\_x, padding\_y = self.padding  w, h = self.size  self.rendered\_text = render(self.text) based on self.text\_size  self.rect = self.rendered\_text.get\_rect()  tempRect = a copy of self.rect  IF a limit has been set to the box:  tempRect.width = min(270, tempRect.width + padding\_x)  tempRect.height = tempRect.height + padding\_y  self.main\_rec = tempRect # this is the text rectangle with paddding added  IF w >0 and h>0: #IF a specIFied size was passed in  IF self.main\_rec.width - arbitary number < w:  self.main\_rec.width = w  self.main\_rec.height = h  self.surface create a Pygame.Surface with size of self.main\_rec  self.background = self.background\_colour |
| show(procedure) | Pygame.draw.rect(self.surface, self.background, self.main\_rec)  Pygame.draw.rect(self.surface, self.border\_colour, self.main\_rec, 5)  text\_x = (self.main\_rec.width - self.rect.width) // 2  text\_y = (self.main\_rec.height - self.rect.height) // 2  self.surface.blit(self.rendered\_text, (text\_x, text\_y))  IF center:  self.x = (canvas width – main\_rec width)//2 # center it  canvas.blit(self.surface, (self.x, self.y)) # blitting the box at the initially specified x and y position |
| **set\_properties (procedure)** | **self.background = background**  **self.border\_colour = border**  **self.hover\_colour = hover** |
| **check\_hover (procedure)** | **mouse\_pos = Pygame.mouse.get\_pos()**  **temp\_rec = self.main\_rec.copy()**  **temp\_rec.x = self.x # the x position of the box**  **temp\_rec.y = self.y # the y position of the box**  **if temp\_rec.collidepoint(mouse\_pos):**  **self.background = self.hover\_colour**  **else:**  **self.background = self.background\_colour** |
| check\_click(procedure) | temp\_rec = self.main\_rec.copy()  temp\_rec.x = self.x # the actual x position of the box  temp\_rec.y = self.y # the actual y position of the box  return (mouse\_pos collided with temp\_rec) and (left mouse button was pressed) |
| Class AutoBox’s methods |  |
| show(procedure) | Pygame.draw.rect(self.surface, self.background, (0,0 self.rect.size))  IF self.border\_color:  Pygame.draw.rect(same parameters as self.rect, border\_radius = 2)  IF self.surf: # this is the surface where text is blitted  x,y = self.dx, 0  center\_x, center\_y = self.center\_text  IF center\_x:  x = (width of main rect - width of text rect)//2  y = (height of main rect - height of text rect)//2  self.surface.blit(self.surf, (x,y, self.text\_rect.size))  main\_surface.blit(self.surface, (self.x, self.y)) # blit onto the main screen |
| add\_text(procedure) | text = split text by spaces  height = height of a rendered random letter  self.surf = Pygame.Surface(...) # a surface of size text\_rect.size  IF len(text) == 1:  rendered = self.font.render(text)  self.surf.blit(rendered,(0,0))  return  line\_count = 0  widths = 0  pointer = 0  letter\_count = 0  loop while pointer < length of text array  current\_word = text[pointer]  temp\_surface = copy of self.surf  font\_letters = [rendered letters for letter in current\_word]    # attempt adding letters onto temporary surface  for each index, letter in enumerate(font\_letter):  newline = False  rect = letter.get\_rect()  IF letter\_count == 0: # IF this is the first line  proposed\_x, proposed\_y = 0, line\_count \* height  letter\_count += 1  widths = 0 # initial padding from the side of the rectangle  else:  proposed\_x = widths # sum of all the widths of letters + padding  IF proposed\_x + rect.width > self.text\_rect.width: # IF the word cannot fit  newline = True  IF newline:  line\_count += 1 # increase the line we're on  widths = 1 # resetting the widths  letter\_count = 0 # reset the letter count to 0 to represent we're on a new line  break the for loop to repeat the whole process again on a new line  blit letter at proposed\_x,proposed\_y on temp\_surface  widths += rect.width + (arbitary number to represent padding between letters)  else: # IF all letters were successfully blitted  IF letter\_count > 0:  widths += (render a space).get\_width()# add space between words  increment pointer # repeat the process again for the next word  self.surf = a copy of temp # save changes |
| **check\_hover(procedure)** | **IF the box type is not 'question':**  **mouse\_pos = get mouse pos**  **IF mouse\_pos collides with self.rect:**  **set the background to hover colour**  **set the border colour to hover border colour**  **else:**  **set the background to default background colour**  **set border colour to None** |
| check\_click(function) | IF self.check\_collision == False or the box type == 'question':  return  IF (mouse\_pos collides with self.rect) and (left mouse button was pressed):  return self |
| Class Tile’s methods |  |
| show(procedure) | surface.blit(self.image, self.rect.topleft) |
| Class Projectile’s methods |  |
| draw(procedure) | Surface.blit(self.image, temp) |
| update(procedure) | invoke self.check\_collision against all enemies  invoke self.check\_collision against player  IF not self.remove:  self.rect.x += self.x\_vel  collision = any([invoke self.mask\_cillision(...) for each TILE in world.all\_tiles])  IF collision:  self.remove = True  IF self.remove:  self.kill() # free up memory space. |
| check\_collision(procedure) | IF objs is an array:  for obj in objs:  collision = invoke self.entity\_collisions(...)  IF collision:  self.remove = True  self.kill()  else:  collision = invoke self.entity\_collisions(...)  IF collision:  self.remove = True  self.kill() |
| entity\_collision(function) | collision = invoke self.mask\_collision(...)  IF collision:  obj.health -= self.shooter's projectile damage  return collision |
| mask\_collision(function) | offset\_x = self.rect.x - target.rect.x  offset\_y = self.rect.y - target.rect.y  collision = check self.mask overlaps with obj.mask with offset (offset\_x, offset\_y)  return collision |
| Class Entity’s methods |  |
| move(procedure) | dx = dy = 0  alive = invoke self.check\_alive()  IF not alive or self.bow\_attack:  return  IF moving\_left:  dx -= self.x\_vel  self.flip\_image = True  self.direction = -1  IF moving\_right:  dx += self.x\_vel  self.flip\_image = False # default direction the entity faces == 1/right  self.direction = 1    IF self.jumping and player is not in the air:  self.y\_vel = -13  self.jumping = False  self.in\_air = True  #gravity / downward acceleration  self.y\_vel = minimum value from(self.y\_vel + GRAVITY, 10) # y\_vel should never exceed 10  dy += self.y\_vel  #check collision with floor  for each TILE in world.all\_tiles:  temp\_rect = self.rect.copy()  temp\_rect.x += dx # proposed move in the x axis  IF TILE.rect collides with temp\_rect:  dx = 0 # dont move the entity because a collision occured  IF the type of this entity is an Enemy:  self.wall\_collision = True  temp\_rect = self.rect.copy()  temp\_rect.y += dy # proposed move in the y axis  IF TILE.rect collides with temp\_rect:  dy = 0  # check if jumping and collision occurs below a tile  IF self.y\_vel < 0:  self.y\_vel = 0  self.rect.top = TILE.rect.bottom    # if the entity is falling  ELIF self.y\_vel >= 0:  self.ground = min(5, self.ground+1)  self.y\_vel = 0  self.in\_air = False  self.rect.bottom = tile.rect.top    IF self.rect.y > (world.height - self.rect.height): # if the entity has falling too low:  self.remove = True # remove the entity  # confirming movements:  self.rect.x += dx  self.rect.y += dy |
| animaton\_handling(function) | invoke check\_alive()  self.shoot\_cooldown\_timer = max(0, self.shoot\_cooldown\_timer - 1) # >= 0  ## update animation based on a timer since last recorded  cooldown\_time = 90  shoot\_projectile = False  self.image = get the current animation image  image\_rect = self.image.get\_rect()  ## keep the entity in the same position. This is if animation sizes are different  IF self.direction == 1:  image\_rect.bottomleft = self.rect.bottomleft  else:  image\_rect.bottomright = self.rect.bottomright  self.rect = image\_rect  self.mask = get mask from image  current\_time = get current time  IF (current\_time - self.time1) > cooldown\_time:  self.animation\_pointer += 1  IF the current action is a combat animation:  IF self.animation\_pointer >= length of current animation:  IF the entity was in bow animation:  shoot\_projectile = True  self.shoot\_cooldown\_timer = self.shoot\_cooldown # start cooldown  set sword attack and bow attack to False  reset self.animation\_pointer and self.current\_action to 0 # idle animation  ELIF the current action is death animation:  when self.animation\_pointer > length of death animation - 1:  self.remove = True  self.animation\_pointer = length of death animation - 1  self.remove = True # remove the entity after death animation has finished  else: ## animations where looping occurs  ## set animation\_pointer to 0 every time animation\_pointer reaches the end of the animation  self.animation\_pointer = self.animation\_pointer % len(self.animations[self.current\_action])  self.time1 = get current time  IF shoot\_projectile:  projectile = instantiate Projectile(...)  return projectile |
| update\_action(procedure) | **IF new\_action is not equal to self.current\_action**  self.current\_action = new\_action  ## reset the index at which the animation for the specific action starts at  self.animation\_pointer = 0  self.time1 = get current time |
| update(procedure) | invoke self.move(...)  IF the entity's obj type is player and they are alive:  IF the player is jumping:  invoke self.update\_action(...) to switch to jumping animation  ELIF the player is falling:  invoke self.update\_action(...) to switch to falling animation  ELIF self.sword\_attack:  invoke self.update\_action(...) to switch to attack animation  ELIF self.bow\_attack:  invoke self.update\_action(...) to switch to bow animation  ELIF moving\_right or moving\_left:  invoke self.update\_action(...) to switch to running animation  else:  invoke self.update\_action(...) to switch to Idle animation |
| get\_animations(procedure) | animation\_scale = {entity: {animation: scale} for each entity in the game} ## 2D dictionary  images = [ ]  img\_path = store path of the current entity's specified animation  img\_list = os.listdir(img\_path) ## get a list of images in img\_path  scale\_info = animation\_scale[obj\_type][animation]  for each IMAGE\_FILE in img\_list:  image = load image and scale it using scale\_info  images.append(image)    self.animations.append(temp) |
| draw(procedure) | image = flip the image on the x-axis if self.flip\_image is True or self.direction == -1  surface.blit(image, self.rect.topleft) |
| check\_alive(function) | IF self.health <= 0: # if the entity has died  self.health = 0 # so no issues arise elsewhere since 0 == dead  death\_index = invoke self.get\_index(...) for death animation  invoke self.update\_action(...) for death\_index  return self.health > 0 |
| get\_index(function) | index = None  IF animation is in self.all\_animations:  index = self.all\_animations.index(animation)  return index |
| sword\_collision(functino) | collision = invoke self.check\_collision(...) for obj  IF collision and obj is in sword aniamtion:  reduce self.health based on obj's current weapon damage  return 1  return 0 |
| check\_collision(function) | dead = invoke self.check\_alive(...) for self  IF dead:  return  obj\_mask = get image mask of obj  offset\_x = obj.rect.x - self.rect.x  offset\_y = obj.rect.y - self.rect.y  current\_mask = get mask based on what direction the entity is facing  collision = check overlap between current\_mask and obj\_mask with offset = (offset\_x, offset\_y)  return collision |
| Class Enemy’s methods |  |
| start\_attack(function) | mask\_collision = invoke self.mask\_collision(...) between self and player  IF self.wait == 0 and mask\_collision and invoke self.check\_alive(...):  set self.sword\_attack to True  invoke self.update\_action(...) for Attack animation  set self.wait to (abritary number suitable for enemy attack cooldown)  return True  return False |
| AI(procedure) | IF falling:  invoke self.move(...) don't move in x or y direction  invoke self.set\_idling(...)  return # don't do anything else related to AI if fallling  self.wait = max(0, self.wait - 1) # stop decreasing once it gets to 0  AI\_moving\_right = False  IF self.current\_action == Attack index: # don't do any movement if attacking  return  alive = self.check\_alive()  IF alive:  IF self.idling:  self.idling\_counter -= 1  IF self.idling\_counter <= 0:  self.idling = False  IF self.change\_direction:  self.direction \*= -1  self.move\_counter \*= -1 # moving in opposite direction    IF self.direction == 1:  AI\_moving\_right = True  AI\_moving\_left = not AI\_moving\_right # mutually exclusive events. When one is True the other is always False.  invoke self.move(...) with AI\_moving\_left and AI\_moving\_right  invoke self.update\_action(...) to 'Running' index  self.move\_counter += 1  self.change\_direction = False  IF self.move\_counter > (self.move\_radius) or self.wall\_collision and (enemy is not jumping/in the air):  self.change\_direction = True  invoke self.set\_idling(...)  self.wall\_collision = False # after changing direction, no longer colliding with wall |
| set\_idling(procedure) | self.idling = True  self.idling\_counter = 50  invoke seld.update\_action(...) to 'Idle' index |
| update(procedure) | invoke self.animation\_handling()  IF self.health > 0:  invoke self.start\_attack(...) with player # to constantly check IF self should attack player    player.sword\_collision(self) # check if enemy has dealt damage to the player  self.sword\_collision(self) # check if player has dealt damage to the enemy  invoke self.AI(...) |
| Class World’s methods |  |
| process\_data(function) | death\_blocks = []  enemies = []  player = instantiate a default Player object in case there isn’t one on the map  portals = [] |

## Usability Features

### Explanation and justification of the design and the user interface

log out

|  |
| --- |
| show leader boards  user clicks on leader board box  login  leader boards  main menu  Platform Access  user clicks on username box  username: points  user can go back after signing up to log in  sign up  show points graph  clicks on a level  killed enemy  level completed or  player died or  player decided to quit  game level  Question Display  show summary  level selection |

The above diagram illustrates the various pages/ sections of the software to which the user can visit. The layout is separated and set out in a fashion that makes it efficient and easy to navigate to increase fluidity when navigating around the platform.

When the user first enters the software, they will be presented with the option to login or sign up. If they choose to log in and their credentials are valid, the user is taken to the main menu screen. If they choose to sign up, they must enter a username and password that fulfil the criteria. The user can then choose to go back or sign up another account before proceeding to the main menu.

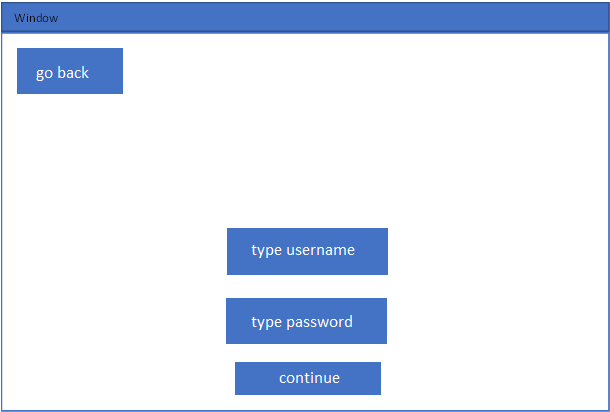
In the game level, every time an enemy is killed, a question is displayed and whenever the game level is terminated the user’s summary statistics are displayed so they are made aware of how they did in the current session.

Window

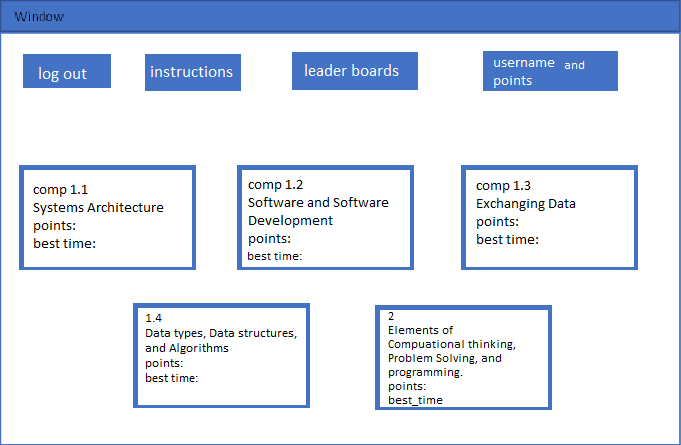
Logo

Login

Sign Up

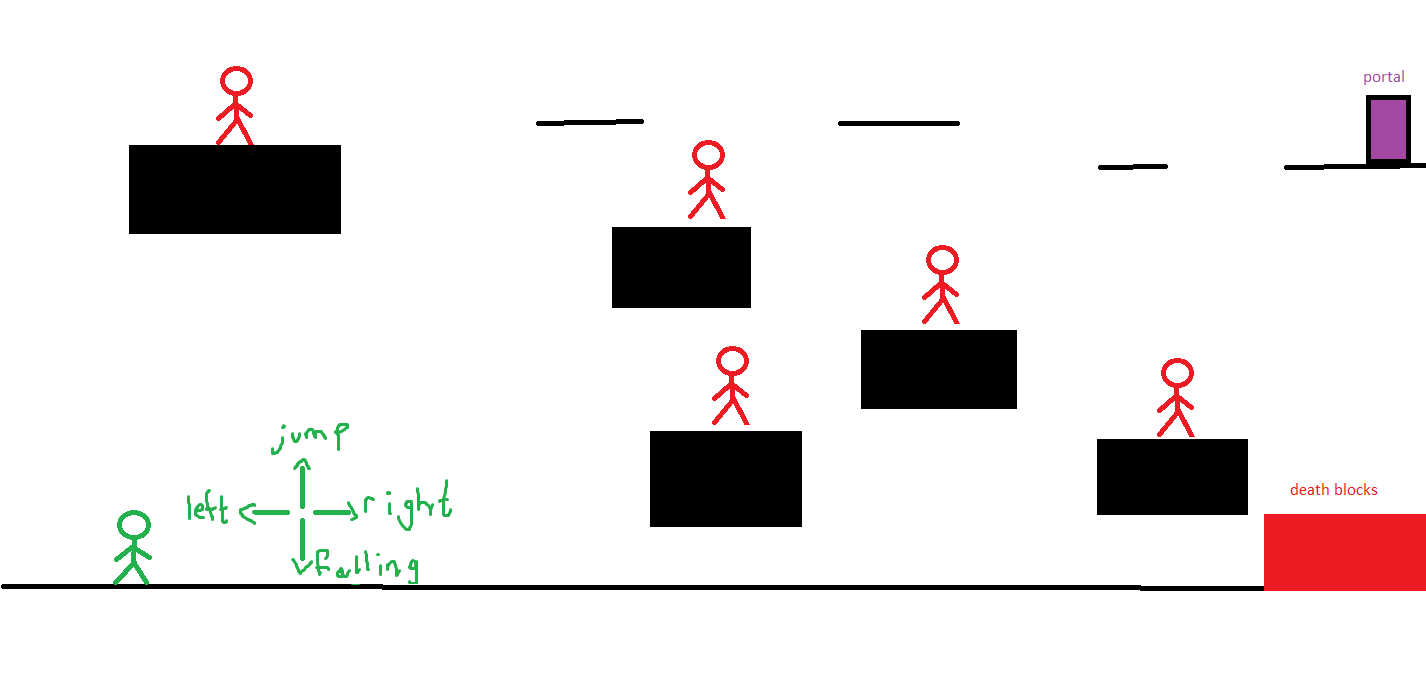
Here is my initial menu design. The menu will consist of a simple design with the game logo/title at the top followed by buttons for login and sign up. The user credentials page will consist of two boxes which include the username and password box which will be re-used in the login and sign-up sections. On the respective page where the user enters credentials, they will also have the option to go back in case they change their mind.

The above is what the credentials page will look like. There are 2 boxes where the user can type in; first is the username box and the second is the password box. The user can then press the continue button to proceed to the next screen. If the user is signing up, don’t proceed, the platform will just tell them whether the sign-up attempt was successful or not. The go back is also placed in a position that is similar in many other platforms so that it is easily identifiable. The background for the platform access and credentials page are subject to change. I will decide whether to use a solid colour or a specific background based on the user input.

****The following is a rough design of what the main menu will look like once the user is logged in.

In the main menu users have the option to view the leader boards, or points progress graph; all the levels will be displayed here in a neat and organised fashion too. The leaderboards will display the 10 highest ranking users based on their total points. The position of the boxes is also indefinite as I will be testing out and implementing auto positioning for boxes based on their widths. I have gone with this design because it clearly displays all the possible options the user can pick and lays them out in an organised manner.

The level gameplay design



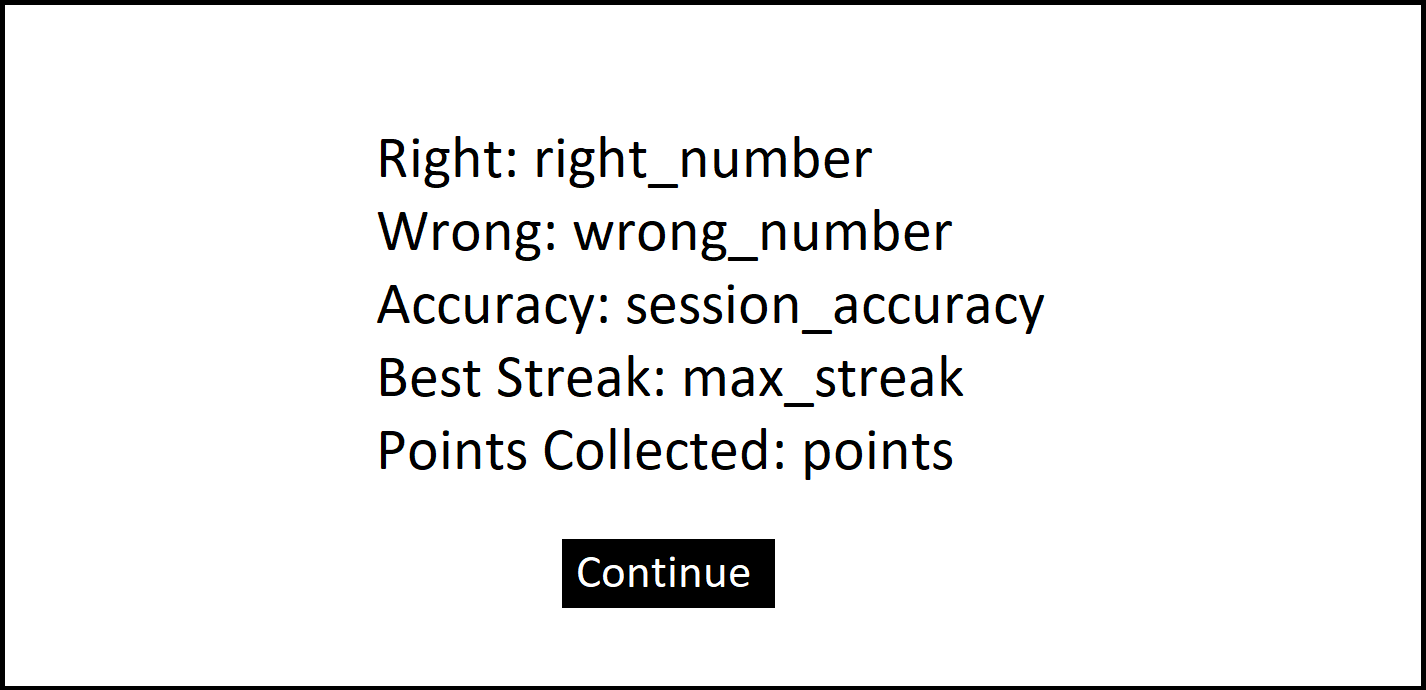
Although not all levels will be the exact same, this is a rough design of what the potential level could look like. In this design enemies are represented by a red outline whilst the player is in green. Death blocks are shown with a solid red fill and to complete the level the player will have to enter the portal as shown on the screen. I have chosen to go with this design because it sets out a clear objective for the player: ‘defeat all enemies and enter the portal to complete the level’. It’s simple yet effective in increasing engagement within the level gameplay as the user faces some challenges to try and get to the portal without dying.

**Graphical user interface, application

Description automatically generated**

Initial question display design

This is the design for what the question display will look like. It will be kept simple and easy to understand such that it is straightforward to navigate. The question display will also contain a feature where the colours of options will change based on what option the user selected. The colours used will be contrasting such as red and green so that this also makes understanding what is right and wrong clear as these are often the standard colours associated with correct/ incorrect results.



The level summary displays relevant information about the quiz such as the total questions correctly answered. The feedback is given in the form of text which is blitted to the screen and is positioned within the feedback rectangle. There will then be a continue button also displayed to allow them to go back to the main menu/ level selection screen.

### Stakeholder Input

Following the drafting of my designs, I sent my stakeholders this email

Good Evening Harshraj,

Apologies for the delayed response, but thank you for your email. I've noticed how you've taken my comments into consideration since the last time I wrote to you, and the development in your ideas are impressive.

The designs are beautiful. I can see that you have yet to add in a colour theme, so I think it might be a good idea to keep a dark colour theme as it might assist in furthering the visual appeal of the platform as a bright coloured theme may be too intensive on the eyes. The ‘go back’ button could also be placed closer to the actual input boxes so that its easier to locate it and click it instead of going all the way to the top left when not needed. I know this is a bit nit-picky, but I would like to see a change in placement of that button. Although I understand that the designs have been abstracted, I’d like to emphasise the use of colours in your platform such as in the main menu or question display. Oh, and on the Student's Quiz page, you have the text and values positioned quite inconsistently which I think is a little displeasing and could be repositioned so that it looks neater. Perhaps something like this:

Right: right\_number

Wrong: wrong\_number

Accuracy: accuracy\_number

etc.

I'm looking forward to seeing these adjustments.

Everything else seems good – feel free to make any other changes that come to your mind

Kind regards,

Mr Waring

Hello both,

Please find the platform's interface designs attached. I also want to point out that these are merely rough drafts that have been greatly abstracted and I'd like to hear both of your thoughts on them before finalising.

Many thanks,

Harshraj

Following up with what Mr Waring said, I decided to go with a dark grey colour theme and resulted in creating this menu screen design.

Hi Harsh,

Im simply amazed by these designs and the features you’ve proposed. I really like your use of points to show a progress graph; one of the features im highly looking forward to is the graph. I also appreciate how you’ve kept the question display simple and lightweight which looks easy to understand, just how I suggseted in our previous meeting. However about the level design it looks like you’ve tried to put all the enemies on the same screen. Although I don’t think this is much of an issue I’d like to see some more depth to the maps maybe something like scrolling so that users can explore around more and just do stuff to perhaps enjoy the game a little bit more.

I’m not too concerned about much else so feel free to keep them as they are or make any changes you see fit. Cheers.

- Dan



I will amend the algorithm for show\_summary such that the formatting is neater, and a consistent whitespace is generated between keys and values. I asked both Dan and Mr Waring for their inputs regarding this option and they approved.

I have also decided to consider what Dan said about the level design so that I can add more tiles by implementing scrolling mechanics, however I will consider adding this feature at a later date.

### Description and justification of the usability features

The usability features I considered improve the platform's usability. Any errors in the credentials provided are clearly highlighted to the user via an appropriate message during the login/registration process. The message displayed when an existing username is entered during signup, for example, must be clearly different from the error message stating that the password entered was rejected. Because error tolerance is such an important component of usability, it is critical that the platform's design addresses such human errors with relevant meaningful messages.

Any text on the design will be designed to be easily read, as well as using a reasonable colour scheme and icons that accurately depict the functionality. The Sans-font family will be used to display most of the text, and the text will be large and clear enough for the user to easily read and understand any text. The menu structure isn't overly complex, and the user interface isn't clumsy. The majority of the pages, such as the menu interface, contain only the most important information and exclude extraneous text information.

I considered a number of factors when designing the buttons. The majority of users should be familiar with the button designs. As a result, I'll use square-cornered filled buttons. The labels on the buttons will clearly describe the action of the buttons. Their size is also an important consideration. All of the implemented buttons will be large enough for users to easily navigate the platform. The location of the most frequently used buttons remains unchanged. The 'Go Back' button, for example, is located to the left of the screen.

Furthermore, by keeping the level design contained to a single size/screen frame it makes it easier for the user to decide on a path to take and locate enemies to defeat to eventually complete the level by entering the portal. I considered many ways in which the user can identify an enemy, so I have decided to outline the enemies in red to make them easily identifiable. However as pointed out by Dan, I am re-considering my decision of keeping the whole level in one section and instead may implement a scrolling mechanism in the future as it would make the usability of the software even better because the player won’t be confined to moving in such a restricted space.

Moreover, by deciding to separate the question display into its own screen instead of prompting a question and options on the game level makes it easier and prevents the game screen from getting too crowded and overloading the screen with too much text and information which could ruin the user’s experience. In addition to that the question display has been kept to only displaying the bare minimum information so it doesn’t distract or overwhelm the user with too much text on the screen. This was done by only blitting (the Pygame vocabulary for display/drawing) the question and its relevant options on screen and displaying feedback on a whole new screen so that its clear when the user receives feedback and gives enough space for descriptive feedback when needed.

### Description and justification of the validation required

The only time keyboard inputs occur are during credential entering and player movement/actions during level gameplay. The text validation when entering credentials include length restraints on the username and password to prevent memory overflow and must have at least more than 4 letters to keep the leader board readable. When signing up, validation also needs to occur to ensure that a unique username is chosen to prevent collision when accessing the data store. **Invalid characters should not be allowed**

During level gameplay there isn’t much validation for keyboard inputs that occur because certain actions will only be carried out when certain keys are pressed. However, there will need to be validation for player movement such that when a player is touching another object such as a tile. This is done to prevent a glitch where the player begins to start phasing through walls and ignore the bounds of the game and lead to unintended outcomes.

**In addition to that, when/if the player falls off the screen, I need to ensure that they don’t keep falling forever as this will never end the game loop and need to add a certain threshold that checks if they have gone below the screen far enough to trigger a removal of the player and recognise them as dead for having fallen off the edges.**

The unintended ability for the player to leave the area/map is a clear issue in any platformer game. To prevent the player from ever leaving the confines of the screen, all maps are designed with invisible blocks on the borders.

Another key case where validation is required is when accessing files. I must ensure that when trying to access a file, it must exist first in order to access it otherwise it can cause unknown and unintended errors throughout the software. Files should also be closed once accessing is finished in order to prevent memory leaks or other unintended errors to arise.

Control validation is also performed when the player touches tiles. This is done to avoid a glitch in which the player starts moving through objects as a result of its hitbox phasing through other hitboxes in the world.

**Furthermore when adding new questions to the data store, I must ensure that all existing users are also added a default [right=0, wrong=0, accuracy=0] for that question to maintain referential integrity when modifying the data store so that when the software tries to access user information for a new question, it doesn’t create any errors that will prevent the question from being displayed or crashing the software. In order to get around this I will develop an algorithm to see if the current user’s data exists for every question in every topic when the user signs in, and if there is a case where the data doesn’t exist, add it in so that I don’t have to manually do this process for time a new question or user is added.**

## Identification and justification of test data to be used during post-development

The objectives table will be used during development. I will review the table below at the post-development stage to make sure all of the requirements have been met. The table has been separated into different features, each with its own set of achievements/actions. These projected outcomes may not reflect the actual implementation as the project proceeds, but they will represent the feature's functionality.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Requirements | Requirement/ module | | Test Objectives | ✔ |
| Platform Access | 1 | Sign Up and Login | Validate credentials and display a suitable error message if an invalid sign up attempt was made |  |
| Buttons should not experience more than a single mouse button click to prevent ‘input\_information’ from being callled multiple times on the stack. |  |
| Depending on the type of button clicked, take the user to the appropriate page. |  |
| Successful sign up attempt should add user data to users.json and all questions for each topic. |  |
| Main Menu | 2 | Level selection | Ensure each level has the correct topic text displayed. |  |
| Each level has a suitable amount of questions to rotate between |  |
| Clicking on a level should take the user to the game level with the correct tile information |  |
| Username and points | Display username and their current points |  |
| Clicking the username and points button should display a graph to show user progress |  |
| 15 | Leaderboards | Display the top 10 users in a suitable format that is easily readable |  |
| User Account and Data store | 3 and 5 | View user statistics | User’s accuracy level should be updated with each attempt at a question/topic, and displayed on the main menu. |  |
| User’s points progress should be trackable by using a graph to plot points against attempts. |  |
| 4 | Store user data | Each user should have their own accuracy levels stored for each question and their best time for each topic |  |
| Use easy-to-remember keys when accessing user and question data. |  |
| Storing questions | Each question should have access to view options, feedback and user information |  |
| Game – Question  Preparation and Display | 6 | Strength based algorithm | Ideally there should be a variety of questions infused in each level such that they match the user’s ability and encourage them to improve by presenting weaker questions, but not overhelming by ONLY providing questions they don’t perform well at. |  |
| The quota for red, amber and green questions should be filled |  |
| Add questions which haven’t been attempted before to green category. |  |
| Loading Questions | Questions should be loaded (using the strength based algorithm) and shuffled in a random order |  |
| The number of enemies and questions selected for a certain level has to be the same. |  |
| 8 | Question Display | Show question and options in a big screen with enough room for a moderate amount of text to be able to fit. |  |
| For certain text boxes, there should be text that is added such that it fits in the box by using an algorithm to add words individually. |  |
| 9 | Question checking | If the user’s answer was incorrect, show a feedback screen as to why they were incorrect. |  |
| Game – Player gameplay and enemy AI | 10 and 13 | User movement  Enemy movement | User should be able to move left and right, with the option to jump. Collisions with certain tiles should restrict these movements |  |
| If the user goes out of bounds, then they die. |  |
| The correct animations should be displayed depending on what the current action is for the user. |  |
| Move enemies within their designated radius |  |
| Update enemy animation based on their current action |  |
| 11 | Damage | Enemy attack deals player designated damage. Player’s attack deals damage according to the weapon selected and that weapon’s damage. |  |
|  | Death | When the health is 0, it should trigger the death animation and remove the enemy/player. Player death should end the level. |  |
| Game – Level generation | 14 | Level creating | An easy to use and effiency software used to create levels and convient export methods to allow reading and loading of levels using pygame. |  |

Text entry boxes will also need to be tested. The test table below outlines a series of valid and invalid inputs that will be entered to determine whether the program has adequate validation on text inputs.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Scene | Test type | | Description | Data | Expected outcome | ✔ |
| Username  Entry | 1 | Out of range | The length is zero | no data entered | Error Message |  |
| 2 | Insufficient length, < 5 | Mike |  |
| 3 | Invalid | Contains only spaces | ‘ ’ |  |
| 4 | Extreme | Borderline length (5) | John5 | Transfers to password page |  |
| 5 | Borderline length (15) | Firestormninja1 |  |
| 6 | Valid | Suitable length | Harsh.310 |  |
| Login | 8 | Invalid | Username not found in file; no account found | johnny.310 | Error Message |  |
| Sign up | 9 | Username found in file; username already taken | John.310 |  |
| Password Entry | 10 | Out of range | The length is zero | no data entered | Error Message |  |
| 11 | Invalid length, < 4 | Ghm |  |
| 12 | Invalid | No lower-case characters | MYPASS310 |  |
| 13 | No upper-case characters | mypass310 |  |
| 14 | No numbers | myPassword |  |
| 15 | Contains only spaces | ‘ ’ |  |
| 16 | Valid | Meets all the criteria | quickPass3 | Accepts the password and transfers to the next page |  |
| 17 | Extreme | Borderline length (4) and meets all criteria | Gib1 |  |
| 18 | Borderline length (15) and meets all criteria | Firestormninja1 |  |
| Login | 19 | Invalid | Username-password combination doesn’t exist | notAmatch3 | Error message |  |

# Development

## Platform Access

### Version 1: Home Page

#### The Development’s Outcome

|  |
| --- |
| This is the platform's home page, from which the user can sign up for a new account or log in to an existing one. The user can select one of the following options by clicking on one of the buttons listed below. The logo and buttons have been positioned such that they are horizontally centred. The initial design is straightforward; a dark gery – blue colour scheme coupled with the logo leading to a perfect blend of casualness and professionalism. |

#### The Development Story

|  |
| --- |
| I first started by creating a class named Display. This is responsible for creating Pygame.Surface objects to draw things on. I decided to start off with this because this is the display that everything will be drawn on and is therefore one of the most important components of the application. I started off by declaring some default class variables and constants that would be used throughout the code: |
|  |
| With the base variables set, the Display also needed adjusting of some class variables and creation of new attributes during instantiation which are defined in the constructor method of the class, this is when an instance of the Display is created. Some of the parameters this takes are background (this can be an image or colour), caption for the application window, size, new\_window and arrow\_pos (this is used to determine of the button to go back to a previous page). |
|  |
| When instantiating an object for this class, the following parameters are taken in by the constructor method. These are all keyword arguments, so in the case some values aren’t passed in they are defaulted to reduce having to re-write the parameters every time a window object is instantiated, this way only changes that need to be made are required to be passed in |
| Whilst coding and attempting to load in images, I came across the issue in my Integrated Development Environment where it required the absolute path to be passed in to load certain images and files. Because my absolute path will not be the same as all the other users who will be using this software, it led me to create the get\_path() function which returns the absolute path of a specified path and raises an error if that path does not exist. |
|  |
| The code above uses the os module in python to return the absolute path of the normal path specified. If the path doesn’t exist, then it raises an error to show that this location doesn’t exist. |
| Once the backbone of the Display class was developed, I moved on to create three very important methods: refresh(), check\_return() and draw\_text(). The draw\_text() method is quite straight-forward, it takes in various arguments (some keyword arguments to reduce what has to be passed in) and draws text of the desired size at the desired location.  The check\_return() method was also developed by using the pseudocode from the Design section making it quick and easy to implement.  However, I decided to add a change to the refresh() method. Initially, this just filled the screen with the background colour, however I added in the ability for the background to be an image which will make using this easier during level generation and thus increase re-usability. |
|  |
| To explain the new refresh method in more detail: This refreshes the screen by drawing over everything with an image or a colour (depending on what the background is) and also draws the back arrow at the coordinates stored in ARROW\_X and ARROW\_Y. These have been stored as constants because they do not change during run time for that specific window instance |
| Following the creation of the display, I created the Button Class using the algorithms in Design to add interactive login and sign-up buttons. This class inherits from the Display class which is stored in a separate file named WINDOW to separate key parts of the code. I did this to inherit some properties such as the fonts and colours to prevent having to re-create the same attributes. In addition to the properties listed in the Design section, I've added a limit attribute to the constructor, which places a certain limit on the width of a textbox if the size of the box is to change in response to the text content inside of it. The logic of the check\_click() method is straightforward. The position of the mouse is retrieved, and the x and y coordinates are checked to see if it falls within the rectangular button.  The rectangular shape of the button is initially drawn in the show() method. For standard buttons, a font object is created, and the text is then rendered into an image. The rectangle of the font image is then added a padding which is specified in the constructor method to create a new rectangle called main\_rec which holds the width and height of the final box. This allows the text to be centred by the padding and blitted to the specified coordinates; the method for creating the outer rectangle is handled by the create\_rect() method. |
|  |
| Next, the show() and check\_click() methods were developed to draw the text and its surrounding box onto the screen and check for collisions with the button. |
|  |
| The show method in the above code handles centring the text within the outer rectangle and then blits it wherever desired. |
| Following the completion of version 1 for Textbox class, I moved on to create a main() procedure which would gather all the separate components developed above and utilise them to create a platform access screen. This will be responsible for showing the options to the screen and depending on what button was clicked, redirect the user to the appropriate credentials page. |
|  |
| The login and sign-up boxes are instantiated outside of the loop, as doing so within the loop wouldn’t cause any errors, but it would lead to inefficient use of computer resources because they will be re-assigned every (frames per second)2 when they don’t need to be. The event capture loop within this procedure is crucial to check when the user wants to exit, which can be in the form of closing the application by pressing the x on the window, pressing return in the platform access or pressing the escape key. This event capture loop also handles the mouse clicks to invoke check\_click() methods for the boxes which prevents multiple button clicks occurring every second. |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Platform access  And Main Menu:  **Requirement 1** | Clicking on login and sign-up boxes | Display “Login/Sign Up button clicked” in the console. In reality this should take the user to a page where they enter their credentials. | Nothing  The clicks aren’t registering. This likely has something to do with the positioning not updating certain values. | ✘ |
| Display login and sign-up boxes positioning. | Both boxes should be positioned on the centre of the screen | Boxes are positioned as intended. | ✔ |
| Text for the boxes is centred according to the padding | The text should be in the middle of textbox relative to the x and y padding | The text is displayed in as intended |  |
| Back button click | Clicking on the back button on this page should exit the program | The program closes | ✔ |
| Credentials Page for users to enter information | Username and password fields for user to enter information | Doesn’t exist in version 1 | ✘ |
| Username should meet the criteria | Only accept usernames with 4-15 characters | Doesn’t exist in version 1 | ✘ |
| Passwords should be moderately strong | Only accept passwords with 4-15 characters with at least 1 number. | Doesn’t exist in version 1 | ✘ |
|  | Store username and password to users.json file | A new record of the username followed by their password, and best times (set to -1) for each topic | Doesn’t exist in version 1 | ✘ |
|  | Store user data in each question | Add a [right, wrong, accuracy] array for the user in each question of each topic. | Doesn’t exist in version 1 | ✘ |

### Version 2: Credentials Page

#### The Development’s Outcome

|  |
| --- |
| The issues from version 1’s failed test cases were addressed immediately. The problem was brough about when the blitting position of the boxes were updated, but their actual x and y coordinates weren’t thus causing an issue when checking mouse collision. As shown below, hovering over a button also works now.  The above is a screenshot of the credentials page where the user enters a username and password. The format of all user credential collection pages is going be the same. The input field is accompanied by a message prompt within the box. The continue button takes the user to the Main Menu, while the previous button takes them to the home page.  The code below shows how the issue was resolved. Instead of using local variables x, and y, the show() method now uses self.x, and self.y which represent its actual position, and self.x is also adjusted according to the centre paramter.    Clicking on the login and sign-up buttons now produces the intended response. |

#### The Development Story

|  |
| --- |
| Upon discussing further with my stakeholders, Dan and Mr Waring both suggested that I add a method of identification to show which box the user is about to select. Therefore, version 2 for Textbox saw the addition of a hover mechanism for the buttons in order to show which ones are currently being selected by changing the background colour |
| The snippet below is the additional method that was added to the Textbox class. This method will be invoked for the boxes outside of the event capture loop, but within the main game loop. This is done to ensure that the hover check occurs constantly instead of just when a mouse button is clicked. |
|  |
| The checking for collision with the mousebutton is the same as check\_click() method so it was quite easy to implement. However, with this whilst there is a collision, there is no return statement; the background colour for the box is changed and when no hovering occurs, the background colour reverts back to the class attribute BACKGROUND. To do this I created one more class variable called hover\_colour. This is used to alternate self.background between the default background colour and the hover colour. |
| The intended functionality of the home page accommodated the transition to the development of the Credentials page. The ability of the Textbox to be used as a button and an input box meant that I could re-use that class in this section of the software by simply altering the text of the box based on the user input and which box they had selected. This allowed me to focus on developing some other key functions and procedures required in the credentials page. |
| The first few sub-routines I coded were ones to do with file handling for the main data store of this project, json files. A read function was created which returns a dictionary of the data of a path, a write procedure which wrote a dictionary to a json file, and a presence checker to see if a certain key existed in a json file. The presence checker function will primarily be used in the credentials page, but the read and write sub-routines are developed in a way such that they can be re-used elsewhere in the project later on. |
| *#------------------------------ get/set details ------------------------------# def* read\_json(path):  details = get\_path(path)  *with open*(details,'r') *as* file:  *return* json.load(file)  *def* write\_json(data,path):  details = get\_path(path)  *with open*(details,'w') *as* file:  file.seek(0)  json.dump(data,file)  *def* presence\_in\_file(username: *str*, data: *dict*):  *return* data.get(username) |
| The ultimate goal of the credentials page is to act as a gateway for the user to access the main menu of the software. Therefore, the first key sub-routine that I coded was the check\_details() function which takes username, password and state as parameters. Depending on the state of the credentials page, (login or sign up) it runs appropriate validation on the username and password before granting access to move on. |
| *def* check\_details(username, password, state):  data = read\_json('user\_info/users.json') *# extract user details* username\_data = presence\_in\_file(username, data) *# .get checks if the key exists, and if it does, it returns the value, else None   if* state=='login':  *if not* username\_data: *return* -1 *# if there is no key that matches to the username, it doesnt exist  return* username\_data.get('password') == password *# check if the username's password is the same as the one entered   elif* state=='sign up':  *if* username\_data: *# if there is a key that matches to the username, it already exists  return* -1  *else*: *# if the username does not exist* add\_info(data, username, password) *# add in user info to users.json and all question files  return* 1  *return* 0 |
| The add\_info() procedure was subsequently created because it follows up on the logic of check\_details() such that when a successful sign-up attempt is made, a new user is added to the users.json. |
| *def* add\_info(data, username, password):  data[username] = {"password": password, "points": [], "1.1": -1, "1.2": -1, "1.3": -1, "1.4": -1, "2": -1} write\_json(data, 'user\_info/users.json') *# if username was added to all question files, officially add them* |
| Once the file handling section of this page was finished, I moved on to develop functions that would be used to validate the username and password entries. The validate\_username() function takes in the username and returns a tuple in the format (length error, unique error).  The validate\_password() function takes in the password as a parameter and uses a regular expression to ensure that the password abides by the criteria. |
| *def* validate\_username(username):  *# length error, not unique error  if not*(4 <= *len*(username) <= 15):  *return* 1, 0  *if* presence\_in\_file(username, WINDOW.read\_json('user\_info/users.json')):  *return* 0, 1  *return* 0, 0 *# there were no issues with the username*  *def* validate\_password(password):  *return* re.match("^(?=.\*[a*-*z])(?=.\*[A*-*Z])(?=.\*\d)[A*-*Za*-*z\d$@$!%\*?&]{4,15}",password) |
| Originally, validate\_password had a similar criterion to the username (which was the length criteria) however, upon further consideration, a more rigid password validation algorithm was enforced, which now uses a more compound criterion. The regex connotes to:  - A minimum of one lowercase letter  - A minimum of one uppercase letter  - A minimum of 1 number  - The option to contain certain symbols |
| The main function for this page is the input\_information() which takes in *state* as a parameter to differentiate between login and sign up. |
| *def* input\_information(state):  username = ''  password = ''  fill\_text = 'Username'  fillpass\_text = 'Password'   username\_click = *False* password\_click = *False* random\_box = Textbox(100,200,text='j',text\_size='medlarge',padding=(0,0),limit=*False*)   username\_box = Textbox(100,460,text=fill\_text.center(15),text\_size='medlarge',size=(300,60))  password\_box = Textbox(100,530,text=fillpass\_text.center(15),text\_size='medlarge',size=(300,60))   continue\_button = Textbox(100,630,text='Continue',size=(200, 50),text\_size='medlarge', padding =(10,0))  continue\_button.create\_rect()  continue\_click = *False* incorrect\_details = *False* delete = *False* delete\_counter = 0 |
| Outside of the loop certain variables are created. These will be used in the main loop to display the boxes, update the text in the boxes and return the outcome of the user’s decision. The two main text boxes for this page (login and sign up which are treated as input boxes, and the continue box which is a button) are created using the Textbox class created in version 1. In order to determine the size of the boxes, I rendered the largest letter (by width) which was ‘W’ used the .get\_width() method on that pygame.Surface object and multiplied it by 15 and added some padding which resulted in the final width of 300 for the username and password boxes.  The fill texts are used so that when the user hasn’t typed anything in a box, the text is changed to the boxes’ default text/label. This makes it easier to identify what each box is for. When the user types something, the fill texts are changed to represent the update and when its empty, it will store the default value. |
| *while True*:  window.refresh(back=*True*, pos=(435,500))  username\_box.text=fill\_text  password\_box.text=fillpass\_text   username\_box.create\_rect()  password\_box.create\_rect()  continue\_button.create\_rect()   continue\_state = username *and* password *# checks if both fields have an input in them*  *# Display username and password boxes*  username\_box.show(window.screen, center=*True*)  password\_box.show(window.screen, center=*True*)  continue\_button.show(window.screen, center=*True*)  username\_box.check\_hover()  password\_box.check\_hover()  *if* continue\_state:  continue\_button.check\_hover(mouse\_pos) |
| At the beginning of each iteration of the main loop, the window is refreshed and the username and password boxes’ text are updated by changing their text attributes and invoking create\_rect() to fit the size of the new text (if no definite size is passed in). The continue state is used to check whether the continue button is clickable or not. This is because it should only be clickable when something has been input in the username AND password boxes. All the boxes are then drawn on the screen and checked for hover. |
| I then moved on to create the event capture loop. The event queue checks multiple things; checks are carried out to see if the x button has been clicked on the window to close it, or the go back arrow to return to the home page. It then checks for key presses where a backspace is used to remove the last character from the username or password box depending on which one is clicked. After that, mouse clicks are checked to update the state of username, password and continue boxes by invoking the check\_click() method. However, to register a valid continue click, there is an extra condition added in for the continue\_click which ensures that the continue button is clickable. Upon clicking on a box again to type, the error message is removed to make it look more professional.  There is also a check to see if the user clicked the back button to take them back to the platform access. |
| *for* event *in* pygame.event.get():  *if* event.type == pygame.QUIT:  pygame.quit()  sys.exit() *if* event.type == pygame.KEYDOWN: *# handle key presses  if* username\_click *or* password\_click: *# if the user is typing in username or password fields* incorrect\_details = *False # removes the error message if user has clicked/ is typing in an inputbox  if* event.key *in* [pygame.K\_RETURN, pygame.K\_SPACE]: *# if they press enter/return key, don't register as valid  continue* character = event.unicode *# collect the string format of the key pressed    if* event.key==pygame.K\_BACKSPACE:  incorrect\_details = *False # remove the error message if it is present* delete = *True  continue # a backspace isn't a valid 'character' that can be added to username and password, therefore   # only add to username and/or password if the character is valid  if* username\_click: username += character  *if* password\_click: password += character  *if* event.type == pygame.KEYUP:  *if* event.key==pygame.K\_BACKSPACE:  delete=*False* delete\_counter = 0  *if* username\_click:username=username[:-1]  *if* password\_click:password=password[:-1]   *if* event.type == pygame.MOUSEBUTTONDOWN: *# check which button has been clicked* mouse\_pos = pygame.mouse.get\_pos()   go\_back = window.check\_return(mouse\_pos)  *if* go\_back:  *return* username\_click = username\_box.check\_click(mouse\_pos)  password\_click = password\_box.check\_click(mouse\_pos)  continue\_click = continue\_button.check\_click(mouse\_pos) *and* continue\_state *# checking if continue button is available and also clicked   if* username\_click *or* password\_click: *# don't show any error messages if they've clicked on a box again to type* incorrect\_details = *False* continue\_button.text = 'Continue' |
| Thereafter, I created a set of conditions that deal with all actions related to changing the contents of the username password variables, along with deleting characters based on how long the delete key is held down for. This is also responsible for changing the border colour of the boxes once they are clicked to show the user which box, they are currently typing in. In addition to the deletion method mentioned in the design section, I decided to add in the option to delete multiple characters by holding down the key button too. The password text is also converted to \* characters to maintain a sense of privacy/security when entering details. |
| *if* delete:  delete\_counter += 1 *else*:  delete\_counter = 0  *if* delete\_counter > 8 *and* (delete\_counter % 2)==0: *# allows for singular + held down deletion  if* username\_click: username=username[:-1]  *if* password\_click: password=password[:-1]  *if not* username: *# if nothing has been typed in the username box, it should display 'Username'* fill\_text='Username' *if not* password:*# if nothing has been typed in the username box, it should display 'Password'* fillpass\_text = 'Password'  *if* username\_click: *# if the user has chosen to type in the username field highlight the border* username\_box.border\_colour = username\_box.GREEN  fill\_text = username *else*:  username\_box.border\_colour = username\_box.default\_border  *if* password\_click: *# if the user has chosen to type in the password field highlight the border* password\_box.border\_colour = password\_box.GREEN  fillpass\_text = '\*'\**len*(password) *# cover the characters in the password else*:  password\_box.border\_colour = password\_box.default\_border |
| Once most of the credential page was functioning, I moved on to developing checks that occur if the user clicks on an active continue button. Firstly, the username and password are validated. A trigger for the error message is set depending on what state they are in and what conditions need to be fulfilled for the error message to trigger. In the case of no errors, this section returns the username. |
| *if* continue\_click: *# if the continue button has been pressed* valid\_username = *sum*(validate\_username(username))==0 *# there were no errors with the username* valid\_password = validate\_password(password)   *if* state=='sign up' *and* (*not* valid\_username *or not* valid\_password):  incorrect\_details = *True   else*: *# if it was sign up or login state and valid info was entered* correct = check\_details(username, password, state)  incorrect\_details = correct < 1   *if* incorrect\_details:  *return* username  pygame.display.update() clock.tick(30) |

#### Testing to inform development

##### Module Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Platform access  And Main Menu:  **Requirement 1** | Clicking on login and sign-up boxes | Display “Login/Sign Up button clicked” in the console. In reality this should take the user to a page where they enter their credentials. | Clicking on the login and sign boxes invokes the input\_information() function as intended. | ✔ |
| Display login and sign-up boxes positioning. | Both boxes should be positioned on the centre of the screen | Boxes are positioned as intended. | ✔ |
| Back button clicks | Clicking on the back button on this page should exit the program | The program closes | ✔ |
| Page for users to enter information | Username and password fields for user to enter information | Interactive username and password boxes which show user input | ✔ |
| Username should meet the criteria | Only accept usernames with 4-15 characters | Shown in text input validation table | ✔ |
| Passwords should be moderately strong | Only accept passwords with 4-15 characters with at least 1 number. | Shown in text input validation table | ✔ |
|  | Store username and password to users.json file | A new record of the username followed by their password, and best times (set to -1) for each topic | Adds the username and other information as intended in a 2D dictionary format. | ✔ |
|  | Store user data in each question | Add a [right, wrong, accuracy] array for the user in each question of each topic. | Doesn’t exist in version 2 | ✘ |
|  | Fill in missing user information for newly added questions | Doesn’t exist in version 2 | ✘ |

##### Username Testing

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test type | | Description | Data | Expected outcome | Actual Outcome | ✔ |
| 1 | Out of range | The length is zero | no data entered | Continue button should not be clickable | Continue button is inactive | ✔ |
| 2 | Insufficient length, < 5 | Mike | Error Message | Displays error message | ✔ |
| 3 | Invalid | Contains only spaces | ‘ ’ | Don’t accept input | Adds in square empty character | ✘ |
|  |  | Inputting tab key | Unicode of tab | ✘ |
| 4 | Extreme | Borderline length (5) | John5 | Accept username | Accepted | ✔ |
| 5 | Borderline length (15) | Firestormninja1 | Accepted | ✔ |
| 6 | Valid | Suitable length | Harsh.310 | Accepted | ✔ |
| 8 | Invalid | Username not found in file; no account found | johnny.310 | Error Message | Displays ‘invalid credentials message | ✔ |
| 9 | Username found in file; username already taken | John.310 | Displays  ‘Credentials do not meet the criteria’ message | ✔ |

##### Password Testing

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test type | | Description | Data | Expected outcome | Actual Outcome | ✔ |
| 1 | Out of range | The length is zero | no data entered | Continue button should not be clickable | Continue button is inactive | ✔ |
| 2 | Invalid length, < 4 | Ghm | Error Message | Displays  ‘Credentials do not meet the criteria’ message | ✔ |
| 3 | Invalid | No lower-case characters | MYPASS310 | ✔ |
| 4 | No upper-case characters | mypass310 | ✔ |
| 5 | No numbers | myPassword | ✔ |
| 7 | Valid | Meets all the criteria | quickPass3 | Accepts the password and transfers to the next page | Accepts all of these | ✔ |
| 8 | Extreme | Borderline length (4) and meets all criteria | Gib1 | ✔ |
| 9 | Borderline length (15) and meets all criteria | Firestormninja1 | ✔ |
| 10 | Invalid | Username-password combination doesn’t exist | notAmatch3 | Error message | Displays  ‘Invalid credentials message | ✔ |
| 11 | Contains only spaces | ‘ ’ | Don’t accept input | Adds in square empty character | ✘ |
| 13 | Inputting tab key | Unicode of tab | ✘ |

I asked Dan and Mr Waring for feedback because most of the objectives for this strand had been met. If more changes are needed, they will be made in the next iteration, along with the correction of any failed input tests. Following their review of this section of the platform, both stakeholders suggested that an identifier be added to the credentials page to indicate which page the user is on, and Dan specifically requested that the error messages be more specific.

### Version 3: Rebranding

#### The Development’s Outcome

|  |
| --- |
|  |
| This version featured a rebranding of the design for the platform access once the core of this section was functional. This colour combination of white, blue, and dark grey is a really fluid, clean, and intuitive colour combination which I believe will increase the end user satisfaction. Improvements have been made to the logo which adds a sense of professionalism but also that casual gaming feel that it’s intended to have, making it a much more welcoming Platform Access page. |
| Screenshots of the login and sign-up pages which now show a line of text stating what page the user is in. I made this changed as requested by both Dan and Mr Waring as it seems there was confusion regarding what page they were in at times. This now makes it easier to remember/identify what part of the platform access the user is in. Furthermore, the continue button’s opacity is also lowered when it is inactive and goes oblique once something is entered in the username and password boxes to signify that it is now interactive. The arrow that represented ‘going back’ is now also replaced with an image that fits with the theme and also contains an icon and text explicitly stating ‘go back’ as that is what its intended purpose is.  Furthermore, in order to increase convenience of entering details the functionality to press the tab key to switch between the input boxes and pressing the enter/return key as an alternative for the continue button. |
|  |
| The credentials pages also show a more descriptive error message as opposed to earlier which makes it clear which criteria are missing from the username or password.  In addition to the error messages, I also changed the border of the username and password box initially to a red colour whilst typing and once criteria for the box they are typing in has been fulfilled, the border is changed to green to signify a valid input. This would take several screenshots to show, hence I will explain this feature in the video proof of the platform’s functionality. |

#### The Development Story

|  |  |
| --- | --- |
| In the previous section the test cases in the input boxes for username and password failed the test cases for instances such as tabs and spaces. To fix this I coded a validate\_character() function that checks if the character is within the valid character set and also an additional check to ensure the addition of the character keeps length of the password or username under 15 characters. | |
| *def* validate\_character(string, click, character):  *# Only allow characters, numbers and certain symbols* valid\_chars = re.match('''[A*-*Za*-*z\d$@$!%\*?&"£%^\*(){}~#:;]+''', character)  *# return True if the username as long as the length is   # under 16 and typing in the specified box and the character is valid  return len*(string+character) < 16 *and* click *and bool*(valid\_chars) | |
| Drawing what page the user is in was simple because the input\_information() function already has a parameter *state* which can be used to identify what page they are accessing. Therefore, all I had to do was pass in the state and pass in a new argument *underline* into the draw\_text() method. | |
| window.draw\_text(f'{state.title()} Page',(0,370),center=(*True*,*False*),size='MEDLARGE',underline=*True*) | |
| By default, the underline parameter is set to False, but if it is true, the method now draws a line representing the width of the text and underneath the text. This is done by passing in starting (x, y) coordinates and the ending (x, y) coordinates | |
| *if* underline:  pygame.draw.line(*self*.screen, (colour), (x, y + rec.h,), (x + rec.w, y + rec.h),2) | |
| In order to elaborate on the error messages, I created 3 new variables which are used to identify the type of error message to be displayed, and two more to represent the text and colour of the message. The alteration of the variables that correspond to an error message occurs when the user clicks on the continue button.  For the username there can be two types of messages. In the previous version I ended up returning a tuple (length error, unique error) which was initially for debugging, but turned out to be useful in this version where I could use the indexes to determine which error message to display in regard to invalid usernames on the sign-up page. There is only one error message for an invalid password, and this states the criteria that the password must fulfil. | |
| *if* continue\_click: *# if the continue button has been pressed* valid\_username = validate\_username(username)  valid\_password = validate\_password(password)   *if* state=='sign up' *and* (*sum*(valid\_username)==1 *or not* valid\_password):  *if* valid\_username[0]: *# if there was an issue with string length* display\_string\_length = *True* display\_password\_text = *False* incorrect\_details = *False   elif* valid\_username[1]: *# if there is already an existing user* display\_string\_length = *False* display\_password\_text = *False* incorrect\_details = *True   elif not* valid\_password:  display\_password\_text = *True* display\_string\_length = *False* incorrect\_details = *False* | |
| Depending on which error message to display, the *message\_text* variable is changed along with the *message\_colour* and is then drawn onto the screen. | |
| *# login page error messages if* incorrect\_details *and* state=='login':  message\_text = 'Incorrect username and password combination'  message\_colour = window.RED  *# sign up page error messages elif* incorrect\_details:  message\_text = 'Username is already taken'  message\_colour = window.RED *elif* display\_string\_length:  message\_text = 'Username must be between 4 and 15 characters long'  message\_colour = window.RED *elif* display\_password\_text:  message\_text='Password must contain 4-15 characters, minimum [one uppercase and lowercase letters, one number] and optional special character'  message\_colour = window.RED  *if* message\_text:  window.draw\_text(text=message\_text, pos=message\_pos,colour=message\_colour,size='MEDIUM',center=(*True*,*False*)) | |
| Adding in the switching using the tab key and function of the return key was quite simple. When the user presses the tab key, it switches the states of the username and password clicks, but the password click is the inverse of the username click because these two events are mutually exclusive. The user is not allowed to type in the username AND password boxes at the same time.  Pressing the return key activates the continue click if the continue button is active. This process was fairly simple it only required me to modify already existing variables using additional event conditions. | |
| *if* event.key == pygame.K\_TAB:  username\_click = *not* username\_click  password\_click = *not* username\_click  *continue  if* event.key == pygame.K\_RETURN *and* continue\_state:  continue\_click = *True* | |
| I then moved on to create all the questions files and named them: ‘1.1’, ‘1.2’, ‘1.3’,’1.4’, and ’2’ which represent the components of the AS Computer Science Specification as stated in the Design section. The creation of these files then allowed me to address a missing, yet crucial, feature, adding user information to questions upon signing up. |  |
| Upon creation of the files, I was able to add in another section in the add\_info() procedure to write new user data to all the questions. With this, the add\_info() procedure was now fully complete | |
| *for* file *in* questions:  file\_info = read\_json(f'Questions/{file}')  *for* question *in* file\_info:  file\_info[question][username] = [0, 0, 0]  write\_json(file\_info, f'Questions/{file}') *# write to all question files* | |
| After registering a new user, it produces the following output within the question file  To elaborate on the formatting of the question file, each question will have:  - A key *options* which will have an array of possible answers stored as strings. The first element will always point to the correct answer.  - Another key, *feedback,* to explain to the user why the correct answer is what it is.  - n number of keys corresponding to each user with values [right, wrong, accuracy] | |
| After I had one question completed, I started adding in more questions and their respective information. But it was during that process that I realised I would also have to add in the default [0, 0, 0] value for each username key otherwise several errors would arise later in the development process. In order to address this issue, I developed a procedure: update\_details() to increase the maintainability and consistency of the data store. | |
| This procedure is invoked each time a user logs in and iterates through each question of every topic. If the user does not have their statistics stored for a question, then it creates a fresh array [0, 0, 0] for that question; setting default values whilst reducing code size is achieved by using the built-in setdefault() method for python dictionaries. Doing this ensures that the data store maintains referential integrity when adding in new questions, as the users.json and {topic number}.json can be considered relational databases. This is because the usernames can be considered as foreign keys. They are an attribute for each question in a topic file but are a primary key in the users.json file. | |
| *def* update\_details():  *# fill in missing user data in question files. ie when a new question is added, don't have to do it manually  for* user *in* read\_json('user\_info/users.json'):  *for* topic\_name *in* os.listdir('Questions'):  question\_path = f'Questions/{topic\_name}'  topic\_data = read\_json(question\_path)  *for* question *in* topic\_data:  topic\_data[question].setdefault(user, [0, 0, 0])  write\_json(topic\_data, question\_path) | |

#### Testing to inform development

There was very little additional testing required for this version. The main one was that user information was correctly stored and automatically filled in where missing.

##### Module Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Platform Access and Main Menu | Store user data in each question | Add a [right, wrong, accuracy] array for the user in each question of each topic. | Stores user statistics as intended. | ✔ |
|  |  | Fill in missing user information for newly added questions | Works as intended | ✔ |

##### Username and Password Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test type | Description | Data | Expected outcome | Actual Outcome | ✔ |
| Invalid | Contains only spaces | ‘ ’ | Don’t accept input | Doesn’t add in character to input boxes | ✔ |
| Inputting tab key | Unicode of tab | ✔ |

## Level Generation and Rendering:

### Version 1: Creating Levels

#### The Development’s Outcome

|  |
| --- |
| The following image shows how the levels will be designed. The software that was used is called ‘Tiled’. This allows for easy creation and modification of levels through its useful features such as layering, tilesets and other key tools that make the development process of levels much easier. |
| Chart  Description automatically generated |

#### The Development Story

|  |  |
| --- | --- |
| Although my initial design was to keep the level restricted to the size of the screen, I thought about what Dan suggest regarding level design and I concluded that having a map which the player can explore in both the x and y direction would be most beneficial for user engagement and satisfaction. Although this would increase the complexity of the next main level gameplay, it was something on of my stakeholders, Dan, suggested and I believe other users of this platform will also appreciate being added. | |
| I started out by downloading a collection of free to use images from itch.io’s assets store where I was able to obtain key images such as player and enemy sprites, along with tile sets for levels. In order to splice the tilesheets for the levels, I resulted to using Microsoft paint, coupled with Piskel which was mainly used to remove the white background from tiles and export all the images.  Using the select tool, I was able to copy and paste these individual tile icons into Piskel where I could then adjust colours and sizing, and finally export all the tiles into a folder  Piskel’s ability to import spritesheets and split them based on frame size made it even easier to splice entity spritesheets as I didn’t have to individually split frames, unlike the tilesets where it was less organised. | |
|  | Exporting the final frames was very straightforward as this could be applied to the tiles and entity images. |
| After having all the images spliced, a new Tiled map was created by adding the attributes from the Display class: MAX\_BLOCKS\_X attribute, MAX\_BLOCKS\_Y, TILE\_DIMENSION\_X, and TILE\_DIMENSION\_Y. This was done to ensure further consistency when I go on to load the game map. | |
| Upon loading in all the images, the tileset stores the ID as indexes which is how they are exported. | |
| After the tile set was created, I was able to create levels and export them in csv format which stored the ID number, not the actual image name. | |
| Exporting the level in csv format produces this result, where each layer is its own file and contains the ID values of the tiles. | |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Game – Level Generation | Create maps | Each tile should be able to be drawn and at different layers | Works as intended | ✔ |
|  |  | Enemy and Player images should be used to point where the players and enemies spawn. | Works as intended | ✔ |
|  | Loading maps | Each tile should be created with its own x and y value. | Does not exist in version 1 | ✘ |
|  |  | Certain tiles should not be checked for collision such as decoration | Does not exist in version 1 | ✘ |
|  |  | The 1st layer tiles should not be checked for collision as that layer is specifically for decoration/aesthetics. | Does not exist in version 1 | ✘ |
|  |  | Draw tiles based on the order layer. Draw tiles on a higher layer above those on a lower layer | Does not exist in version 1 | ✘ |

### Version 2: Loading Levels

#### The Development’s Outcome

|  |
| --- |
| This version featured the loading of the game map. All the main types of tiles are displayed here. Normal tiles, the portal, decoration, and the death block(green water). The other Forest theme will also feature the same type of tiles, only the images will be different |

#### The Development Story

|  |  |
| --- | --- |
| I first started out by changing all the image names to numbers. This way the tile IDs in Tiled and the images were matching making it easier to reference which images were to be loaded.  I started out by creating a *Tile* class which will be used to store all the essential information about tiles. I followed the pseudocode generated in the Design section which made this process much more efficient. | |
| The Tile class inherits from the pygame Sprite class in case I want to use in built methods specifically designed for sprites later on during development. The basic template is the same as the one made in the Design section. Within the constructor method, the position of the tile is centred so that tiles such as decorations or portals have a more natural placement because not all tiles will be (46x46) pixels.  There is a draw method which draws the tile according to its x and y position. | |
| *class* Tile(pygame.sprite.Sprite):  *def \_\_init\_\_*(*self*, img, x, y, item\_type):  pygame.sprite.Sprite.*\_\_init\_\_*(*self*)  *self*.x, *self*.y = x, y  *self*.image = img  *self*.mask = pygame.mask.from\_surface(*self*.image)  *self*.rect = *self*.image.get\_rect()  *self*.rect.midtop = (x + Display.TILE\_DIMENSION\_X // 2, y) *# make it so that its at the center of a tile, even if the size isn't the same  self*.initial\_time = pygame.time.get\_ticks()  *self*.obj\_type = item\_type   *def* draw(*self*, surface, target):  surface.blit(*self*.image, *self*.rect.topleft) | |
| In addition to a normal Tile class, I also created an AnimatedTile class which is used for animated tiles such as portals or other tiles I may decide to add later on.  This class is similar to the Tile class, so it inherits from it. I decided to do this, because tile collision will need to be implemented later on and they will both have similar collision detection, hence the use of inheritance. The constructor method for this class is slightly different because, I used the logic of entity animation form the Design section for the animated tiles. There will be multiple images for this animated tile, therefore all of the images will be stored in an array, using the get\_animations method(), and upon a certain cooldown time, they will be updated using the animation\_handling() method. | |
| *class* AnimatedTile(Tile):  *def \_\_init\_\_*(*self*, img, x, y, item\_type):  pygame.sprite.Sprite.\_\_init\_\_(*self*)  Tile.\_\_init\_\_(*self*, img, x, y, item\_type)  *self*.animations = []  *self*.get\_animations()  *self*.animation\_pointer = 0  *self*.image = *self*.animations[*self*.animation\_pointer]  *self*.mask = pygame.mask.from\_surface(*self*.image)  *self*.rect = *self*.image.get\_rect()  *self*.rect.midtop = (x + Display.TILE\_DIMENSION\_X // 2, y) *# make it so that its at the center of a tile, even if the size isn't the same   def* get\_animations(*self*):  item\_path = os.path.join(image\_path, *self*.obj\_type) *# the image path of the item* images = os.listdir(item\_path) *# get a list of the image names for the animation  for* image *in* images:  image = pygame.image.load(os.path.join(item\_path, image))  *self*.animations += [image]   *def* animation\_handling(*self*):  cooldown\_time = 90 *# cooldown time  if self*.obj\_type == 'Portal':  cooldown\_time = 100 *# increase rate of change for frames if its a portal* current\_time = pygame.time.get\_ticks()   *if* current\_time - *self*.initial\_time >= cooldown\_time: *# if the cooldown time has finished  self*.animation\_pointer = (*self*.animation\_pointer + 1) % *len*(*self*.animations) *# increase the animation pointer  self*.initial\_time = current\_time *# change the time   self*.image = *self*.animations[*self*.animation\_pointer] *# update the image   self*.mask = pygame.mask.from\_surface(*self*.image) *# update the mask  self*.rect = *self*.image.get\_rect()  *self*.rect.midtop = (*self*.x + Display.TILE\_DIMENSION\_X // 2, *self*.y) *# make it so that its at the center of a tile, even if the size isn't the same* | |
| Before loading in the levels, I created a new file for each level named tile\_info.json which stores information about tiles for example, here is the tile\_info.json for a level I designed initially: | |
| {  "obstacle": "0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 22 23",  "decoration": "21 28",  "kill\_block": "26 27",  "coin": "",  "player": "24",  "portal": "30",  "enemy": "25",  "enemy\_scale": [168, 92],  "player\_scale": [50, 83],  "move\_radii": [3, 1, 2, 2, 0, 2, 0, 0, 1, 0, 1, 1] } | |
| Once the important information and templates for the Tiles were set up, I moved on to create the World class which will be used to store all the important information about the current map. The constructor method is used to split the tiles. The tiles that will be checked for collision are obstacle\_list, ones where no collision should occur are no\_collide and a final array, all\_tiles, to store all the tiles. The | |
| *class* World:  *def \_\_init\_\_*(*self*):  *self*.obstacle\_list = []  *self*.no\_collide = [] *# blocks that shouldn't be checked for collision  self*.height = WINDOW.Display.MAX\_BLOCKS\_Y  *self*.all\_tiles = []  *self*.game\_level = *self*.load\_level(level) | |
| The first function for the class World is load\_level(). This has been assigned a staticmethod decorator because although it belongs to the World class, it has no specific use of any class attributes or methods; it’s a stand-alone function that just returns map data. It reads all the files in the level path and orders from lowest to highest based on the layer number, returning a 3D array of map data to represent layers and their respective tiles on the layer. | |
| @staticmethod *def* load\_level(level):  layers = []  path = f'levels/{level}'  files = os.listdir(path)   ordered = *sorted*(files,key=*lambda* i: *int*(i.split('\_')[1][:i.split('\_')[1].index('.')])) *# sort layers based on numbers  # highest number == highest layer -> prioritised/placed above everything else  # so by sorting from lowest -> highest, lowest layer is blitted first, and highest layer is blitted last/ on top of everything else* files = ordered *# sort the tiles such that highest layer is prioritised/ blitted over the other layers  for* index, file *in enumerate*(files):  *with open*(os.path.join(path, file)) *as* file:  level = csv.reader(file, delimiter=',')  layers.append([\*level])  *# sys.exit()  return* layers | |
| After having a method for loading level data, I proceeded to creating a process\_data() method which is used to translate the information from self.game\_level into actual tile images and assign the tiles to their respective categories. This is done by reading data from tile\_info.json for that level. This method takes in an img\_dict parameter which used to obtain the image that points the filename of a tile to its image.  The x coordinate is calculated by multiplying the index of the tile in the array and the y coordinate is calculated by multiplying the index of current array: ie: | |
| [  [1,2,3],  [4,5,6]  ] | Each sub-array represents a row of the map, and each element is the column. Therefore, the index of the tile in that row \* the tile width == x coordinate. The index of the sub-array \* the tile height == y coordinate. The outer for loop oversees the order of drawing tiles over other tiles to allow a layered effect. |
| Because there are no entity classes created yet the section under ‘player’ and ‘enemy’ is left blank and will be added to later.  The method then returns all the arrays of tiles which store the necessary information. | |
| *def* process\_data(*self*, tile\_info, img\_dict, player\_img, enemy\_img):  data = *self*.game\_level  enemy\_counter = 0  *# iterate through each value in level data file* decorations = [] *# add in all the tiles that don't need to be checked for collision* death\_blocks = []  enemies = []  portals = []   *for* ind, layer *in enumerate*(data):  *for* y, row *in enumerate*(layer):  *self*.height = *len*(layer)  *for* x, tile *in enumerate*(row):  *if* tile == '-1':  *continue* img = img\_dict[tile] *# get the image from the list of images* img\_rect = img.get\_rect()img\_rect.bottomleft = (x \* window.TILE\_DIMENSION\_X, y \* window.TILE\_DIMENSION\_Y + 46)   obj = *None  if* tile *in* tile\_info['obstacle'].split():  obj = Tile(img, \*img\_rect.topleft, 'Obstacle')  *if* ind != 0: *self*.obstacle\_list.append(obj) *# if it isn't the 1st layer (for no collisions)  elif* tile *in* tile\_info['decoration'].split(): *# grass / no collision decoration* obj = Tile(img, \*img\_rect.topleft, 'Decoration')  *elif* tile *in* tile\_info['kill\_block'].split(): *# water* obj = Tile(img, \*img\_rect.topleft,'Death Block')  death\_blocks.append(obj)  *elif* tile *in* tile\_info['player'].split(): *# create player if there's one on the map  pass  elif* tile *in* tile\_info['enemy'].split():  *pass* enemy\_counter += 1  *elif* tile == tile\_info['portal']:  portals += [AnimatedTile(img, img\_rect.x, img\_rect.y, 'Portal')]  *if* obj: *self*.all\_tiles.append(obj)  *return* player, decorations, death\_blocks, enemies, portals | |
| With the core of the loading levels completed, I moved on to actually displaying this information onto the screen. This was accomplished by creating the play\_level() function. This function creates variables that point to player and enemy image images along with getting a dictionary of all the image file names and loading in their images using pygame.image.load(). After this is done, a World object is instantiated into the world variable and calls the process\_data() method with the tile information, image dictionary and the player and enemy images. | |
| *def* play\_level(username, user\_id, level):  ENEMY = 'samurai' *# pick a random enemy* PLAYER = 'player'   LEVEL = 2 *# random.randint(1,4) # choose a random map layout* world = World(LEVEL)  TILE\_TYPES = os.listdir(f'level\_config/{LEVEL}/Tiles') *# get a list of all the tiles* background = pygame.transform.scale(pygame.image.load(WINDOW.get\_path(f'level\_config/{LEVEL}/background.png')),window.SIZE).convert\_alpha()   *# load in game data* img\_dict = {}  *for* i *in* TILE\_TYPES:  img = pygame.image.load(WINDOW.get\_path(f'level\_config/{LEVEL}/Tiles/{i}')).convert\_alpha()  name = i[:i.index('.')]  img\_dict[name] = img tile\_info = WINDOW.read\_json(f'level\_config/{LEVEL}/tile\_info.json')  *# sprite groups* player, decorations, death\_blocks, enemies, portals = world.process\_data(tile\_info, img\_dict, PLAYER, ENEMY)  run = *True* | |
| The main loop invokes the world.draw() method with the background image passed in. This draws the tiles stored in world.all\_tiles() onto window.screen. window which is an instance of the Display class is instantiated outside of the function to allow the World to draw the tiles where required without having it being passed in as a parameter. | |
| *while* run:window.refresh()  world.draw(background)   *for* event *in* pygame.event.get():  *if* event.type == pygame.QUIT:  run = *False*   pygame.display.update() *# make all the changes* clock.tick(FPS) *return* | |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
|  | Loading maps | Each tile should be created with its own x and y value. | Tiles’ coordinates are correctly calculated and drawn. | ✔ |
|  |  | Certain tiles should not be checked for collision such as decoration | Prevented decoration tiles from being added to obstacles list. | ✔ |
|  |  | The 1st layer tiles should not be checked for collision as that layer is specifically for decoration/aesthetics. | Skip adding any tiles on the first layer to obstacles list. | ✔ |
|  |  | Draw tiles based on the order layer. Draw tiles on a higher layer above those on a lower layer | Tiles which do not fill up a full tile and are drawn above a tile that does fill up a fill tile are shown above. | ✔ |

## Level Gameplay and Mechanics

### Version 1: Player

#### The Development’s Outcome

|  |  |
| --- | --- |
|  | The image to the left shows the player sprite interacting This version featured the successful implementation of spawning the player, player movement, changing player animations, and attacking options. |

#### The Development Story

|  |  |
| --- | --- |
| The development process for this iteration started out by creating an Entity class. As stated in the Design section, I will encapsulate a majority of the player and enemy features into the entity class which will reduce the need to repeat code and improve efficiency through inheritance. | |
| The constructor method for the Entity class turned out to be much more complex than I initially thought. The key attributes such as *all\_animations* holds an array of strings of the animations that the entity can undergo. This array also has values assigned to it by default if nothing is passed in and the object created is a for a player. By doing this it reduces the need to pass in animations constantly, and only requires passing if the entity will have different animations.  The constructor is als0o responsible for creating several motion states, and changes to the entity image through self.*flip\_image* or self.direction. This is also responsible for assigning a default weapon: sword and creating a dictionary to store weapon damage. This was done to allow adding of additional weapons in the future if I decide to do so. | |
| *class* Entity(pygame.sprite.Sprite):  *def \_\_init\_\_*(*self*, x, y, obj\_type, scale, max\_health=100, x\_vel=6, all\_animations=*None*, combat\_animations=*None*,  melee\_dps=33):  *self*.JUMP\_Y = 14.7  *self*.GRAVITY = *self*.JUMP\_Y / 20  *self*.all\_animations = all\_animations  *self*.combat\_animations = combat\_animations  *if* all\_animations *is None and* obj\_type == 'player':  *self*.all\_animations = ['Idle', 'Running', 'Jumping', 'Falling', 'Melee', 'Bow', 'Die']  *if* combat\_animations *is None and* obj\_type == 'player':  *self*.combat\_animations = [*4]*  pygame.sprite.Sprite.*\_\_init\_\_*(*self*)  *self*.max\_health = max\_health  *self*.health = *self*.max\_health  *self*.y\_vel = 0  *self*.obj\_type = obj\_type  *self*.animations = []  *self*.animation\_pointer = 0  *self*.current\_action = 0  *self*.time1 = pygame.time.get\_ticks()  *for* animation *in self*.all\_animations:  *self*.get\_animations(obj\_type, animation, scale)  *self*.image = *self*.animations[*self*.current\_action][*self*.animation\_pointer]  *self*.rect = *self*.image.get\_rect()  *self*.rect.topleft = (x, y)  *self*.x\_vel = x\_vel  *self*.direction = 1 *# [1,-1] = [facing right, facing left]  self*.flip\_image = *False  self*.jumping = *False  self*.in\_air = *True  self*.sword\_attack = *False  self*.current\_weapon = 1 *# 1 == sword, 2 == Bow  self*.mask = pygame.mask.from\_surface(*self*.image)  *self*.current\_weapon\_damage = {1: melee\_dps} *# sword deals 50 damage*  *self*.remove = *False* | |
| The first method to be created was the draw() method. This takes in a surface and draws the current image making any adjustments needed such as flipping it based on the direction of motion. | |
| *def* draw(*self*, surface):  surface.blit(pygame.transform.flip(*self*.image, *self*.flip\_image *or self*.direction == -1, *False*), self.rect.topleft) | |
| With the basic draw method created for the player, I proceeded to instantiate the player in the World’s process\_data() method to return a player object at the correct position, and invoked the draw method to see if it worked as intended. | |
| *elif* tile *in* tile\_info['player'].split(): *# create player if there's one on the map* player = Entity(img\_rect.x, img\_rect.y, player\_img, tile\_info['player\_scale'], melee\_dps=500)  In the main loop:  player.draw(window.screen) | |
| Doing so produced the following result: | |
|  | The player is drawn at the position that was specified in the tile maker. The direction of the image is also flipped when I changed the player’s direction attribute also meaning that I had little tampering to do with the draw method. |
| To prevent the player from constantly floating in the air, I proceeded to develop the move() method for the entity class. | |
| The move method uses two determinate variables to adjust the entity’s position: *moving\_left* and *moving\_right.* The rate at which the player’s distance changes is affected by the self.x\_vel and self.y\_vel variables. Instead of moving the entity’s position, two variables: dx, and dy are used which represent the proposed change in direction. Using these, depending whether the player is moving left or right, dx will be negative or positive respectively depending on self.x\_vel.  During jumping motions, the y\_vel is initially set to a value which causes the player to move upwards. However, gravity is constantly applied to the entity to keep them on the ground/ fall when they aren’t in contact with the ground, this also causes that initial jump to eventually fall back down, but in gradually accelerated motion due to the force of gravity. During this motion, other attributes are also altered to lock the jump action from occurring again until the entity has reached the ground from the jump. | |
| *def* move(*self*, moving\_left, moving\_right, world): *# handle player movement*  *# reset movement variables* dx = dy = 0  check = *True  if self*.health <= 0:  *return   # horizontal movement   if* moving\_left *and* check:  dx = -*self*.x\_vel  *self*.flip\_image = *True  self*.direction = -1   *if* moving\_right *and* check:  dx = *self*.x\_vel  *self*.flip\_image = *False  self*.direction = 1   *# jumping/vertical movement  if self*.jumping *and* (*not self*.in\_air) *and* check:  *self*.y\_vel = -*self*.JUMP\_Y  *self*.jumping = *False  self*.in\_air = *True   # gravity/ downward acceleration*  *self*.y\_vel = *min*(*self*.y\_vel + *self*.GRAVITY, 10)  dy += *self*.y\_vel | |
|  | |
| Collision checks are done independently; first collision in the x axis is checked, and then in the y axis. If there is a collision when the entity is moved in the proposed direction, that change in the direction is set to 0 to reverse the effects of the movement that causes the collision and the player’s position is set so that it just about touches the tile but not to an extent that will trigger collisions again. dx, and dy are and then applied to the entity’s position at the end of the procedure | |
| *# check collision with floor for* tile *in* world.obstacle\_list:  *# check collision in x direction  if* tile.rect.colliderect(*self*.rect.x + dx, *self*.rect.y, *self*.rect.w, *self*.rect.h):  dx = 0  *# If AI collision with wall, turn em around   if* tile.rect.colliderect(*self*.rect.x, *self*.rect.y + dy, *self*.rect.w, *self*.rect.h):  dy = 0  *# check if jumping and collision occurs below ground  if self*.y\_vel < 0:  *self*.y\_vel = 0  *self*.rect.top = tile.rect.bottom   *# if they are falling  elif self*.y\_vel > 0:  *self*.y\_vel = 0  *self*.in\_air = *False  self*.rect.bottom = tile.rect.top *# update player position self*.rect.x += dx *self*.rect.y += dy | |
|  | |
| Outside the main game loop, two new variables: *moving\_left* and *moving\_right* were set to False by default. Inside of the main game loop, within the event capture loop a condition was added that would invoke the jumping action. | |
| *if* event.key == pygame.K\_w:  player.jumping = *True* | |
| Outside of the event capture loop, the moving\_left and moving\_right variables are adjusted constantly according to which keys were being pressed. This is because unlike the jump button which doesn’t need to be held down, it just needs to be pressed to trigger the jump animation. However, moving requires the continuous holding down of the appropriate keys. If this were to be added in the event capture loop, each press would trigger one *self.x\_vel* movement in the direction, meaning they would have to continuously press and release the movement keys. This approach avoids that outcome | |
| Underneath player.draw(), player.move() is invoked with the appropriate arguments. The death\_blocks are also drawn and checked for collision with the player. | |
| keys = pygame.key.get\_pressed() moving\_left = keys[pygame.K\_a] *and* attack\_conditions moving\_right = keys[pygame.K\_d] *and* attack\_conditions  player.draw(window.screen, camera) player.move(moving\_left, moving\_right, world)  *# do death block checking for player for* tile *in* death\_blocks\_group:  tile.draw(window.screen) | |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Spawning player | Player should be spawned at the intended location specified in the level creation | Works as intended | ✔ |
| Enemy Spawning | Enemies are spawned at their specified location in the level creation | Doesn’t exist in version 1 | ✘ |
| User movement | Using the appropriate keys moves the player in that respective direction. | Works as intended | ✔ |
| Flip the image of the player and enemies based on the direction they are facing. | Works as intended | ✔ |
| After jumping once, the player shouldn’t be allowed to jump again. | Player is able to jump as many times as they want – even in the air. | ✘ |
| Update player and enemy animations based on what action they’re currently in. | Does not exist in version 1 | ✘ |
| Player dies if they fall off the map. | Player keeps falling infinitely, not triggering death/ removing player. | ✘ |
| Plyer dies If they collide into a death block | Doesn’t exist in version 1 | ✘ |
| Player collision with the tile prevents them from phasing through tiles and stops movement | The dx and dy variables are changed to 0, causing no further movement if a collision in proposed direction occurs | ✔ |
| Combat | Enemy hits reduce player health, and player hits reduces enemy health. | Does not exist in version 1 | ✘ |
| Enemy death triggers question display | Does not exist in version 1 | ✘ |
| Enemy AI | Move enemies based on their assigned radius, and flip direction after this distance is covered. | Does not exist in version 1 | ✘ |
| Trigger combat when player collides within a certain distance | Does not exist in version 1 | ✘ |
| Randomly idle for a certain period of time to create disharmony between all enemy movements. (don’t make enemy movement look in-sync) | Does not exist in version 1 | ✘ |
| Camera | Follow the player | Does not exist in version 1 | ✘ |

### Version 2: Animations

#### The Development’s Outcome

|  |
| --- |
| This version featured the addition of text such as the current weapon, points and the time elapsed. It also saw the introduction of animations and fixed some logic errors that arose in the previous version. The screenshots above show some of the animation states of the user. |

#### The Development Story

|  |
| --- |
| The development process started out by fixing some of the key issues from the previous version. The first issue was double jumping. In order to fix this, an additional variable called ground\_conditions is assigned a Boolean True or False value at each iteration of the main game loop to check that the player is on the ground, alive and isn’t in the air. |
| ground\_conditions = *not* player.in\_air *and* player.health *and* (  player.y\_vel <= player.GRAVITY) *and not* start\_fade *# making sure player isn't in the air and is still alive* |
| This also led me to pre-emptively create move\_conditions to check if they player is not in the attack animation.  move\_conditions = *not* (  player.sword\_attack) *and not* start\_fade *# only allow attacking if not already in attack animation -> ADD INTO ITERATIVE DEVELOPMENT* |
| ground\_conditions and move\_conditions are added into the condition statement for when the jump key is pressed so that jumps are only allowed when the player is on the ground. However, horizontal movement keys only had the move\_conditions check added in to prevent the player from moving whilst attacking, this way it also allows them to move whilst jumping. |
| moving\_left = keys[pygame.K\_a] *and* move\_conditions moving\_right = keys[pygame.K\_d] *and* move\_conditions |
| The infinite falling was a fairly easy fix. Since I added a height attribute to the class World, I was able to utilise this so that when the player’s feet have reached the bottom of the map (the height (number of tiles in the y direction) \* the tile’s y dimension), the remove attribute is set to True. The health isn’t reduced to 0 here because the when the death animation is added, realistically there is no surface for the player’s death animation to occur, hence the instant removal of the player. |
| *# check if going off the sides if self*.rect.y > world.height\*Display.TILE\_DIMENSION\_Y - *self*.rect.h: *# if the player is off screen  self*.remove = *True  # self.kill()*  *return* |
| With the logic errors addressed, I first created a custom error that is used for debugging. I found myself using quite a lot of print statements during the development stage of previous sections, hence I decided to add a more robust and professional form of error checking. This way execution stops and displays an appropriate message based on the error.  *class* StopRunning(*Exception*):  *"""raised to display an error message about something and stop execution. Avoiding print statements"""  pass*  A pseudocode example that can be used during development:  IF conditions is not met:  raise StopRunning(Error message for why the execution of the program has stopped) |
|  |
| After implementing that debugging feature, I moved on to implement the animations for entities. To store all the animations of the current entity, I created a class attribute *animation* which is a 2D array storing arrays of the framers for each of the animations. In order to fill the *animations* array, I developed the get\_animations() method. |
|  |
| This method takes in: obj\_type: string; the animation: string; and the scale: tuple to which the image is to be resized. Although there is a scale, a dictionary *animation\_scale* was created to store altered scale sizes for the different animations. This is because not all the image sizes for the animations were the same, therefore they were re-scaled according to the current scale passed in. This way no matter what the scale parameter, the secondary scale for the animations will be relative to that parameter. Next, the function gets a list of all the images and retrieves the scale from the animation\_scale dictionary to which the current animation needs to be re-scaled to. After applying this to all the images, the images are stored in a *temp* array which is then added the main *self.animations* array to create a 2D array. |
| *def* get\_animations(*self*, obj\_type, animation, scale): *# add animations for this sprite into a list* animation\_scale = {...}  temp = []  img\_path = f"images/mobs/{obj\_type}/{animation}"  images = os.listdir(img\_path) *# get a list of images* scale2 = scale  scale\_info = animation\_scale.get(obj\_type)  *if* scale\_info:  scale\_data = scale\_info.get(animation)  *if* scale\_data: scale2 = [\**map*(*int*, scale\_data)]  scale2 = [\**map*(*int*, (scale2[0] \* 0.8, scale2[1] \* 0.8))]   *for* image *in* images: *# iterate through the images in this directory* image = pygame.image.load(os.path.join(img\_path,image))  temp += [pygame.transform.scale(image,scale2).convert\_alpha()]  *self*.animations += [temp] |
| After the completion of this method, I invoked it within the constructor method. |
| *for* animation *in self*.all\_animations:  *self*.get\_animations(obj\_type, animation, scale) |
| Subsequently, the update\_action was created which will be used to switch between animations, by changing the value of *self.current\_action* and resetting the frame pointer *animation\_pointer* to 0. This method also performs a check such that the animation that is being changed to isn’t the same as the current animation to prevent the entity being stuck in the first frame of an animation. |
| *def* update\_action(*self*, new\_action: *int*, world=*None*):  *""" check if the new action is different to the new action  # if the new action is the same as the old action, it would set animation pointer to 0 every time so only the  # first frame of the animation would be shown. By adding this check, it makes it so that the animation pointer and animation is changed/reset  # only if there is a change in the player action. """*  *if* new\_action != *self*.current\_action:  *self*.current\_action = new\_action  *# reset the index at which the animation for the specific action starts at  self*.animation\_pointer = 0  *self*.time1 = pygame.time.get\_ticks() |
| With the functionality of the animation loading and switching, I moved on to switching between the actual frames within an animation. This was accomplished by changing the *self.animation\_pointer* action value (*self.current\_action* is used to point to the index in *animations*  from *all\_animations* whilst *animation\_pointer* is used to switch between frames of an animation. |
| This feature was accommodated by the animation\_handling() method. This initially ensures that the entity is alive by invoking self.check\_alive() which at the time only had a pass statement as the feature was yet to be implemented. It also establishes a cooldown time and amends it if a certain animation needs longer cooldown times. *self.image* and *self.rect* are also altered here depending on the animation being changed to and are positioned such that when a change occurs depending on the direction the entity was facing it keeps them on the same position because, as stated earlier, not all images are the same size. |
| *def* animation\_handling(*self*): *# updates the animation frame  self*.check\_alive()  *# update animation based on a timer since last time recorded* cooldown\_time = 90 *# every 120 main game loops change animation frame  if self*.current\_action == *self*.get\_index('Idle'):  cooldown\_time = 120   *# update entity image  self*.image = *self*.animations[*self*.current\_action][*self*.animation\_pointer]  image\_rect = *self*.image.get\_rect()  image\_rect.midbottom = *self*.rect.midbottom  *if self*.obj\_type != 'stormy':  *if self*.direction == 1:  image\_rect.bottomleft = *self*.rect.bottomleft *# keep the entity on the ground  else*:  image\_rect.bottomright = *self*.rect.bottomright   *self*.rect = image\_rect  *self*.idle\_rect.midbottom = image\_rect.midbottom *# move the collision rectangle   if self*.obj\_type =='player': *# only the player's hit box should be dynamic, for the rest it should be normal  self*.idle\_rect = image\_rect *self*.mask = pygame.mask.from\_surface(*self*.image)  current\_time = pygame.time.get\_ticks()  death\_index = *self*.get\_index('Die') |
| Once the alteration of the image and mask is finished, the main section of code is run. This is responsible for changing the actual animation frame. Once the cooldown time has passed, *self.animation\_pointer* is incremented by 1. For animations where looping doesn’t occur additional checks are carried out and appropriate attributes are changed. Specifically, when attack animations finish, the respective attack attribute is set to False, and when the entity’s death animation is finished it is signalled to be removed/ officially considered dead.  For looping animations, the modulous operator is utilised so that when the animation pointer reaches the last frame, adding one more to it will loop back to the beginning of the array.  At the end *time1* is changed to the current time, ready to check for the next change in *animation\_pointer*. |
| *# change animation frame if* (current\_time - *self*.time1) > cooldown\_time:  *self*.animation\_pointer += 1 *# add one to animation pointer  if self*.current\_action *in self*.combat\_animations: *# if its a combat animation  if self*.animation\_pointer >= *len*(  *self*.animations[*self*.current\_action]): *# if at the last frame of animation  self*.sword\_attack = *False # no longer attacking with a weapon  self*.animation\_pointer = 0 *# reset animation pointer  self*.current\_action = 0 *# go back to idle position   elif self*.current\_action == death\_index: *# death animation  if self*.animation\_pointer > *len*(*self*.animations[death\_index]) - 1:  *self*.animation\_pointer = *len*(*self*.animations[death\_index]) - 1  *self*.remove = *True   else*: *# looping animations  self*.animation\_pointer = *self*.animation\_pointer % *len*(*self*.animations[*self*.current\_action])   *self*.time1 = pygame.time.get\_ticks() |
| After the core of the animation handling was finished, I was able to move on to the method that would actually the trigger which animations should occur and when. But before that I made a get\_index() method which is used to the index of an animation from *animations* instead of having to use/remember numbers, referencing the actual name is much easier especially since the indexing of *all\_animations* is the same as *animations,* index 0 of *all\_animations* represents idling, and index 0 of *animations* holds the frames for idling. |
| *def* get\_index(*self*, animation: *str*): *# get the death index from the list of animations* index = *None  if* animation *in self*.all\_animations:  index = *self*.all\_animations.index(animation)  *return* index |
| With that done, I proceeded to the development of the update method. I decided to move the self.move() method which was previously in the main loop into here to reduce the amount of code need in the main loop, furthering the use of encapsulation. This update method is specifically for a player and will be overwritten in the Enemy class. Checks are performed by using condition statements to determine the state of the player; it invokes the self.update\_action() passing in the return value of self.get\_index() for the action/animation to be changed to. If none of the conditions are fulfilled, the player is put in the ‘Idle’ state. |
| *def* update(*self*, moving\_right: *bool*, moving\_left: *bool*, world):  *self*.move(moving\_right, moving\_left, world)  *self*.animation\_handling()  *# update player animations  if self*.obj\_type == 'player' *and self*.health:  *if self*.in\_air *or self*.y\_vel > *self*.GRAVITY: *# if jumping or falling the 0.75 is due to gravity  if self*.y\_vel < 0: *# if going upwards  self*.update\_action(*self*.get\_index('Jumping')) *else*: *# if falling  self*.update\_action(*self*.get\_index('Falling')) *elif self*.sword\_attack:  *self*.update\_action(*self*.get\_index('Melee'),  world) *elif* moving\_right *or* moving\_left:  *self*.update\_action(*self*.get\_index('Running'))  *else*:  *self*.update\_action(*self*.get\_index('Idle'))  After the addition of the update method, I finally proceeded to making the check\_alive() method which is used to check if the player is alive and activates the death animation by checking if they’re health is so. The result of this is also returned as it used in several other instances to validate entity actions. |
| *def* check\_alive(*self*): *# check if the entity is alive* alive = *self*.health > 0  *if not* alive: *# if they've died  self*.health = 0 *# health is back at 0* death\_index = *self*.get\_index('Die') *# get the index of where the 'Die' animation is  self*.update\_action(death\_index) *# update the series of images that contain the death animation  return* alive |
| The main game loop’s player handling section was then restructured to reflect these changes: the check\_alive() method is invoked at the very beginning of the loop.  player.check\_alive()  *# game logic*  *. . .*  *# player handling* player.draw(window.screen, camera) player.update(moving\_left, moving\_right, world) |
|  |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Enemy Spawning | Enemies are spawned at their specified location in the level creation | Doesn’t exist in version 2 | ✘ |
|  | User movement | After jumping once, the player shouldn’t be allowed to jump again. | Player is able to jump as many times as they want – even in the air. | ✔ |
| Update player and enemy animations based on what action they’re currently in. | Does not exist in version 1 | ✔ |
| Player dies if they fall off the map. | Player keeps falling infinitely, not triggering death/ removing player. | ✔ |
| Player collision with the tile prevents them from phasing through tiles and stops movement | The dx and dy variables are changed to 0, causing no further movement if a collision in proposed direction occurs.  During the jumping motion, whilst falling and the user collides with the corner of a tile, it triggers collision and phases them to the top, because they think player is falling and colliding with ground. | ✘ |
|  | Plyer dies If they collide into a death block | Doesn’t exist in version 2 | ✘ |
| Combat | Enemy hits reduce player health, and player hits reduces enemy health. | Does not exist in version 2 | ✘ |
| Enemy death triggers question display | Does not exist in version 2 | ✘ |
| Enemy AI | Move enemies based on their assigned radius, and flip direction after this distance is covered. | Does not exist in version 2 | ✘ |
| Trigger combat when player collides within a certain distance | Does not exist in version 2 | ✘ |
| Randomly idle for a certain period of time to create disharmony between all enemy movements. (don’t make enemy movement look in-sync) | Does not exist in version 2 | ✘ |
| Camera | Follow the player | Does not exist in version 2 | ✘ |

### Version 3: Camera movement, Text, and Death Blocks

#### The Development’s Outcome

|  |
| --- |
| In this feature, I further considered Dan’s recommendation about a dynamic map; a map that has some sort of scrolling mechanism. In response to that, I added in a camera that follows the player. Whilst the player’s position changes, the camera adjusts its position too. All other objects that have a draw method will have their drawing positions also changed to create a scrolling animation, but their actual x and y coordinates remain as they were originally (with the exception of entities] which maintains collision detection. |

#### The Development Story

|  |
| --- |
| This development process started out by addressing the issues in the previous section. Although it was a feature that was working properly, there was just one case where phasing through tiles occurred. This is because the size of the fall and jump animations wasn’t equal which resulted in a chain reaction of collisions. As shown in the images below, the rect of the jumping and falling is unequal (zoom in to 200% to see). Hence, when checking tile collision and the player is falling, its assumed they are landing on the ground, but in this case that isn’t occuring, hence they’re moved to the top of the tile they collided with and causing a chain reaction with each tile above it until reaching the top. |
| The fix was quite easy. I re-sized the motion images and then scaled them proportionally to maintain consistency amongst themselves. Idle, Running, Jumping, and Falling now all have the same width and height to prevent this issue from arising again. |
| animation\_scale = {  'player': {  'Melee': (scale[0] \* 1.4, scale[1]),  'Idle': (48, scale[1]),  'Die': (scale[0] \* 1, scale[1] \* 0.8),  'Running': (48, scale[1]),  'Jumping': (48, scale[1]),  'Falling': (48, scale[1])  }, |
| After fixing of corner tile collision, it was followed by the development of the camera. The camera movement will be handled by a class which moves relative to the player. It’s x and y position are adjusted to reflect that of the player based on the direction they’re facing. This is done by initiating this. The constructor method takes in a target which should be an object contains a rect and adjusts the position of the camera so that its centred relative to the screen. |
| *class* Camera:  *def \_\_init\_\_*(*self*, target):  *self*.rect = target.rect.copy()  *self*.x, *self*.y = target.rect.topleft  *self*.rect.x = window.WIDTH // 2  *self*.rect.y = window.HEIGHT // 2   *def* update(*self*, target):  *if* target.current\_action *not in* target.combat\_animations:  *# make sure the camera doesn't jitter after switching animations. Keeps the camera in place  if* target.direction == 1:  *self*.rect.x = target.rect.x  *else*:  *self*.rect.x = target.rect.right  *# self.rect.y = window.HEIGHT // 2 + 200*  I tested this out by instantiating the Camera outside of the main game loop and invoking camera.update() with the player object passed in as an argument. Doing so successfully followed the player in the x direction. |
| Following the success of the scrolling in the x direction, I moved on to implement the vertical scrolling too. This was straightforward because pygame’s rect objects have in-built attributes that return certain coordinates. Thus, all I had to do was change the old statement to self.rect.bottomleft = target.rect.bottomleft.  In addition to that, I also added python’s in built assertions as a precaution for when the camera class is used. This is done so that when the target is passed in, it ensures that the target is a rect object before proceeding further. Unlike raising errors through Exception, assertions only occur when the conditional statement is met. On the other hand, Exceptions can be used at any point. Another precaution was the checking that the target had combat animations. This is because during combat animations, it would change the x and y values because the image sizes aren’t the same.  Furthermore, I removed the need to pass in the target when updating the camera; the target attribute is created in the constructor method. This is because python’s objects are call by reference; this means that the object’s memory address is referenced thus, whenever any changes are made in the main game loop to the target (player) these changes will be accessible by the Camera class through the self.target attribute. |
| *class* Camera:  *def \_\_init\_\_*(*self*, target):  *assert hasattr*(target, 'rect'), 'target needs a rect object'  *self*.target = target  *self*.rect = target.rect.copy()  *self*.x, *self*.y = target.rect.topleft  *self*.rect.x = window.WIDTH//2  *self*.rect.y = window.HEIGHT // 2   *def* update(*self*):  *assert hasattr*(*self*.target, 'rect'), 'target needs a rect object'  *if hasattr*(*self*.target, 'combat\_animations') *and hasattr*(*self*.target, 'current\_action'):  *if self*.target.current\_action *not in self*.target.combat\_animations:  *# make sure the camera doesn't jitter after switching animations. Keeps the camera in place  if self*.target.direction == 1:  *self*.rect.bottomleft = *self*.target.rect.bottomleft  *else*:  *self*.rect.bottomright = *self*.target.rect.bottomright  *# self.rect.y = window.HEIGHT // 2 + 200  else*: *#  if self*.target.direction == 1:  *self*.rect.bottomleft = *self*.target.rect.bottomleft  *else*:  *self*.rect.bottomright = *self*.target.rect.bottomright |
| With the Camera class finished, I instantiated it outside of the main game loop with the player passed in as an argument and invoked the update() method at the beginning of the main game loop. |
| camera = Camera(player) *while* run:  player.check\_alive()  camera.update() |
| With the camera now set up, I changed the draw method for the Tile and Entity class, whilst adding the functionality of the death blocks in the Tile class. |
| The draw method now calculates the new drawing position of the tile, relative to the camera movement. This same logic is also applied for the entity class. |
| *def* draw(*self*, surface, target):  temp = *self*.rect.copy()  x, y = target.rect.topleft  temp.x = temp.x - x + Display.WIDTH//2  temp.y = temp.y - y + Display.HEIGHT//2  surface.blit(*self*.image, temp.topleft)  Entity draw method:  *def* draw(*self*, surface, target):  temp = *self*.rect.copy()  temp.x = temp.x - target.rect.x + Display.WIDTH // 2  temp.y = temp.y - target.rect.y + Display.HEIGHT // 2  surface.blit(pygame.transform.flip(*self*.image, *self*.flip\_image *or self*.direction == -1, *False*), temp.topleft)  *# pygame.draw.rect(surface, (0,255,255), temp, 1)* |
| In addition to the change in the draw() method, within the Tile class, the update method was added to trigger collisions and animation\_handling() for tiles with animations. The check\_collision() method is specifically made to check if most of the entity’s body is submerged within the tile to a certain extent (50%) and then reduces the health to 0 if the tile was collided into was a death block. The collision is carried out using pygame’s masks for pixel perfect collision in the mask\_collision() method. |
| *def* update(*self*, obj, camera=*None*):  *# do the animation handling  if hasattr*(*self*, 'animation\_handling'):  *self*.animation\_handling()  *# check for collision* collision = *self*.check\_collision(obj)  *return* collision  *def* check\_collision(*self*, objs):  collision = *self*.mask\_collision(objs) *and* objs.rect.y + objs.rect.h\*0.5 >= *self*.rect.y *# if more than half of body is submerged  if* collision:  *if self*.obj\_type == 'Death Block':  objs.health = 0 *return True*  *return True  def* mask\_collision(*self*, obj):  mask = pygame.mask.from\_surface(*self*.image)  offset = (obj.rect.x - *self*.rect.x, obj.rect.y - *self*.rect.y)  *return* mask.overlap(obj.mask, offset) |
| Afterwards, I moved on to display the text on the screen about key information. I also realised in this iteration that the background draws over the back button, hence it needs to be called after invoking the draw() method for the world.  The current weapon and points were easy to implement because there are already variables that exist. The Current weapon is made so that adding weapons later on and displaying which one is selected is easier. All I have to do is add the name of the weapon to the array in the format string statement. The coordinates are arbitary as they were tested to see what position they first best in. |
| *# draw text*  window.draw\_back() window.draw\_text(f'Current Weapon: {["Sword"][player.current\_weapon - 1]}', (200, 5)) window.draw\_text(f'Points: {points}',(490,5)) |
| Adding the timer required some extra steps. I first created a time1 variable outside of the main game loop. Within the main game loop, the current time is calculated and subtracted from the initial time. If the time passed is greater than or equal to 1000 milliseconds, the timer is incremented by 1, and the time1 variable is updated to match the current time. |
| *if* (pygame.time.get\_ticks() - time1) >= 1000 *and not* paused: *# 1 ticks == 1 millisecond, 1000 millisecond = 1 second, update timer every second* timer += 1 *# account for the time in the question screen* time1 = pygame.time.get\_ticks()  Displaying the timer alone would be showing the seconds passed, thus it led me to create a function that is used to format the time in a readable format. This function converts the seconds passed into hours, minutes and seconds format. It also fills in leading 0s for singular digits.  This allowed the time to be drawn on the horizontal center of the screen, with the converted time format. |
| *def* convert\_time\_format(time):  h, r = *divmod*(time, 3600) *# calculate how many hours, and the remainder is the number of minutes (in seconds)* m, s = *divmod*(r, 60) *# convert the remaining seconds into minutes, and the remainder is the amount of seconds  return* f'{h:02}:{m:02}:{s:02}' *# format each value by filling in missing number(s) of maximum 2 numbers with leading 0s.* |
| *# draw text*  *...*  window.draw\_text(text=f'Time: {WINDOW.convert\_time\_format(timer)}', pos=(670,3), size='MEDIUM', center=(*True*,*False*)) |

|  |
| --- |
| Within the main game loop, the invocation of the draw methods for the player, death blocks and tiles in the class World, now have an additional required parameter *target* which is the camera. |
| In World’s draw method:  *for* tile *in self*.all\_tiles:  tile.draw(window.screen, target)  Main game loop:  player object calling draw method:  player.draw(window.screen, camera)  iterating through death blocks:  *for* tile *in* death\_blocks\_group:  tile.draw(player, camera)  tile.update(player, camera) |

|  |
| --- |
| The final feature that was implemented was alternate key mappings. This was made so that the user can choose to use the arrow keys or the [w, a, d] keys for movement. This was made because whilst observing Mr Waring and Dan, they found themselves instinctively using the arrow keys to move until I informed them about the movement keys. |
| The addition of this was very simple as there are only three instances where user input for movement occurs: jumping, moving left, and moving right: the jumping condition block was updated so that it contained the Up key as a possible key which could trigger the event by using the Boolean ‘or’ operator. |
| *if* (event.key == pygame.K\_w *or* event.key == pygame.K\_UP) *and* ground\_conditions *and* move\_conditions *and not* player.in\_air:  The movement keys followed the same logic: the addition of optional LEFT and RIGHT key checks are added by using the Boolean ‘or’ operator. |
| moving\_left = (keys[pygame.K\_a] *or* keys[pygame.K\_LEFT]) *and* move\_conditions moving\_right = (keys[pygame.K\_d] *or* keys[pygame.K\_RIGHT]) *and* move\_conditions |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Enemy Spawning | Enemies are spawned at their specified location in the level creation | Doesn’t exist in version 3 | ✘ |
|  | User movement | Player collision with the tile prevents them from phasing through tiles and stops movement | Collision works as intended. | ✔ |
| Plyer dies If they collide into a death block | Player death animation is triggered and is unable to move from there on. | ✔ |
|  | Player can use different key sets to move | Works as intended. | ✔ |
| Combat | Enemy hits reduce player health, and player hits reduces enemy health. | Does not exist in version 2 | ✘ |
| Enemy death triggers question display | Does not exist in version 2 | ✘ |
| Enemy AI | Move enemies based on their assigned radius, and flip direction after this distance is covered. | Does not exist in version 2 | ✘ |
| Trigger combat when player collides within a certain distance | Does not exist in version 2 | ✘ |
| Randomly idle for a certain period of time to create disharmony between all enemy movements. (don’t make enemy movement look in-sync) | Does not exist in version 2 | ✘ |
| Camera | Follow the player | Follows the player and adjusts positions to center the player on the screen. Also maintains collision detection with tiles. | ✔ |

### Version 4: Enemy AI with animations

#### The Development’s Outcome

|  |
| --- |
| This version introduced enemies and additional user interface features such as health bars. Enemies have their own individual movement patterns and also attack the player upon collision within a certain distance. Enemy animations are also fully functional due to the efficient use of inheritance from the Entity class. |

#### The Development Story

|  |
| --- |
| As all the errors were addressed in the last iteration, my first action was creating the Enemy class. This inherits from the Entity class because there are a large number of similarities, but due to some of the differences in the behaviour of the enemies, it constituted for a child class to be made. The constructor method experienced the addition of attributes that weren’t initially in the Design of the Enemy class. A majority of these, similar to the Entity class, are used to determine the state. However, some of the key attributes were the move\_radius, attack\_radius as these would be responsible for movement and triggering combat.  Initially, the design for enemy attacking was that when the masks(pixels) collide, enemy combat is engaged, however when discussing with Dan, he said that he’d like some more space to try and dodge attacks if possible. This led to the creation of the attack vision; its calculated by using the attack\_radius which is used to create a rectangle that points outwards from the enemy in the direction of motion and is used to check for collision.  The attack\_radius is calculated by retrieving the largest width from all the images with the addition of padding. |
| *class* Enemy(Entity):  *def \_\_init\_\_*(*self*, x, y, obj\_type, scale, max\_health=100, x\_vel=2, all\_animations=*None*, attack\_radius=150,  move\_radius=3, melee\_dps=30):  *super*().*\_\_init\_\_*(x, y, obj\_type, scale, max\_health, x\_vel, all\_animations, melee\_dps=melee\_dps)  *self*.move\_counter = 0  *self*.idling = *False  self*.idling\_counter = 0  attack\_animations = *self*.animations[*self*.get\_index('Attack')]  idle\_image = *self*.animations[*self*.get\_index('Idle')][0]   attack\_radius = *max*(attack\_animations, key=*lambda* image: image.get\_width()).get\_width() + 5  attack\_radius -= idle\_image.get\_width() - 10  *self*.attack\_vision = pygame.Rect(0, 0, attack\_radius, 3)  *self*.combat\_animations = [*self*.get\_index('Attack')]  *self*.attacked = *False  self*.wait = 0  *self*.change\_direction = *False  self*.wall\_collision = *False  self*.move\_radius = move\_radius |
| The first method to be developed was the AI for an enemy instance. Initially it checks if the player is falling and invokes the move method with no movement in the x or y direction and sets the motion to idling using the set\_idling() method.  In addition, when the entity is during the attack animation, it skips any other movement related statements as this avoids the enemy’s image triggering unintended collision during an attack animation.  The attack vision also adjusted so that it matches the direction of motion |
| *def* AI(*self*, world):  *if self*.in\_air *or self*.y\_vel > *self*.GRAVITY: *# if the enemy is falling  self*.move(0, 0, world) *# don't move in any direction  self*.set\_idling(world)  *return # don't do anything else related to AI movement  self*.wait = *max*(0, *self*.wait - 1)   AI\_moving\_right = *False  if self*.current\_action *in* [*self*.get\_index('Attack')]:  *return   # set up attack radius  self*.attack\_vision.center = *self*.rect.center  *if self*.direction == 1:  *self*.attack\_vision.left = *self*.rect.right  *else*:  *self*.attack\_vision.right = *self*.rect.left |
|  |
| The above in an image of how the attack vision should look like (although the actual drawing of the rectangle will be omitted). |
| After the checks are performed and adjusting the attack vision, I started creating the main algorithm that handles enemy AI. The enemy is set into a state of idling randomly and remains in that state until the idling time is up. Additional checks are performed during the idling motion to check once it’s finished if the enemy is to change direction.  The enemy’s move counter denotes how much they have to move. Upon moving their assigned move\_radius or colliding with a wall, the enemy’s move counter is multiplied by -1 which makes them walk twice the distance in the opposite direction to reach move\_radius again. The AI\_moving\_left and AI\_moving\_right variables are determined by the direction the enemy is facing. The move\_counter is incremented by the enemy’s x velocity. |
| *if self*.check\_alive():  *# checking if enemy is not already idling and not falling/jumping  if* random.randint(1, 500) == 1 *and not self*.idling *and not self*.in\_air:  *self*.set\_idling(world) *# if the enemy is in idling motion:  if self*.idling:  *# self.update\_action(0)  self*.idling\_counter -= 1  *if self*.idling\_counter <= 0:  *self*.idling = *False  if self*.change\_direction:  *self*.direction \*= -1  *self*.move\_counter \*= -1  *return # don't attempt to move the player in idling animation   if self*.direction == 1:  AI\_moving\_right = *True* AI\_moving\_left = *not* AI\_moving\_right  *self*.move(AI\_moving\_left, AI\_moving\_right, world)  *self*.update\_action(*self*.get\_index('Running'), world)  *self*.move\_counter += *self*.x\_vel   *self*.change\_direction = *False  if* (*self*.move\_counter > (*self*.move\_radius) *or self*.wall\_collision) *and self*.y\_vel <= *self*.GRAVITY:  *self*.change\_direction = *True  self*.set\_idling(world)  *self*.wall\_collision = *False* |
| Since the AI method uses the set\_idling() method quite frequently, I promptly started developing the set\_idling() method. This updates the state of the enemy to idling, sets a counter for how long the enemy is to remain in the idle state and updates the animation to idling. Because I inherited from Entity, I didn’t have to re-write the update\_action() method again for this class. |
| *def* set\_idling(*self*, world):  *self*.idling = *True  self*.idling\_counter = 50  *self*.update\_action(*self*.get\_index('Idle'), world) |
| Similar to the Entity class I created the update method for this class which invokes animation\_handling() and AI() . |
| *def* update(*self*, player, surface, world):  *self*.animation\_handling()   *# combat checking  if self*.health > 0:  *pass*  *self*.AI(world) *# do enemy AI* |
| After that, as shown in the snipper above, the combat checking was the only missing feature for the enemies and the player as there was no way to register damage taken. The main method that is invoked is combat\_collision. This invokes the check\_collision method used to check collision between two entities, and deals damage to the current entity if they were dealt damage based on the opposing entity’s current weapon damage.  The check\_collision() method uses pygame’s mask class and the offset between the two entities’ positions to calculate pinpoint pixel collision to register damage and returns the overlapping pixels. The Boolean form of the overlap will either be True or False; True == pixels overlapped, False == no pixels overlapped. |
| *def* combat\_collision(*self*, obj): *# check for sword attack collision  if self*.check\_collision(obj) *and* obj.sword\_attack:  *# self.collisions += 1 and obj.sword\_attack and self.direction != obj.direction  self*.health -= obj.current\_weapon\_damage.get(obj.current\_weapon) / 25  *return* 1  *return* 0  *def* check\_collision(*self*, obj):  *if self*.health <= 0:  *return* obj\_mask = pygame.mask.from\_surface(pygame.transform.flip(obj.image, obj.direction == -1,  *False*)) *# flips the mask of the image during collision detection* offset\_x = obj.rect.x - *self*.rect.x  offset\_y = obj.rect.y - *self*.rect.y  current\_mask = pygame.mask.from\_surface(pygame.transform.flip(*self*.image, *self*.direction == -1,  *False*)) *# flips the mask of the image during collision detection* collision = current\_mask.overlap(obj\_mask, (  offset\_x, offset\_y)) *# making sure player is in sword animation  return bool*(collision) |
| Once the combat collisions were finished, I created a method that initiates enemy attacking the player: start\_attack(). If the player crosses the enemy’s attack vision, then the enemy is put into an attacking state and puts a cooldown until they can again the player next to prevent continuous attacking. |
| *def* rec\_collision(*self*, obj):  *# check if obj is within enemy's attack vision*  *return self*.attack\_vision.colliderect(obj.rect)  *def* start\_attack(*self*, obj, world):  *# if attack cooldown is over, obj within attack vision and enemy(self) on the ground*  *if self*.wait == 0 *and self*.rec\_collision(obj) *and self*.check\_alive() *and* (*not self*.in\_air *and self*.y\_vel >= *self*.GRAVITY):  *self*.sword\_attack = *True   self*.update\_action(*self*.get\_index('Attack'), world) *# change the animation to attack animation  self*.wait = 100 *# initiate cooldown* |
| Once the core of the collisions was finished, I added in the missing algorithms into the update() method. This invokes start\_attack() to initiate combat if another entity is within attack range, it then checks damage on itself and on the opposing entity: the player, to successfully register damage. |
| *# combat checking if self*.health > 0:   *self*.start\_attack(player, world) *# check if player collision has* player.combat\_collision(*self*) *# check if self has dealt damage to player  self*.combat\_collision(player) *# check if player has dealt damage to self* |
| With that, I was finished with the Enemy class. |
| Before proceeding to adding the enemies, I created a class to Group all these different objects into one to reduce having to individually write for loops for each object. This not only reduces the code size in the main loop but also the readability which greatly helped in debugging during testing. |
| This class takes in an unspecified number of parameters that will be objects. Each object is stored in *self.sprites* by unpacking the arguments into a python List. There are several other methods which some objects have and some don’t. To avoid issues arising, the in-built hasattr() function is used to check whether a certain object (sprite) has the method to be performed. If there is then the parameters and key word parameters in the Group’s methods are passed in as arguments for the respective sprite’s method. |
| *class* Group:  *def \_\_init\_\_*(*self*, \*args):  *self*.sprites = [\*args]   *def* update(*self*, \*args, \*\*kwargs):  *for* sprite *in self*.sprites:  *if hasattr*(sprite, 'update'):  sprite.update(\*args, \*\*kwargs)   *# this is for images  def* draw(*self*, surface, target=*None*):  *for* sprite *in self*.sprites:  *# Check if the sprite has a `draw` method.  if hasattr*(sprite, 'draw'):  sprite.draw(surface, target=target)  *else*:  surface.blit(sprite.image, sprite.rect)  *def* check\_death(*self*): *# used to check if an entity has died  for* obj *in self*.sprites:  *if* obj.remove:  entity\_died = *True* obj.kill()*# free up memory space*  *self*.sprites.remove(obj) *# remove the obj from the array*  *return* entity\_died |
| The check\_death() function has little use right now but will be used later on when enemy deaths will trigger the question display. Regardless, the Group class was then instantiated outside of the main game loop for all the different objects: |
| After that I added the code in the class World which adds enemy objects to a list of enemies.  This is done within the process\_data() method when the world map is created. All enemies will only contain 4 animations: Idle, Die, Running, and Attack. The move radius of the enemy is determined by reading the tile\_info.json and getting the index of the current enemy which is held in the enemy\_counter local variable. |
| *elif* tile *in* tile\_info['enemy'].split():  enemies += [Enemy(img\_rect.x, img\_rect.y, enemy\_img,   tile\_info['enemy\_scale'],  all\_animations=['Idle', 'Die', 'Running', 'Attack'],  max\_health=100, x\_vel=2, move\_radius=tile\_info['move\_radii'][enemy\_counter], melee\_dps=35)]  enemy\_counter += 1  To reflect all of these changes, here is the updated section of the play\_level () function |
| *# sprite groups* player, decorations, death\_blocks, enemies, portals = world.process\_data(tile\_info, img\_dict, PLAYER, ENEMY)  death\_blocks\_group = Group(\*death\_blocks) enemy\_group = Group(\*enemies) portal\_group = Group(\*portals)  *while* run:  *same code as earlier ...*  *# player handling* player.draw(window.screen, camera)  player.update(moving\_left, moving\_right, world)   *# enemy handling* enemy\_group.update(player, window.screen, world)  enemy\_group.draw(window.screen, target=camera)   *# death block handling* death\_blocks\_group.draw(window.screen, target=camera)  death\_blocks\_group.update(player, camera) *# do death block checking for player* |
| Before moving onto the quiz window section, I added in a check when updating animations to combat animations that checks whether the action will cause a collision with a tile, and if it does then don’t change to the attack animation. This is similar to the previous bug when the player image sizes weren’t the same. In the Idle state the player may be just about touching the wall, but because the attack animations have a greater width, it would instantly trigger tile collisions. |
| The frame\_change\_collision() method was created to address the issue. It gets the rectangle of the frame and checks whether it causes a tile collision or not. If it does, then True is returned, else False. |
| *def* frame\_change\_collision(*self*, image, world):  image\_rect = image.get\_rect()  *if self*.direction == 1:  image\_rect.bottomleft = *self*.rect.bottomleft *# keep the entity on the ground  else*:  image\_rect.bottomright = *self*.rect.bottomright   *for* tile *in* world.obstacle\_list:  *if* image\_rect.colliderect(tile.rect):  *return True  return False* |
| This method is invoked within the update\_action() method. This is the additional code that was added to the method at the top. It first gets all the images. If any of the frames collide with a tile, then the *sword\_attack* state is set to False and stops executing any further lines from this method. If no collisions occurred, it proceeds as usual and update the animation if it wasn’t the same as the previous one. |
| *def* update\_action(*self*, new\_action: *int*, world=*None*):melee\_index = *self*.get\_index('Melee')  *if* new\_action == melee\_index *and* world:  images = *self*.animations[melee\_index]  *if any*(*self*.frame\_change\_collision(image, world) *for* image *in* images): *# if wall collisions have occured in any of the frames  self*.sword\_attack = *False  return   if* new\_action != *self*.current\_action:  *self*.current\_action = new\_action  *# reset the index at which the animation for the specific action starts at  self*.animation\_pointer = 0  *self*.time1 = pygame.time.get\_ticks() |
| The final feature that was added in this version was the draw\_health\_bar(). This feature was specifically requested by Dan, but Mr Waring also approved of it. During gameplay, it was hard to see how much health the player and enemeis had remaining, thus an indicator for their healths was brought up during the previous meeting with my two stakeholders.  The draw\_health\_bar() method was created within the entity class. This also takes a target as a parameter and follows the same logic for the change in drawing coordinates as the draw() method which was changed in the previous version. self.health\_rect’s position was modified to reflect this change and adjusts its position depending on the obj\_type of the entity.  It then creates health\_colour and lost\_health\_colour which are assigned RGB values. The player has a greenish tone, whereas other entities will have a redder tone. These values were initially completely green and completely red, with the lost\_health\_colour being black. However, Dan and Mr Waring both preferred that there be colours that are less vibrant and also the lost health colour should match the health colour, but with a different tone. After hearing back from them I assigned new RGB values for the health bars as shown in the snipped below.  First, a rectangle representing the maximum health is drawn and has the lost\_health\_colour filled to it. The current health bar is drawn on top of the maximum health bar, and has its width adjusted proportional to the (current health)/(maximum health). Finally, a thin black border is drawn around the health bars. |
|  |
| *def* draw\_health\_bar(*self*, surface, target):  temp = *self*.idle\_rect.copy() *# copy the rect of the current entity* temp.x = temp.x - target.rect.x + Display.WIDTH / 2.0  temp.y = temp.y - target.rect.y + Display.HEIGHT // 2 - 10   *if not self*.check\_alive(): *# if the entity is dead, don't draw a health bar  return   self*.health\_rect.midtop = temp.midtop  *if self*.direction == -1 *and self*.obj\_type *in* ['knight', 'samurai']: *# adjusts the position of the health bar if need be  self*.health\_rect.topright = temp.topright    health\_colour = (16, 130, 0)  lost\_health\_colour = (173, 181, 172)    *if self*.obj\_type != 'player': *# enemy health bar colours* health\_colour = (255, 38, 0)  lost\_health\_colour = (219, 182, 175)   pygame.draw.rect(surface, lost\_health\_colour, *self*.health\_rect)  current\_health = *self*.health\_rect.copy()  current\_health.w = (*self*.health / *self*.max\_health) \* *self*.health\_rect.w *# the % of health \* full width* pygame.draw.rect(surface, health\_colour, current\_health)  pygame.draw.rect(surface, (0, 0, 0), *self*.health\_rect, 2) |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Enemy Spawning | Enemies are spawned at their specified location in the level creation | Loads in enemies at the positions specified in the level creation application. | ✔ |
| User movement | Player collision with the tile prevents them from phasing through tiles and stops movement | Collision works as intended. | ✔ |
| Plyer dies If they collide into a death block | Player death animation is triggered, and player is unable to move from there on. | ✔ |
| Combat | Enemy hits reduce player health, and player hits reduces enemy health. | Damage taken on an entity is reflected on their health bar. | ✔ |
| Enemy death triggers question display | Does not exist in version 3 | ✘ |
| Enemy AI | Move enemies based on their assigned radius, and flip direction after this distance is covered. | Enemies only move the number of blocks (left and right from the starting position) assigned to them | ✔ |
| Trigger combat when player collides within a certain distance | Enemy’s attack players before their actual rectangles/ pixels collide. | ✔ |
| Randomly idle for a certain period of time to create disharmony between all enemy movements. (don’t make enemy movement look in-sync) | Enemies randomly idle during motion, and then continue their motion. | ✔ |
| Camera | Follow the player | Follows the player and adjusts positions to center the player on the screen. Also maintains collision detection with tiles. | ✔ |

## Quiz

### Version 1: Question screen + introduction of AutoBox class + Option handling

#### The Development’s Outcome

|  |  |
| --- | --- |
| Graphical user interface, website  Description automatically generated |  |
| The first features implemented for the Quiz gameplay was the question display, and the ability to pick an option which then shows the user if that selected option was wright or wrong, followed by showing the state of all other options too: correct/incorrect. This is made clear by a correct option being filled with a green background, whilst the rest are filled with red. | |

#### The Development Story

|  |
| --- |
| To start the development process, I created a new class for text named AutoBox. This inherits from the Textbox, but because this is used in the question display, there are some functionalities that are different and specific to this class, this has been achieved by polymorphism through overwriting. |
| Outside of the constructor method, a few class attributes are created. Within the constructor method, key attributes such as the font, rectangle size, text rectangle size. In addition to these, the add\_text() method is called which is used to draw the text within the text rectangle. |
| *class* AutoBox(Textbox):  default\_background = Textbox.default\_background  incorrect\_colour = (204, 51, 0)  correct\_colour = (51, 153, 51)  hover\_colour = (65, 114, 191)  hover\_border = (207, 234, 255)   *def \_\_init\_\_*(*self*, x, y, size, obj\_type, text='', font\_size='MEDIUM',center\_text=(*True*,*True*),colour=default\_background):  *self*.background\_colour = colour  *self*.border\_colour = *None   if isinstance*(font\_size, *int*): *# if a custom font size is passed in  self*.font = pygame.font.SysFont('Sans', font\_size)   *elif* font\_size.upper() *in* 'MEDIUM LARGE MEDLARGE SMALL'.split():  *self*.font = *eval*(f'self.{font\_size}\_FONT')   *else*:  StopRunning('Invalid text size passed in')   *self*.obj\_type = obj\_type  *self*.x, *self*.y = x, y  *self*.rect = pygame.Rect(x, y, \*size)  *self*.surface = pygame.Surface(*self*.rect.size)  text\_rect\_size = [0.9\**self*.rect.w,0.9\**self*.rect.h] *# create a padding for where the text will be placed   # calculate the offset to place the text rectangle in the center of the main box rectangle  self*.dx = (*self*.rect.w - text\_rect\_size[0])//2  *self*.dy = (*self*.rect.h - text\_rect\_size[1])//2  *self*.text\_rect = pygame.Rect((*self*.rect.x + *self*.dx, *self*.rect.y + *self*.dy, \*text\_rect\_size)) *self*.text=text  *self*.center\_text = center\_text  *self*.add\_text(text)  *self*.background = *self*.background\_colour  *self*.check\_collision = *True* |
| The first method I developed was the add\_text() method. This takes in a text string as a parameter, splits the text by spaces and renders the text onto a surface: self.surf -> this is a surface of the text rectangle but with the features of a pygame.Surface object. It then tries to blit each letter of each word individually on a copy of self.surf, and if a letter is unable to fit within the text rectangle, algorithm attempts to repeat the process of drawing the word on a new line (technically, lower down the screen by increasing the y coordinate at which the text is being blitted). If a word is blitted successfully, the changes are saved to self.surf. A newline can also occur with a special character ‘\\n’. This wasn’t there initially, but whilst adding questions; splitting certain lines of feedback text makes it easier to read and more organised. |
| *# this allows text to fit in a box. def* add\_text(*self*, text):  *"""  \\n = newline  break up the text and try to blit each word individually; if at any point any of the letters for a word goes out,  then repeat the blit attempt process for the word again but on a new line/y-coordinate.* text = text.split() *# split all the words  # modification variables* add\_y = 0  widths = 0  *# text variables* pointer = 0  letter\_count = 0  height = *self*.font.render('h',1,(255,255,255)).get\_height()  *self*.surf = pygame.Surface(*self*.text\_rect.size, pygame.SRCALPHA, 32)  *self*.surf.convert\_alpha()   *if self*.obj\_type == 'button' *and len*(text)==1:  rendered = *self*.font.render(text[0],1,(255,255,255))  *self*.surf.blit(rendered,(0,0))  *return   # to evaluate: this can go on forever if the box is too small for all of the text to fit in* last\_word\_attempt = 0  max\_x = 0  *while* pointer <= *len*(text) - 1:  current\_word = text[pointer]  *if* last\_word\_attempt > 2: *# if 2 new line attempts have been made then it means the word still cannot fit  # break out of the while loop and stop attempting to infinitely trying to add the word  break* temp = *self*.surf.copy() *# create a temporary surface where letters will be blitted* font\_letters = []   *for* letter *in* current\_word:  font\_letters += [*self*.font.render(letter, 1, (255, 255, 255))]   *for* index,letter *in enumerate*(font\_letters): *# iterating through each rendered letter* newline = *False* rect = letter.get\_rect()  *if not* letter\_count: *# if the row has no letters (if its the first letter)* proposed\_x, proposed\_y = 0, add\_y \* height  letter\_count += 1  widths = 0 *# initial padding from the side of rectangle  else*:  proposed\_x = widths *# sum all the widths + padding of previous letters* proposed\_y = add\_y \* height  *# if adding a character of a word will overflow, add the whole word to the next line  if* proposed\_x + rect.w > *self*.text\_rect.w:  newline = *True   if* newline *or* current\_word == '\\n':  max\_x = *max*(max\_x, *min*(*self*.text\_rect.w, proposed\_x+rect.w))  add\_y += 1 *# increasing the y* widths = 1 *# resetting the widths to 1* letter\_count = 0 *# reset the letter count on the row  if* pointer == *len*(text)-1: *# if its the last word and still not fitting* last\_word\_attempt += 1  *if* current\_word == '\\n':  pointer += 1  *break # repeat the process again for this word but on a new line* max\_x = *max*(max\_x, *min*(*self*.text\_rect.w, proposed\_x+rect.w))   temp.blit(letter, (proposed\_x, proposed\_y)) *# blit it on the temp surface* widths += rect.w + 1 *# +1 is for the padding between letters   else*: *# if the loop finished iterating meaning that all letters were successfully blitted  if* letter\_count: *# if there is a first character on the row  # self.surf.blit(self.font.render(' ',1,(255,255,255)), (proposed\_x, proposed\_y))* widths += *self*.font.render(' ',1,(255,255,255)).get\_width()*# this is the "space" between each word* pointer += 1 *# once a word has finished blitting, move onto the next one  self*.surf = temp.copy() *# if a word was fully blitted, then add it onto the main canvas/surface of this box   self*.text\_rect.h = (add\_y+1) \* font\_letters[0].get\_height()  *self*.text\_rect.w = max\_x  *def* update\_text(*self*, text):  *self*.add\_text(text) |
| The update\_text() method is used to change the text of a textbox. This isn’t as useful for the question display, but it may be useful in cases where the text of a box requires changing. |
| The next method that was added was the show() method. As the method’s name suggests, this is used to draw self.surface onto a specified surface. Self.surface contains self.surf. By splitting up the different surfaces it allowed for individual management and control of the surfaces which would have otherwise been much more cumbersome. This method also centres the text rect if specified in the constructor method. |
| *def* show(*self*, surface):  *# self.surface.fill(self.background)* pygame.draw.rect(*self*.surface, *self*.background, (0, 0, *self*.rect.w, *self*.rect.h))  *if self*.border\_colour: *# if there is a border colour* pygame.draw.rect(*self*.surface, *self*.border\_colour, (0, 0, *self*.rect.w, *self*.rect.h), 3)   x, y = *self*.dx, 0  *if self*.center\_text[0]:  x = (*self*.rect.w-*self*.text\_rect.w)//2  *if self*.center\_text[1]:  y = (*self*.rect.h-*self*.text\_rect.h)//2  *self*.surface.blit(*self*.surf, (x, y, *self*.text\_rect.w, *self*.text\_rect.h))  *# blit everything onto the specified surface* surface.blit(*self*.surface,(*self*.x,*self*.y)) |
| Next, the interactivity with the box was added through the check\_click() and check\_hover() methods. The logic behind these is the same as the Textbox methods but there are additional checks for what type of object it is and change in the alpha value of the surface.  Mainly, the check\_click() method returns self, which is the AutoBox instance. This is used later on in the main quiz window loop in checking box click results. Both of these methods ensure that the object being clicked or hovered isn’t a question because these actions should not be applicable for the question box. |
| *def* check\_hover(*self*, mouse\_pos=0):  *self*.surface.set\_alpha(190)  *# if the mouse position is over the rectangle, change colour, otherwise change it back to normal  if self*.obj\_type != 'question':  mouse\_pos = pygame.mouse.get\_pos()  *if self*.rect.collidepoint(mouse\_pos) *and self*.check\_collision:  *self*.background = *self*.hover\_colour  *self*.border\_colour = *self*.hover\_border  *else*:  *self*.background = *self*.background\_colour  *self*.border\_colour = *None  def* check\_click(*self*, mouse\_pos=0):  *if* (*not self*.check\_collision) *or* (*self*.obj\_type == 'question'): *return* mouse\_pos = pygame.mouse.get\_pos()  *if self*.rect.collidepoint(mouse\_pos) *and* (pygame.mouse.get\_pressed()[0]):  *return self* |
| Similar to the entity Group class, I created a class to group boxes together to avoid having to repeat the same method calls and assignments for each individual box. The update\_boxes() method ensures that if check\_hover() is to be called, the box should have collision checking enabled, and when clicking, it ensures that another box hasn’t already been clicked to prevent selecting multiple options. |
| *class* BoxGroup:  *def \_\_init\_\_*(*self*, \*args):  *self*.objects = [\*args]   *def* update\_boxes(*self*, surface):  box\_clicked = *False  for* box *in self*.objects:  *if hasattr*(box, 'show'):  box.show(surface)  *if hasattr*(box, 'check\_hover') *and* box.check\_collision:  box.check\_hover()  *if hasattr*(box, 'check\_click') *and not* box\_clicked: *# if a box hasn't already been clicked  if* box.check\_click():  box\_clicked = *True # a box has been clicked   def* check\_clicks(*self*):  box\_clicked = *False  for* box *in self*.objects:  *if hasattr*(box, 'check\_click') *and not* box\_clicked:  *if* box.check\_click():  box\_clicked = box  *return* box\_clicked   *def* get\_list(*self*):  *return self*.objects |
| Upon completion of the AutoBox class, before proceeding any further, a function get\_boxes() was created which would handle the instantiation of AutoBox objects and automatically format them. I was going to manually position the boxes before, but upon further consideration I deemed it easier to automatically format them which is determined by two variables: x\_padding and y\_padding.  Initially, this creates a question box with the text argument being the question paramter passed into the function. The options are also collected by navigating to the question data in the file and shuffled in a random order using the sample method from the random module. After creating a question box, the dimensions of the option boxes are calculated by using a proportion of the question box, and positioned according to their dimensions and paddings. |
| *def* get\_boxes(question\_data, question, window):  w, h = window.width, 200   text = question  options = question\_data[question]['options']  options = random.sample(options, 4) *# select a random order from the options, shuffling the order of options* question\_box = AutoBox(0, 0, (w, h\*0.8), obj\_type='question', text=text)   *# calculate dimensions of the boxes.* x\_padding = 35  x1 = x\_padding  x2 = (w//2) + x\_padding  width = (w//2) - (2\*x\_padding)  option\_w, option\_h = (width,(window.HEIGHT-h)\*0.47)  y\_padding = (window.HEIGHT - question\_box.rect.height - (2\*option\_h))//3   *# creating all the option instances, and positioning them respectively* option\_1 = AutoBox(question\_box.rect.x + x1, question\_box.rect.bottom + y\_padding, (option\_w, option\_h), obj\_type='option', text=options[0])  option\_2 = AutoBox(x2, question\_box.rect.bottom + y\_padding, (option\_w, option\_h), obj\_type='option', text=options[1])  option\_3 = AutoBox(option\_1.rect.left, option\_1.rect.bottom + y\_padding, (option\_w, option\_h), obj\_type='option', text=options[2])  option\_4 = AutoBox(option\_2.rect.left, option\_2.rect.bottom + y\_padding, (option\_w, option\_h), obj\_type='option', text=options[3])  main\_group = BoxGroup(option\_1, option\_2, option\_3, option\_4, question\_box)  *return* main\_group  Finally, a BoxGroup object is created which is assigned the main\_group variable and is returned. |
| With the functionality of the boxes finished, the start\_question() function was then developed. The purpose of this function is to display appropriate buttons, text and return the result of the user input for the question.  At the beginning of the function the correct answer is retrieved by accessing the first element of the options array for that question. After that the get\_boxes() function is called to store a collection of the boxes to be displayed when the quiz window first appears. Thereafter, additional variables are created to handle the timer. |
| *def* start\_question(question, question\_data, timer=0):  *# ---------- key variables -------------#*  correct\_answer = question\_data[question]['options'][0] *# correct answer will always be at first index of options* main\_group = get\_boxes(question\_data, question, window)   continue\_button = Textbox(100, 0.9\*window.height,text='Continue',text\_size='medlarge',padding=(10,10))  options\_screen = *True* result = *None* check\_click = *True* time1 = 0  continue\_button.create\_rect()  start\_fade = *True*  going\_back = *False* time1 = time1  timer = timer  pause\_timer = *False* |
| The main quiz window loop. This is responsible for invoking methods used to show the boxes on the window and the update\_boxes() method to enable interactivity amongst the boxes. There are also several conditions also put into place before proceeding with these actions. The event capture loop’s main feature is to check the user’s selected box and change the box properties according to that. |
| *while True*:  window.refresh()  *for* event *in* pygame.event.get():  *if* event.type == pygame.QUIT *and* result *is None*:  *return* timer  *# sys.exit()  if* event.type == pygame.KEYDOWN:  *if* event.key == pygame.K\_ESCAPE:  *return* timer   *if* event.type == pygame.MOUSEBUTTONDOWN:  *# check for the continue button click on the main/ options screen  if* pygame.time.get\_ticks() - time1 > move\_to\_feedback *and* time1 *and* options\_screen: *# wait a bit of time  if* continue\_button.check\_click():  options\_screen = *False # don't display options anymore, signalling the feedback screen to show   # check for continue button click on the feedback screen  if* continue\_button.check\_click() *and not* options\_screen *and not* move\_to\_feedback:  start\_fade = *True* going\_back = *True* clicked = main\_group.check\_clicks() *# check if an option has been clicked  if* clicked *and* check\_click: *# if an option has been clicked* result = clicked.text == correct\_answer *# check if the clicked option's text matches to the correct answer* time1 = pygame.time.get\_ticks()*# initiate time for continue button to show*  check\_click = *False # don't check for more clicks on any other options   # change the options' properties based on the outcome  for* option *in* main\_group.objects:  *if* option.obj\_type=='question': *# if the box is a question, then don't make any changes  continue   if* option.text != correct\_answer: *# incorrect answer* option.background = option.incorrect\_colour  option.hover\_colour = option.incorrect\_colour  *else*: *# correct answer* option.background = option.correct\_colour  option.hover\_colour = option.correct\_colour  move\_to\_feedback //= 2 *# if they're right, show the continue button faster* option.check\_collision = *False # don't check for collisions with the selected option/button anymore* pause\_timer=*True # don't continue the timer after the question has been answered to allow the user to absorb info without worrying about time* clicked.border\_colour = clicked.hover\_border *# highlight the selected colour so the user doesn't forget what they clicked* clicked.border\_radius = 7  *if* options\_screen: *# if the user is still on the options screen* main\_group.update\_boxes(window.screen) *# update the boxes (draw them) onto the screen   elif not* options\_screen: *# if an option has been picked, and it was incorrect show the feedback text  if not* feedback.text:  going\_back = *True # go back   if not* going\_back:  feedback.show(window.screen) *# show the feedback* feedback\_continue.check\_hover(pygame.mouse.get\_pos())  feedback\_continue.show(window.screen, center=*True*)  move\_to\_feedback = *False   # after a option is picked, and after a certain time, move to feedback screen, and display its continue button  if* pygame.time.get\_ticks() - time1 > move\_to\_feedback *and* time1 *and* options\_screen:  continue\_button.check\_hover(pygame.mouse.get\_pos())  continue\_button.show(window.screen, center=*True*)  *for* box *in* main\_group.get\_list():  *if* box.obj\_type =='question': *# keep the question opacity the same  continue* box.surface.set\_alpha(60) *# reduce the opacity of other options   if* start\_fade: *# do the fade animation  if* going\_back:  fade.direction = -1 *# fading out  if* fade.fade(window.screen):  start\_fade = *False  if* going\_back: *# if they're exiting the question screen  return* result, timer   *if* (pygame.time.get\_ticks() - time1) >= *and* move\_to\_feedback *not* pause\_timer: *# 1 ticks == 1 millisecond, 1000 millisecond = 1 second, update timer every second* timer += 1 *# account for the time in the question screen* time1 = pygame.time.get\_ticks()   window.draw\_text(text=f'Time: {WINDOW.convert\_time\_format(timer)}', pos=(670,3), size='MEDIUM',center=(*True*,*False*))  pygame.display.update()  clock.tick(FPS) |
|  |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Question preparation and display | Selecting questions | Select questions based on the red, amber, green quotas | Doesn’t exist in version 1 | ✘ |
| Select one question in Quiz window | Select a random question and be able to access its data | Works as intended | ✔ |
| Show question and options | Display the question and options in a suitable format | The question and options are displayed with padding and readable font size. | ✔ |
| Determine if a selected option was correct | A visible indication of getting a question right or wrong | The border of the correct and incorrect options is adjusted after an option is selected. | ✔ |
| Move onto feedback screen | Clicking on the continue button displays the feedback text | The continue button works (clicking on it no longer shows the option screen), but the feedback text is not displayed in this version. | ✘ |
| Text string contains a singular word which should be too large to fit in the box. | Text should not be drawn in the box if it doesn’t fit. | The add\_text()’s while loop doesn’t stop running because the word never fits. | ✘ |
| First word of a sentence is too large to fit in a box. | ✘ |
| Pause timer | When an option is selected, pause the timer | Works as intended | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Combat | Enemy death triggers question display | Does not exist in version 1 |  |
| Getting a question correct | Increase user points | Doesn’t exist in version 1 | ✘ |
| Recover a portion of user health | Doesn’t exist in version 1 | ✘ |
| Show summary statistics | A page which shows how the user performed in the most recent session. | Doesn’t exist in version 1 | ✘ |

### Version 2: Feedback

#### The Development’s Outcome

|  |  |
| --- | --- |
|  |  |
| This version saw a re-design of the indication if the user got a question right or wrong, this was suggested by Dan because with the updated design, it makes it much easier. Upon clicking the continue button, it now shows a feedback screen. Initially, if the user got the question right, it would skip the feedback screen, but Mr Waring suggested that if a user guessed the right answer, they wouldn’t know why they got it right. This was also supported by Dan’s comments as he also found instances where he guessed and answer but wanted to know why it was correct. | |
|  | |

#### The Development Story

|  |
| --- |
| This version started out by fixing the infinite while loop bug in the previous version. The fix was simple. When word is attempted to be drawn onto the screen, and it isn’t successful in two lines, the loop breaks. This was added at the beginning of the while loop to prevent constantly repeating the same blitting process on one word indefinitely. |
| *while* pointer <= *len*(text) - 1:  current\_word = text[pointer]  *if* last\_word\_attempt > 2: *# if 2 new line attempts have been made then it means the word still cannot fit  # break out of the while loop and stop attempting to infinitely trying to add the word  break* |
| Next, I moved on to changing the properties that were altered in the previous version to denote correct and incorrect options to make it more visually explicit. |
| *if* option.text != correct\_answer: *# incorrect answer* option.background = option.incorrect\_colour *else*: *# correct answer* option.background = option.correct\_colour  move\_to\_feedback //= 2 *# if they're right, show the continue button faster*  BoxGroup’s update\_boxes() method  *def* update\_boxes(*self*, surface):  ...  *if hasattr*(box, 'check\_hover') *and* box.check\_collision:  box.check\_hover() ... |
| Next the feedback screen was developed. In order to avoid having to create a separate loop, a check was made with the options\_screen variable. Since the program already stops displaying the question and options once the use clicks on continue and options\_screen is set to False, an additional selection statement was added which shows the feedback\_box. The feedback box is instantiated outside of the quiz loop. |
| feedback = AutoBox(0,40,(window.SIZE[0],window.SIZE[1]-40),text=feedback\_text, obj\_type='feedback',font\_size=29,center\_text=(*False*,*False*),colour=Display.BACKGROUND)  The code snippet below contains the addition of showing the feedback. An addition check is made in the event loop where if the continue button is clicked on the feedback screen, then going\_back is set to true. Another instance where going\_back is set to true is if there is no feedback text to show. Once going\_back is True, the function returns the result and timer. This is used so that the timer in the quiz window and the game level remain in sync. |
| *while True*:  window.refresh()  *for* event *in* pygame.event.get():  ...  *# check for continue button click on the feedback screen  if* continue\_button.check\_click() *and not* options\_screen *and not* move\_to\_feedback:going\_back = *True*  ...  *if* options\_screen: *# if the user is still on the options screen* ...  *elif not* options\_screen: *# if an option has been picked, and it was incorrect show the feedback text  if not* feedback.text: *# if there is no feedback*  going\_back = *True # go back   if not* going\_back:  feedback.show(window.screen) *# show the feedback* continue\_button.check\_hover(pygame.mouse.get\_pos())  continue\_button.show(window.screen, center=*True*)  move\_to\_feedback = *False   # after an option is picked, and after a certain time, move to feedback screen, and display its continue button  if* pygame.time.get\_ticks() - time1 > move\_to\_feedback *and* time1 *and* options\_screen:  ... *if* going\_back: *# if they're exiting the question screen  return* result, timer  pygame.display.update()  clock.tick(FPS) |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Question preparation and display | Selecting questions | Select questions based on the red, amber, green quotas | Doesn’t exist in version 2 | ✘ |
| Select one question in Quiz window | Select a random question and be able to access its data | Works as intended | ✔ |
| Show question and options | Display the question and options in a suitable format | The question and options are displayed with padding and readable font size. | ✔ |
| Determine if a selected option was correct | A visible indication of getting a question right or wrong | The border of the correct and incorrect options is adjusted after an option is selected. | ✔ |
| Move onto feedback screen | Clicking on the continue button displays the feedback text | Works as intended. Upon clicking on the continue button in the feedback screen, it returns the result and timer as intended | ✔ |
| Pause timer | When an option is selected, pause the timer | Works as intended | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Combat | Enemy death triggers question display | Does not exist in version 2 | ✘ |
|  | Getting a question correct | Increase user points | Doesn’t exist in version 2 | ✘ |
|  |  | Recover a portion of user health | Doesn’t exist in version 2 | ✘ |
|  | Show summary statistics | A page which shows how the user performed in the most recent session. | Doesn’t exist in version 2 | ✘ |

### Version 3: Selecting questions

#### The Development’s Outcome

|  |
| --- |
| This version introduced enemy deaths triggering questions successfully along with a strength-based algorithm to pick questions for the user based on their accuracies and a quota for the different categories. Getting questions right also increases the users’ points and helps them recover some health. The health regeneration was suggested by Dan during the development process as he found himself dying quite frequently despite getting questions right. |

#### The Development Story

|  |  |
| --- | --- |
| In order to select questions, a sorting algorithm is needed. Thus, the first algorithm that I started developing was the 2D bubble sort. This bubble\_sort2D() function works as a normal bubble sort but compares the 1st index of the array. This is because this will mainly be used to sort arrays where the 0th index is a name and the 1st index is the data, hence by comparing the data, the sorted array from highest to lowest is returned. | |
| *def* bubble\_sort2D(array) -> *list*: *# sort a 2D array  # We want to stop passing through the list  # as soon as we pass through without swapping any elements* swapped = *True  while* swapped:  swapped = *False  for* i *in range*(*len*(array) - 1):  *if* array[i][1] < array[i+1][1]: *# comparing the key index, can be points or accuracy* array[i], array[i+1] = array[i+1], array[i]  swapped = *True  return* array *# return the sorted 2D array* | |
| Next, I began by writing a function: get\_questions() which takes in level and username as parameters. The initial design was that this would select the 12 worst questions that the user performed at. This was accomplished by using a 2D bubble sort (saved in the WINDOW.py file as it may be used elsewhere too). | |
| *def* get\_questions(level, username): question\_data = read\_json(f'Questions/{level}.json') questions = WINDOW.bubble\_sort2D([[question, question\_data[question][username][2]] *for* question *in* question\_data])  final\_list = questions[:12]  *return* final\_list, question\_data | |
| However, upon further consolidation with Mr Waring, he suggested that students should not be overwhelmed with questions they perform bad at and should instead have a mix of questions with a range of accuracies. This led to the incorporation of a more refined algorithm which splits questions up based on the current user’s accuracy and selects an equal amount from all of them  Before the return statement, final\_list is now an array which contains a mix of questions from the 3 different categories: green, amber and red. The random.sample() method is invoked to select a random sample of 3 questions from that category. There is also an additional check to see if a question has been attempted before. If it has never been attempted, it’s automatically placed in the green category. | |
| *# separate the questions based on which category they fit in* green = [] amber = [] red = [] *for* question, accuracy *in* questions:  total\_attempts = *sum*(question\_data[question][username][:2]) *# sum of right and wrong attempts  if* accuracy >= 0.75 *or* (accuracy == 0 *and* total\_attempts == 0): *# never attempted or strong knowledge* green.append(question)  *elif* 0.5 <= accuracy < 0.75: *# moderate knowledge* amber.append(question)  *else*:  red.append(question) *# if the accuracy is less than 50%; absolute shit-housery* final\_list = [] *# a list of the final questions* accuracy = *lambda* i: question\_data[i][username][2] categories = "green amber red".split() *for* j,i *in enumerate*([green,amber,red]):  questions = random.sample(i, 3)  *print*(f'{categories[j]}: {[accuracy(question) *for* question *in* questions]}')  final\_list.append(questions)  Running the code produced the following result in the console. | |
| green: [0, 0, 1.0] amber: [0.6, 0.6, 0.6] red: [0.2, 0.0, 0.2] | The print statement was used to ensure that the intended number of questions were picked from each category and that the accuracies were correctly categorised. |
| Although this worked in when there were enough questions to pick from each category, when invoked on a user that hadn’t attempted these questions before a ValueError was raised by the random’s sample() method: This meant that the constant sample size for each category was sometimes bigger than the sampling frame (the number of questions in the category). | |
|  | |
| In order to address this, I developed a more robust algorithm which not only ensures that there are enough questions to be picked, but also utilises carry-overs. This means that if a certain category cannot fulfil its quota, the spaces that aren’t able to be filled are passed over to the next category. However, for this to work, I first had to sort the categories based on their lengths from smallest to largest so that the carry-over process would be efficient and require minimal code. | |
| The code above was replaced with the newer version which also utilises max\_quota. This is done so that the number of questions can be easily adjusted depending on the number of enemies on the map, by simply changing that one variable. An additional change is made to the quotas if everything has been carried over the last category and that category doesn’t have enough questions to fulfil the quota, the quota for that category is changed to the length of the category so that it fits however much it can.  In addition to that, the sub-routine was turned into a python generator object which yields values. This meant that I longer had to use a ‘final\_list’ variable nor waste any memory by doing so as the generator will yield the set of questions when the in-built next() function is called or when it is iterated through. | |
| lists = *sorted*([green, red, amber], key=*lambda* i: *len*(i)) *# smallest -> largest lists* max\_quota = 4 *# maximum number of questions to select from each category* quotas = [max\_quota]\*3 *# [max\_quota, max\_quota, max\_quota] -> [red, amber, green]*  *# find out how many of each category should be picked for* index, category *in enumerate*(lists[:-1]): *# loop until n-1  if len*(category) < quotas[index]: *# if the current category doesn't have enough questions for its assigned quota* quotas[index+1] += quotas[index] - *len*(category) *# carry over the remaining ones to the next category* quotas[index] = *len*(category) *# change the quota for the current category to its length as it can only fit that much* quotas[-1] = *min*(quotas[-1], *len*(lists[-1])) *# if everything has been carried over to the last one, it should only pick however much it can  for* index, category *in enumerate*(lists):  current\_quota = quotas[index] *# get the quota that points to the current question* sample = random.sample(category, current\_quota) *# select a random order of (n = quota) questions from the current category  yield* [\*sample] *# yield the set of questions*  The final section of the code randomly selects questions from each category but now the sample() method has the sample size argument as the quota for that current category which no longer causes the error that would arise before, where the sample size was bigger than the actual sampling frame. | |
| After having successfully retrieved questions from the data store, the triggering of questions upon enemy death was implemented. First a check is performed to see if any enemies had died. | |
| enemy\_dead = enemy\_group.check\_death() *if* enemy\_dead *and* questions: *# if there are still questions left, then do the animation to goto question screen* show\_question = *True* | |
| Although the number of enemies and questions should be the same, in order to prevent any errors, I added in the additional conditions of ‘and questions’ which ensures that the questions array isn’t empty. | |
| After both of those conditions have been met, the question display is invoked within another condition statement later on in the game loop.  Firstly, in that condition block, an assertion is made to ensure that there are questions remaining as an extra precaution during the development process to aid in debugging.  After the assert statement is passed, a question is popped from the questions array, which is treated as a stack data structure, and has the start\_question() function invoked (which is stored in another file to optimise organisation) with the current timer, question, and question\_data passed in as arguments.  Once the user has completed the question stage, the return values are stored in the QuestionWindow\_values if the result and timer are returned then this will be in the form of a tuple. In this case, update the user’s statistics base  d on the result in the local dictionary question\_data and add points, increase streak, and recover player health if the question was correct. If no result was returned; the user backed out, then do nothing and only adjust the timer to reflect the time spent on the question page. | |
| *if* show\_question:  *assert bool*(questions), 'No more questions left' *# making sure there are still questions left to pop* current\_question = questions.pop() *# pop the question at the top of the stack* QuestionWindow\_values = QuestionWindow.start\_question(question=current\_question, question\_data=question\_data, timer=timer)   *# extract current stats for the question and adjust them based on result of the user's choice  if isinstance*(QuestionWindow\_values, *tuple*): *# if the result and timer was returned* result = QuestionWindow\_values[0] *# the outcome of the question displayed* user\_right, user\_wrong, user\_accuracy = (question\_data[current\_question])[username]  *if* result:  user\_right += 1; points += 10; total\_right+=1; streak += 1 *# if they got the question right, add 10 points, increase streak, and recover health* player.health = *min*(player.health + 15, player.max\_health)  *else*:  user\_wrong += 1; total\_wrong += 1; streak = 0  *if* user\_wrong!=0 *or* user\_right!=0:  user\_accuracy = user\_right/(user\_right+user\_wrong) *# to ensure that the denominator is not 0* total\_accuracy = *round*(total\_right/(total\_right+total\_wrong), 2)  question\_data[current\_question][username] = [user\_right, user\_wrong, user\_accuracy] *# update statistics of the user on the question displayed* timer = QuestionWindow\_values[1]  *else*:  timer = QuestionWindow\_values  max\_streak = *max*(streak, max\_streak)   show\_question = *False # don't trigger a chain reaction of displaying questions* | |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Question preparation and display | Selecting questions | Select questions based on the red, amber, green quotas | Works as intended, with the additional functionality of carry-overs when quotas are not met. | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Getting a question correct | Increase user points | Adds 10 points to the user’s score each time they get a question right. | ✔ |
|  | Recover a portion of user health | User’s health gets increased by 15. A bug exists where player’s health can exceed above its maximum due to this feature | ✘ |

### Version 4: Transitions

#### The Development’s Outcome

|  |
| --- |
| After having a majority of the core gameplay features functioning. This version introduced a smoother transition mechanism when loading in the game level and moving between the game level and question window screen. A meeting with both my stakeholders suggested that abrupt changes in the window made their experience of the platform less enjoyable. Upon adding in these transitions both my stakeholders were satisfied with the implementation of the screen fading in and out, which led to “much smoother gameplay” as commented by Dan. |

#### The Development Story

|  |
| --- |
| The first change made within this version was addressing the increase of health over the maximum after getting questions right. This was a simple fix:  player.health = *min*(player.health + 15, player.max\_health)  This way, the player’s health will always be lower than or equal to their max\_health. |
| Next, I moved on to develop the transition mechanism. This feature wasn’t initially in the Design of the platform, hence it required extra planning to produce efficient and re-suable code. Thus, I decided to create a class for this specific feature instead of functions like I did for displaying contents for other sections of the platform. The development process started by creating a ScreenFade class. |
| Fade logic: |
| Within the constructor method of this class, the time for the animation is the key variable which will determine the speed of the animation, as shown by the use in the calculation of the self.speed\_x and self.speed\_y attributes. The fade\_counter variabels are used to store the offset/ change in position from the original position of the transition. This enables for horizontal and vertical movement of boxes to occur separately. |
| *class* ScreenFade:  *def \_\_init\_\_*(*self*, direction, colour, time=0.6):  *self*.direction = direction *# 1 == fade outwards, -1 == fade inwards* time = time\*60 *# 60 is the FPS  self*.colour = colour  *# adjust the speed such that the vertical and horizontal walls reach the sides at the same time  self*.speed\_x = *round*((window.WIDTH//2)/time)  *self*.speed\_y = *round*((window.HEIGHT//2)/time)  *self*.fade\_counter\_x = *self*.fade\_counter\_y = 0  *self*.acceleration = 2 |
| The one and only method that this class contains is the fade method. This method is responsible for incrementing the counters and displaying the respective boxes. Initially, there were horizontal and vertical boxes that made up the transition animation but after asking Dan about it, he suggested that vertical animations made the transition look much sleeker and simpler which suited his preference. At the beginning, fade\_complete is set to False, but at the end, because only vertical animations are occurring, the counter\_y variable is checked to see if the boxes have moved a sufficient distance away from the center of the screen in which case the variable states are reset and fade\_complete variable is set to True. |
| *def* fade(*self*, surface):  fade\_complete = *False  self*.fade\_counter\_x += *self*.speed\_x \* *self*.acceleration  *self*.fade\_counter\_y += *self*.speed\_y \* *self*.acceleration  *# noinspection PyTypeChecker  self*.acceleration = *max*(1, *self*.acceleration\*0.97)  counter\_x = *self*.fade\_counter\_x  counter\_y = *self*.fade\_counter\_y  *if self*.direction == 1: *# outwards fade  # horizontal movement  # pygame.draw.rect(surface, self.colour, (0 - counter\_x, 0, window.WIDTH//2, window.HEIGHT))  # pygame.draw.rect(surface, self.colour, (window.WIDTH//2 + counter\_x, 0, window.WIDTH//2, window.HEIGHT))  # vertical movement* pygame.draw.rect(surface, *self*.colour, (0, -window.HEIGHT//2 - counter\_y, window.WIDTH, window.HEIGHT))  pygame.draw.rect(surface, *self*.colour, (0, window.HEIGHT//2 + counter\_y, window.WIDTH, window.HEIGHT))   *else*: *# fade going inwards  # pygame.draw.rect(surface, self.colour, (-window.WIDTH//2 + counter\_x, 0, window.WIDTH//2, window.HEIGHT))  # pygame.draw.rect(surface, self.colour, (window.WIDTH - counter\_x, 0, window.WIDTH//2, window.HEIGHT))* pygame.draw.rect(surface, *self*.colour, (0, -window.HEIGHT + counter\_y, window.WIDTH, window.HEIGHT))  pygame.draw.rect(surface, *self*.colour, (0,window.HEIGHT - counter\_y, window.WIDTH, window.HEIGHT))   *if* counter\_y - 100 > window.HEIGHT/2:  fade\_complete = *True  self*.acceleration = 2  *self*.fade\_counter\_x = 0  *self*.fade\_counter\_y = 0  *return* fade\_complete |
| Within the game\_level() function, an object of this newly created class is instantiated with the direction (1 == boxes move outwards == intro fade) and colour being black (0, 0, 0) , with the start\_fade variable being set to True. This allows for a smooth transition into the game level. |
| fade = ScreenFade(1, (0, 0, 0)) start\_fade = *True* |
| The ground\_ and move\_ conditions are changed so that the player is not allowed to move or attack during the animation period. Outside of the while loop, the first section where a fade animation is triggered is during the portal interaction. If the user successfully enters a portal, the animation should be changed to -1 which is an outro fade (boxes moving in towards the center). |
| *while* run:  ...  *# only perform actions based on these conditions* ground\_conditions = *not* player.in\_air *and* player.health *and* (  player.y\_vel <= player.GRAVITY) *and not* start\_fade *# making sure player isn't in the air and is still alive* move\_conditions = *not* (  player.sword\_attack) *and not* start\_fade *# only allow attacking if not already in attack animation -> ADD INTO ITERATIVE DEVELOPMENT*  ...   *for* event *in* pygame.event.get():  ...   ...  *# portal handling* portal\_group.draw(window.screen, target=camera)  *# if the user hasn't already entered the portal*  *for* portal *in* portal\_group.sprites:  portal\_collision = portal.update(player) *and not* portal\_enter  *if* portal\_collision *and* (*not* questions *or not* enemy\_group.sprites): *# if all questions are answered or enemies are dead, then allow the user to enter the portal* start\_fade = *True # start the transition* fade.direction = -1  show\_question = *False # don't show a question* portal\_enter = *True # mark the portal as entered* |
| The next instance where the fade is changed is during the show\_question condition block. After a question display has finished, the intro fade is started. In addition to this, when the player dies or an enemy is killed, the outro fade occurs which is used when exiting the game level. |
| *if* show\_question:  ...   *# inwards fade* start\_fade = *True* fade.direction = 1  show\_question = *False   # start fade animation if player has died  if* player.remove *and not* start\_fade:  start\_fade = *True* fade.direction = -1   *# if an enemy has died, present a question* enemy\_dead = enemy\_group.check\_death()  *if* enemy\_dead *and* questions: *# if there are still questions left, then do the animation to goto question screen* start\_fade = *True* fade.direction = -1  ...  *# draw text and back button*  ... |
| The actual fade handling occurs within the start\_fade condition block. This is responsible for checking if the fade animation has finished, and if so, determine what to do after the animation has finished depending on the game state. All the following selection statements were coded to handle an outro animation as that is the only time when crucial decisions need to be made. This will determine if a question is to be shown (when an enemy dies), or stop running the while loop and don’t show any questions (the player has entered the portal or they have died as these are the only non-go back button instances when the game level can end). |
| *# do fade animation  if* start\_fade:  *if* fade.fade(window.screen): *# if the fade has completed* start\_fade = *False # don't show the intro fade anymore*  *# Outro animation finished and portal is entered or player dies  if* (portal\_enter *or* player.remove)*and* fade.direction == -1:run = *False* show\_question = *False   # if fading out, check if there are still questions and enemies left to show a question, otherwise its just a normal fade  elif* fade.direction == -1 *and not* player.remove:  show\_question = *True False*   pygame.display.update() *# make all the changes* clock.tick(FPS) |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Getting a question correct | Recover a portion of user health | User’s health gets increased by 15 and stops after reaching maximum health | ✔ |
|  | Show summary statistics | A page which shows how the user performed in the most recent session. | Shows a brief overview of the stats for the sessions | ✘ |
|  |  | Wait a small amount of time before showing continue button | Works as intended | ✘ |

### Version 5: Summary statistics

#### The Development’s Outcome

|  |
| --- |
|  |
| These are the final vesrions of what the summary stastics look like. It shows the key user statstics formated in a readable and user-friendly appraoch with a font size that’s not too large nor too small. It also has the go back button repositioned to the center as this felt intuitive during the development process which turned out to be favoured by Dan and Mr Waring too. |

#### The Development Story

|  |
| --- |
| The implementation of the summary statistics was quite straightforward. This is because the structure was already established in pseudocode in the Design section, all that was left to do was converting it into python code. This was accomplished within the show\_summary() procedure. This is invoked once the main game loop in play\_level finishes.  show\_summary(total\_right, total\_wrong, max\_questions, total\_accuracy, max\_streak, points, timer, player.remove, portal\_enter)  Outside of the main loop for this screen, some key variables were defined about the users statstics, and adjustments are made such the time’s units are calculated using the get\_time\_units() function. The get\_time\_units() function returns the highest unit of time in English. |
| *def* show\_summary(right, wrong, total\_questions, accuracy, streak, points, timer, player\_died, portal\_enter):  timer = WINDOW.convert\_time\_format(timer)  accuracy = *str*(accuracy)+'%'  units = get\_time\_units(timer)  timer = *str*(timer) + units  show\_back = *False* initial\_time = pygame.time.get\_ticks()  attempted = f'{right+wrong}/{total\_questions}'  *def* get\_time\_units(timer):  time\_split = timer.split(':')  units = ''  *if* time\_split[2].lstrip('0'): *# remove leading 0s* units = ' seconds'  *if* time\_split[1].lstrip('0'): *# remove leading 0s* units = ' minutes'  *if* time\_split[0].lstrip('0'): *# remove leading 0s* units = ' hours'  *return* units |
| Within the event capture loop within the main loop in the show\_summary() procedure, the primary form of user input to be checked is the mouse click on the go back button. Outside of the event capture loop, the user data is formatted in a way such that there is sufficient gap between the statistic and the value. In addition to that, depending on certain conditions, text is displayed to tell the user about why the level ended which is also colour coded (green meaning completed/good and red meaning something bad happened/ incomplete level). This was brought about when Dan was testing the gameplay and often asked me why the level ended, and as a result there are three possible messages to be displayed which are not statistics: death message, level quit message and level completion message.  Furthermore, the back button isn’t immediately shown. The program gives the user some time to process their stats before being given the option to proceed. This was suggested by Mr Waring as he brought up that some students may skip useful information about their sessions performance if the skip button is immediately presented to them, hence the adjustments were made so that after 2 seconds the back button is shown. This can be represented by the ‘back=show\_back’ as the keyword argument when the refresh() method is invoked for window. |
| *while True*:  window.refresh(back=show\_back, pos=(650, 480))  *for* event *in* pygame.event.get():  *if* event.type == pygame.MOUSEBUTTONDOWN:  *if* window.check\_return():  *return   if* event.type == pygame.QUIT:  sys.exit()   initial\_x = 430  *for* i,line *in enumerate*(['Attempted', 'Right', 'Accuracy', 'Best Streak', 'Points Collected', 'Time Survived']):  window.draw\_text(line+':', (initial\_x,150+(50\*i)), center=(*False*,*False*))  *for* j,value *in enumerate*([attempted, right, accuracy, streak, points, timer]):  window.draw\_text(*str*(value),(initial\_x\*2,150+(j\*50)))  *if* player\_died:  window.draw\_text('You Died',(0,50),center=(*True*, *False*), colour=(255, 0, 0), size='MEDLARGE')  *elif* portal\_enter: *# if the player was alive and entered the portal* window.draw\_text('Level Completed!', (0, 50),center=(*True*, *False*),colour=(0, 255, 0), size='MEDLARGE')  *else*:  window.draw\_text('Level Quit. No points will be added', (0, 50),center=(*True*, *False*),colour=(255, 0, 0), size='MEDLARGE')  *if* pygame.time.get\_ticks() - initial\_time >= 2000 *and not* show\_back: *# if back button is not being shown and 2 seconds have passed* show\_back = *True* pygame.display.update() |
| After having the summary statstics displayed, once the user clicks on go back their user data is updated. This is done using the local dictionary that was used to save changes during the session. Once the session is completed its invoked after the show\_summary() procedure finishes executing.  update\_data(question\_data, questions, max\_questions, level, username, points, timer, portal\_enter, total\_accuracy, user\_quit) |
| Using pre-existing functions such as read\_json and write\_json made this process very quick. When updating the data, main information to check was that the user entered the portal and got everything right to be able to register/ update their best time record for that level.  Initially, my Design was that if the user’s clearance time was lower than the previous time, then it would be updated. However, whilst testing, upon clearing a level with a lower time level but worse accuracy would also update the best time when it shouldn’t have. As such, due to two unknowns initially, I decided to have one constant – the accuracy. The user must get everything right (100% accuracy in the session) in order to improve their best time as adding in accuracy would require a multi-variable equation which would lead to complexities outside the scope of this project.  Furthermore, when adding points onto the user data, it’s also checked that they at least attempted a question and that they didin’t quit the level. Quitting the level will save their question progress but won’t add any points which creates a disincentive to quit a level. |
| *def* update\_data(question\_data, questions, max\_questions, level, username, points, timer, portal\_enter, total\_accuracy, user\_quit):  *# update question data and user data when/ if run == False, if they just finished level/died* write\_json(question\_data, f'Questions/{level}.json')  user\_info = read\_json(f'user\_info/users.json')  current\_best = user\_info[username][level]   *# only update the completion time if the user answered all the questions/ defeated all enemies and they made it to the portal  if* portal\_enter *and* total\_accuracy == 100:  *if* current\_best == -1:  current\_best = timer  *else*:  current\_best = *min*(current\_best, timer)   user\_info[username][level] = current\_best *# update time if it was lower  if len*(questions) != max\_questions *and not* user\_quit: user\_info[username]['points'].append(points) *# adding points onto the player's history for graph plotting; if they answered questions* write\_json(user\_info, f'user\_info/users.json') *# save all the changes* |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Game -  Player gameplay, enemy AI, and Points | Show summary statistics | A page which shows how the user performed in the most recent session. | Shows a brief overview of the stats for the sessions | ✔ |
|  |  | Wait a small amount of time before showing continue button | Works as intended | ✔ |
| User Account and Data store | User data stored when player dies | Correct points added, question data updates, no best time updated | Works as intended. | ✔ |
| User data stored when player completes level | Correct points added, question data updates, best time checked before updated | Works as intended | ✔ |
| User data stored when player quits level | No points added, question data updates, no best time updated | Works as intended | ✔ |

## Main Menu

### Version 1: Topic boxes

#### The Development’s Outcome

|  |
| --- |
|  |
| The Main Menu was the last component of the platform that was developed. The first version of consisted of the topic boxes for all the available topics along with the user’s accuracy and best time displayed for each topic. The first row boxes are positioned such that the padding between them is equal. The same applies to the second row too; this creats a neat and tidy look to the meny interface as its automatically done and thus, can be adjusted if need be. Each box also has a message for the best time if the user hasn’t got one already because this issue was brought up by both Dan and Mr Waring who found it difficult to understand why they didn’t get their best times updated despite completeing a level (without a 100% accuracy). |

#### The Development Story

|  |  |
| --- | --- |
| The development process was initiated by creating a function: get\_topic\_boxes() which is used to retrieve the topic boxes based on the algorithm made in the Design section which returns a list. The logic of the algorithm is that for each row a width for the boxes is selected and depending on the width, a suitable padding between the boxes is established. | |
| To explain the logic behind how the formatting of boxes is calculated for a certain row. If the padding is fixed, then the widths of the boxes can be calculated by using the first formula above. Otherwise, if the width is decided to be fixed, the padding can be calculated using the second formula. There cannot exist two unknown variables during the calculation process. When implementing the algorithm, I decided to use the latter; the widths of the boxes were fixed and so the padding was calculated. | |
| The logic above is applied to the box formatting of AutoBox class instances. This is because the AutoBox is a box where the text can adapt to fit the size of the box, unlike the normal Textbox which is used for static texts. In Addition to the formatting, the appropriate text is also displayed for the accuracy and best time according to the user’s current data.  The y padding was a value I determined through trial and error because there are only 2 rows for this version of the platform, hence it would be futile to generalise a formula for y padding as well. | |
| *def* get\_topic\_boxes(username, user\_data) -> *list*:  default\_message = 'Achieve 100% accuracy and enter the portal in the level to unlock'  topics = []  row\_1 = ['Systems Architecture', 'Software and Software development', 'Exchanging Data']  row\_2 = ['Data types, Data structures, and Algorithms',  'Elements of Computational thinking, Problem solving, and programming']   accuracy\_1 = []  *for* topic *in* row\_1:  topic\_number = get\_topic\_number(topic)  question\_data = WINDOW.read\_json(f'Questions/{topic\_number}.json')  accuracy\_1.append(get\_accuracy(question\_data,username))   accuracy\_2 = []  *for* topic *in* row\_2:  topic\_number = get\_topic\_number(topic)  question\_data = WINDOW.read\_json(f'Questions/{topic\_number}.json')  accuracy\_2.append(get\_accuracy(question\_data, username))   *# arrange topic selection* width1 = 430  row\_1\_num = *len*(row\_1)  padding1 = (window.WIDTH - row\_1\_num \* width1) // (row\_1\_num + 1)  *for* i *in range*(3):  current\_time = user\_data[username][get\_topic\_number(row\_1[i])]  *if* current\_time == -1:  current\_time = default\_message  *else*:  current\_time = WINDOW.convert\_time\_format(current\_time)  topics.append(  AutoBox(padding1 \* i + (width1 \* i) + padding1, 200, (width1, 0.4 \* width1), row\_1[i], text=row\_1[i]+f' \\n \\n Accuracy: {accuracy\_1[i]} \\n Best Time: {current\_time}',center\_text=(*False*,*True*),font\_size=22))   width2 = 520  row\_2\_num = *len*(row\_2)  padding2 = (window.WIDTH - row\_2\_num \* width2) // (row\_2\_num + 1)  padding\_y = 200   *for* j *in range*(2):  current\_time = user\_data[username][get\_topic\_number(row\_2[j])]  *if* current\_time == -1:  current\_time = default\_message  *else*:  current\_time = WINDOW.convert\_time\_format(current\_time)  topics.append(AutoBox(padding2 \* j + (width2 \* j) + padding2, topics[1].rect.h + padding\_y + padding1,(width2, 0.35 \* width2), row\_2[j], text=row\_2[j]+f' \\n \\n Accuracy: {accuracy\_2[j]} \\n Best Time: {current\_time}',center\_text=(*False*,*True*),font\_size=22))  *return* topics | |
|  | |
| The get\_accuracy() and get\_topic\_number() functions were just an empty function that returned nothing when the get\_topic\_boxes() function was completed, as these were coded this way to maintain the flow of logic, and I decided to develop those after the get\_topic\_boxes() function was finished. | |
| Hence, once I finished that function, I first created the get\_accuracy() function. This function was initially used to return the accuracy of a topic by calculating the average accuracy across all the questions. Initially, there was a bug where the average considered questions that had never been attempted before which led to a misleading accuracy representation.  This was fixed by adding an attempt when iterating through each question to ensure that there were attempts made before considering the accuracy to the average. If there were attempts made, the accuracy is added to the total accuracy and total attempted questions.  The accuracy is also rounded to 2 decimal places to prevent percentages with recurring decimals taking up unintended space. The return format of this function is a string which adds a percentage symbol (if there is a valid accuracy) or ‘N/A’ if the topic has never been attempted before. | |
| *def* get\_accuracy(question\_data, username) -> *str*:  total\_attempted = 0  accuracy = 0  *for* question *in* question\_data:  stats = question\_data[question][username]  attempts = stats[0] + stats[1]  *if* attempts > 0: *# if attempts were made* total\_attempted += 1  accuracy += stats[2] *# the index that points to the accuracy* accuracy = (accuracy/total\_attempted)\*100   *if* total\_attempted: *# if they've answered at least one question and got it right/wrong* accuracy = *str*(*round*(accuracy, 2))+'%'  *else*:  accuracy = 'N/A'   *return* accuracy  Whilst testing this, topics where no questions were attempted, the total attempted questions were 0, which crashed the program. In order to fix this, when calculating the accuracy an additional check was placed ensuring that the denominator isn’t 0 to prevent a division by 0 error. | |
| *if* total\_attempted > 0: *# making sure denominator is not 0, and that they have attempted questions*  accuracy = (accuracy/total\_attempted)\*100 | |
| Whilst ensuring that the accuracy calculations were correct, there were logic errors that arose. | |
|  | The lists show the [right, wrong, accuracy] for each question in the first topic. The last line shows the total number of right and wrong questions. This was used to check the accuracy. Whilst the accuracy should be 70%, the calculated accuracy displayed was 73%. This likely arose because I used the average of the accuracies.  To fix this I changed how the accuracy was calculated. Instead of using the average of accuracies, I made it so that it now collects the total right and total wrong which is then used to calculate an accurate accuracy without rounding errors which may have occurred previously. |
| The refined accuracy method. This now uses right and wrong values for each question cumulative local variables: right and wrong. | |
| *def* get\_accuracy(question\_data, username) -> *str*:  right = wrong = 0  accuracy = 'N/A' *# default accuracy is non applicable if no questions are attempted  for* question *in* question\_data:  stats = question\_data[question][username]  right += stats[0]  wrong += stats[1]   total\_attempted = right + wrong   *if* total\_attempted > 0: *# making sure denominator is not 0, and that they have attempted questions* accuracy = (right/total\_attempted)\*100   *if* total\_attempted: *# if they've answered at least one question and got it right/wrong* accuracy = *str*(*round*(accuracy, 2))+'%'   *return* accuracy | |
| The only remaining function left to code was the get\_topic\_number() function. This is used to get the topic number for a topic. The function was created because the question files were created using the specification topic numbers to prevent having to use long names.  Using the global dictionary topic\_num, the function invokes the get() method on dictionary for the specific topic. The get() method was used instead of indexing using [] because if a key doesn’t exist, the get() method returns None, whereas indexing would raise a key error. | |
| topic\_num = {  'Systems Architecture':"1.1",  'Software and Software development': "1.2",  'Exchanging Data': "1.3",  'Data types, Data structures, and Algorithms': "1.4",  'Elements of Computational thinking, Problem solving, and programming': "2"  }  *def* get\_topic\_number(topic):  *return* topic\_num.get(topic) | |
| With the core of the topic boxes developed, I moved on to displaying them on the screen. The first step was creating a show\_menu() function. Then, the topic boxes’ list was retrieved by invoking get\_topic\_boxes() outside the main while loop. Using the BoxGroup class created during the Quiz Window, I created a BoxGroup instance outside the main loop here as well, utilising pre-coded sub-routines. | |
| user\_data = WINDOW.read\_json('user\_info/users.json') topics = get\_topic\_boxes(username, user\_data) all\_boxes = BoxGroup(\*topics)  back\_pos = *None* | |
| I then proceeded to create the main while loop which will display the main menu. Within the event capture loop, the mousebutton clicks is used to check if a topic was clicked. As a precautionary measure, using the get\_topic\_number() function for the box’s question is stored in the variable corresponding\_num. After getting the level number, the play\_level() function is invoked in the game\_level file with the username and corresponding\_num passed in so that the function knows which question file to access. | |
| *while True*:  window.refresh(back=*True*, show\_mouse\_pos=*True*, pos=back\_pos)   *for* event *in* pygame.event.get():  *if* event.type == pygame.QUIT:  pygame.quit()  sys.exit()   *if* event.type == pygame.KEYDOWN:  *if* event.key == pygame.K\_ESCAPE:  pygame.quit()  sys.exit()   *if* event.type == pygame.MOUSEBUTTONDOWN:  *# check for button clicks* clicked = all\_boxes.check\_clicks()   *# check for back click  if* window.check\_return():  *return   elif bool*(clicked): *# if something was returned* corresponding\_num = get\_topic\_number(clicked.obj\_type)  *if* corresponding\_num: *# if the clicked box is a topic* game\_level.play\_level(username, corresponding\_num)  *# update user information after a change has been made by finishing a level don't do it constantly*  topics = get\_topic\_boxes(username,user\_data) *# updates the text\_box size and text after user completes a level* all\_boxes = BoxGroup(\*topics, username\_box, leaderboard\_box, instructions\_box)  all\_boxes.update\_boxes(window.screen)  pygame.display.update()  clock.tick(FPS) | |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Platform Access  and Main Menu  **Requirement 2** | Topic box interactivity. | Clicking on a topic box takes the user to the gameplay section with the correct questions loaded in. | Works as intended | ✔ |
| Update user statistics after game level. | Topic boxes should display changed accuracy and/or best times. | When coming back from level gameplay, the accuracy isn’t updated nor is the best time. Despite, the changes being saved in the actual files but not reflected in the menu. | ✘ |
| Leaderboard | Show leaderboard button | Doesn’t exist in version 1 | ✘ |
| Show leaderboard data | Doesn’t exist in version 1 | ✘ |
| User progress | Show username and points button | Doesn’t exist in version 1 | ✘ |
| Show points and sessions graph | Doesn’t exist in version 1 | ✘ |
| Instructions | Show instructions button | Doesn’t exist in version 1 | ✘ |
| Show instructions image | Doesn’t exist in version 1 | ✘ |

### Version 2: Leaderboard and Instructions

#### The Development’s Outcome

|  |  |
| --- | --- |
|  |  |
|  | |
| As shown by the pictures above, this version featured the blitting of the leaderboards, instructions, and username box and upon clicking. Although the username box wasn’t made interactive in this version, clicking on the leaderboards box showing a high scores table of the top 10 users, and clicking on the ‘How To Play’ box shows the instructions. The usernames and points are positioned such that there is sufficient space between them with alignment being maintained, along with the rankings which are numbered to easily identify the current player’s position. In addition to the new boxes, the topic boxes also show the accurate user information upon completion of a game level.  The instructions page was made by taking a screenshot of a demo level to allow a brief overview of what to expect, instead of just text which the user may not remember. This was much appreciated by both my stakeholders as it helped clarify all the key features of the game. | |

#### The Development Story

|  |  |
| --- | --- |
| The development process for this started by addressing the issue of topic boxes not updating their data after a level session was completed. The fix was quite easy for this. I just had to ensure that after returning back from the game level, the contents of the topic boxes were updated. This was done by re-invoking the get\_topic\_boxes() function and recreating the all\_boxes BoxGroup object within the code block where the play\_level() procedure is called. | |
| topics = get\_topic\_boxes(username,user\_data) *# updates the text\_box size and text after user completes a level* all\_boxes = BoxGroup(\*topics, username\_box, leaderboard\_box, instructions\_box) | |
| Next, I moved on to creating the boxes and variables to indicate which page to show. Outside of the main loop, the formatting of the three boxes is calculated. The username box’s width is calculated by rendering a temporary text with ‘username: ‘ followed by 15 ‘W’s which is the largest character by width to calculate the largest width the username box needs to be. Their appropriate variable names: leaderboards, instructions and graph are set to False to signify that none of these are being currently being shown. | |
| rec = topics[1].rect *# the 2nd topic's rect* rec2 = topics[2].rect *# the 3rd topic's rect* username\_width = AutoBox.MEDIUM\_FONT.render(f'username: {"W"\*15} ', 1, (255, 255,255)).get\_width() + 45 *# finding out the maximum width for a username since 'W' is largest width character* username\_text = f'Username: {username} \\n Points: {*sum*(user\_data[username]["points"])}'  leaderboard\_width = AutoBox.MEDIUM\_FONT.render(f'Leaderboards', 1, (255, 255, 255)).get\_width() \* 1.3 *# get a slightly scaled up width of leaderboards font  # position the leaderboards box at the centered x position relative to the 2nd topic box* leaderboard\_pos = (rec.x + (rec.w - leaderboard\_width)//2, 40) leaderboard\_size = (leaderboard\_width, topics[1].rect.h // 2) leaderboard\_box = AutoBox(\*leaderboard\_pos, leaderboard\_size, 'leaderboard', text='Leaderboards', center\_text=(*True*, *True*))  *# space between leaderboard box and username box == space between leaderboard box and instructions box* username\_pos = (rec2.x + (rec2.w - username\_width)//2, 40) username\_size = (username\_width, topics[1].rect.h // 2) username\_box = AutoBox(\*username\_pos, username\_size, 'username', text=username\_text, font\_size=23, center\_text=(*False*, *True*))  instructions\_pos = (leaderboard\_box.rect.left - (username\_box.rect.x - leaderboard\_box.rect.right) - leaderboard\_width, 40) instructions\_size = (leaderboard\_width, topics[1].rect.h // 2) instructions\_box = AutoBox(\*instructions\_pos, instructions\_size, 'instructions', text='How To Play', center\_text=(*True*, *True*))  all\_boxes = BoxGroup(\*topics, leaderboard\_box, username\_box, instructions\_box) leaderboards = *False* graph = *False* instructions = *False* back\_pos = *None* | |
| Within the main loop, the *leaderboards*, *instructions*, and *graph* variables are used to deduce what is to be shown. If none of these variables are True, then the topic selection boxes are shown. If one of these variables is True, then it shows the appropriate information. There is also an additional section of code which updates the user’s points display according to the level gameplay’s events. This was handled pre-emptively as a similar logic error occurred in version 1 with the topic boxes’ accuracies and times not updating, hence I ensured the same mistake didn’t occur in this version too. | |
| *while True*:  window.refresh(back=*True*, show\_mouse\_pos=*True*, pos=back\_pos)  *for* event *in* pygame.event.get():  ...   *if* event.type == pygame.MOUSEBUTTONDOWN:  *# check for button clicks* clicked = all\_boxes.check\_clicks() *# check for back click*  *# clicking back on non-topic screens takes user back to topics screen. If on the topic screen, takes the user back to Platform Access screen.*  *if* window.check\_return():  *if* leaderboards:  leaderboards = *False  elif* graph:  graph = *False  elif* instructions:  instructions = *False    else*:  *return   elif bool*(clicked): *# if something was clicked* corresponding\_num = get\_topic\_number(clicked.obj\_type)  *if* corresponding\_num: *# if the clicked box is a topic*  *# play level code* ...  *# update user information after a change has been made by finishing a level don't do it constantly*  user\_data = WINDOW.read\_json('user\_info/users.json')  username\_box.update\_text(f'Username: {username} \\n Points: {*sum*(user\_data[username]["points"])}')  *# change state of the variables depending on the conditions.*  *elif* clicked.obj\_type == 'leaderboard':  leaderboards = *True   elif* clicked.obj\_type == 'username':  *pass*   *elif* clicked.obj\_type == 'instructions':  instructions = *True   if* leaderboards:  show\_leaderboards()  StopRunning('leaderboards')   *elif* graph:  show\_graph()  StopRunning('graph')  *elif* instructions:  show\_instructions()  StopRunning('instructions')  *else*:  all\_boxes.update\_boxes(window.screen)    pygame.display.update()  clock.tick(FPS)  At that point in time, the procedures in red had yet to be created, but were just empty function blocks with pass statements to ensure the logic of the program worked. This produced a main menu with three more boxes which had clicks and hovering working. | |
| Once the boxes were drawn where intended and were registering clicks successfully as the StopRunning exception was triggered when clicked on the appropriate box. | |
| I then moved on to develop the show\_leaderboards() and show\_instructions() procedures to display the appropriate data. First, I coded the show\_instructions() procedure as this was fairly straight forward. I went into a game level, took a screenshot, and edited it to display the key information. Then I made it so that the procedure loads in an image and displays it on the specified surface at the top left of the screen. There is also a transformation that occurs which scales the image to the fit the size of the surface in case the image size isn’t already to scale. | |
| *def* show\_instructions(surface):  instructions = pygame.image.load('images/instructions.png')  *# scale up the image to fit the window size* instructions = pygame.transform.scale(instructions, surface.SIZE)  *# draw the image on the topleft of the surface* surface.screen.blit(instructions, (0, 0)) | |
| Next, the show\_leaderboards() function was created which again takes in a surface as an object and the user data that is to be displayed. The 2D bubble sort algorithm is used again in this section too. After extracting the required data from user\_data and storing it within a 2D array, the top ten players are selected by extracting all indexes less than 10 (0 to 9) which is stored in a new array: top\_ten. After determining the top ten users, the longest name is used here again, similarly to the when determining the size of the username box, but here it’s used to determine the space between the usernames and the points. After calculating the width of the longest possible name, a starting x and y position were created and the titles: ‘Username’ and ‘Points’ are displayed with an underline using the start\_x coordinate (+ the padding\_x for the Points title), whilst both being at the same y position. | |
| *def* show\_leaderboards(surface, user\_data) -> *None*:  # extract username and points as an array for each user, forming a 2D array  user\_points = [[username, *sum*(user\_data[username]['points'])] *for* username *in* user\_data] *# create a 2D array with [[username,points] for each user]* top\_ten = WINDOW.bubble\_sort2D(user\_points)[:10] *# highest -> lowest of the top 10* font = pygame.font.SysFont('Sans', 30)  longest\_name = font.render('W'\*15, 1, (255, 255, 255)).get\_rect() *# longest name's data* padding\_y = 20  padding\_x = longest\_name.width + 20  start\_x = (window.WIDTH - padding\_x) // 2  start\_y = (window.HEIGHT - (padding\_y \* 6) - longest\_name.height \* 10) // 2 *# position all names to the vertical center  # draw text username and points titles* username = font.render('Username', 1, (255, 255, 255))  points = font.render('Points',1,(255,255,255))  surface.blit(username, (start\_x, 60))  surface.blit(points, (start\_x+padding\_x - 20 ,60))  pygame.draw.line(surface, (255,255,255),(start\_x-13,65+username.get\_height()),(start\_x+padding\_x+points.get\_width()-10,65+username.get\_height())) | |
| I tested out the code above and it produced the following results: the username and points titles where correctly drawn with an underline. | |
|  | With the titles and coordinate determination working, I proceeded to code the for loop that would iterate through each username and points combination in the top 10, render the text and display it onto the screen. Using the for loop’s indexing the start\_x and start\_x + padding remained the same for each username and point. The only coordinate changing was the y position. Each iteration the y value is multiplied by *i* which is the counter in the loop and is used to calculate how many times the y\_padding must be added to draw the current username and points below the previous ones. |
|  | |
| *# draw the actual values for each user for* i *in range*(*len*(top\_ten)):  rendered\_name = font.render(f'{i+1}) {top\_ten[i][0]}', 1, (255, 255, 255))  rendered\_points = font.render(*str*(top\_ten[i][1]), 1, (255, 255, 255))  surface.blit(rendered\_name, (start\_x, (padding\_y \* i) + (longest\_name.height \* i) + start\_y))  surface.blit(rendered\_points, (start\_x + padding\_x, (padding\_y \* i) + (longest\_name.height \* i) + start\_y)) | |
| Upon finishing with that, the sub-routine calls in the main game loop were updated, with the required arguments being passed in, with the exception of the show\_graph() which was yet to be coded. | |
| *if* leaderboards: *# display the leaderboards* show\_leaderboards(window.screen, user\_data)  *elif* graph: *# display the graph* show\_graph()  StopRunning('graph')  *elif* instructions: *# display the instructions* show\_instructions(window) | |

|  |
| --- |
| Whilst testing the different sections, there was a bug that even if the instructions or leaderboards were displayed, clicking a position on the screen where a Topic box was also placed would trigger the level gameplay to occur when it shouldn’t have. To fix this issue, it took longer than expected as debugging error was very hard to identify. However, eventually I came to the conclusion that this must have occurred somewhere in the event capture loop when the mouse button was pressed. |
| It was in that loop where I realised that all\_boxes has the check\_clicks() method invoked and the clicked value will return the position of the box if the mouse button’s click position overlapped even if the topics weren’t being displayed. In order to fix this a compound selection statement was implemented that made sure the user wasn’t in any of the information-based pages. If they were, then clicked is set to False, in which case the selection block of bool(clicked) doesn’t execute. |
| clicked = all\_boxes.check\_clicks() *# don't register non go back clicks if not in the topic selection if* clicked *and* (leaderboards *or* graph *or* instructions):  clicked=*False*  *elif bool*(clicked): *# if a valid click was made*  ... |

#### Testing to inform development

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Strand | Test | Expected Outcome | Actual Outcome | ✔ |
| Platform Access  and Main Menu  **Requirement 2** | Topic box interactivity. | Clicking on a topic box takes the user to the gameplay section with the correct questions loaded in. | Works as intended | ✔ |
| Update user statistics after game level. | Topic boxes should display changed accuracy and/or best times. | Player data now updates accurately | ✔ |
| Leaderboard | Show leaderboard button | Works as intended | ✔ |
| Show leaderboard data | Works as intended. Also added the addition of numbering the order of players to make it easier to identify ranks | ✔ |
| User progress | Show username and points button | Works as intended | ✔ |
| Show points and sessions graph | Doesn’t exist in version 2 | ✘ |
| Instructions | Show instructions button | Works as intended | ✔ |
| Show instructions image | Works as intended | ✔ |

### Version 3: Points Graph and Transition

#### The Development’s Outcome

|  |
| --- |
|  |
| This version featured the implementation of a graph that can be used to monitor user progress across different sessions. |

#### The Development Story

#### Testing to inform development

# Evaluation

## Meeting the Success Criteria

## Limitations

## Maintenance

## Potential Features

- allow teacher to add custom questions for any subject.