A2 Computer Science Project

Contents

Section 1: Analysis

3.1.1 – Problem identification

Introduction to the problem

The problem is that in the past two years, the releases of 2D puzzle games have been very limited. There have been many games released with puzzle aspects, such as puzzle-platformers, but we haven't seen a big release of a pure, 2D puzzle game recently. Despite the fact that both genres are hugely popular and historically successful ('Portal' and 'Minesweeper' being prime examples), there is a huge lack of recent 2D puzzle games in the market today. I searched on Steam with the tags "2D" and "Puzzle", to see the release dates of the popular 2D puzzle, most relevant games on Steam today. I found that the five listed releases are as follows:

1) Gems of War – Puzzle RPG (released 20/11/2014, free to play)
2) Puzzle Pirates (released 31/08/2011, free to play)
3) Ori and the Will of the Wisps (released 11/03/2020, £24.99)
4) Papers, Please (released 08/08/2013, £6.99)
5) There Is No Game: Wrong Dimension (06/08/2020, £10.29)

I looked at all these games and realised that, immediately, three of these games have a release date that is before 2015. 'Gems of War – Puzzle RPG', 'Puzzle Pirates' and 'Papers, Please' have release dates that can be considered non-recent, as they fall outside of a five year bracket. 'Ori and the Will of Wisps' and 'There Is No Game: Wrong Dimension' were indeed released this year, but are not pure puzzle games – rather, they incorporate puzzle game aspects as part of the game. For instance, 'Ori and the Will of the Wisps' is significantly platforming-based, with the player traversing a huge 2D environment in a side-scrolling world. 'There Is No Game: Wrong Dimension' involves many genres of videogame, such as open-world, detective and many more. Hence, these games cannot be classified as "pure 2D puzzle games", as the puzzle game is not the main feature of the videogame. This further reinforces my point on the lack of 2D puzzle games released, and justifies my problem.

Introduction to the stakeholders

My target audience for my end user group are mainly teenagers, specifically aged from 14 to 18. This is because teenagers typically play video games a lot, with statista.com stating that Americans aged 15 to 19 years old spend "49 minutes on gaming or leisurely computer use during an average weekday," and "more than 90 minutes doing so during weekends and holidays." This shows very clearly that videogames are an integral part of the average teenagers' life.

In terms of gender, my game will be aimed at either gender. Puzzle games as a genre are aimed at both males and females; both genders can play and enjoy the challenges that the games bring, there are no real aspects that appeal to one sex more than the other. I have chosen to make my target audience gender-neutral in order to broaden my potential market, rather than limiting myself to one gender.

For this project, I have identified a group of five people who accurately represent my target audience through their diversity as a group, consisting of males and females, people that are younger and older on the limited given age spectrum (14 to 18). I shall be making use of iterative development in order to fine-tune prototypes of the final product so I can integrate my stakeholders very well into an iterative development process, in order to ensure the final product meets their needs best. Their

feedback will be used for the development of any subsequent iteration(s). All of these individuals are gamers, and enjoy videogames as a frequent part of their lives. Their first names, ages, genders and a brief description into their gaming experience is below:

- [redacted] (14, Male)

[redacted] is a young gamer who enjoys playing more relaxing video games. He likes games that are mildly challenging, such as “Quell”, but don’t require huge amounts of effort or intensity to play. He prefers less competitive, more mentally stimulating games.

- [redacted] (16, Female)

[redacted] enjoys games like “Candy Crush”, which involve solving problems and making conscious decisions to reach a successful outcome.

- [redacted] (17, Female)

[redacted] is a gamer who enjoys more challenging videogames as they make her think more. An example of this is “Candy Crush”, especially heading towards harder levels as it makes her think more strategically about her next move, and she loves the challenge of completing a difficult task.

- [redacted] (17, Male)

[redacted] enjoys games that mentally challenge him and help improve his mental sharpness, as well as being fun at the same time. Usually these are math, word or puzzle games.

- [redacted] (18, Male)

[redacted] plays games such as “Into The Breach”, a real time strategy game which involves thinking several moves ahead in order to beat the opponent. He enjoys games which involve a lot of strategic thinking.

Looking at the history of all of my stakeholders, I can confirm that they are suitable to talk to for feedback and testing on my project. They have enough knowledge and appreciation of videogames, and feature preferences that make them perfect for 2D puzzle games, such as problem solving and strategic thinking. Furthermore, outside of gaming, all of these people have basic IT and cognitive skills. All are more than capable of using a basic computer. Because of this I know that all of them are fully able to play and comprehend the game itself.

They will make use of the proposed solution since they are all gamers who are interested in the genre of puzzle games, and I have inquired further to confirm this. They will be testing and evaluating my product, providing feedback to refine further iterations. They are part of the final target audience, and will be able to play the game if it is (hypothetically) released to the public. It is appropriate to their needs because it solves the issue of the lack of releases of fully puzzle-based 2D games, and fills the gap in the market for such a product.

I inquired further into their opinion on the problem, to see how they felt about the given issue with the lack of 2D puzzle game releases in the past five years. I all my stakeholders if they felt as though there had been a lack of puzzle game releases lately, and if they could name a recently released puzzle game.

[redacted]: “I can’t name a puzzle game released very recently, although I know Candy Crush is a puzzle game that is still popular. I definitely think there hasn’t been enough puzzle games released recently.”

[redacted]: “I think 2048 was released in 2018… I’m not too sure though. I do think that there is a space in the market for more 2D puzzle games.”

[redacted]: “I definitely think there is a huge room in the market for more 2D puzzle games. All the games I’ve played recently that have been advertised as 2D puzzle games are more of other genres, with puzzle game elements.”

[redacted]: “I really can’t name any recent 2D puzzle game releases, especially not for PC. I think there are a few released on mobile, but these aren’t very good.”

[redacted]: “I play a lot of old 2D puzzle games, like Minesweeper, but I haven’t got a clue about recent releases. I think there is a big space for one in the market.”

([redacted] was wrong. ‘2048’ was released in 2014.) From this, I can see very clearly that my stakeholders believe that my problem is both viable and real. They all acknowledge the severe lack of recent releases, and my solution will solve this problem, since it will be a 2D puzzle game. The stakeholders will make use of this proposed solution through iterative testing, and providing feedback on what can be improved for subsequent iterations (as I previously mentioned.) Furthermore, since my stakeholders are all members of the target audience, and will be able to play the game for their own leisure and enjoyment after the final version has been released.

3.1.1a – The problem area

I need to investigate my problem area, and look at 2D puzzle games. From this, I need to identify the essential fundamental features of a 2D puzzle game, explaining and justifying each of their purposes. To do this, I will look at previous 2D games, and take features from these examples, choosing only the essential features.

I researched online as to what makes a good 2D game. I looked at various forums and websites.

- The most common answer was “working, engaging gameplay.” This basically means that a game has to work properly, and has to actively make the user think and do something. This means that it will need to take inputs from the user frequently.
- According to a forum from construct.net, most users prioritised gameplay over any other factor in a video game. One user stated “Amazing graphics are nice to have (…) but not at the cost of gameplay.” This means that even though graphics and the general appearance and aesthetic of a game hold some importance, the overall gameplay and experience is considered the most engaging and important factor when creating a 2D game.
- A blog article on pluralsight.com echoed this idea, talking about how “A hit game can be built entirely on gameplay without any story.” This website referenced ‘Angry Birds’, stating that it “doesn’t focus on a compelling story”, instead having “incredible gameplay”, making it so popular and successful.
- The article later defined good gameplay as having “clear controls and good feedback to the player.”
- According an article on serc.carleton.edu: “A good game designer gives his players continuous challenges, each of which leads to another challenge, to keep them “hooked on playing a game.” This links to the idea that gameplay and the actual experience of a game is

the most important factor and the idea that the gameplay itself needs to be engaging and actively involving of the user.
- This brings about the idea that players need to be able to understand their objective or purpose within the game. Looking at different classically renowned games, almost all of them feature some clear form of progression or levels, or something to indicate the player's progress or objective.

From this, I can identify that clear controls are an essential feature of my solution. In order to have "clear controls", I will need to choose a method of input that is conventional and accessible to my users. This means I will need to carry out further research on what are the best methods of input, and which are most suited to my users. I will also interview stakeholders to identify this.

Secondly, I need to ensure that my main gameplay works. To do this, I need to have inputs that produce a consistent output every time – for instance, if a user presses the "UP" arrow key, the player should always move up. It's no use if the game produced a different output for the same inputs every time, since the game would be unfair and completely unenjoyable. The user would have no way of knowing what would happen every time they pressed a button.

I will need to have some way of telling the player their progress/difficulty, giving them an idea of where they are and where they stand with the game. This is needed to help motivate the player and allow them to feel a sense of progression, keeping them 'hooked' to the game.

Furthermore, in order to allow the player to progress I will need to ensure that the puzzle in the game is always solvable. The reasoning behind this is that if the puzzle is unsolvable, the game will be unplayable at times – the player won't be able to beat certain levels, they won't be able to progress past certain points, and gameplay will quickly stagnate and their enjoyment will be ruined.

Since my solution is a game, I will also need to save user data, either temporarily or permanently (should I choose to add a leader board function). Temporary saving of data is needed in order to keep track of the player: their location, their progression, the current state of the game and their status (such as health) are all pieces of data that need to be retained during the duration of the gameplay.

An essential feature of almost all games is the loading of objects and data, as well as displaying them on screen. These are essential features because, without them, the game is pretty much non-existent. A UI (user interface) is also another essential feature for ANY system, since it allows a user to interact with the system.

Lastly, specific to the puzzle game genre: I will need some way of randomly generating a puzzle that is solvable (and hence not impossible.) The reasoning behind this has been discussed above.

On the flip side, I also need to know what I should avoid in my game. I researched what constitutes poor game design and what makes a "bad videogame". I read a blog on Windows called "The 7 sins of bad game design".

- Poor controls in a videogame immediately disconnect the player from the experience. It makes it difficult for them to play the game with satisfaction, and may render the game completely unplayable.
- The blog also talked about needing a clear objective in the game. I need to communicate clearly to the user what they must do to solve the puzzle.

- Repetitive gameplay ruins the user experience because it takes away the excitement from the game.

From this, I need to make sure controls are accessible, and will likely use controls that are universally recognised. This has already been discussed previously.

However, I haven't got any way of telling the user what their objective is. To do this I must include an information/tutorial screen telling the player what to do, so that they know what they must do to win the game.

I must somehow make sure the gameplay is not repetitive. This could be done by randomly generating puzzles every time, so that every puzzle is different to the last. This will keep the game fresh, exciting and challenging for the users.

Collated below, are the essential features of the solution:

Essential features	**Justification**
Methods of input and the machine interpreting these inputs	The user needs to be able to control the system through inputs, allowing them to play the game.
Functional gameplay that makes use of consistent inputs	Inputs need to be consistent to allow the user to properly understand what the click/press in order to provide a certain result.
Some way of telling the player their difficulty/level/progression	The user will have to know where they are in the game in order to know what they have to do next in order to progress. They also need to be able to understand what difficulty level they are playing at in order to customise the challenge for themselves (should I choose to include a difficulty selection screen).
Storage of data/user data	Data will need to be constantly stored and updated, either temporarily or permanently. This will allow gameplay to take place and user data/progression to be saved.
Generation of elements on screen/sound elements	Generation of UI/gameplay elements is essential because gameplay is entirely visual-based, and objects need to appear and interact on screen. This also includes detection of collision, which is needed to trigger certain in-game events, such as death.
The use of a user interface to allow the user to interact with the system	The user needs to be able to interact with the system to select, play and understand the game. The use of a user interface will enable the user to carry these out.
Generation of a solvable puzzle	The puzzle needs to be unique and solvable every time to provide a fair but challenging experience to the user. If the puzzle is not solvable, the game is pointless.
Inclusion of a tutorial screen	The player needs to have a clear objective and understanding as to how to solve the puzzle. If not, gameplay will be difficult to understand and not enjoyable.

Non-repetitive gameplay	Repetitive gameplay will quickly bore a user and not be a challenge.

Visuals, colours and other important factors

As part of visuals, the colour and appearance of a game is very important. Certain colours are assigned certain attributes in the human mind – this is part of colour psychology. For example, the colour red is often associated with anger, violence, power and danger, all of which are more dominant and powerful emotions. As a result, for a threatening character/object in my game, I can use the colour red as the primary colour for that object. This will influence the user to understand and recognise the threat, and make manoeuvres to escape/eliminate the perceived threat. I asked my stakeholders what they associate with the following colours.

	Red	Orange	Yellow	Green	Blue	Indigo	Violet	Black	White
[redacted]	Anger	Warm	Sunlight	Nature	Water	Royalty	Feminine	Darkness	Clean
[redacted]	Danger	Warm	Positivity	Nature	Sad	Winter	Comfort	Empty	Purity
[redacted]	Anger	Neutral	Attention	Positive	Calm	Authority	Serenity	Unknown	Known
[redacted]	Anger	Neutral	Happy	Strength	Peace	Neutral	Neutral	Darkness	Light
[redacted]	Anger	Happy	Happy	Happy	Calm	Tired	Tired	Peace	Peace

From this table, I can deduce that…

- The best colour to denote a threat entity/object would be red. Red has consistently shown to be a colour that represents threats, danger and generally offensive/aggressive concepts.
- For the entity that the player controls, I have decided that the best colour would be green. This is because green is a colour that has been consistently attributed to positive qualities and concepts, such as nature, happiness and strength.
- The best colour for the background would most likely be black or white. This is because black and white won't distract from the main game. I could've used an image for the background, but I ultimately decided against it because it is unnecessarily complex and goes against the idea of abstracting away unneeded details.

I also must consider the actual resolution of my game. According to a blog on hobo-web.co.uk, from a sample of 451'027 users…

- 19.53% of users use 1920×1080 as their screen resolution.
- 15.07% use 1366×768.
- 9.65% use 1440×900.
- 7.26% use 1536×864.

From this, I can see that 51.51% of the sample falls in the four shown resolutions.

- These resolutions have aspect ratios of 16:9, 683:384, 8:5 and 16:9 (again).
- On average, they have an aspect ratio of 1.733550347 (width to height).
- I can identify that all these resolutions have a minimum of 1000 pixels in width and 700 pixels in height.

From this, I have chosen that the resolution of the game screen will be 1000×700. This resolution maintains a 10:7 aspect ratio, which is 0.3049789187 away from our previously calculated average. It maintains a window that is wider than it is tall, and as a result, will fit onto virtually any generic monitor.

For the puzzle game, I only want the user to be able to move up, down, left and right. This eliminates the need for omni-directional controls, such as the mouse or an external controller, since only four directions of movement are needed. Furthermore, programming for an external controller is very difficult and time consuming, and will not yield better results than simply using a keyboard.

I researched what the actual controls should be for the keyboard, and found that the two most popular configurations for 4/8 directional keyboard controls are WASD and the arrow keys. I asked my stakeholders which controls they prefer. The results are below:

- [redacted] prefers WASD.
- [redacted] prefers the arrow keys.
- [redacted] prefers the arrow keys.
- [redacted] prefers WASD.
- [redacted] is indifferent to either; he said that both are easy to use and accessible.

Since there is a clear 50/50 split in these answers, I have decided that I will allow users to choose their input method to be either WASD or the arrow keys. Both are universal keyboard controls, and cater well to my entire target audience, hence why I have made this decision.

Features of our solution

Here, I have collected and compiled a list of the elementary and basic features of the solution. The next section will describe how these features make the solution suitable to a computational solution.

- 2D gameplay
- Methods of input and the machine interpreting these inputs
- Functional gameplay that uses consistent inputs
- Telling the player their progression/difficulty
- Storage of data
- Generation of elements (visual or auditory)
- The use of a UI
- Generation of a solvable puzzle
- Tutorial screen
- Non-repetitive puzzles
- Red to represent enemies; green to represent the player
- WASD/Arrow keys as inputs

3.1.1b – A computational approach

Problem recognition

As aforementioned, the overall problem is developing a 2D puzzle-based game for an audience of both genders, from the ages 14 to 18 years old. The game itself is supposed to be a video-game that is played on a computer, since this aligns with what the stakeholders expect. It shall also be designed and created on a computer. The type of puzzle game that I shall choose is currently unknown, as the

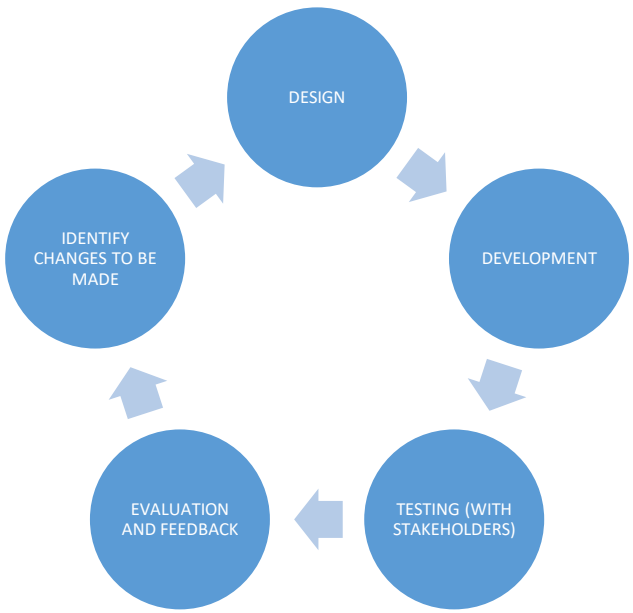
puzzle game genre itself is built up of very diverse sub-genres. This shall be decided later on in the analysis section.

Firstly, the most basic question: why am I using a computer to solve my problem?

- It allows for development and implementation of instructions must faster than any other medium available right now, and computers allow for generation of different puzzles, every time. A physical puzzle game only has one or few real solutions; a computer can generate theoretically infinitely many puzzles, within certain bounds.
- My target group of ages 14-18, are increasingly interested in playing games on machines rather than a physical form. For instance, chess, mah-jong, solitaire and many other games that are traditionally played in a physical form, are now available virtually, and are wildly popular as a result, even today.
- It is much easier to design, implement, test and refine on a computer rather than a physical counterpart.
- Looking at one of my essential features, the need for inputs (WASD/arrow keys) cannot be done without using a computer in conjunction with a keyboard. The use of four directional controls only exists on computer systems.
- The use of a computer system allows for the rapid storage and retrieval of data compared to any manual solution. A computer program can also hold variables, and modify these variables at any time, making it amenable to my game and its essential feature of data handling.
- The use of a computer system allows dynamic visual elements to be easily implemented, with ease of implementing colours and visual objects using a programming language, allowing me to select what visual elements are coloured.
- With the use of variables in conjunction with visual elements, I can also easily tell the player their current level/difficulty as they progress through the game.

<u>Problem decomposition</u>

This problem can be broken down into smaller steps, as I develop the program:



Because two of my essential features are the use of a UI and the use of a tutorial screen, I can split the development of different parts of the game (e.g. the main menu, the tutorial the UI, the gameplay, the ending screen) into different parts, and develop them as such, rather than trying to do it all at once. This will help maximise efficiency, make the development faster, and avoid any potential confusion that may lead to mistakes and bugs.

I can also decompose the system into steps. Here is a concept of what these steps might look like:

1) Load puzzle data/visual data from assets directory
2) Generate puzzle as appropriate (visual, for user to see)
3) Enable keystroke detection and input
4) Allow player to move and interact with the puzzle (denying them any moves that violate rules)
5) Detect when the puzzle is solved/detect a game over
6) Save score to appropriate location

The first three steps are solely the computer's job and do not require any user interaction or input in order to work/function. They can be executed fairly quickly; the assets directory will be included as part of the program file, for rapid loading times. Steps 4 and 5 are dependent on the user, as this is the actual gameplay and where the puzzle is put into action. The game will be constantly waiting for user input and will execute it as it comes in, without any delay. The storage of data is covered in step 6, tackling another essential feature.

Divide and conquer

During development, I can split the game up into different parts to make it easier and more efficient to develop.

- I can design the UI (user interface) separately to the puzzle algorithm, and then implement together, rather than trying to design the whole program at once.
- It's easier to modify code when moving to the next iteration if it is separated into its different components and parts, rather than all blocked together as one huge chunk of code, as well as being easier to read in the future.

Abstraction

One of the essential features of my game is that it is 2D. 2D is abstracted from 3D, in the sense that an entire dimension is 'filtered out'. I have done this because:

- 3D puzzle will likely provide the user with unnecessary information that they do not need, and could potentially confuse them, as well as taking longer to develop.
- Puzzles in 2D are easier to develop, and can convey their information much easier than their 3D counterparts, hence making them more suited for the user. Any information that the user does not need to see (for example, variables stating the x and y coordinates of objects on the screen) will be hidden away from the user, in order to avoid any kind of confusion or unwanted user interaction with the hidden code.
- The program will be controlled via user input from the keyboard and mouse, which is a form of abstraction, in the sense that the user will not need to enter raw inputs, but rather to press designated keys and buttons, making the game easier to play and more suited for its audience, as well as helping fulfil the previously discussed idea of having 'clear, understandable controls and mechanics' in the section 'What makes a good game?'.

- Graphically, it puts less stress on a user's system, and as a result lowers system requirements for the game to run. A 3D puzzle game, rich with (unnecessary) details and a high framerate would likely require a reasonably advanced GPU, which can cost the user a lot of money.
- 3D does not make any actual improvements to the overall gameplay or experience of the puzzle game.
- Users do not often play puzzle games for dazzling graphics and textures, as shown in prior secondary research, but rather focus on 'working, engaging gameplay'.

By recognising abstraction to 2D from 3D, my problem can be solved more easily. 3D is not necessary for the quality of a game to be improved, particularly with a puzzle game, and I can render designs more easily and succinctly in 2D, in comparison to 3D. Hence, I have made use of abstraction as a computational method to help solve my problem.

Thinking procedurally

One of my essential features is the generation of visual and auditory elements; the other looks at the generation of a solvable puzzle that is non-repeating. These are all amenable to procedural thinking.

- I can have certain procedures that, when called, draw elements and sprites to the screen. Different procedures may draw different objects; for instance, there may be a procedure for drawing the background and puzzle, and a separate procedure for character sprites/objects.
- Certain procedures can be triggered by in-game events, such as collisions, and can run to trigger changes in the system, such as the press of a button moving the tutorial into the main game.
- I can use procedural generation to generate a solvable puzzle, as certain algorithms can be used to make sure that the puzzle generated is always one that is solvable, while being unique every time by randomising certain parameters.
- Certain methods/procedures can be called to allow movement when certain inputs are detected. For instance, if a player presses the UP arrow, a method can be called to move the on-screen player object up.
- By ensuring the same procedure is called every time a particular input is given, I can ensure gameplay always uses consistent inputs, making gameplay playable for a user.

Algorithms, program logic, branches and loops

- I can use particular algorithms to allow the generation of a solvable puzzle – I know heuristic methods will not always give a correct generation in a reasonable amount of time.
- I may also need to use algorithms to manage data, such as sorting arrays of data into a particular order.
- My problem is also suited to the use of branches and loops, since I can make use of loops in algorithm implementation or a game loop to refresh the screen.
- My solution will also feature a lot of decision-making, and hence will need to make use of conditional "if", "else" and "elif" statements.

3.1.2 – Stakeholders

3.1.2a – The stakeholders

Stakeholder:	Description:	Interest:
[redacted]	Male, 14. A more laidback player, who enjoys slower-	Wants a puzzle game that allows him to play at his own

	paced puzzle games.	pace.
[redacted]	Female, 16. An amateur gamer who enjoys casual gaming, particularly on her laptop.	Wants a puzzle game that can be played quickly and on-the-go.
[redacted]	Female, 17. An occasional gamer who likes visually impressive games.	Wants a puzzle game with a variety of colours and interesting designs.
[redacted]	Male, 17. A fairly serious gamer who enjoys challenges.	Wants a puzzle game which is challenging and addicting.
[redacted]	Male, 18. A very seasoned gamer who enjoys solving highly complex problems.	Wants a new, challenging puzzle game, with a high level of difficulty.

These stakeholders have been selected because they are all examples of the final consumer, while fulfilling a variety of personalities within the target audience.

Because I have a range of personalities, I am able to consult the group for advice and guidance on how I may improve the product, and am guaranteed to get a good response. For example, [redacted] can provide strong aid on how the game should look visually, while [redacted] and [redacted] can help when it comes to designing and implementing the levels of difficulty.

3.1.3 – Research the problem

3.1.3a – Researching the problem and potential solutions

Interview questions

I also must do my own research on the problem, talking to my stakeholders. I am going to make use of Google Forms, to create a survey to ask them 9 questions, as I cannot interview them in person due to restrictions imposed by the COVID-19 outbreak. The following questions shall be presented to them, in order to help identify requirements for the project.

1) What is your name?
2) Have you played any puzzle games before? If so, can you name them?
3) What was good about these games?
4) What could have been improved?
5) What, in your opinion, would appeal to you in a puzzle game?
6) What puzzle games that you have previously played would you say were ‘good’? What factors contributed to this?
7) What puzzle games that you have previously played would you say were ‘bad? What factors contributed to this?
8) What features do you think should be in a puzzle game?
9) Anything else you want to add?

I have selected these questions based on what I think I need to know from gamers themselves, rather than what articles online have told me from secondary research. The responses of all the users were recorded with their permissions.

Interview answers

Here I have collated answers from every interview.

[redacted] (16, female):

1) What is your name?

[redacted].

2) Have you played any puzzle games before? If so, can you name them?

Tetris, Candy Crush, Mah-jong.

3) What was good about these games?

They get your brain thinking and I like how they are all about logical thinking.

4) What could have been improved?

After a while of playing to start to pick up trends that make the game easier or slightly irritation for example you may notice that hints don't actually provide you the best next move.

5) What, in your opinion, would appeal to you in a puzzle game?

A game that is all based on thinking about logical moves and best strategies.

6) What puzzle games that you have previously played would you say were 'good'? What factors contributed to this?

Online Solitaire is a good example as it is all based on not simply making a move but the right one that would benefit you later on in the game and it gets you thinking about all options available while still taking your time to see how quickly you can do so.

7) What puzzle games that you have previously played would you say were 'bad? What factors contributed to this?

I haven't played any explicitly bad ones however some flaws I noticed in the ones I've played have been that at times things like hints are not as useful as they make themselves to be and it actually tends to be not the more smarter and logical answer. Furthermore once you get to harder levels games tend to get repetitive and easy.

8) What features do you think should be in a puzzle game?

I think all puzzle games should have a time element whether it is a time limit or a record of how long you took to complete the puzzle.

9) Anything else you want to add?

That's all, I don't really have anything to add.

[redacted] (14, male):

1) What is your name?

[redacted]

2) Have you played any puzzle games before? If so, can you name them?

Candy Crush, Minesweeper, Portal 2, Hitman GO, 2048, Monument Valley 2

3) What was good about these games?

Hitman GO had a fun idea and way to play, amazing graphics and soundtrack. Portal 2 allowed you to play co-op with your friends. Candy Crush is simple and addictive.

4) What could have been improved?

2048 is very frustrating and I haven't beat it yet. Monument Valley 2's controls were terrible.

5) What, in your opinion, would appeal to you in a puzzle game?

In puzzle games, co-operative modes are very appealing to me because they let me play with friends. Simplicity is also very important.

6) What puzzle games that you have previously played would you say were 'good'? What factors contributed to this?

As previously stated, I really enjoyed Portal 2 because it let me play with my friends.

7) What puzzle games that you have previously played would you say were 'bad? What factors contributed to this?

Monument Valley 2 had controls so bad they rendered the game nearly unplayable in my opinion.

8) What features do you think should be in a puzzle game?

Simple controls, a good soundtrack/music.

9) Anything else you want to add?

Gameplay is also really important. Co-operative modes, while nice are not essential.

[redacted] (18, male):

1) What is your name?

[redacted] [redacted].

2) Have you played any puzzle games before? If so, can you name them?

Yes, I've played Welltris, Keep Talking and Nobody Explodes, and Quell.

3) What was good about these games?

In Quell, the relaxing atmosphere paired with simple objectives made for a great little game. The touch/swipe controls were well suited for the mobile devices that Quell was released on, and the goal was clear & easy to understand.

In Keep Talking and Nobody Explodes, the gameplay was challenging and unorthodox, and relied on clear communication between players. This created a unique experience, which was very fun and entertaining for all involved.

4) What could have been improved?

In Welltris, the controls were unclear, as was the objective. This made for a confusing game, as there was no explanation as to what you needed to do, you had to figure it out yourself. The game itself wasn't bad, on the contrary it was quite fun, however the initial barrier meant that the game was difficult to get into.

5) What, in your opinion, would appeal to you in a puzzle game?

A clear, obvious and simple objective, with a catch that makes completing the objective a challenge.

6) What puzzle games that you have previously played would you say were ‘good’? What factors contributed to this?

Hitman GO is another puzzle game I enjoyed. The reason for this was because of the thought process that went into choosing which piece to move, where and when - you had to plan several steps ahead and predict what the enemy might do in order to successfully eliminate your target.

7) What puzzle games that you have previously played would you say were ‘bad? What factors contributed to this?

Infinity Loop was a very addicting and fun game to begin with. However, it suffered from repetitive gameplay. As the game went on, levels would become very repetitive and the game lost its edge as the lack of new mechanics/challenges made playing the game more of a chore.

8) What features do you think should be in a puzzle game?

Introduction of mechanics that make the player feel as though they understand the game better than they actually do.

9) Anything else you want to add?

Nothing else. I look forward to seeing your result!

[redacted] (16, female)

1) What is your name?

[redacted]

2) Have you played any puzzle games before? If so, can you name them?

Yes, I have. I've played Tetris, minesweeper, Candy crush and cut the rope.

3) What was good about these games?

Games like candy crush and cut the rope not only get increasingly difficult as you play them, like Tetris, they also have mini rewards after a certain number of levels, which encourages the player to continue playing the game, even though certain levels are very difficult to beat.

4) What could have been improved?

5) Games like minesweeper are at set difficulty, no matter how long you play them for, which can lead to the game getting boring over time.

6) What, in your opinion, would appeal to you in a puzzle game?

Games that push the player by increasing difficulty, this prevents the game from getting boring and it feels rewarding to overcome a difficult level.

7) What puzzle games that you have previously played would you say were ‘good’? What factors contributed to this?

I would say that Candy Crush is a good game as it not only increases difficulty and has a small reward system (of like , it also has events that aren't related to the main game), which keeps the game interesting as well.

8) What puzzle games that you have previously played would you say were ‘bad? What factors contributed to this?

Minesweeper would be one, as it doesn’t get harder over time, so it starts getting boring.

9) What features do you think should be in a puzzle game?

A consistent challenge and rewards especially after more difficult levels.

10) Anything else you want to add?

I haven't got anything to add.

[redacted] (16, male)

1) What is your name?

[redacted] [redacted]

2) Have you played any puzzle games before? If so, can you name them?

I enjoy trying a crossword on paper and on the internet every so often. I am interested in any number or word games that challenge me, my favourite is probably solitaire.

3) What was good about these games?

I like to be challenged with problems that require me to take time to figure out an answer. I enjoy the feeling of knowing that I am improving my cognitive skills for other areas of life while still having fun. For instance word games can improve reading ability in those who are younger.

4) What could have been improved?

Some of the games have an interface that is clunky and slow meaning that it is sometimes harder to focus on the puzzle. I believe that a more minimalistic approach to the design of a puzzle will allow me to do better as I can focus more on the question, be more likely to answer it correctly and as a consequence be more fulfilled.

5) What, in your opinion, would appeal to you in a puzzle game?

Any type of puzzle that mixes a simple puzzle that may have already been thought of with a new and fresher aspect that can help me to enjoy the game more. For example, playing a game like Tetris, but you have to answer mathematics questions to slow down the game.

6) What puzzle games that you have previously played would you say were ‘good’? What factors contributed to this?

I feel that "good" games are those that are simple, they may entertain you for a small amount of time but eventually become repetitive and therefore aren’t as engaging as other puzzles in a similar subject. For example a word search while being a puzzle cannot be classed as extremely interesting as there is little variety in game play.

7) What puzzle games that you have previously played would you say were ‘bad? What factors contributed to this?

A "bad" puzzle game for me is one that doesn’t contribute to cognitive exercise in any way and is purely luck based or not interesting at all. There can be partial elements of luck but if a puzzle is not

inherently "mentally challenging" then it cannot be classed as a puzzle. For example I would class "snap" the card game as bad.

8) What features do you think should be in a puzzle game?

For me the most important features to include in a puzzle game are multiple aspects that challenge both the numerical and literary cognitive function. For instance a game like scrabble (which may not be classed as a puzzle) but forces you to think about word placement for a numerical score but also test a vast breadth in vocabulary.

9) Anything else you want to add?

These questions are good and engaging but if you want a large amount of answers that you can easily find a trend from, make the questions more simple and still open but maybe more guided as to what you want them to say.

Analysis of answers

From this information, we can produce the following…

The puzzle game **MUST**:

- Provide the user with a challenge
- Not become repetitive
- Not be based on purely on luck – must have some user input that determines the outcome

The puzzle game **SHOULD**:

- Evolve in difficulty in some way
- Feature some level of audio/sound design

The puzzle game **COULD**:

- Feature a cooperative mode, allowing users to play alongside their friends

From these, I can derive certain parts of my success criteria. I can also raise the following points:

- The main issues that I must tackle are the ability to provide the user with a challenge; for this, I must choose a puzzle game that can provide an actual viable challenge.
- I think that allowing for the evolution of difficulty is a good idea, however, this does not necessarily have to be in the form of evolution as the user plays, continuously and infinitely (like Tetris). Instead, I think that the use of either difficulty selection or a levelling system is a much better idea. I have made this decision because implementing an infinite puzzle which gets progressively more difficult is much harder than a simple levelling system, and has more room for error.
- The use of levels/difficulty selection is a much safer option with reduced risk, and allows for users to tailor their experience around their own choices and skill level. Hence, I will include either a levelling system or difficulty selection.
- I think that the inclusion of sound design is fairly important, but I'm not sure if it's fully beneficial. I will decide if this is a justified decision later in this section, after enquiring further with the stakeholders.

- I do not think that implementing a cooperative mode is a valid, feasible idea. This is a limitation, specifically to do with how it is more difficult to use networking to allow for cooperative play, and the time and resources this will take, as well as the impact on the overall project.

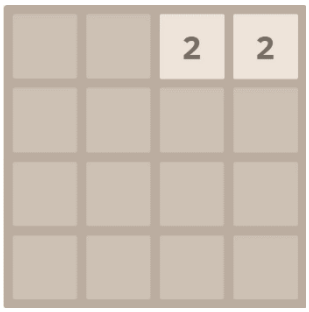
Analysis of existing solutions

I also need to take a look at existing solutions myself. This will involve me looking at different 2D puzzle games and analyse the features of them that are good, and hence could potentially be implemented in my own game, and features that are undesirable, and hence should be avoided as much as possible.

Furthermore, analysis of a wide range of existing puzzle game solutions can allow me to personally explore different sub-genres of puzzle game, and justify a choice as to which sub-genre I shall build my game around.

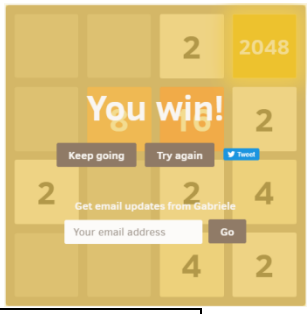
2048

The first game that I shall analyse is called '2048'.



- It was developed by Gabriele Cirulli and released on 9th March 2014.
- It is a single-player puzzle 2D videogame, which has become extremely popular, with versions being released for Microsoft Windows, Android, iOS and Nintendo 3DS.
- The objective of 2048 is to match blocks in a four-by-four grid, all of which have number values on them. Initially, two blocks spawn with either 2's or 4's on them, as shown below in fig 1.1.2:
- 2 and 4 can spawn anywhere in the four-by-four grid. As you match identical blocks, they increase in value, doubling every time.
- The game technically goes on infinitely, but the primary objective is to obtain a block with the number 2048 on it. Afterwards, there is an option to continue, until there are no moves left.

Once the user reaches a 2048 block, they are presented with the winning screen and the option to continue, as shown on the bottom right:



Advantages	Disadvantages
It is highly addictive, and features a simple concept and objective; move and match blocks, try and obtain 2048.	Remarkably difficult to beat using ad-hoc methods; this was very frustrating.
The design is very aesthetically pleasing, it's symmetrical and features a soft, warm colour	Playing with no real strategy other than attempting to match would result in very, very

scheme, with a font that isn't sharp or over-the-top, but rather gentle and simple.	short games.
The higher value blocks feature a soft 'glow', giving the user a sense of achievement and making them feel as though they're actively progressing.	Even with the use of structured methods and strategy, I found that I only ever won the game twice.
Random blocks spawning into free spaces means there is a level of strategy to the game; you try to limit where they can spawn to your advantage.	The results can become very repetitive, with the same outcome occurring several times due to its huge difficulty.
Random spawning of blocks also means the user has to constantly adapt their strategy to suit the board at hand, making the game less predictable and more exciting.	There isn't any real solution to make the game easier.

From this, I can take the following things that I could take from the game:

- The simplistic, 2D aesthetic
- Random generation of blocks
- The ability for the game to continue the game/make the game last forever, for the user to try to beat their own high score (potentially make use of a leaderboard)
- The use of mathematics and number/logic as part of the puzzle (this fits into what one of the stakeholders said, that they would like a puzzle game with a mathematical aspect)

From this, I can deduce the following disadvantages, or things I should ideally avoid:

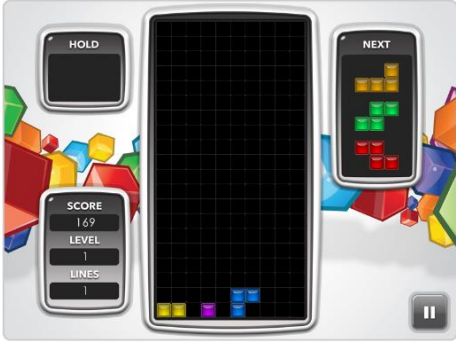
- Difficulty; the game is virtually impossible to beat via ad hoc methods.
- Repetitive outcomes

A lot of the listed characteristics of 2048 have been described by my stakeholders previously. For example, [redacted] talked about minimalistic designs and mathematical problems being positive aspects of a puzzle game, while [redacted] talked about 2048's difficulty being too much. This reinforces the decisions I've made above, as my stakeholders agree.

<u>Tetris</u>

The second game I shall play is called 'Tetris'.

- Developed by Alexey Pajitnov in 1984.
- Tetris involves falling blocks in a 10 by 20 grid, with 7 different types of block, each a different shape, as shown below. These are called 'Tetrominoes'.
- The objective of Tetris is to try to score as high a score as possible.
- The player is guaranteed to lose eventually.
- The player scores points by dropping Tetrominoes in arrangements such that they fill up an entire row in the 10 by 20 grid.
- The player can rotate the blocks in any way, store them and see what blocks they have next.



- A player has lost the game if the entire grid is filled such that there are Tetrominoes touching the top.

I played Tetris several times, and it is very easy to see why the game is so popular even after several decades.

Advantages	**Disadvantages**
The concept is simple but challenging, providing you with few but effective tools.	I found it difficult to understand which Tetromino was going to fall next, meaning that I often made mistakes in my planning.
I really liked the ability to see a 'ghost' of where your piece lands, as this really helped with decision-making in the game.	Another issue I had was that I didn't have the ability to simply use a held piece, without holding another piece in its place.
The ability to 'hard drop' or 'soft drop' a piece allowed the player to control the speed at which they played.	The design in this particular version was too unnecessarily detailed.
I liked these two mechanics in particular because they allow the user to control the pace of the game.	
The game gets faster and thus more difficult as the player progresses.	

I can take the following from this game:

- Some way to see your next move
- The ability to move your pieces faster
- The ability to see where your current piece will go to
- Set pieces that don't vary
- Progressive difficulty

I should avoid the following:

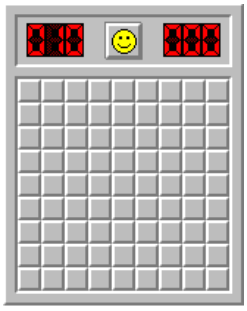
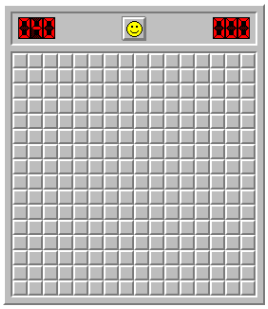
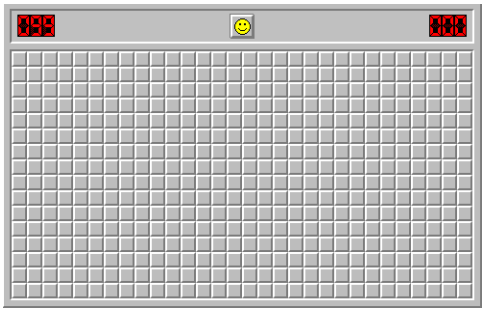
- Lack of clarity on how a particular mechanic works
- A mechanic that impedes gameplay, or gets in the way of another mechanic/ability
- Unnecessary detail

[redacted] had already mentioned set difficulty as a negative, meaning I should make use of progressive/adjustable difficulty.

<u>Minesweeper</u>

A third puzzle game I shall take a look at is 'Minesweeper'.

- Released in 1989.
- The player is presented with a grid, with the objective of searching for hidden mines in this grid.
- There are three difficulty variants, with different dimensions and a different number of mines.
- Beginner is 9 by 9, with 10 mines; intermediate is 16 by 16, with 40 mines; expert is 16 by 30, with 99 mines.

- When you click on a tile in Minesweeper, the area around the tile clicked disappears, revealing numbers. These numbers indicate how many bombs are near to the particular tile containing the number.
- From here, the player has to use the given information to click tiles, avoiding bombs, until the entire field has been cleared except for the bombs.
- If the player clicks on a bomb, they lose.

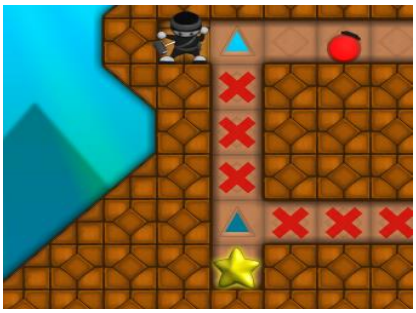
Advantages	**Disadvantages**
Allows the user to adjust the difficulty settings and create a custom game, where they can choose the height, width and number of mines on the board.	A lot of the game came down to guesswork as to where bombs may be.
Allows players to 'flag' certain locations where they think there may be a bomb, to avoid accidentally clicking on and detonating the bomb.	

I can take the following from this game:

- The ability to adjust and change difficulty
- Some way to minimise accidental clicks
- Increasing resolution of problem to increase difficulty

I should avoid the following:

- Parts of the puzzle which involve guesswork



Ninja Painter 2

The fourth game that I have chosen to look at and analyse is called 'Ninja Painter 2'.

- A flash game on a website called coolmathgames.com.
- Maze-based

- Has a prequel, Ninja Painter. I've chosen to analyse the second game over the first because the latter features more advanced and interesting mechanics that I can look at.
- The objective is to paint walls in a maze with an appropriate colour.
- You pick up buckets of paint of different colours and paint walls with those designated colours. You can lose the level by flying out of a window.
- Once the ninja (player) has been told to go on a particular direction, they will continue in that direction until they hit a flat surface.
- The ninja can 'deflect' off of angled surfaces, pick up keys that open locks, and hit switches that trigger particular corresponding blocks to appear or disappear.

As shown below, there are three stars to collect in every level. They serve as an extra challenge for the player. Also in the image, the paint bucket (the red circle) and walls to paint (the red crosses) are shown.

As shown, there are multiple colours used to paint the walls, as well as angled walls for the ninja to 'deflect' off of. In this particular image, the game is in progress, as the pink walls have already been painted. It also shows the combo that the player has achieved, and how many points they have scored as a result.

Advantages	**Disadvantages**
Featured an actual soundtrack and sound design.	The soundtrack in this case was loud and irritating, and did not fit the theme of the game at all.
Featured stars as side-challenges.	The scoring system did nothing to the gameplay whatsoever.
Sound design for interacting with objects was very appropriate.	The achievements system was trivial.
The use of mechanics such as the switches and doors made the game challenging and added extra depth to the experience.	
The ability to choose the character's gender allowed the game to appeal to a wider audience.	

I can take the following from this game:

- Interesting mechanics that change or influence gameplay
- Sound design and sound effects

I should avoid the following:

- Useless scoring/achievements systems
- Overbearing and harsh soundtracks

<u>Lemmings</u>

The fifth game I shall play is called Lemmings.

- A puzzle-platformer, released in 1991.
- The objective is to get a group of lemmings on each level to a designated exit, avoiding environmental hazards and obstacles along the way.
- Lemmings can individually be granted different abilities and traits to aid them. There are eight skills that can be applied to the lemmings.
- Climber lemmings can climb until they hit an obstacle.
- Floater lemmings can float down from heights.
- Bomber lemmings explode after a five second delay, destroying a small area around them.
- Blocker lemmings force all lemmings that contact it to change direction.
- Builder lemmings build a stairway with 12 steps.
- Basher lemmings dig horizontally.
- Miner lemmings dig downwards diagonally.
- Digger lemmings dig directly downwards.
- The game features four difficulty settings, with set levels in each.
- The difficulties are (in order from easiest to hardest): fun, tricky, taxing and mayhem.
- The player also has the ability to 'nuke' all lemmings, killing them all after a five second delay.

Advantages	Disadvantages
'Side-scrolling' feature, as shown above, with a scroll bar.	Occasionally, the mechanics would work incorrectly. Sometimes builder lemmings would build in the wrong direction, sometimes the nuke button wouldn't work, sometimes basher lemmings would give up digging and walk away from the wall without a reason.
Depth of information provided; the player is told about virtually everything going on in-game.	The number of tools felt overwhelming at times.
Huge freedom with approach to a problem.	
Restriction of use of certain lemmings, to make the game harder.	

I can take the following from this game:

- Plentiful but useful mechanics
- Impose restrictions to force players to adapt on certain levels

I should avoid the following:

- Potentially game-breaking bugs and issues with certain mechanics and objects
- An overwhelming number of tools

3.1.3b – Essential features

Firstly, looking at these puzzle games, it is evident that they are all different subgenres of puzzle game.

- 2048 was a block/arithmetic based game
- Tetris was a block/tile matching game
- Minesweeper was a tile uncovering game
- Ninja Painter 2 was maze-based
- Lemmings was side-scrolling

This meant that I must make a decision as to what type of puzzle game my final product will be.

1) Side scrolling games
   a) I can identify that implementation of a side-scrolling game would be the most difficult, even on a 2D plane. I would need to create a camera object in order to follow the progression of the game, and generate the next part of the puzzle via some form of procedural generation, which is very difficult.
   b) This is also very taxing for the computer system, since it has to both record and generate the puzzle, which is growing in size.
   c) For these reasons, I have chosen to eliminate side-scrolling games as a genre of puzzle games that I could potentially develop. This is one of the limitations.
2) Tile uncovering games
   a) I inquired with my stakeholders about tile uncovering games (Minesweeper). They all disliked this genre generally, because of how repetitive it generally is.
   b) Because this violates one of the previously mentioned essential features, I will not be making this type of puzzle game.
3) Arithmetic games
   a) Arithmetic games are a good idea, because the stakeholders generally had a positive reception towards them.
   b) However, they may be difficult to code on a large scale, and given the time restrictions, this can be very damaging.
   c) On a smaller, more limited scale, these would likely be very repetitive, which is a violation of the essential features.
   d) As a result, I will not be making an arithmetic based game. This is another limitation, as the stakeholders ([redacted]) generally preferred puzzle games with arithmetic elements.
4) Block/tile matching games
   a) I pitched this idea to my stakeholders; I had a 50/50 split. Some liked the idea; others claimed it would become too repetitive and boring very quickly.
   b) Thinking about implementing this type of game, it lends itself fairly well to most algorithms, but has a lot of room for error. It is very difficult to build an algorithm that recognises when blocks should be eliminated every time, since it may make a mistake in different circumstances.
   c) It's very difficult to design an idea that is original with this sub-genre. 'Tetris', '2048' and 'Candy crush' have already pretty much used every possible idea in this genre.
   d) Even though we decided that would be a better idea to look for another genre, I will be taking some elements from tile-based games, in order to help with generation of puzzles.
5) Maze games
   a) Maze games lend themselves very well to my requirements and computational methods, since there are algorithmic methods of generating mazes.
   b) They can also be adapted to feature different obstacles within the maze, making the game more interesting and challenging.

c) I presented the idea of a maze game to my stakeholders. It was very enthusiastically received, with all of them fully supportive of the idea.
d) For these reasons, I will be developing a 2D, primarily maze-based puzzle game.

I ran this idea by my stakeholders, and got the following responses.

[redacted]: “I am in agreement with your conclusion of a maze game, incorporating elements of arithmetic/tiles. I have said multiple times in my feedback that I am one to enjoy games that include problem solving and a lot more thinking and I think that your conclusion of a choice of game style would do just that.”

[redacted]: “I think this idea would allow for the inclusion of some of the good factors mentioned last time, like the ability to make the levels progressively more difficult and making levels completely different from each other which keeps the game from getting boring over time. I also think the arithmetic tiles would also help with, making it easier for the player to learn how to get better at the game, as there are rules/patterns that are followed.”

[redacted]: “A maze game sounds very fun! Mazes are very interesting to explore, so a maze based puzzle game is a great idea. I think that it will pair nicely with an arithmetic based puzzle, as the player has to both navigate and solve problems, which sounds exciting.”

[redacted]: “I like this idea because it makes me think, like a puzzle game should.”

[redacted]: “Yes, I agree, that sounds like a good mix between an interesting game that both engages the user intellectually and also allows the user to be interested.”

As a result of this, I will be making a 2D puzzle game with a maze as the main feature.

I also asked my stakeholders about whether I should use numbered levels or levels named by difficulty (easy, medium and hard).

[redacted]: “I like the idea of having easy, medium and hard difficulties because I feel like it will give me a chance to choose how I want to play rather than strict level continuously increasing in difficulty.”

[redacted]: “I would prefer the difficulty of the game to be measured in easy/medium/hard difficulties, rather than levels, as it allows the player to choose the difficulty they wish to play at, preventing the game from getting boring, for those that are good at it.”

[redacted]: “Having discrete levels makes it unclear which level is supposed to be easier/harder. I want to be able to quickly decide which difficulty I’d like to play on rather than having to try every level and find out.”

[redacted]: “I prefer using difficulty levels rather than set levels because set levels are invariant while easy, medium and hard give more customisation options because they are random.”

[redacted]: “I feel I respond better to worded difficulties because it helps me to more accurately understand what to expect and also gives more sense of competitiveness. Also, most games have worded settings, like online Scrabble for instance.”

As a result, I will be making use of labelled levels rather than numbered levels.

Finally, I asked proposed the following idea to my stakeholders as the final detail for the game's nature.

- (This game assumes the idea of four directional movement in a square maze.)
- The objective of the player is to traverse through the generated maze, from one end to the other.
- They move using WASD/arrow keys.
- Everywhere the player traverses, a red tile will be produced. These red tiles are dangerous and will kill the player if traversed on again. This forces the player to make correct decisions and think critically.
- In order to prevent a player from stalling infinitely, I have decided to include a timer that kills them if they remain in place for too long. The timer will be displayed on-screen to show how much idle time the player has left.
- Upon death/completion, the player will be able to advance to the next difficulty/restart the current level. They will also be able to exit to the main menu.
- In order to evolve difficulty of the game, I will increase the resolution of the maze for difficulties "easy", "medium" and "hard". Furthermore a harder difficulty (and hence a higher resolution maze, with more detail) will force the users to make even more decisions in the given time.
- To make the game even harder, I am going to make the given time period shorter for harder difficulties. I will need to include a timer to show how much time a player has left to remain idle before they are eliminated.
- I took this idea of forcing players to make quick decisions from Lemmings; in Lemmings, if the player doesn't make a decision fast enough, is idle for too long, or doesn't assign a particular lemming an ability in time, the lemmings will often fall to their deaths or run into hazards.
- I took the idea of "painting" traversed cells/spaces from Ninja Painter 2. In this case, traversed cells become the threat entity that the player must now avoid.
- [redacted] came up with the idea of calling the red cells 'lava', and the player simply has to 'escape the lava'. The stakeholders really like this idea of giving context to the game, and came up with the name "The Floor is Lava". Since this was agreed to appropriate, I greenlit this.
- A lot of the games I looked at featured power-ups; I have not yet considered this idea fully, or inquired with my stakeholders about it. As a result, the initial prototype will not contain any kind of pickups or power-ups, but subsequent iterations may contain it, depending on how the stakeholders receive my product.

Here are the stakeholders' reception to the above.

[redacted]: "I like the idea of the game because it is a challenge. It is a creative and different approach to the maze game idea, and I have never played a game like this before. The idea of killing the player adds a level of difficulty and strain on them, pushing them to their limits to find a correct way out quickly."

[redacted]: "This game idea pushes you to think ahead, and forces you to make decisions. I like this because it can be intense or relaxing, depending on the difficulty selected. I particularly liked the way that different mazes will be generated every time because it keeps the idea fresh and exciting."

[redacted]: “I think the ability to adjust difficulty is really good because if a level is too easy for me I can choose a harder maze. I also think that the idea of including a timer to force moves is cool, and it’s a good idea for lower difficulties to have longer timers.”

[redacted]: “I like the concept, it seems pretty interesting and fun. The idea of having to make split second decisions is very intriguing, combined with a randomly generated maze means that there’s a lot of fun to be had. I also appreciate the different difficulty levels, I’m not very quick with reactions so I will probably need to choose the lower difficulty, making it more accessible to a wider variety of users.”

[redacted]: “To me, the concept of time being the enemy is refreshing. It creates a stimulating atmosphere by breaking the stereotype of playing against a concrete opponent (e.g. chaser) by introducing an abstract opponent that is time. Furthermore, the ‘no back-tracking rule’ adds to the challenge and I believe that it will play a role in enhancing decision making and reaction time within players.”

Since all of my stakeholders agree with the proposed idea, I will continue with this. Of course, I have only outlined the essential features of the gameplay thus far. The gameplay and concepts will be expanded in the next section. I do still need to clarify, and make decisions based on my research, some other aspects of the game.

Now that I have identified my puzzle game as a 2D maze game, I can identify further essential features. I can also make decisions on what may or may not be included, taking elements from the games shown above.

Essential feature	**Justification**	**How I will solve this**
2D gameplay	The entire project is based around the need for a 2D puzzle game system.	I will use a module/programming language that allows for development of 2D games.
Methods of input and the machine interpreting these inputs	The user needs to be able to control the system through inputs, allowing them to play the game.	I will store inputs for the player in an array, so that they are grouped together, but when a particular input is given, a corresponding method is called.
Functional gameplay that uses consistent inputs	Inputs need to be consistent to allow the user to properly understand what the click/press in order to provide a certain result.	Ensuring that methods for certain inputs are ONLY called when that input is given.
Telling the player their progression/difficulty	The user will have to know where they are in the game in order to know what they have to do next in order to progress. They also need to be able to understand what difficulty level they are playing at in order to customise the challenge for themselves (should I choose to include a difficulty selection screen).	Allowing selection of difficulty at the main menu, and having a small textbox displaying the player’s current difficulty in-game.

Storage of data	Data will need to be constantly stored and updated, either temporarily or permanently. This will allow gameplay to take place and user data/progression to be saved.	Use of arrays to group relevant data, and storing information as discrete variables for single pieces of data.
Generation of elements (visual or auditory)	Generation of UI/gameplay elements is essential because gameplay is entirely visual-based, and objects need to appear and interact on screen. This also includes detection of collision, which is needed to trigger certain in-game events, such as death.	Use of a module/programming language/library that allows the generation of computer graphics and sound.
The use of a UI	The user needs to be able to interact with the system to select, play and understand the game. The use of a user interface will enable the user to carry these out.	Classes for different UI elements, the use of computer graphics and text.
Generation of a solvable puzzle	The puzzle needs to be unique and solvable every time to provide a fair but challenging experience to the user. If the puzzle is not solvable, the game is pointless.	Implementation of some algorithm that guarantees random generation of perfect mazes.
Tutorial screen	The player needs to have a clear objective and understanding as to how to solve the puzzle. If not, gameplay will be difficult to understand and not enjoyable.	Information on the game's objective in textboxes, so the user knows what they must do.
Non-repetitive puzzles	Repetitive gameplay will quickly bore a user and not be a challenge.	Using an algorithm that generates mazes randomly so that they are different every time.
Red to represent enemies; green to represent the player	Research has shown that all of my stakeholders believe red to be a colour that is threatening and green is a colour that is friendly – making them appropriate for those roles.	Use of tuples to represent RGB values of colours, and use a programming language that allows computer graphics.
Black/white background	Black/white backgrounds are better than coloured backgrounds/image backgrounds because they don't impede the view of the objects in the foreground and don't distract from the game.	Setting background colour value to (255, 255, 255) or (0, 0, 0). (I use tuples because they cannot be accidentally modified.)
WASD/Arrow keys as inputs	These are the two most dominant and desired methods of input by my stakeholder group. Allowing users to select between them guarantees user comfort with controls.	Storing these inputs in two separate arrays of 4 elements, and making one array valid for inputs depending on user choice.
Square maze	Cells in the maze will be square shaped, because they tend themselves to "tiles" well, and more algorithms allow for generation of	Use of an algorithm that generates square mazes.

	square mazes.	
Four-directional movement	Square mazes only allow for four-directional movement, and WASD allows this perfectly. (UP/DOWN/LEFT/RIGHT)	Use of a square maze (above).
Bird's-eye view	The game literally has to be in bird's-eye view, since it's a 2D game – there is no other approach with view that can be taken to present a maze in 2 dimensions.	Generating the maze on screen and keeping it as a static object, from a bird's-eye view.
Sound design/effects	Sound design has been agreed by my stakeholders to be important to making a game more immersive and proved to be beneficial for my play through of Ninja Painter 2.	Using a programming language/module that allows me to import and use sound effects.
Increasing resolution of maze for higher difficulties	With Minesweeper, using a higher resolution grid shown to be effective in making a level more difficult, since it provides more room for error - making a maze higher resolution will do the same, and is easy to implement with an algorithm.	Changing variables for maze resolution, increasing/decreasing them as appropriate for every maze generation.
Random generation of mazes	As well as using the algorithm to ensure the maze generated and unique every time, the maze will need to be random in order to ensure uniqueness and an extra level of difficulty.	Implementing methods that follow algorithmic steps to generate a solvable maze, randomising certain parameters/variables for every generation.
Main Menu	This is included as a central hub to allow the user to move either to the controls menu or the tutorial screen, or to select the difficulty and play the game. This is a UI/user experience feature.	Use buttons that allow the user to navigate to different parts of the system.
Text boxes	These will be used in order to provide the user with essential information to be read. This is needed for the user to know what stage of the game they are on.	Creation of text box classes, complete with methods and attributes for a text box.
Interactive buttons	In order to allow the user to navigate throughout the system (outside of the actual main gameplay), I must provide them with labelled buttons to click to move throughout the system. This is a UI/user experience feature.	Creation of button classes, complete with methods and attributes for a button.
Difficulty selection named - 'easy', 'medium' and 'hard'	Explicit naming of difficulties allows the user to understand what level of challenge to expect, versus using a number system. Furthermore, the stakeholders all strongly endorsed this idea.	Use of separate buttons which each generate games of the same specification as the selected difficulty.
Generation of lava cells	As the player moves through the maze, lava cells are left in their wake. This is used as the primary threat entity to establish some level of challenge to the game, in order to prevent	Use of an array to store locations of lava cells, appending this array with the previous

	them from backtracking.	location of the player.
Lava cells killing the player	The lava needs to kill the player upon contact, or it's not a real threat and the game has no real challenge.	Use of an array to store locations of lava cells, to kill a player when their location matches that of any lava cell in the array.
A timer	This is to prevent infinite stalling on the player's part, and to force them to move and keep the game moving.	Use of an integer/float variable that is displayed on-screen, and that decreases by 1 every second.
Expired timer killing the player	When the timer expires, the player will need to die since they have spent too long in one place.	When the timer's variable for time reaches 0, call a method that ends the player's life.
Game over screen appearing on death	After the player dies, they need to be given the option to restart or return to the main menu, or they have nowhere to go in the system.	Use of a method that detects death, and brings about the game over screen, with options to return to the main menu or restart.

After I collated all of these ideas, I sent them to my stakeholders to look at. They sent me their responses, giving their opinions on the points I raised.

[redacted]: "Having looked at the tables I can happily approve of your ideas to include and avoid. I particularly like how you are considering adding features that look at the user experience too. I also like how difficulty is adjustable for the users, and how bigger mazes are used for harder difficulties."

[redacted]: "I like the random generation of mazes because it makes sure the gameplay never gets old. I agree with the idea of avoiding repetitive outcomes because it makes the game boring and makes me not want to play the game."

[redacted]: "I appreciate the 2D aesthetic - my laptop is not very fast so I can't run more complex, 3D games. Having a 2D game with simpler graphics means that I don't have to worry about performance issues, and all my friends can also play as it is easier to run. I also like that the difficulty is customisable. I don't like it when a game is too difficult, as it means that it becomes a chore to play and so isn't fun."

[redacted]: "I like that you've used labelled buttons to let the user navigate from the main menu to the actual game and back. This makes using the game much easier for me."

[redacted]: "It is clear that careful consideration had been taken into the initial planning and layout of what the final project will look like. A wide array of aspects has also been mentioned showing everything has been considered."

After looking at my stakeholders' responses to my ideas, I can verify that all of the ideas listed are fully validated and justified.

3.1.3c – Limitations of the solution

However, there are a few limitations of my solution. These are things that my stakeholders would've wanted me to do, but I will not be doing, due to certain issues/restrictions.

- I will not be developing a game with side-scrolling features, despite [redacted] saying that he quite enjoyed side-scrolling games, because development of these games on a 2D plane would need a camera element, which will take time and is difficult to develop. It's not worth the investment as it is not integral to the puzzle game experience.
- [redacted] mentioned that he enjoys games with arithmetic elements. I won't be including these because there is no real beneficial application of this in a maze game.
- [redacted] mentioned that he would like a game with elements that allow a player to play with their friends. I am not including this because to do this would require either network support or support for multiple input controllers; I have not got the resources for either. Neither of are worth the time spent developing, and may impede the final product instead.
- The ability to select a character will not be included, despite being popular amongst stakeholders. This is because it doesn't help the gameplay enough to be included. I can spend the time on refining and testing gameplay for bugs.
- The game will not be ported to non-PC platforms because I am working under time constraints, and porting to other platforms will take time and will result in a less refined game. Also, I don't have easy access to resources that allow me to easily port my game; acquiring and understanding these will take a lot of time.

3.1.4 – Specify the proposed solution

3.1.4a – The solution requirements and 3.1.4b – Success criteria

Requirements specification

This is a list of objectives that specify what the final product will do for the user, and how the success of each of these requirements will be measured.

Requirement ("The user must be able to…")	**How will this be measured/tested?**
Access different parts of the game via the main menu, as well as being able to return to the main menu.	Having all stakeholders navigate to every part of the game via the main menu and return to the menu.
Exit the game from the main menu.	Having all stakeholders try to exit the game using the 'EXIT' button on the main menu.
Select their control scheme (WASD or arrow keys)	Having stakeholders choose between the two different control schemes and checking if they work.
Select difficulty	Having stakeholders choose the difficulty of the game, and check if the appropriate maze and timer have been loaded.
Have a maze generated based on their selected difficulty	[Covered above.]
Move using their selected control scheme	Having stakeholders check that all inputs match the correct outputs (e.g. UP arrow moves the character up).
Be disallowed from traversing through walls of the maze	Having stakeholders attempt repeatedly to traverse through walls
Have a timer that tracks how much time they	Having stakeholders test whether the timer uses

have left before death	the correct duration for their selected difficulty, and that it counts down correctly.
Have lava cells generated in their wake	Having stakeholders traverse through the maze, checking that there are always lava cells being generated behind them as they move.
Die (when either running into a lava cell or expending their allocated time)	Having stakeholders test both methods of death, and that they are truly dead (unable to move, triggering of the 'game over' screen).
Win (upon traversing the maze correctly, without dying)	Having stakeholders test all difficulties, ensuring the winning screen ensues when the player does complete the maze.
Return to main menu or retry from the 'game over' screen	Having stakeholders attempt to do both from the game over screen.
Return to main menu or proceed to next level from the winning screen	Having stakeholders attempt to do both from the winning screen.

System requirements

In order for the game to run on a system, I need to outline hardware and software specifications, as system requirements, to make sure the user knows whether or not their computer is capable of running the game.

Hardware requirements

Since the idea of the game is to be very simplistic and basic, the system requirements are not expected to be incredibly high-end or extreme. In fact, the system requirements are ideally expected to be incredibly low and basic, meaning the game can be run on virtually any modern machine.

Component of computer	**Minimum requirements**	**Recommended requirements**
CPU	Single core, 2GHz	Dual-core, 2GHz
RAM	2GB	4GB
GPU	N/A	N/A
Disk space	25MB	50MB
Peripherals	Display monitor (1000×700), keyboard and mouse	Display monitor (60Hz+ refresh rate), (1920×1080), keyboard and mouse, speakers/headphones

As shown above…

- The system requirements are very elementary, and there won't really be a system incapable of running the game.
- The keyboard is needed to actually play the game; the mouse is needed for navigation.
- The speakers/headphones are needed for any auditory output, such as sound effects. This is not necessary to the experience, but does improve it.
- The screen size is to fit a window of 1000×700, as previously specified.
- I have chosen to have the game in a window instead of full screen because monitor displays vary in size, and may cause problems with aspect ratios in relation to my game. Having the game in a window will eliminate all of these problems, without impacting the performance or experience.
- The disk space is estimated roughly to account for the entire game and all of its files.

- The user doesn't need a GPU, since there are literally no advanced graphics to be processed at all and no simple, repeated calculations either.
- The game can only be run on computers; it is not supported on any kind of console or other platform. This is done to make development of the game much easier, as I won't have to port it to other platforms.
- The game will be running at 60 FPS. I have chosen 60 FPS because this is the standard today for most games. Framerates higher than 60 FPS are unnecessary for such a simple game; hence it will be locked at 60.
- This means that the user should (ideally) have a monitor with a 60Hz+ refresh rate, in order to experience the game in its best possible form. This isn't a minimum requirement, and the user may use a monitor with a lower refresh rate, such as a 50Hz monitor, this won't hugely affect experience. However, it is a recommended requirement to have a 60Hz+ monitor, in order to experience the game in its best possible state.

Software requirements

Software	**Minimum Requirement**
Operating system	Windows, Mac OS, or Linux
Python	Latest version
Pygame	Latest version

- Python code can be compiled and run by Windows, Mac OS or Linux, as appropriate file extensions (.EXE for Windows, .DMG for Mac OS and .PY for Linux). As a result, my solution will be usable on any of these operating systems.
- I have chosen to develop the game in Python 3, through the Pygame module, because Pygame allows the use of 2D computer graphics and sound libraries.
- I have extensive experience in Python and Pygame, which means development of the final product will be significantly faster, since I don't have to learn a new language.
- The user will need the latest version of both Python and Pygame, since I will be using them. Older versions of Python and Pygame may not be compatible with my code as a result, and the user will need the newest versions of both installed, to prevent problems.

Finally, I need success criteria to evaluate the overall success of each iteration. These success criteria need to be specific and measurable, in order for me and the stakeholders to be able to identify if they have been met or not, thus determining the overall outcome and success of an iteration/the project.

I have chosen to decompose my success criteria into smaller parts, relating to separate elements of the game.

DESIGN CRITERIA:

Criteria:	**How to evidence:**
The final product must be a 2D puzzle game.	Screenshot of main gameplay
The window for the game must be 1000×700.	Screenshot, and confirmation from stakeholders
The game background must be black or white.	Screenshot of main gameplay
The primary threat (lava) in the game must be coloured red.	Screenshot of main gameplay
The main character/object of the game must be coloured green.	Screenshot of main gameplay

The menu must feature buttons for the player to click on to make decisions.	Screenshot of main menu

DEVELOPMENT CRITERIA:

Criteria:	**How to evidence:**
The game must be developed on Python, using the Pygame module.	Screenshot of code in Python, using the Pygame module
Development must feature prototypes, and improve these prototypes through iteration.	Screenshots and reviews of every iteration, with criticisms from the stakeholders
Development must take feedback from the selected stakeholders, and make changes to iterations based on this feedback.	Screenshots and reviews showing how each iteration has been improved based on previous feedback

INPUT/OUTPUT CRITERIA:

Criteria:	**How to evidence:**
The player must be able to choose between WASD and the arrow keys as their input controls.	Screenshots of control selection menu, stakeholder reviews ensuring controls work
The game must output to a screen.	Picture of game outputting to screen
The game must run in windowed form.	Screenshot of the game window
The game must feature some audio, in the form of basic sound effects.	Stakeholder review confirming the presence of audio for some events

GAME CRITERIA:

Criteria:	**How to evidence:**
The game must feature a title screen.	Screenshot of title screen
The game must feature an introductory screen.	Screenshot of introductory screen
The player must be able to choose their difficulty level.	Screenshot of difficulty selection screen, with stakeholder review to ensure that it works
The introductory screen must allow the user to view a tutorial screen with basic instructions.	Screenshot of tutorial screen with instructions, clarifying with stakeholders that instructions are understandable
The introductory screen must allow the user to exit the application completely.	Screenshot of 'EXIT' button and stakeholders confirming button's functionality
The game must feature a main game screen, where the user can play.	Screenshot of game screen, stakeholders confirming that the game is playable
The game must feature a game over/winning screen.	Screenshot of the game over screen; screenshot of the winning screen
The algorithm must generate mazes that are always solvable.	Screenshots of all difficulty mazes to show they're all fully solvable
Any tiles/spaces/cells that the player traverses must become red.	Screenshots of a player as they traverse through the maze, and how lava is left in their wake
The player must not be allowed to traverse lava without dying.	Stakeholder review confirming that they die whenever they attempt to touch lava cells
The game must feature a timer to force players to make moves.	Stakeholder review confirming that they die if the timer reaches 0; confirming that the timer resets with every move

Harder difficulty mazes must be a higher resolution (more detailed, with more cells) than their lower difficulty counterparts.	Screenshots of all maze difficulties [covered above], showing increased resolution for higher difficulty settings
Harder difficulties must feature a shorter timer than their lower difficulty counterparts.	Screenshots and confirmation from stakeholders that higher difficulty mazes have a shorter timer

Section 2: Design

3.2.1 – Decompose the problem

3.2.1a – Breaking down the problem

Introduction to Design

- Now that we have outlined specifications and understood the objectives, needs and requirements of the stakeholders for our final product, we need to actually design a feasible and real solution.
- I shall continue to maintain close contact to the stakeholders, to ensure that the product designed is as close to their desired outcome as possible, as well as continuing to follow the success criteria set out in the analysis section.

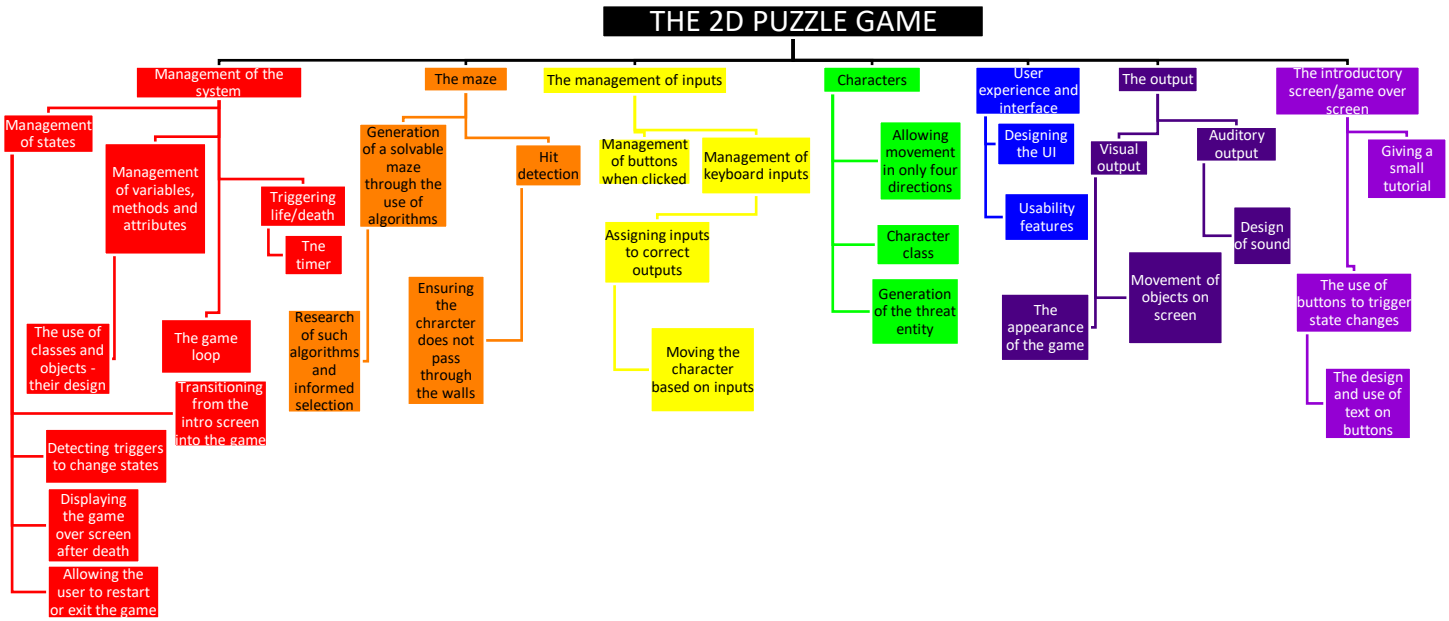
Decomposing the problem

Looking at the different areas of the requirement specification and success criteria, I can identify different jobs that need to be done in order to meet all stakeholder needs and wants. Below, I have split these all into different areas, in order to decompose the problem.

(Note: development criteria have not been included here – they are not relevant to the main structure/behaviour of the system.)

Area	**Requirement/Success Criteria**
Management of the system	Access different parts of the game via the main menu, as well as being able to return to the main menu.
Management of the system	Exit the game from the main menu.
The management of inputs	Select their control scheme (WASD or arrow keys)
User experience and interface	Select difficulty
The maze	Have a maze generated based on their selected difficulty
The management of inputs	Move using their selected control scheme
The maze	Be disallowed from traversing through walls of the maze
Management of the system	Have a timer that tracks how much time they have left before death
Characters	Have lava cells generated in their wake
Characters	Die (when either running into a lava cell or expending their allocated time)
Characters	Win (upon traversing the maze correctly, without dying)
Management of the system	Return to main menu or retry from the ‘game over’ screen
Management of the system	Return to main menu or proceed to next level from the winning screen
User experience and interface	The final product must be a 2D puzzle game.
The output	The window for the game must be 1000×700.
User experience and interface	The game background must be black or white.
Characters	The primary threat (lava) in the game must be coloured red.
Characters	The main character/object of the game must be coloured green.
User experience and interface	The menu must feature buttons for the player to click on to make decisions.
The management of inputs	The player must be able to choose between WASD and the arrow keys as their input controls.
The output	The game must output to a screen.

The output	The game must run in windowed form.
The output	The game must feature some audio, in the form of basic sound effects.
User experience and interface	The game must feature a title screen.
User experience and interface	The game must feature an introductory screen.
Management of the system	The player must be able to choose their difficulty level.
The introductory screen/game over screen	The introductory screen must allow the user to view a tutorial screen with basic instructions.
Management of the system	The introductory screen must allow the user to exit the application completely.
User experience and interface	The game must feature a main game screen, where the user can play.
The introductory screen/game over screen	The game must feature a game over/winning screen.
The maze	The algorithm must generate mazes that are always solvable.
The maze	Any tiles/spaces/cells that the player traverses must become red.
The maze	The player must not be allowed to traverse lava without dying.
Management of the system	The game must feature a timer to force players to make moves.
The maze	Harder difficulty mazes must be a higher resolution (more detailed, with more cells) than their lower difficulty counterparts.
Management of the system	Harder difficulties must feature a shorter timer than their lower difficulty counterparts.



- Above, I have decomposed the previous points, simplifying them and categorising them into primary areas.

- I have chosen to decompose my problem in this way because it allows me to consider individual parts of the problem much easier.
- I have decomposed in the shown categories because I can quickly account for the all aspects of the program by looking at it in this way.
- For instance, by compartmentalizing “outputs”, I can further decompose this aspect into auditory output and visual output. From here, I can focus on smaller issues, such as the need to design sounds before implementing them.
- With this, I can begin to understand how this may be done; sound design will be discussed later in this section.
- Through the use of decomposition, I can build up smaller, simply solutions to work towards a product that solves our complex problem.

Red – The management of the system

- I need to come up with some way of managing all the different states of the game.
- I need to design classes for different elements of the solution, and the objects derived from these classes.
- I need to design the timer, and make sure life/death are triggered correctly.

Orange – The maze

- I need to make sure the character can’t move through the maze freely, phasing through walls.
- I need to research and make a selection on an algorithm that will generate the maze.

Yellow – The management of inputs

- I need to make sure any given input always gives its correct output.
- I need to make sure any given input is managed correctly, and that invalid inputs do not affect anything.
- I need to make sure the user can give inputs via the keyboard.

Green – Characters

- I need to ensure that lava is generated wherever the player goes.
- I need to make sure the player can’t move in any directions other than up, down, left and right.
- I need to design the character class that holds all of the information for the character.

Blue – User experience and interface

- I need to design the screens of the system that the user will encounter, and make sure they’re all suitable for my stakeholders.
- I need to consider usability features, to make the game easier to use.
- I need to consider fonts, colours, shapes and all other elements that make up the user interface.

Indigo – The output

- I need to design all audio for the system.
- I need to design the appearance of the game.
- I need to ensure objects on-screen can move.

Violet – The introductory screen/game over screen

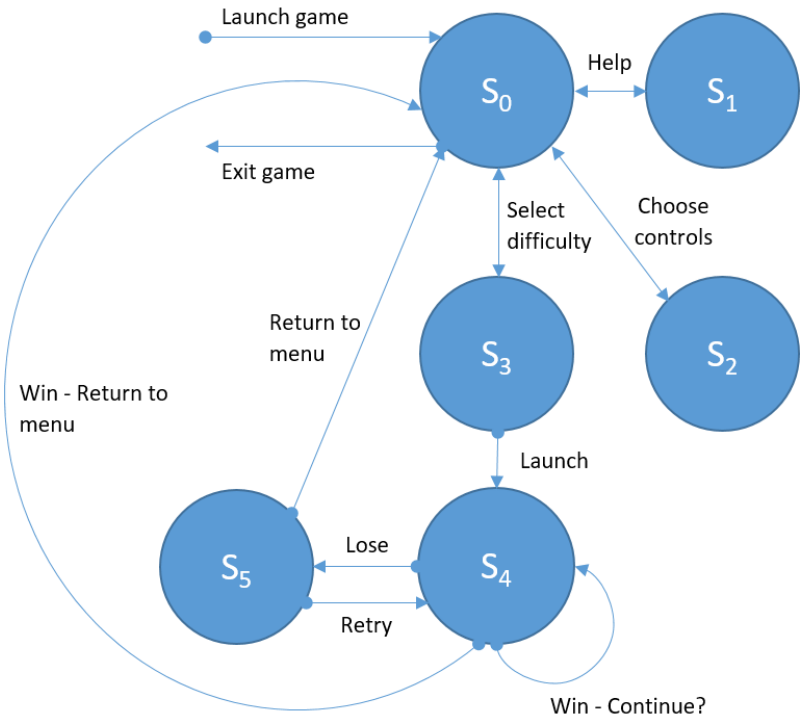
- I need to make sure the tutorial screen is included, and design/specify the information included on this screen.
- I need to design buttons and textboxes for my system.
- I need to design all screens for my system.

3.2.2 – Describe the solution

3.2.2a – The structure of the solution

Management of states

- The different parts/states of the game need to be able to work together, and allow the user to move between them, because if not, the user may end up locked in one state permanently.
- To solve this issue, I looked into ways of allowing a system to move between states and found a viable solution: a finite-state machine, more simply known as a state machine.
- A state machine consists of a finite number of states, and transitions between these states when certain axioms (requirements) are met.
- For example, a system might initially be in “state0”, until a button is pressed, upon which it transitions to “state1”. When this button is pressed again, it may move back to “state0”.
- My game itself is indeed finite, and upon loading in, the user will be presented with the “starting screen”, containing a set of options.
- As previously mentioned, these options will allow the user to view a basic information screen, launch the game into easy/medium/hard difficulties, adjust their controls between WASD and the arrow keys, and to quit the game.
- Below is a potential, elementary idea of how I could implement this. The different states are denoted by their names and “S_n”, where n is the number assigned to that particular state, relative to the introductory screen (S_0).



Since there wasn't enough space on the diagram to write the whole names of each state, I shall write them below:

- S_0 = Introductory screen/Main Menu
- S_1 = Tutorial
- S_2 = Adjust controls
- S_3 = Difficulty selection
- S_4 = Game state
- S_5 = Game over

Summed up:

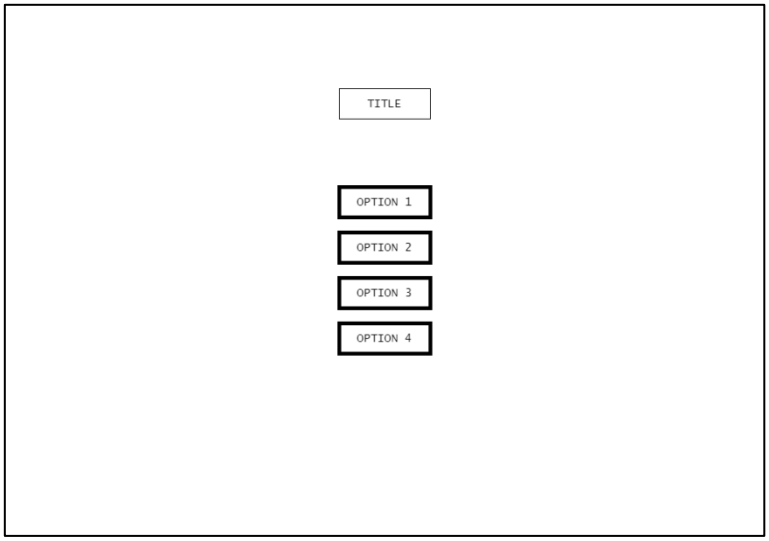
- When a user enters the game, they are greeted with the introductory screen.
- They will then have four choices: they can choose to launch the game by selecting a difficulty, they can adjust their controls, they can view the tutorial section, or they can exit the application.
- When choosing the tutorial/controls sections, they will have to be able to move back into the introductory state; hence the arrows are double-ended.
- With difficulty selection, they may change their mind and want to go back to the introductory screen, hence the double-ended arrow.
- Once they enter the game state, the maze will be generated and play will be ensued.
- If the player wins, they have the option to continue, or to return to the main menu.
- If they lose, they have the option to retry (at the same difficulty) or return to main menu. (NOTE: retrying does not mean the player gets another chance at the same maze. A new maze will be regenerated at the same resolution as their former difficulty selection.)

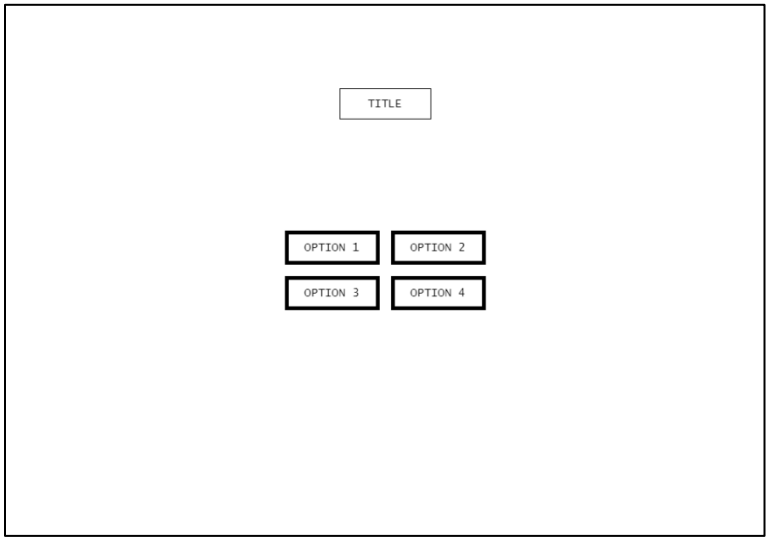
The use of a state machine handles all the problems listed under this subheading, as it allows the game to transition from the intro screen to the game, displays a game over screen, and from here it allows the user to restart or exit the game (via the main menu/introductory screen). It transitions between states upon certain triggers – these "triggers" are the user clicking on certain (labelled) buttons displayed on the screen.

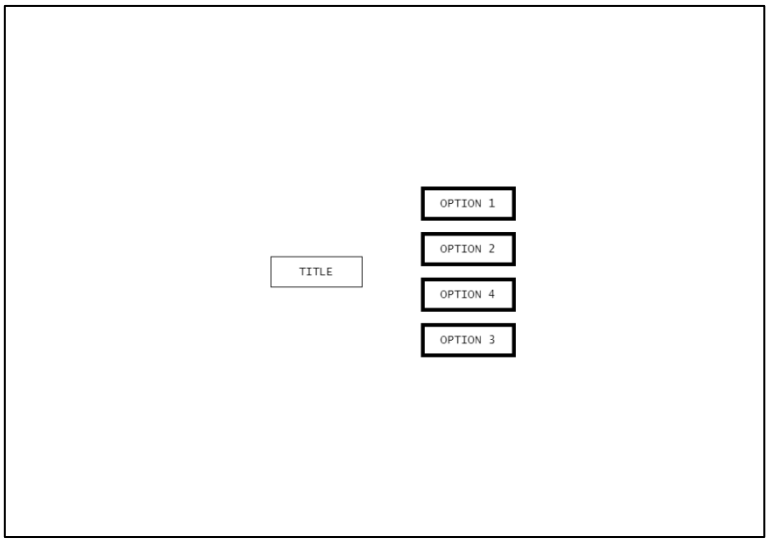
<u>Design of the user interface</u>

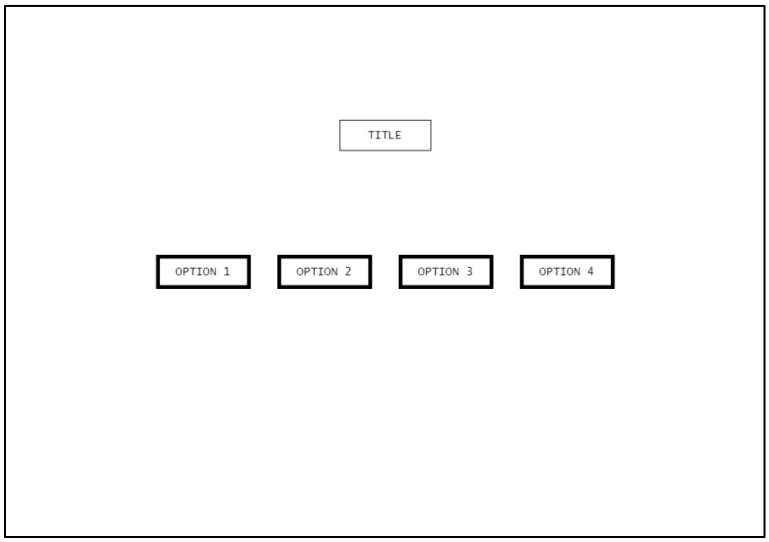
- Here I will show designs for the user interface, to show what the user will see when using the program.
- I will present these designs to the stakeholders.

For the title screen, I came up with 4 different layout designs. I then asked my stakeholders which one they preferred the most and why.







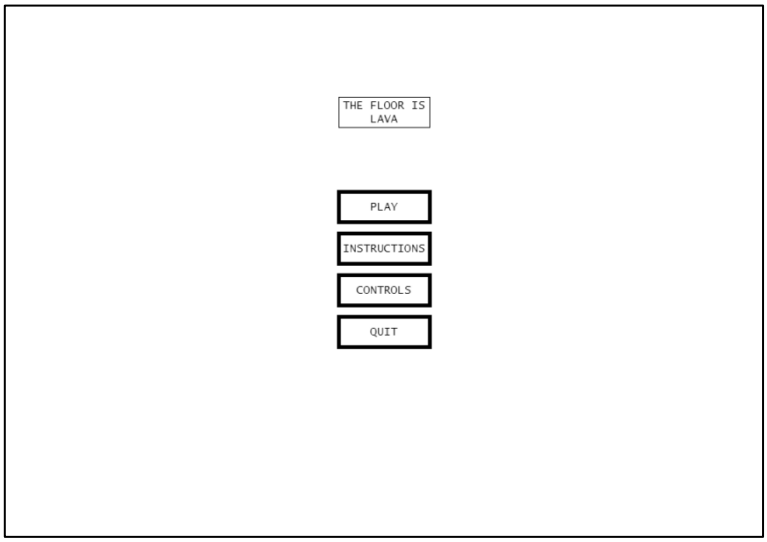


(Options are read in an 'S-shape' – top-left = first option, top-right = second, bottom-left = third, bottom-right = fourth.)

- [redacted]: "The first option is the best in my opinion. This is because it's super simple and clean, without any unnecessary details or clutter. It communicates all the options to the user effectively."
- [redacted]**:** "I personally like the third format a lot, because it's easy to read, but still creative and different – I've never seen that kind of layout with any videogame title screens."
- [redacted]**:** "I think the first format is the best, because it's easy to read, navigate and the hierarchy of different options is clear."
- [redacted]**:** "The first one is preferable to me, because it's the most generic and hence familiar layout for a title screen. Most videogames that I've played have a layout like this, meaning I'm more comfortable."

- [redacted]: “I think the second format is the best, because it has all the options grouped together in a compact way, making it look very simple. Furthermore, the title is easily distinguishable.”

As a result of their choices and justification, I will be using the first format shown above. It’s the most popular with stakeholders, and all of them agreed that the simple design was easy to use and understand.

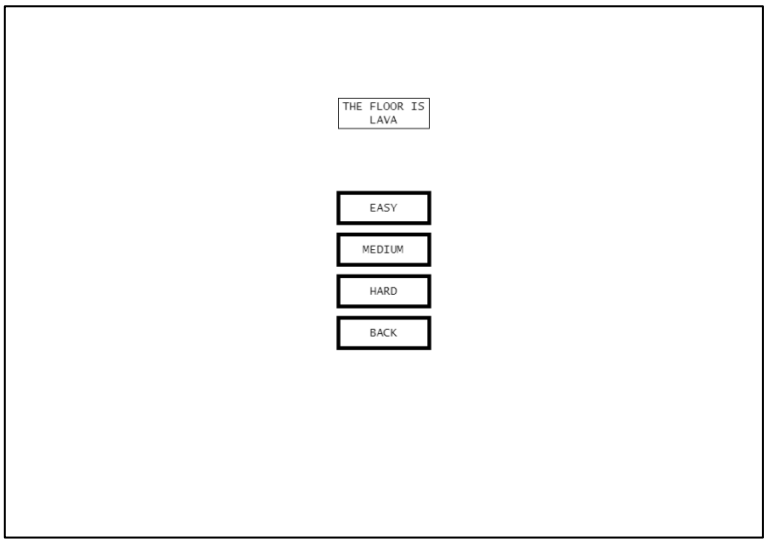


Here is an abstract diagram illustrating the options for the main menu in the previously chosen format.

As well as the main menu, I also need to consider other screens.

- The difficulty selection screen allows the user to select the difficulty at which they wish to play at.
- It is triggered when the user clicks on the “PLAY” button in the menu above.
- I produced a difficulty selection menu design based on the design above and proposed this to the stakeholders.

This design is shown on the right. Below are their responses.



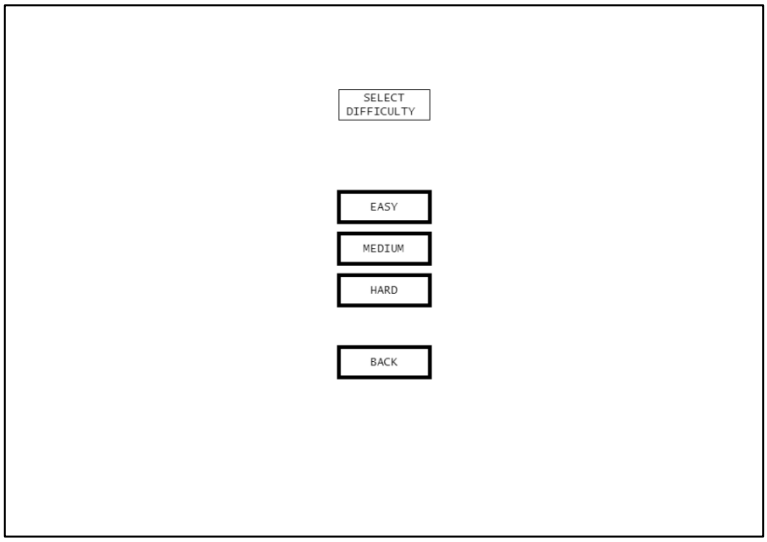
- [redacted]: “I like this design a lot because it maintains consistency to the previous menu, rather than looking random and disorganised.
- [redacted]: “This menu is great 4because it clearly communicates what all the buttons do.”
- [redacted]: “I like how you’ve kept the idea of hierarchy, with ‘easy’ at the top and ‘hard’ at the bottom.”
- [redacted]: “Including the option to go back was definitely a good idea; kudos for that. However, the user doesn’t exactly know their whereabouts in the system – could you add something which tells them that they’re currently selecting difficulty?”

- [redacted]: “You should probably separate the “back” button from the difficulty buttons, because they’re in different categories.”

I redesigned the difficulty menu with these considerations in mind, as shown here, on the right.

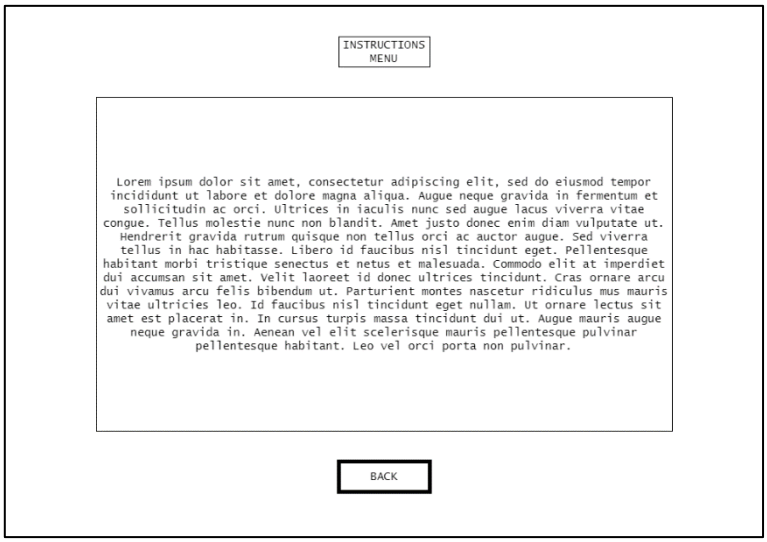
In this version, I added a slight separation between the ‘back’ button and the other options, and switched the title for ‘SELECT DIFFICULTY’.

The stakeholders all preferred this version of the menu, agreeing that it’s more user-friendly and clear.



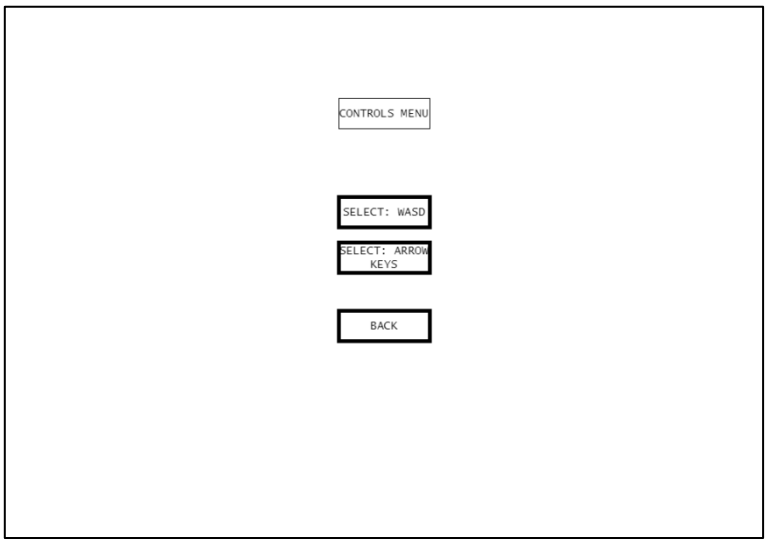
Next is the ‘instructions menu’. This menu is unique, since it features a large textbox, instead of a small one.

- This menu is accessed after clicking the ‘INSTRUCTIONS’ button in the main menu.
- It provides information on how to play.
- In this screenshot, the large textbox is filled with lorem ipsum text – in the final design, this will contain a text description on how to play the game.
- The ‘BACK’ button allows the user to return to the main menu.
- All stakeholders approved of this design.



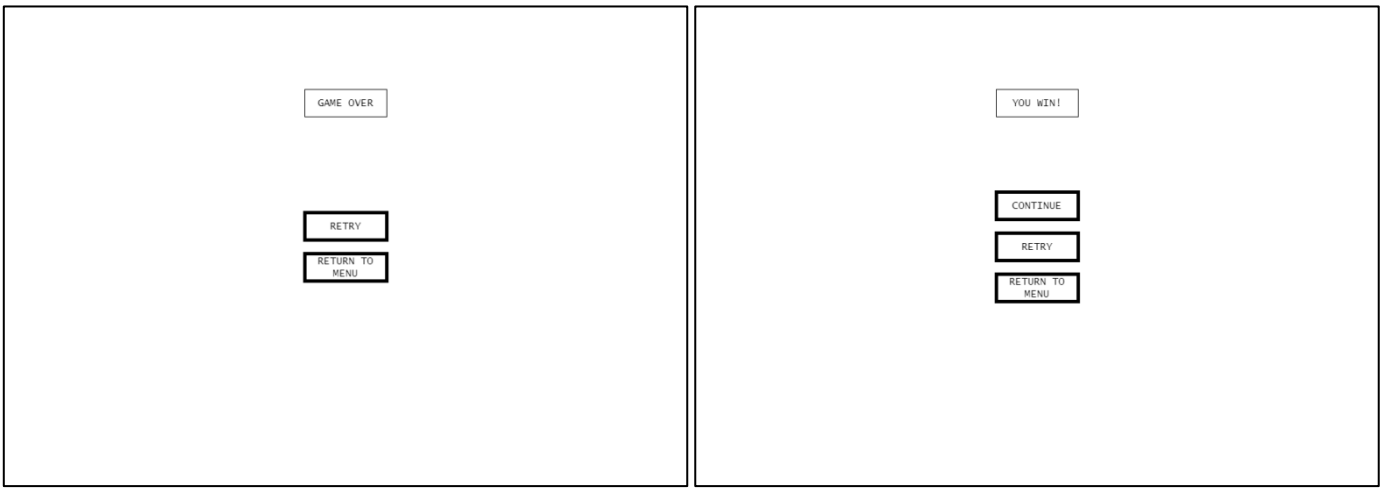
The next screen is the controls screen. This allows the user to select their control scheme, either WASD or the arrow keys. This menu is shown below.

- This menu adopts the previous idea of separating the ‘BACK’ button from the main centralised options from that screen.



- By clicking on either buttons, users can switch their control schemes between the two, according to their own comfort.

Finally, I have to create designs for the game over screen and winning screen.



- On the left is the 'game over' screen; on the right is the winning screen.
- These screens are unique – they are triggered depending on the outcome of a game.
- If the player loses, regardless of difficulty, they are presented with the game over screen. (The 'RETRY' button will not give them the same maze, but a maze of the same difficulty instead.)
- If the player wins, they will be presented with the winning screen. This screen gives the option to continue – this allows them to generate a maze of the same selected difficulty. (In this case, the 'RETRY' button WILL give them the same maze.)
- The 'RETURN TO MENU' button simply returns the player to the main menu.

<u>Sound design</u>

I asked the stakeholders where the best places to include sound are. Their responses are recorded below:

- [redacted]: "I think it's good to include sound when the user wins/dies. This is good because different sound effects can be used here, which can give the user some sense of achievement when they win."
- [redacted]: "It would be nice if you included a sound effect when people click on something."
- [redacted]: "It's a good idea to include some small sound effect when the user clicks on something, since this can let them know that their selection can be processed. Furthermore, I think it's important to include some kind of sound effect when the timer is running low, since this will alert the user to the fact that they have to make a decision soon."
- [redacted]: "I think you should include a sound effect when the user is consumed by lava, to show that they've lost the game."

- [redacted]: “Including a sound effect on start-up would be great.”

I will be taking all of these ideas into account and including sound effects as specified.

Due to time constraints, I cannot design all of the sounds on my own. Instead, I will be making use of an open source sound library, so I don’t run into copyright issues. I will be using opengameart.org’s sound effects library.

I showed my stakeholders some sound effects from here – [redacted] critiqued some sound effects, either for being inappropriate for the game, annoying to hear repetitively or just too long. Eventually we came across a collection of 11 sounds called ‘UI sound effects pack’. The stakeholders really liked four of the sounds from the pack.

These sounds were grouped in two; one pair was said by [redacted] to be appropriate for ‘going forward and back in the menu’, while the other pair was appropriate for ‘the winning and losing screens’. All other stakeholders agreed with her, stating that these sounds all had a consistent theme to them.

[redacted] identified a last pair of sounds (from the same pack) that could be played when the users entered/exited the game.

Finally, we found a ‘ticking’ sound effect from a different pack of sound effects for the timer. Initially, I proposed having the ticking playing whenever the timer is active; [redacted] disagreed with this, saying this would be irritating. Instead, he proposed that this sound will be played whenever there is less than a third of the time left on the timer. The other stakeholders agreed.

My stakeholders all really liked these sound effects, and agreed that they were appropriate for their designated uses. The points at which sound effects are played are collated below.

- When the user enters the game: play “entry”.
- When the user exits the game: play “exit”.
- When the user selects a menu/option: play “menu forward”.
- When the user clicks the ‘back’ button on any menu: play “menu back”.
- When the timer is at less than a third remaining: play “ticking”.
- When the player loses: play “you lose”.
- When the player wins: play “you win”.

<u>Designing the timer</u>

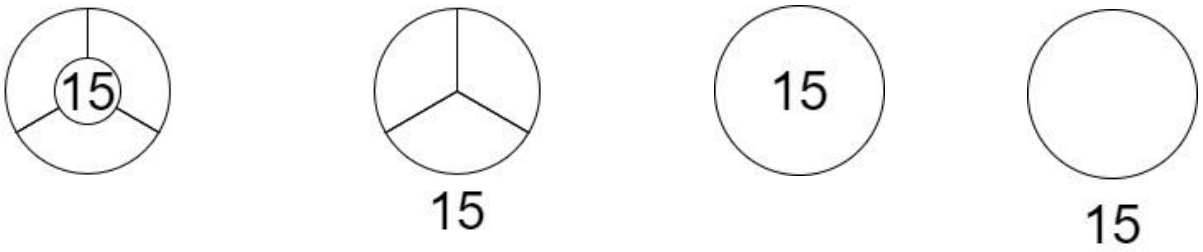
As decided earlier, the timer will start making a ‘ticking’ noise when it falls beneath a third of the time remaining. As a result, I decided it would be a good idea to design the timer in segments of three, as a circle. This design is shown on the far right.

Here are the stakeholder responses to this design:

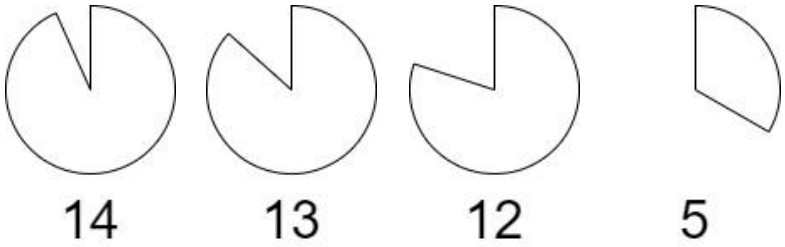
- [redacted]: “I think its good idea, but really doesn’t tell the user how much time they have left at all. Maybe if you could add numbers that say how much time is left, that would help.”
- [redacted]: “I like how it looks, but it doesn’t seem to be very informative.”
- [redacted]: “This design is really good because it makes it clear when the user has one-third of their time left, but doesn’t really tell how much time they have as a numerical value.”

- [redacted]: “I like how it’s split into segments, but I think it would be better if it was interactive – like, the circle shrinks as the time decreases. That would be a lot better than just being split into thirds.”
- [redacted]: “You need to add numbers to help the player actually know how much time they have.”

Based on this feedback, I produced four more designs for the timer – two based on the original timer (left), and two based on [redacted]’s idea (right).



All of these newer designs feature a displayed number, indicating the number of seconds the user has before their death. However, with the design on the left, the circle ‘shrinks’ (examples shown below) as the time reduces, proportional to the time expended and time remaining.



The above illustrations show what the third design would look like while time is being expended, with the remaining time at the bottom. I asked the stakeholders which set of designs they preferred – all answered that they preferred the second set of designs, with the shrinking circle. I asked which of these they preferred and why:

- [redacted]: “I prefer the design with the timer outside of the circle, because it’s clear and simple.”
- [redacted]: “I prefer these designs over the previous designs because they provide more information on how much time is left. The version with the number inside looks better.”
- [redacted]: “The design with the number in the middle is super concise and minimalistic, but may look strange as the timer shrinks; the central vertex with the number on top of it would look awkward. As a result, I’m going to go with the design where the number is outside of the circle.”
- [redacted]: “I agree with [redacted], while the first design does look ‘cool’, it isn’t as practical, and the number might be quite hard to read when the timer shrinks.”
- [redacted]: “I like that you took my feedback into account. I think both timers look good, but keeping the number outside looks a lot clearer.”

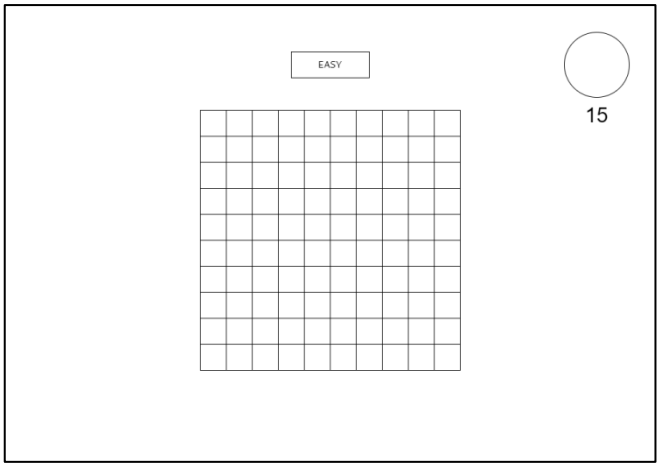
As a result, I will be using the design shown above, with the shrinking circle proportional to the amount of time left, and the number of seconds remaining directly beneath the circle.

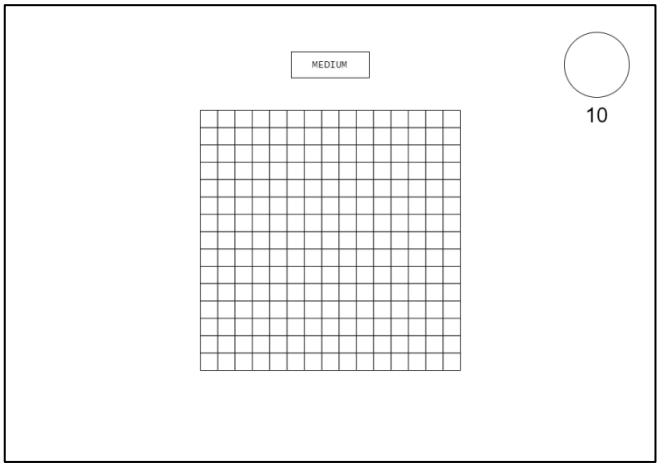
The main game

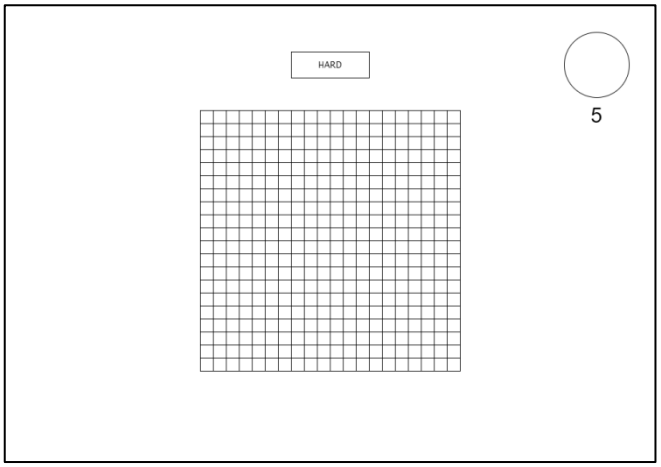
As mentioned before, the maze itself will have different resolutions and different timers, dependent on the difficulty level. These are listed below:

- Easy: 10×10 maze; 15 second timer
- Medium: 15×15 maze; 10 second timer
- Hard: 20×20 maze; 5 second timer

The basic appearance of these games is shown below. With higher resolutions, the grids of the mazes remain the same size, but the size of the individual cells shrink.







- On the top right is the design for the 'easy' game. It shows the 15 second timer, and the 10×10 grid from which the maze will be derived.
- On the right is the design for the 'medium' game. It shows the 10 second timer, and the 15×15 grid from which the maze will be derived.
- On the bottom right is the design for the 'hard' game. It shows the 5 second timer, and the 20×20 grid from which the maze will be derived.
- Please note that these designs don't show the mazes, but only the grids from which the mazes are generated.

I asked my stakeholders what they thought of these designs; their responses are recorded below.

- [redacted]: "I really like these designs because they are clear."
- [redacted]: "The timer is clear in the corner and large, so it's easy to see how much time the user has left."
- [redacted]: "I like how you haven't included anything distracting on the screen, just what's important."
- [redacted]: "These look great!"
- [redacted]: "It's good that you've included the name of the difficulty level for clarity."

Since the stakeholders are all satisfied, I will be going ahead with these ideas.

The game loop

- The game loop is a part of the program which loops infinitely (until manually stopped), updating all objects on screen, data and many more factors.
- A single loop is known as a “frame”.
- Because I specified earlier that my game runs at 60 frames per second, I will need to have a complete loop run 60 times every second.
- Firstly, I will need to determine what breaks the game loop.
- When the game is exited, I will need to break the loop in order to allow the player to leave.

3.2.2b – Algorithms

Generation of the maze

- Of course, in order for the game to actually function and be playable, there must be a solvable maze for the user.
- However, not all drawings of mazes are solvable, and prebuilding solvable mazes into the game will mean that it will quickly become repetitive, due to the lack of range and difficulty within the game itself.
- The stakeholders clearly talked about not wanting a game that is easy or repetitive, and stated that puzzle games that ‘challenge their ability’ are the most enjoyable. As a result, I need to find a way to generate solvable mazes on the spot, as the game is launched.
- To do this, I need to make use of an algorithm. More specifically, I need to make use of a maze generation algorithm. A maze generation algorithm generates solvable mazes, every time.
- I researched a wide range of maze generation algorithms, and derived the following options.
- From this, I analysed each algorithm, and produced small fact files on each one, so that I can make a choice of which is most appropriate for my project, looking at advantages and disadvantages, how they work, and ease of implementation.

Recursive backtracker

Overview: Recursive backtracker, also known as “depth-first search”, is a remarkably popular maze generation algorithm because of its simplicity and ease of implementation. It involves digging paths in a field.

How it works:

Step 1) Select a random point on the grid, noting its location on the x and y axes. Choose a random direction to move in (up, down, left, right).

Step 2) Check if two steps ahead of current location (conserving selected direction) is either outside of the boundaries, a wall, or a path. If a wall is ahead, you can set the next two cells in the direction as paths.

Step 3) Update current location.

Step 4) If a dead end is reached, backtrack (hence the name) onto previously covered path cells, and continue in the opposite direction. Update current location.

Step 5) Repeat from Step 2. If no more possible paths exist on the given plane, end.

Implementation: This algorithm is known for being fairly easy to implement. As the name suggests, it uses recursion, hence making programming the algorithm a lot easier, as it follows its steps more independently rather than relying on hard-coded instructions excessively.

Advantages:

- Fast and easy to implement
- Very good for most mazes, fairly efficient
- Produces generally unbiased mazes, for a maze of average size

Disadvantages:

- On a small scale, produces very simple mazes that are easy to solve
- Can be memory intensive for larger scale mazes, and quite inefficient
- Mazes deteriorate in speed and efficiency of generation, proportional to size

Other characteristics/info:

- Carves mazes from a grid
- Starts from a point and burrows its way around
- Generates fewest dead-ends, resulting in more “long, winding” passages

Eller’s algorithm

Overview: Contrary to the previous algorithm, Eller’s algorithm is much more difficult and complex, but is remarkably fast, even for larger mazes.

How it works:

Step 1) Assign all cells in the first row into individual sets. These sets can be labelled with numbers.

Step 2) Join adjacent cells randomly, ONLY if they aren’t in the same set. Once two adjacent cells are joined, merge their sets into one set. Assign them all under a single numerical value; they are now the same set.

Step 3) For every set, randomly select and create connections to the row beneath, so that every set has at least one connection to the row below. As a result, the cells in the row below will be part of the same set (and share the same numerical value) as the cells that they are connected to directly above.

Step 4) Assign new sets to any remaining cells in the next row down, thus creating the process again.

Step 5) Repeat until the last row is reached.

Step 6) In the final row, join all adjacent cells that are parts of different sets, and eliminate vertical connections.

Implementation: This algorithm is much harder to implement than the previous. I’ll need to focus on building sets and understanding them, as well as making sure that only sets that are non-identical are allowed to merge.

Advantages:

- Very fast

- Can work for theoretically infinite mazes efficiently, hence making it very suited for large mazes
- Only needs to examine one row at a time, making it less demanding for a system

Disadvantages:

- Difficult to understand and implement, very complex

Other characteristics/info:

- Carves mazes from a grid
- Reliant on sets, operates in rows, travelling downwards
- Unbiased and generates a combination of dead-ends and long paths

Kruskal's algorithm

Overview: This is an algorithm that is based off of graph theory, and that I have had experience with in the past. In graph theory, it is used to generate a minimum spanning tree from a given graph. The original version of Kruskal's algorithm would select edges with the lowest weight (value), but the version for a maze will need to select edges randomly, in order to produce and generate mazes uniquely every time.

How it works:

Step 1) Start with a grid, all the lines are called "edges". Create a list of every existing edge.

Step 2) Create a set for every cell.

Step 3) If two adjacent cells belong to different sets, remove the wall between the two cells, and join them, also merging them into a single cell.

Step 4) Repeat step 3 until maze is complete, and all cells belong to the same set.

Implementation: Implementing Kruskal's algorithm is not particularly difficult, but doing it efficiently and optimising sets is difficult, almost as difficult as the implementation of Eller's algorithm.

Advantages:

- Produces generally unbiased mazes
- An easy to understand algorithm, I have prior experience

Disadvantages:

- Difficult to implement efficiently, due to sets

Other characteristics/info:

- Makes use of sets and dividing walls/merging
- Produces mazes with lots of dead-ends

Prim's algorithm

Overview: Another graph theory algorithm, also originally created to find the MST of a given graph. However, Prim's algorithm works outwards from a random point, instead of breaking walls between sets.

How it works:

Step 1) Choose a random point in the grid. All walls of this cell are added to a list of walls.

Step 2) While there are walls in the list, pick a random wall from the list and check the cell on the other side of this wall. If the cell isn't part of the maze, remove the wall to make a path, and mark the cell as part of the wall. Add the walls of this new cell to the list of walls.

Step 3) Remove this wall from the list, run step 2 again until there are no remaining cells.

Implementation: Implementation of Prim's algorithm will need to focus on frontier cells, as a set. Cells and walls need to be classified into lists, in order for the algorithm to know what to do next. Other than that, implementation is very easy.

Advantages:

- Easy implementation
- An easy to understand algorithm, I have prior experience

Disadvantages:

- Doesn't hold any huge advantages over other maze algorithms

Other characteristics/info:

- Makes use of sets, grows from a point outwards
- Produces mazes structurally similar to Kruskal's, since they are both MST algorithms, with lots of dead-ends

<u>Recursive division</u>

Overview: Unlike any other algorithm we've looked at so far, this algorithm starts with a blank space and builds walls, rather than carving paths or destroying walls between sets in a grid.

How it works:

Step 1) Start with a blank space/field.

Step 2) Draw a line from one end of the field to the other, using either a horizontal line or a vertical line. This is our wall.

Step 3) Break a single passage in the wall.

Step 4) Repeat step 2 with the remaining areas, until maze reaches desired resolution.

Implementation: This algorithm is recursive, meaning it is easier to implement, but doesn't use defined sets or a grid. Overall, this could be either easy or hard to implement, depending on how well I write my code.

Advantages:

- Can be used for theoretically infinitely-sized mazes
- Very easy to understand
- Mazes can be made as detailed or difficult as desired
- Has a huge range of possibility with difficult adjustments and levels

Disadvantages:

- Doesn't use a grid
- Could potentially be difficult to implement

Other characteristics/info:

- Builds walls without using a grid, but rather from empty space
- Infinite detail/scale

Aldous-Broder algorithm

Overview: This algorithm also draws from graph theory, but instead investigates spanning trees. I have no prior experience with this one.

How it works:

Step 1) Select a random cell on a grid. Mark this as the current cell. Add it to the list of visited cells.

Step 2) Randomly choose any adjacent cell from the current cell, and make this the current cell. If the current cell is not in the list of visited cells, add it to the list of visited cells, and remove the wall between the current cell and the previously visited cell.

Step 3) Repeat step 2 until all cells are in the list of visited cells.

Implementation: This algorithm is probably one of the easiest to implement, in part due to its simplicity.

Advantages:

- Very simple
- Very easy to implement

Disadvantages:

- Painfully slow
- Painfully inefficient
- Can technically run forever, in worst case scenario

Other characteristics/info:

- Uses a grid and cells
- Uses moving out from a certain point, one cell at a time

Wilson's algorithm

Overview: Similar to the previous algorithm, this algorithm was originally based on USTs (uniform spanning trees). However, this one is slightly more effective than the Aldous-Broder algorithm, and is more likely to finish faster.

How it works:

Step 1) Choose a random cell and add it to the list of visited cells.

Step 2) Select randomly a vertex that has not already been visited and randomly walk until you visit a cell that is in the list of visited cells.

Step 3) Add all cells visited over the walk to the visited cell, along with their paths. Maintain walls that have not been traversed across directly.

Step 4) Repeat steps 2 and 3 until all cells have been added to the list of visited cells.

Implementation: This algorithm is pretty hard to implement, because you have to keep track of the random walk, and all the nodes visited, as well as paths. Because of this, it can be quite excessive in terms of data.

Advantages:

- Effective

Disadvantages:

- Very inefficient
- Inconsistent, as it can take a while to find a white space

Other characteristics/info:

- Uses a grid, burrows it's way around
- Is unique in approach, using a random walk that takes place away from a target point

<u>Hunt and kill</u>

Overview: This algorithm is much more similar to recursive backtracker, but instead of backtracking, features a unique "hunt mode" to find new blank cells upon hitting a dead end.

How it works:

Step 1) Choose a random starting cell in the grid.

Step 2) Walk randomly, creating passages to unvisited neighbours, until the current cell has no unvisited neighbours, and hence nowhere to go. Enter hunt mode.

Step 3) In hunt mode, scan the entire grid vertically, until a cell is found that is unvisited, while next to a visited cell. When found, carve a passage between the two cells, let the unvisited cell be the new starting point.

Step 4) Repeat steps 2 and 3 until hunt mode scans the entire grid and finds no unvisited cells.

Implementation: This algorithm is pretty easy to implement, and makes use of two loops, the walk (kill) and the search (hunt).

Advantages:

- Efficient and quick
- Makes use of hunt mode as a quick way of finding new cells

Disadvantages:

- Inferior to recursive backtracker in some situations

Other characteristics/info:

- Similar to recursive backtracker, produces few dead-ends, and hence more "long and winding" passages.
- Features "hunt mode", unique to this algorithm

Growing tree

Overview: This algorithm is unique because it can be configured in certain ways to mimic behaviour of other algorithms.

How it works:

Step 1) C is a list of cells, empty to begin with. Add any random cell to list C.

Step 2) Choose a cell in C and carve a passage to any unvisited neighbour of this cell. If this cell has no unvisited neighbours, remove the cell from C.

Step 3) Repeat step 2 until list C is empty.

Implementation: This algorithm is incredibly simple to implement, and can be implemented to behave like recursive backtracker or Prim's algorithm, depending on how cells are selected from list C. (If the newest cell is chosen, the algorithm will behave like recursive backtracker. If a random cell is chosen, the algorithm will behave like Prim's algorithm.)

Advantages:

- Remarkably easy to implement
- Huge range of customisability in cell selection methods

Disadvantages:

- Fast, but not as fast as other algorithms

Other characteristics/info:

- Customisable cell selection methods, different methods can even be used in conjunction with each other

Binary tree

Overview: This algorithm is unique in the sense that all it can build the entire maze while only reading one cell at a time, individually.

How it works:

Step 1) For a cell in grid, randomly carve either north or west. Repeat until maze is complete.

Implementation: This algorithm is widely considered the easiest to implement out of all the algorithms on this list. It features only one real command, in a loop, and repeats until a maze of the desired resolution is complete.

Advantages:

- Incredibly easy to implement
- Ridiculously easy concept

Disadvantages:

- Huge directional bias

Other characteristics/info:

- This algorithm can be tailored to randomly carve north/west, north/east, south/west or south/east
- This algorithm has bias in the direction of how the maze was carved. As a result, traversing the maze in the same direction as the maze generation is trivially easy; traversing in the opposite direction is much more difficult

Sidewinder

Overview: This algorithm allows for taller mazes, while being quick and easy to implement. Related to Binary Tree.

How it works:

Step 1) Start at cell (0, 0). Create a “run” set.

Step 2) Add the current cell to the “run” set. For any current cell, randomly decide to carve east or not.

Step 3) Randomly choose to carve east (or not) for the current cell.

Step 4) If the current cell was carved (east), make the new cell the current cell, and repeat steps 2-4.

Step 5) If not, choose a random cell in the “run” set, and carve north from this cell. Empty the “run” set.

Step 6) Repeat steps 2-5 until all cells have been processed.

Implementation: The concept of run sets means I will likely have to make use of data structures like lists or arrays. This implementation isn’t particularly difficult, and I will need to make use of ‘if’ statements to check conditions in steps 4/5.

Advantages:

- Easy implementation
- Allows ‘tall’ mazes

Disadvantages:

- Holds few advantages over Eller's algorithm and Binary Tree
- All mazes generated will have vertical bias

Other characteristics/info:

- Travelling from north to south will mean a user will not run into dead ends

I have to make a choice of which maze algorithm I will use.

- Immediately, I am going to be eliminating the Aldous-Broder algorithm and Wilson's algorithm. My reasoning behind this is that both of these algorithms are unacceptably inefficient, and grant no guarantee that they will even finish in a reasonable amount of time.
- Next I will look at how well algorithms suit themselves to my game. I will eliminate recursive division because it doesn't use a grid, and hence isn't suitable to my previously decided class setup.
- I will also eliminate heavily biased algorithms, this is because they are too easy to traverse for the user and as a result won't be a challenge for the user. This means that Sidewinder and Binary tree will not be used.
- I will not be using Eller's algorithm because of how difficult it is to implement – this is a limitation of my solution. Had I had more time, I likely would've used this algorithm.
- I will not be using Prim's or Kruskal's algorithms, because they are both biased towards dead-ends.
- This process of elimination has left me with three possible algorithms: Recursive backtracker, Hunt and Kill, and Growing Tree. These three algorithms are very similar in nature, and are all fairly easy to implement.
- Because of how flexible and easy to implement Growing Tree is, I shall be using this algorithm. Its ease of implementation means that I can spend more time working on other parts of the program, rather than worrying about the maze algorithm, and its flexibility allows me to produce mazes more suitable for my game.
- Below is the pseudocode for the Growing Tree algorithm, for a 10 × 10 grid.

```
array grid[9, 9]
list C[]
// empty cells in the grid are represented by a '?' symbol
// carved cells in the grid are represented by a '#' symbol
for i = 0 to 9
	for j = 0 to 9
		grid[i, j] = "?"
	next j
next i
// the above simply fills the grid with blank spaces
function carve(grid, C) // this function carves out the entire grid
	x = random.integer(0, 9)
	y = random.integer(0, 9) // these are inclusive
```

```
        while grid[x, y] != “?” then // checks if the space is still an empty cell
                x = random.integer(0, 9)
                y = random.integer(0, 9)
        endwhile
        C.append((x, y)) // appends the location of the random cell to C
        grid[x, y] = “#” // makes this cell carved
        while length(C) != 0
                current_cell = C[length(C)-1] // stores the location of the last cell in C
                a = current_cell[0] // stores the first part of the index
                b = current_cell[1] // stores the second part of the index
                z = random.integer(1, 4) // inclusive – chooses a direction to carve in
                if z == 1 and grid[a+1, b] == “?” then // checks if a neighbour  is above
                        C.append((a+1, b))
                elif z == 2 and grid[a-1, b] == “?” then // checks if a neighbour  is below
                        C.append((a-1, b))
                elif z == 3 and grid[a, b+1] == “?” then // checks if a neighbour  is right
                        C.append((a, b+1))
                elif z == 4 and grid[a, b-1] == “?” then // checks if a neighbour  is left
                        C.append((a, b-1))
                elif grid[a+1, b] != “?” and grid[a-1, b] != “?” and grid[a, b+1] != “?” and
        grid[a, b-1] != “?” then
                        C.pop() // removes the last cell in C – it has no neighbours
                else
                        z = random.integer(1, 4) // tries again with another random direction
        if length(C) == 0 then // when C is empty, the algorithm is complete
                return grid
endfunction
carve(grid, C)
```

For the pseudocode above, this only works if the grid is visualised – the cells do not become walls, but the spaces between the cells do. The code serves for visualisation purposes rather than a practical implementation. I will probably have to adapt this in the final code; this is a very basic demonstration of how the algorithm is implemented.

Sorting algorithms

As well as the maze algorithms, I also need to consider sorting and searching algorithms, to manage my data.

Below are some examples of sorting algorithms I may want to use in my project:

Algorithm name:	Description:	Type:	Order (worst case):
Bubble sort	Move through the list repeatedly, swapping adjacent items in a list as appropriate until the list is sorted.	Comparison sort	$O(n^2)$, where n is the length of the list.
Shuttle sort	Same as bubble sort, but reset to start the start of the list with every swap.	Comparison sort	$O(n^2)$, where n is the length of the list.
Quick sort	The first value becomes a 'pivot'. Write all values smaller than the pivot in one sub-list, and all values greater on another. Apply this procedure to all sub-lists, until there only exist sub-lists with one entry; the elements are now sorted.	Divide-and-conquer	$O(n^2)$, where n is the length of the list.
Insertion sort	Builds the sorted list by inserting one element at a time into its correct position on the list.	Comparison sort	$O(n^2)$, where n is the length of the list.
Merge sort	Split a list of elements into individual parts, comparing adjacent parts and sorting as appropriate, before merging. Repeat this process until the entire list is sorted.	Divide-and-conquer	O(n log n), where n is the length of the list.

I will likely need to use these algorithms when sorting data in data structures (e.g. for the generation of my grid, I will need a sorted list of cell coordinates.)

- I will not make use of insertion sort, shuttle sort or bubble sort. This is because, while they are amenable to small lists, I will likely be handling larger lists, potentially with 625 items (if my grid was a 25 by 25 grid). These algorithms are all slow and inefficient for lists of these sizes.
- Instead, I will make use of quick sort and merge sort. Quick sort is fast and has an average performance of O(n log n), while merge sort has a worst case performance of O(n log n); both of these being faster than $O(n^2)$. They are also more suitable to larger lists.

<u>Searching algorithms</u>

I will need to search through data structures to find and identify a particular element/value within them. To do this, I will need to use searching algorithms.

Algorithm name:	Description:	Type:	Order (worst case):
Linear search	Traverse the list from left to right, looking for the desired value until it is found. If it isn't found, it is not in the list.	Search	O(n), where n is the length of the list.
Binary search	The list must be sorted into ascending order before this algorithm is applied. Select the middle element of the given list, and compare it with the desired value. If the middle value is greater than the desired	Search	O(log n), where n is the length of the list.

	value, eliminate all values on the right. If the middle value is less than the target value, eliminate all values on the left. If the middle value is the target value, the target value has been found. Repeat until the target value is found. If it isn't found, it is not in the list.		

- While binary search is undoubtedly faster than linear search, it requires the list to be sorted first.
- My lists are not sufficiently large enough that using binary search instead of linear search (and hence sorting the list first) is worth using – this will barely save any time, and may actually be slower than simply using linear search,.
- As a result, I will be using linear search for searching lists – binary search simply isn't useful for my system.

All of these algorithms will help me build up the solution, because they make the problem more easily and efficiently solvable, and relate to almost all aspects of the game. They also guarantee that jobs (e.g. maze generation, data sorting) will be finished within a reasonable amount of time, meaning the user will not have to wait long periods for the computer to finish processing elements.

3.2.2c – Usability features

Here I will discuss features of the final solution that make it easier for the user to use. I need to focus on consistency, to make sure the system is easy to use for a user.

<u>The tutorial screen</u>

As aforementioned, the final product will feature a tutorial screen, which explains how the game should be played. This is done by clarifying controls, explaining the objective of the game, and the threat entity and timer.

The tutorial screen text is as follows:

"Welcome to 'The Floor is Lava.' You are trapped in a maze, with only one exit. You can traverse through the maze freely, but watch out; deadly lava spawns in your wake, meaning once you make a move, there's no going back. You can move up, down, left or right, but be warned… stall for too long, and you'll be consumed by the deadly heat.

Move using WASD or the arrow keys, depending on your selected control scheme. Your control scheme can be changed in the controls menu. You can select your difficulty according to your comfort and skill level, and keep an eye out for the timer on the top right – this will tell you how much time you have left before you meet your end. Good luck."

I asked my stakeholders about their opinion on the above text – here are their responses:

- [redacted]: "Wow, I really like how descriptive this text is, while still being slightly creepy. Nice work!"
- [redacted]: "It provides all the information needed to play the game."
- [redacted]: "I like how you've split it into two parts, the first explaining the theme, and the second explaining the technical side."

- [redacted]: “Really eerie, it’s good that you’ve explained the premise of the game AND the controls.”
- [redacted]: “Good job, fits into the theme really well.”

As a result, this text will be used in the final game, for the tutorial screen.

Font

For the game, I decided that the inclusion of only two fonts was a good idea; this was because I wanted to keep the game simple and easy to understand for the player. The ‘secondary font’ will be used for headings and titles, while the ‘primary font’ will be used for information boxes and larger bodies of text.

I selected a sample of 10 fonts from the Pygame library, based on how thematically appropriate they are, in my own opinion. These are shown below:

- Segoe UI 15
- **Segoe UI Black**
- AMD RTG
- Gadugi
- Centaur
- **STENCIL**
- **Impact**
- Ebrima
- **Snap ITC**
- Lucida Fax

I asked my stakeholders what they thought the fonts should be, based on readability and how well they fit into the theme. Their answers are recorded below:

- [redacted]: “Both ‘Segoe UI Black’ and ‘Impact’ are really good for the title text, because they’re eye-catching and fit the theme well. ‘Gadugi’ is my favourite font for the body text because of its simplicity.”
- [redacted]: “My favourite font for the heading text is ‘Impact’ because it really puts emphasis on the title. My favourite font for the body text is ‘Segoe UI’ because it’s simple, and easy to read.”
- [redacted]: “I think ‘AMD RTG’ is super unique, but really not suited to body text at all. It would be excellent for title text, because it is fits the theme really well. For the body text, I think ‘Segoe UI’ is the best option, because it’s readable and simple.”
- [redacted]: “I think the combination of ‘Segoe UI Black’ for title text and ‘Segoe UI’ for body text will look be perfect for the user, since they are both similar fonts and look super good together. They’re also very readable, and ‘Segoe UI Black’ fits the ‘square’ theme really well.”
- [redacted]: “For title text, ‘Segoe UI Black’ would be great because of it gets the user’s attention. For body text, ‘Ebrima’, ‘Segoe UI’ or ‘Gadugi’ are all appropriate; they’re all readable and simple.”

From all of this, I have decided that ‘Segoe UI’ is the most appropriate primary font (for large bodies of text and information). This font was hugely popular amongst stakeholders because of its simplicity and readability.

For the secondary font (for titles), I have decided that 'Segoe UI Black' is the best option. Like mentioned by [redacted], when used in conjunction with 'Segoe UI', it will maintain consistency in the games aesthetic, since they are from the same family of fonts. Furthermore, this font is super easy to read.

I will import these fonts using the 'pygame.font', the Pygame module for loading and rendering fonts.

Colour

I already mentioned colours in the previous section and made some design decisions.

- The main colour of the player object will be green.
- The main colour of the lava will be red.
- The main colour of the background will be black.
- The main colour of the maze will be white.
- The main colour of the timer will also be varying between green, yellow and red, depending on the amount of time the player has left until immediate death.

White will allow the player to clearly see the maze, and hence see where they can travel/traverse, and where they can't. It'll also allow them to easily assess how far away a threat object is.

The timer's colour will vary according to how much time the player has left. It will be green if the player has over two-thirds of their allocated time left, yellow if they have between a third and two-thirds, and red if they have under a third. I've done this because red is a colour that represents an alert, and the player needs to be made aware if they have little time left to move.

Black as a background allows all foreground colours (red, green, yellow, white) to be easily visible, since they all contrast the background well. It also isn't distracting from the actual content on-screen. It also fits into the theme of 'fire/lava' very well, much better than white.

I also need to consider colours of UI elements. I asked the stakeholders what colours they believed to be most appropriate for different UI elements (the title, the textboxes and the buttons). Their responses are recorded below.

Title(s):

- [redacted]: "The title needs to grab the user's attention, so it should probably be a primary colour. I think red is the best colour, since the title is "The Floor is Lava", and hence red is the most appropriate thematically."
- [redacted]: "The title should be fiery colours, like red, orange and yellow, since they fit the theme of fire and lava."
- [redacted]: "The title should be red, because it's bold and attention-grabbing."
- [redacted]: "The title should be white, because it'll stand out from the black background. Alternatively, red or yellow would be a good idea, seeing as they stand out just as well."
- [redacted]: "Using white for the title is a good idea, because it's a stark contrast from the black background."

From this, I selected red to be the title. This is because red is a very bold primary colour, and fits the theme perfectly.

Textboxes:

- [redacted]: “I think keeping the textboxes and buttons red is a good idea, because then they’ll all stand out and fit the theme.”
- [redacted]: “I think green is a good idea for the textboxes, since it matches the character in the game.”
- [redacted]: “You should use white, with black text, for the textboxes. These will stand out and can’t be missed by the user.”
- [redacted]: “Try yellow/orange for the textboxes – this will establish some feel of ‘hierarchy’ to the UI elements, where red is the most important, then orange, then yellow… all while maintaining the ‘fire’ theme.”
- [redacted]: “I think orange should be used for the textbox, since it continues the fire theme.”

The other stakeholders really liked [redacted]’s idea of using orange/yellow for the textboxes. From this, we decided that the textboxes should be yellow, since they are less important than buttons – buttons are interactive, and hence need more attention from the user.

Buttons:

- [redacted]: “I think orange should be used for the buttons, based on [redacted]’s idea.”
- [redacted]: “I agree with [redacted].”
- [redacted]: “I agree with [redacted].”
- [redacted]: “Use orange for the buttons if you’re going to use yellow for the textboxes.”
- [redacted]: “Orange/yellow is the best option based on the previous idea.”

As a result, orange will be used for the buttons.

Details of textboxes and buttons

Textboxes need to be distinguishable from buttons in order to be recognisable for the user, and so the user can know what to click on.

- Buttons will have a thick border, while textboxes do not.
- When clicked, buttons will ‘flash’ – their colours and text will invert, in order to indicate that they have been clicked.
- The font will be Segoe UI, as they are not headings or titles (title textboxes will make use of Segoe UI Black). These are in accordance with previous stakeholder decisions.
- Both the buttons and the textboxes will have black text, since it contrasts their respective colours well, and are readable.
- Width:height ratio of 3:1 for buttons and textboxes (containing small information.)
- All of these decisions are collated in the table below.

Entity	Size	Colour	Font	Text colour	Shape	Reasoning
Titles	Large	Red	Segoe UI Black	Red	N/A (Text shape)	Is clear, unique and attention-grabbing, while maintaining the theme.
Subtitle textboxes	Small	Yellow	Segoe UI Black	Black	Rectangle; ratio 3:1	Bolder font than regular textboxes allows them to be viewed as more important.
Information textboxes	Small*	Yellow	Segoe UI	Black	Rectangle; ratio 3:1*	Maintains a consistent theme for the entire game (shape and font).
Buttons	Small	Orange	Segoe UI	Black	Rectangle; ratio 3:1	Maintains a consistent theme for the entire game (shape and font).

*dependent on content involved

Above are two monochromatic images of a button and textbox side-by-side. They have been designed in this way in order to be distinguishable for the player. As specified by the previous table, the final designs will have their font as 'Segoe UI' (as decided previously), black text, and will be coloured orange for buttons, and yellow for textboxes.

I asked my stakeholders what their opinions are on these designs. Their responses are recorded below.

- [redacted]: "I really like these designs because they show a clear distinction between what I can click on and what I can't."
- [redacted]: "I like the idea of the button 'flashing' when clicked on, this is a really nice feature and helps the user know when they've clicked on something."
- [redacted]: "I like how black text has been used on a yellow background (for the textboxes), because white text is really hard to read on yellow backgrounds. Black is much more readable."
- [redacted]: "These are good, keep it up."
- [redacted]: "I like how you've kept the fonts and colours consistent for the textboxes and buttons, but changed the border to make them distinguishable, since it keeps the overall theme, but still makes it clear to the user as to what is what."

For the previously shown UI designs in 2.2.2a:

- Text item "THE FLOOR IS LAVA" in the main menu is a title, and hence employs the aesthetics described in the table above.
- Text items "SELECT DIFFICULTY", "INSTRUCTIONS MENU", "CONTROLS MENU", "YOU WIN!" and "GAME OVER" (from their respective screens) are all subtitle textboxes, and hence employ the aesthetics described in the table above.
- The large textbox on the instructions menu is a large information textbox, and hence employs the aesthetics described in the table above. However, it does not maintain the 3:1 ratio, because it carries a large body of text.
- All buttons showed in the aforementioned UI designs employ the aesthetics described in the table above.

Adjustable controls

As aforementioned, the user will be given the option to choose between using the WASD controls or the arrow keys. This is done to allow the user to have their own playstyle and play the game in a way that is comfortable for them, rather than being forced to play the game one way.

Adjustable difficulty

This allows the user to choose their own difficulty level, making the game fair for them; it lets them choose how difficult they wish for the game to be, to let them understand the game at their own rate. By doing this, I also avoid 'scaring off' players early – they can select the lower difficulties in order to slowly get used to the game, before allowing them to tackle the harder difficulties when they feel confident.

3.2.2d – Data structures and data

Naming conventions

As previously mentioned (in the section on classes), different variables, constants, data structures, methods, functions and procedures need to be appropriately named, to make further modification easier for me as a programmer. In order to ensure appropriate naming, I will need to consider naming conventions.

When naming data items, I will have to ensure they are all named consistently, to help allow code to be understandable and readable. Spaces are not allowed between the names of data items; they need to be changed so that the computer can understand them. When dealing with combining several words into a single string, there are four options:

- Camel case: Remove the space, capitalize the first letter of every word, not including the first word. An example would be “camelCase”.
- Pascal case: Remove the space, capitalize the first letter of every word, including the first word. An example would be “PascalCase”.
- Snake case: Replace all spaces with underscores. No capital letters. An example would be “snake_case”.
- Snake case (all caps): Replace all spaces with underscores. Make all letters capital. An example would be “SNAKE_CASE”.
- Kebab case: Replace all spaces in the phrase with a dash. No capital letters. An example would be “kebab-case”.

There is no single ‘best’ naming convention – it is really all down to preference. However, it is important that all data items are named in the same manner, in order to make the code easily modifiable for the future iterations.

I cannot use kebab case, because the dash symbol is an operator in Python, and hence cannot be used without causing errors. In a lot of other programs I have created in Python, I have made use of snake case, and hence am preferential towards this convention. As a result, I am going to use snake case as my naming convention for all data items.

Data dictionary

Below is a table of all known variables and constants that are to be used in my program.

Name of data:	**Data type:**	**Example:**	**Validation rules:**	**Purpose/justification:**
x_pos	Variable (integer)	x_pos = 20 (the x position of the current object is 20 pixels across from the top-left corner.)	Only accepts positive integer values.	Used for the measurement of the x-coordinate of some object. Named as such.
y_pos	Variable (integer)	y_pos = 20 (the y position of the current object is 20 pixels down from the top-left corner.)	Only accepts positive integer values.	Used for the measurement of the y-coordinate of some object. Named as such.
width	Variable (integer)	width = 20 (the width of the	Only accepts positive	Used for the width, in pixels, of some object.

		current object is 20 pixels.)	integer values.	Named as such.
height	Variable (integer)	height = 20 (the height of the current object is 20 pixels.)	Only accepts positive integer values.	Used for the height, in pixels, of some object. Named as such.
alive	Variable (Boolean)	alive = True (the player is currently alive.)	Only accepts Boolean values (True or False).	Used to determine whether the player is alive or dead (if True, the player is currently alive; else, they are dead.) Named as such.
touching	Variable (Boolean)	touching = False (the current object is not touching the other object.)	Only accepts Boolean values (True or False).	Used to determine whether or not an object is touching a particular other object (if True, they are touching.) Named as such.
clicked	Variable (Boolean)	clicked = True (the current object has been clicked on by the user.)	Only accepts Boolean values (True or False).	Used to determine if a particular object is clicked or not (if True, it is clicked.) Named as such.
x_cells	Variable (integer)	x_cells = 20 (the current grid has 20 cells in its x direction.)	Only accepts positive integer values.	For a grid object, used to determine the number of cells that the grid has on its x-direction. When this value is multiplied by 'y_cells', this gives the area of the grid. Named as such.
y_cells	Variable (integer)	y_cells = 20 (the current grid has 20 cells in its y direction.)	Only accepts positive integer values.	For a grid object, used to determine the number of cells that the grid has on its y-direction. When this value is multiplied by 'x_cells', this gives the area of the grid. Named as such.
duration	Variable (integer)	duration = 15 (the current timer has an initial duration of 15 seconds.)	Only accepts positive integer values.	For a timer object, used to determine the initial duration that the timer must count down from. Named as such.
duration_remaining	Variable (integer)	duration_remaining = 10 (the current timer has a remaining duration	Only accepts positive integer values.	For a timer object, used to determine the duration remaining on the timer. Named as

		of 10 seconds.)		such.
button_name	Variable (string)	button_name = “Button” (the current button has the name “Button”.)	Only accepts string values.	For a button object, used to identify the name of the button. Named as such.
box_name	Variable (string)	box_name = “Textbox” (the current textbox has name “Textbox”.)	Only accepts string values.	For a textbox object, used to identify the name of the textbox. Named as such.
text	Variable (string)	text = “Click here!” (the current button/textbox has text “Click here!” written on it.)	Only accepts string values.	For a textbox or button object, used to determine the text written on the object. Named as such.
running	Variable (Boolean)	running = True (the current state is running.)	Only accepts Boolean values.	For a state object, used to determine whether the current state is running or not. Named as such.
current_state	Variable (string)	current_state = “Intro” (the current state name is “Intro”.)	Only accepts string values of the known states.	For a state object, used to determine the name of the current state. Named as such.
next_state	Variable (string)	next_state = “Tutorial” (the next state name is “Tutorial”.)	Only accepts string values of the known states.	For a state object, used to determine the name of the next state. Named as such.

Below is a table of all known data structures that are to be used in my program.

Name of data structure:	**Type of data structure:**	**Purpose/justification:**	**Other details:**
colour (e.g. red, orange, yellow)	Tuple (of integers)	Used for the RGB value of the colours in the game. Tuples are used because they are unchangeable after being defined. Named as such.	Tuple size: 3 items
controls	Tuple (of strings)	Used to store the four control directions, which can be either ‘[K_w, K_a, K_s, K_d]’ or ‘[K_UP, K_LEFT, K_DOWN, K_RIGHT]’, depending on the selected control scheme (WASD or arrow keys.) A tuple is	Tuple size: 4 items

		used because these are unchangeable after being defined. Named as such.	
grid_cells	List (of tuples (of integers))	Used to store the coordinates (in the form of tuples) of all cells in the grid. A list is used because they can be appended to and have a huge length, meaning I am guaranteed to be able to store all coordinates. Named as such.	Tuple size: 2 items List size: 536,870,912 items
lava_cells	List (of tuples (of integers))	Used to store the coordinates (in the form of tuples) of all the lava cells. A list is used because they can be appended to and have a huge length, meaning I am guaranteed to be able to store all coordinates. Named as such.	Tuple size: 2 items List size: 536,870,912 items
maze_cells	List (of tuples (of integers))	Used to store the coordinates (in the form of tuples) of all the maze cells. A list is used because they can be appended to and have a huge length, meaning I am guaranteed to be able to store all coordinates. Named as such.	Tuple size: 2 items List size: 536,870,912 items
walls	List (of tuples (of integers))	Used to store the coordinates (in the form of tuples) of all the walls (from start to end coordinate). A list is used because they can be appended to and have a huge length, meaning I am guaranteed to be able to store all	Tuple size: 4 items List size: 536,870,912 items

		coordinates. Named as such.	

Below is a table of all known libraries that are to be used in my program.

Name of library:	**Purpose:**
Pygame	To allow the use of visual objects on the screen and as a platform for the entire game.
time	To allow the measurement and use of time within the system.
random	To allow the generation of random numbers (for the maze).
math	To allow access to mathematical functions.

Management of variables, methods and attributes

Another thing I need to make sure of is that the system handles variables, methods and attributes correctly.

- I will need to make sure that all methods and data are encapsulated and separated from any other methods/data; to do this, like previously specified, I will be make use of object-oriented programming.
- I need to ensure encapsulation so that I can modify my program with later iterations; by organising my data and methods together and keeping them isolated from each other, I can more easily and efficiently make modifications to my program, meaning less time is wasted.
- Because of this, I will need to design classes for all respective objects, specifying data and methods for these respective classes, their purposes, and justifying why I have made the choices I have, and how it helps me solve my overall problem.
- This will be done through the use of class diagrams.

A class diagram is a design method for visualising object-oriented systems. They allow me to illustrate how each class in my system will behave and what methods/attributes they contain.

- My program is suited to be illustrated by a class diagram because it is an object-oriented system, and the use of visual elements and objects lends very well to OOP.
- In order to design class diagrams, I need to identify the main objects/classes in my program, and identify their methods and attributes, as well as objects derived from these classes.
- (Note: In order to manage my program with more ease, I will be splitting the program across several files. This is done to make the program easier to view and modify in the future, which will be useful for future iterations.)
- (Note: not all items in the game are stored in object form – certain parts of the game will be procedurally programmed. This is because they are purely method-based, containing little to no data. For example, the game loop itself may be procedurally programmed, and hence not be an object derived from a class.)

The major objects (derived from class) in my game will be:

- The character – derived from the character class
- The timer – derived from the timer class
- The grid – derived from the grid class

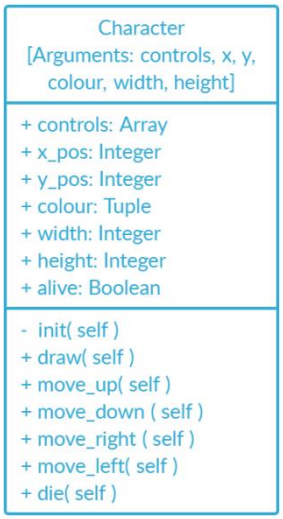
- The lava (threat entity/red area) – derived from the lava class
- The navigation/selection buttons – derived from the button class
- The maze – derived from the maze class

(Please note: there are more classes, relating to how the system itself is managed and the use of states. These classes are discussed later, since they make use of inheritance and don't relate to physical objects as part of the game, but rather concepts.)

- Some attributes may be shared between classes; for example, it is logical to share the "grid_cells" attribute from the Maze class with the "cells" attribute from the Grid class, since they will be holding the same data, for the same purpose.
- The diagrams shown below are formatted such that the top cell is the class name, the middle cell is the attribute(s), in the form "Attribute_name: Attribute_type".
- The bottom cell is the method(s).
- '+' depicts a public class member (attribute/method), while '-' represents a private class member.
- For these diagrams, I've assumed all class members (apart from initialisation methods) to be public. This may change in the future. Please note that these are all initial designs based on predictions – these may vary or more classes may be added.

On the right is the class diagram for the character class.

- Its arguments (values/variables that are passed in) are "controls", "x", "y", "colour", "width" and "height".



Attributes

1) The "controls" attribute is needed to pass the controls into the class for the object: the character.
   a) It is stored in an array because there are four buttons needed, and hence cannot be stored in any other data type.
   b) I didn't use a list because lists cannot handle mathematical operations, and I need my data structure to be able to do so, hence, an array was more suited. Furthermore, because they are stored in an array, they can be accessed individually, and called by index.
2) The "x" and "y" values are passed in for "pos_x" and "pos_y", and the "colour", width" and "height" arguments are passed in for the respective attributes.
   a) The "x_pos" attribute represents the horizontal position of the character. This can and will change. I have stored it as an integer because I will need to perform mathematical calculations with it, when moving the character object, for instance.
   b) The same applies to "y_pos", except with the vertical position. I need these co-ordinates because pygame uses visual elements, and everything inside of the window is tracked by coordinates. In order to correctly position items on screen, I need to use x and y coordinates.
3) The "colour" attribute stores the RGB value of the colour of the character (green).

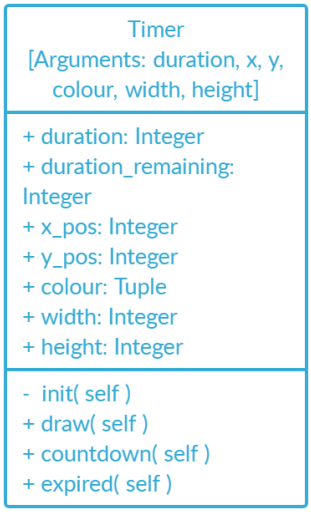
a) I have chosen to store it in a tuple because I can predetermine colours in the initialisation section of the program, and they won't be modified – will remain constant. This is because tuples are non-modifiable.
b) Furthermore, tuples allow me to easily store three values together, and hence are perfect for storing RGB colours.

4) The "width" and "height" attributes denote the height and width of the character in pixels.
   a) These are both needed because I will need to ensure that the character is drawn to a correct, consistent size, and by keeping them as variables, I can ensure that these sizes remain constant.
   b) These are stored as integer values because I will need to perform arithmetic on them, when moving the character object.
5) "alive" is a Boolean value that simply tracks whether the player is alive or not (True/False).

Methods

1) The "init" method initialises all the given attributes/variables, and the class object itself. It is a requirement with Python.
   a) The "draw" method draws the character object to the screen whenever it is called. I will have it in the game loop, to draw the character 60 times a second.
   b) This will allow the character to be visible and mobile, as the coordinate attributes will also be updated on the screen, shown by the positioning of the character.
2) The "move_up()", "move_down()", "move_left()", "move_right()" methods all move the character in the specified direction, when the corresponding "controls" inputs are given.
   a) For instance (on WASD settings), if W is pressed, the "move_up()" function will be called, changing the "x_pos" variable by the appropriate amount.
   b) These methods are needed in order for the player to be able to control the character.
3) Finally, the "die" method kills the character once a global variable from another class/method is received. This can be either from the "touching_player()", from the "Lava" class, or "expired()", from the "Timer" class, as both of these are triggers to kill the player.
   a) When this method is called, the player will be rendered unable to move, "alive" will be set to False, and the game will move into the next state.

This is the timer class.



- It has arguments "duration", "x", "y", "colour", "width" and "height" passed into it. The "duration_remaining" or a related value for the same purpose does not need to be an argument, as the class will calculate this attribute on its own.

Attributes

1) The "duration" attribute is how much time the user has been allocated to remain idle at any given time.
   a) This attribute resets every time the user moves.
   b) It is an integer because it will be used in mathematical calculations, as explained with the following attribute.
2) The "duration_remaining" attribute is similar to the "duration" attribute, except it represents the duration remaining rather than the

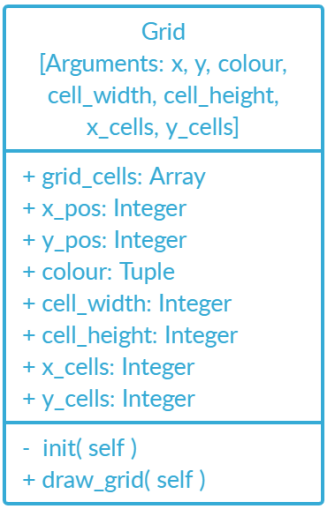
entire allocated duration that the user can remain inactive at any given point in time.
   a) Since it is incremented by -1 every second, I need to ensure that it is an integer type value, as this is a mathematical calculation.
3) "colour", "x_pos", "y_pos", "width" and "height" all behave the same as the attributes from the "character" class, but instead attributed to the timer and its physical behaviours on the pygame window.

Methods

1) Like all other classes, the timer class has an "init()" method.
2) The "draw()" method draws the timer on the screen.
3) The "countdown()" increments the "duration_remaining" attribute by -1 every time it is called. It is called every 1 second, when the game is active.
   a) This is to time how long the player has left until death by remaining idle.
4) Finally, the "expired()" method checks if "duration_remaining" is less than or equal to 0.
   a) If it is, the player is dead, and the "expired()" method will call any other methods to ensure that this is upheld (calling the "die()" method in the character class).

This is the grid class.

(I did some brief research on how mazes are generated, and found that, the majority of the time, they are carved from grids. As a result, I have included a grid class in my design. If this is not needed, it will be omitted in the future.)



- The arguments of this class are "x", "y", "colour", "cell_width", "cell_height", "x_cells" and "y_cells". These arguments are assigned to "x_pos", "y_pos", "colour", "cell_width", "cell_height", "x_cells" and "y_cells" respectively.

Attributes

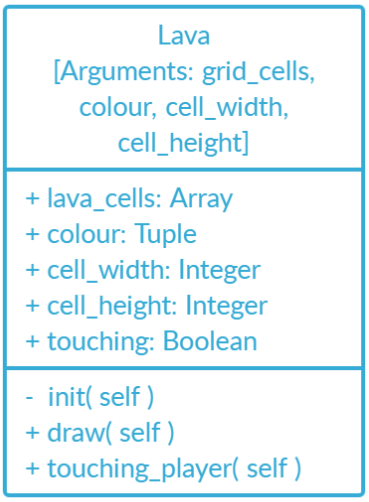
1) The "cells" attribute is for keeping track of what cells are in the grid, using pygame coordinates.
   a) It is an array because with arrays I can hold many pieces of data. The coordinates themselves will be tuples.
   b) Furthermore, I can search these coordinates, and select their x-component or y-component, and use them for mathematical operations (hence they will be integers).
2) The "grid_cells" attribute is essentially an array that holds tuples of two, which hold integers to represent pygame coordinates.
   a) This holds all coordinates for cells in the grid.
3) The "x_pos" and "y_pos" attributes represent the x and y coordinates of the origin point of the grid itself.
   a) These are integers, for mathematical calculations.
4) "cell_width" and "cell_height" represent the width and height of an individual cell in the grid (all cells are the same size), as integers.
5) "x_cells" and "y_cells" represent the number of cells in the grid in the horizontal and vertical directions.

a) These are needed when drawing the grid.

Methods

1) The only method (besides "init()") in this class is the "draw_grid()" method.
   a) As the name suggests, this constructs the grid of a specified resolution.
   b) From this grid (if certain algorithms are used) I can carve the maze.

This is the lava class.



- Its arguments are "grid_cells" (derived from the previous Grid class), "colour", "cell_width" and "cell_height". "grid_cells" is needed as an argument in order to ensure all lava cells fall within the grid.

Attributes

1) Similar to the grid class, it contains an array as an attribute, called "lava_cells".
   a) Responsible for keeping track of all cells that have been turned red (and are hence lava cells).
   b) Cells will be appended into this array every time the player moves.
   c) Stored as tuples, holding integer coordinates in the form of an x and y component (for the reasons aforementioned).
   d) This will allow me to see if the player is touching any lava cells, and hence kill them if they are.
2) "colour", "cell_width", "cell_height" have all already been explained.
   a) The colour of the lava is red (255, 0, 0).
3) The "touching" attribute is a Boolean value (True or False) that is True if the player is touching the lava (and hence is supposed to be dead).

Methods

1) The "draw()" method draws lava (and appends any lava cells) where the player goes.
   a) This is done so that the player can see the lava, and also so that locations of all the lava cells are known.
2) "touching_player()": this method checks if the player is on the same location as any cell in "lava_cells".
   a) If they are, the player is on a lava cell. This will set the "touching" attribute to True, and this will call "die()" method in the character class.

This is the button class.

- Buttons are for the player to click on, in order to make selections throughout my game.

- Its arguments are “x”, “y”, “colour”, “text”, “name”, “width” and “height”, correspondent with the appropriate attributes.

Attributes

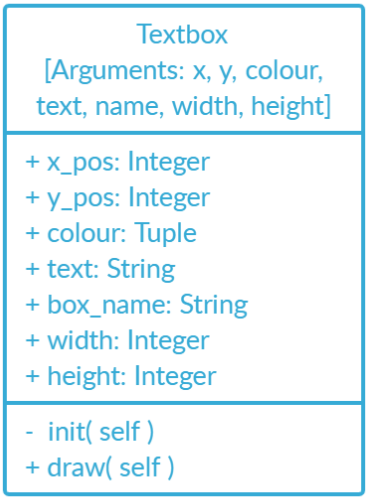
1) The “x_pos” and “y_pos” integer attributes are for the coordinate positioning of a button in the pygame window.
2) The “text” attribute is simply whatever the button has written on it, often information on what clicking the button will do, or what option it represents in the UI.
   a) As a result, it has to be a string, as it represents text data, without any need for calculations.
3) The “button_name” attribute is a string value that identifies the instance of the button in the system.
4) The “width”, “height” and “colour” attributes have already been explained.
5) “clicked” is a Boolean attribute, which detects if the player has clicked on the button.
   a) As a result, it only needs to be True or False.

Methods

1) “draw()”: this draws the button on screen when called.
2) “is_clicked()”: if the button is clicked, and hence “clicked” is set to True, the button will do whatever it needs to do/trigger.
   a) This could be moving states, making selections or even exiting the game. This is needed in order for the user to have control and selection over the system.

This is the textbox class.

- The previous button class is essentially a text box that can be clicked on in order to trigger some event; the textbox simply is a formatted box with a piece of text in it.
- The arguments of this class are “x”, “y”, “colour”, “text”, “name”, “width”, “height”, assigned to the appropriate attributes.



Attributes

1) “box_name” is the name of the particular instance of box in the system, and hence is a string.

Methods

1) “init()”: for initialising variables
2) “draw()”: for displaying the box on the screen.

This is the maze class.

- This class contains an array called “grid_cells” – this identical to the array in the grid class, called “grid_cells”. This array was passed in as one of the arguments. This is needed to ensure

that the maze doesn't exceed/attempt to use cells that are non-existent in the grid, as this will lead to a lot of errors and an unplayable game.
- The other arguments are "x", "y", "colour", "cell_width", "cell_height", "x_cells" and "y_cells", all assigned to the appropriate attributes.

Attributes

1) The "x_pos", "y_pos", "colour", "cell_width" "cell_height", "x_cells" and "y_cells" attributes have all already been explained in the grid class.
2) The "walls" array is needed to keep track of walls in the maze, to prevent the player from being able to traverse through walls.
   a) These walls will likely be stored as tuples.
3) The "maze_cells" attribute is another array, containing all the cells in the maze (likely in order of generation).
   a) These two arrays ("walls" and "maze_cells") are needed in order to keep track of the formation of the cell, and to prevent it from being violated.

Methods

1) The "generate_maze()" method calls the algorithm that will build the maze from (or independently of) the grid.
2) The "draw()" method draws the maze to the screen.
   a) These are needed to actually build the maze and put it on the screen so that the user can play the game.

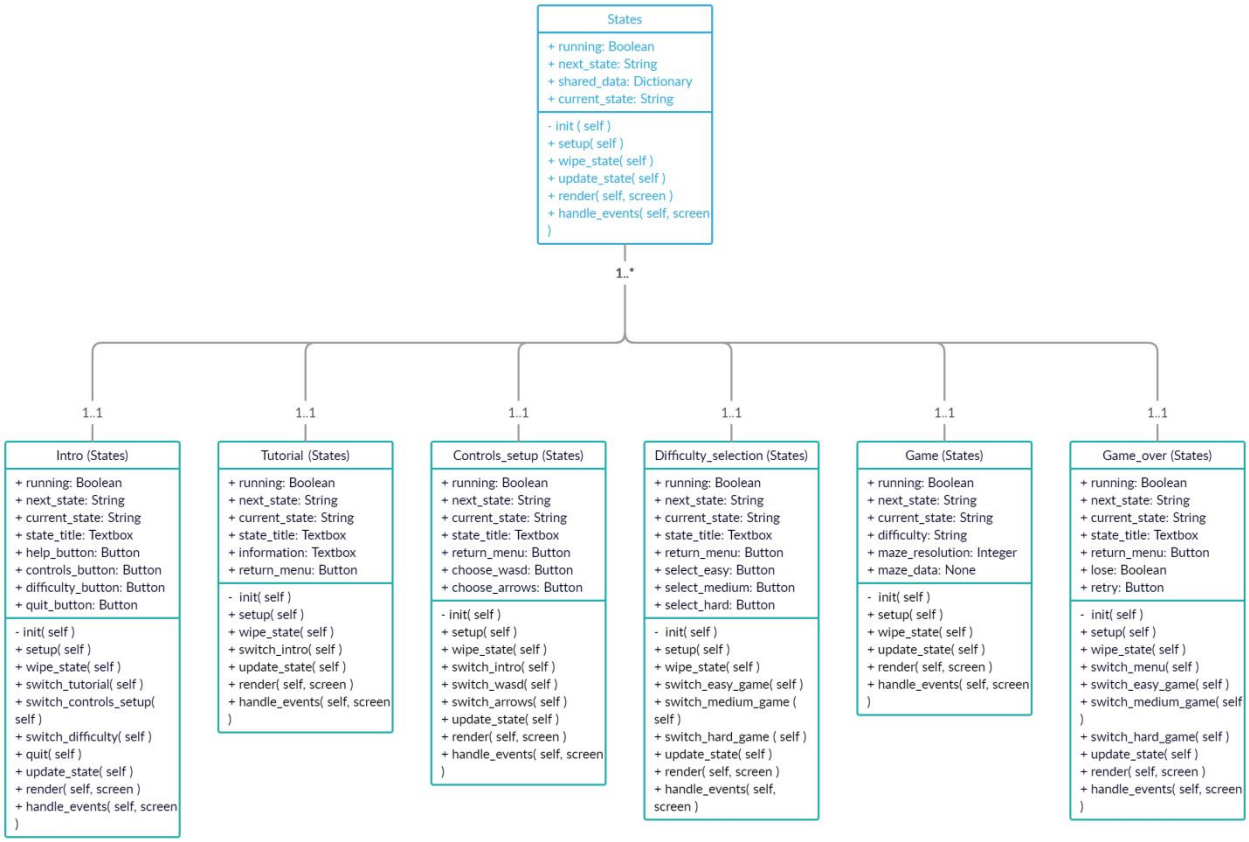
Even though I've listed all of the (predicted) classes for the main game and menu, I still need to consider the classes relating to states and their behaviours. I researched into the OOP implementation of state machines, and found that it's different to any other class structure previously mentioned: it makes use of inheritance, subclasses and super-classes.

I found that in the state-related classes, I will need to implement a state super-class, followed by all the states I will be using. There are 6 different states as aforementioned. From these states, I can derive the following sub-classes:

SUPERCLASS: State

1) Intro
2) Tutorial
3) Controls_setup
4) Difficulty_selection
5) Game
6) Game_over

From the state super-class, all the classes listed above will inherit the characteristics of methods and attributes.



(Please note that with the subclass names, the bracketed name simply denotes the inheritance of methods and attributes from the state superclass. These are listed in the class diagrams.)

3.2.3 – Describe the approach to testing

3.2.3a – Test data

In order to ensure my system meets all the previously mentioned success criteria (3.1.4), and to prevent future bugs and problems, I will need to carry out a variety of tests. These tests will measure the degree of success for each of the individual success criteria, and will be carried out by the stakeholders and myself.

Please note that not all of these criteria are applicable to testing – these have been included nonetheless, but will be marked as not applicable.

DESIGN CRITERIA:

Criteria:	**How to evidence:**	**Test:**
The final product must be a 2D puzzle game.	Screenshot of main gameplay	N/A
The window for the game must be	Screenshot, and confirmation	Playing through the game

1000×700.	from stakeholders	and ensuring the window never changes in size.
The game background must be black or white.	Screenshot of main gameplay	N/A
The primary threat (lava) in the game must be coloured red.	Screenshot of main gameplay	Having stakeholders traverse through the maze, and ensure all lava cells generated are red, and are always fully visible.
The main character/object of the game must be coloured green.	Screenshot of main gameplay	Having stakeholders play the game, and ensure the main character is always fully visible and is green.
The menu must feature buttons for the player to click on to make decisions.	Screenshot of main menu	Checking that buttons are all fully operational and carry out their job.

DEVELOPMENT CRITERIA:

Criteria:	**How to evidence:**	**Test:**
The game must be developed on Python, using the Pygame module.	Screenshot of code in Python, using the Pygame module	N/A
Development must feature prototypes, and improve these prototypes through iteration.	Screenshots and reviews of every iteration, with criticisms from the stakeholders	N/A
Development must take feedback from the selected stakeholders, and make changes to iterations based on this feedback.	Screenshots and reviews showing how each iteration has been improved based on previous feedback	N/A

INPUT/OUTPUT CRITERIA:

Criteria:	**How to evidence:**	**Test:**
The player must be able to choose between WASD and the arrow keys as their input controls.	Screenshots of control selection menu, stakeholder reviews ensuring controls work	Ensuring that the system only allows the player to move when the correct inputs are entered, for the selected control scheme, and checking that inputs only affect their designated direction.
The game must output to a screen.	Picture of game outputting to screen	N/A
The game must run in windowed form.	Screenshot of the game window	Checking that full-screen cannot be toggled.
The game must feature some audio, in the form of basic sound effects.	Stakeholder review confirming the presence of audio for some events	Checking audio effects only playing at designated point, with the correct sound effects.

GAME CRITERIA:

Criteria:	How to evidence:	Test:
The game must feature a title screen.	Screenshot of title screen	Checking all elements of the title screen function correctly.
The game must feature an introductory screen.	Screenshot of introductory screen	Checking all elements of the introductory screen function correctly.
The player must be able to choose their difficulty level.	Screenshot of difficulty selection screen, with stakeholder review to ensure that it works	Checking that every difficulty selection generates the appropriate game (maze and timer).
The introductory screen must allow the user to view a tutorial screen with basic instructions.	Screenshot of tutorial screen with instructions, clarifying with stakeholders that instructions are understandable	Checking that instructions are displayed on the introductory screen, and are fully readable.
The introductory screen must allow the user to exit the application completely.	Screenshot of 'EXIT' button and stakeholders confirming button's functionality	Checking that the exit button allows the user to fully exit and close the game immediately.
The game must feature a main game screen, where the user can play.	Screenshot of game screen, stakeholders confirming that the game is playable	Checking that the game is fully playable; that all inputs are always registered.
The game must feature a game over/winning screen.	Screenshot of the game over screen; screenshot of the winning screen	Checking that the appropriate screen is shown as a result of the game (winning screen for winning; game over screen for losing).
The algorithm must generate mazes that are always solvable.	Screenshots of all difficulty mazes to show they're all fully solvable	Checking that the chosen algorithm always generates fully solvable mazes.
Any tiles/spaces/cells that the player traverses must become red.	Screenshots of a player as they traverse through the maze, and how lava is left in their wake	Checking that all traversed cells become lava cells.
The player must not be allowed to traverse lava without dying.	Stakeholder review confirming that they die whenever they attempt to touch lava cells	Checking that the player dies when they attempt to traverse the lava.
The game must feature a timer to force players to make moves.	Stakeholder review confirming that they die if the timer reaches 0; confirming that the timer resets with every move	Checking that the correct timer is generated; checking that the timer counts down as appropriate, and starts at the correct time; checking that the player dies when the timer expires (for all

		difficulties); checking that the timer resets with every move; checking that the reset timer functions correctly and kills the player.
Harder difficulty mazes must be a higher resolution (more detailed, with more cells) than their lower difficulty counterparts.	Screenshots of all maze difficulties [covered above], showing increased resolution for higher difficulty settings	Checking that appropriate mazes are generated; checking that mazes disallow players from traversing through their walls.
Harder difficulties must feature a shorter timer than their lower difficulty counterparts.	Screenshots and confirmation from stakeholders that higher difficulty mazes have a shorter timer	Checking that the correct timer is generated for all difficulties.

As well as the above tests based on the success criteria, I will need to test that other parts of the system function correctly. These are listed below:

- Checking that all the “BACK” buttons function correctly.
- Checking that the “RETURN TO MENU” buttons function correctly.
- Checking that the “RETRY” buttons (in the end-game screens) function correctly – that retrying in a lost game generates a new maze of the same difficulty, while retrying a won game regenerates the same maze.

Lastly, I need to consider robustness tests to try to ‘break’ the software. These will ensure quality and durability of the final product, and how it handles unexpected inputs and behaviour. These are listed below:

- Repeatedly and rapidly traversing through different parts of the system, checking if any UI elements get ‘mixed up’ or if any menus are generated at an inappropriate time.
- Repeatedly navigating back and forth (from all menus), checking if any UI elements are ‘missed’, or not generated.
- Repeatedly switching between control schemes and playing the game, checking if at any time, the non-selected controls become useable (e.g. while WASD is selected, checking if the arrow keys remain inactive, and vice-versa).
- Checking that the same maze is never generated 3 times in a row hard difficulty, 4 times in a row for medium difficulty, and 5 times in a row for easy difficulty.

(These tests will be carried out and checked during test plan four – acceptance testing)

These tests are only generic tests thus far; I need to describe my approach to testing through the use of test plans.

Test plan one – during development (white box testing)

White box testing involves looking at the code and ensuring that it'll all work, regardless of circumstance. As a result, this form of testing will be conducted during development, since it's most appropriate where the code is visible to me.

Test:	**Justification:**	**Data used:**	**Expected outcome:**	**Where this is carried out (page):**
Checking that the algorithm always generates solvable mazes	To ensure the game always winnable	Visual observation, maze traversal	A fully solvable maze	166-167
Checking that the screen window is at a resolution of 1000 by 700	To make sure the screen size is always according to the original success criteria	Visual observation, checking code	A window of resolution 1000 by 700	167
Checking that the timer counts down correctly, and resets whenever the player moves	To ensure that the timer works correctly, and that the player actually has a chance of winning the game	Visual observation, screenshots as proof, repeated testing in different scenarios, checking code	A timer that counts down one second at a time, killing the player at zero.	167
Checking that buttons transition between states as needed	To ensure the user can navigate the system as needed	Checking by clicking buttons, testing every button in the system and recording results in a large table*	All buttons navigate to the correct screen/have the correct effect on the game	168-171
Checking UI elements are drawn to the screen in the correct position, and in the correct colour	To ensure all information is displayed on the screen as necessary	Screenshots as proof	All UI elements appearing on screen	171-175
Checking that any lava cell kills the player upon contact	To ensure the player can actually lose the game	Repeated testing, visual observation, screenshots as proof	Player dies upon contacting any lava cell on screen	175-176

*table to be collated post-development, testing is carried out during development

Test plan two – post development (black box testing)

After development, black box testing will be conducted (again, by myself). This will involve treating myself as a user, without looking at or interacting with the code, and testing the program in this manner.

Requirement ("The	**Test/data used:**	**Justification:**	**Expected outcome:**	**Where this is**

user must be able to…")				**carried out (page):**
Access different parts of the game via the main menu, as well as being able to return to the main menu.	Navigating to all the different parts of the system via the main menu, using the buttons (compilation of aforementioned table will be done here)	The user needs to be able to have access to every part of the game for the full experience, and the main menu should facilitate this.	Buttons allow access to every game state available to the user	178-179
Exit the game from the main menu.	Pressing the 'QUIT/EXIT' button, and checking if I leave the game.	The user needs to be able to exit the game.	When clicked, the button should allow the player to leave the game.	179
Select their control scheme (WASD or arrow keys)	Selecting each control scheme and checking they both work	The user needs to be able to choose their controls scheme, as per the requirements.	The selected control scheme should be able to control the character ONLY	179
Select difficulty	Checking that all difficulties generate different mazes as appropriate, screenshots as evidence	The game needs to have varying difficulty as one of its requirements, and the user needs to have the ability to select this	Mazes and timers should be created according to the selected difficulty	179-180
Have a maze generated based on their selected difficulty	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	
Move using their selected control scheme	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	
Be disallowed from traversing through walls of the maze	Attempting to traverse through walls, recording results	The user cannot be allowed to traverse through walls – if they did, the game would provide no challenge	When a player attempts to traverse through a wall, nothing happens	180-181
Have a timer that tracks how much time they have left before death	Testing that the timer actually kills the player, testing that it resets upon movement	Without a timer, the player could stall infinitely, making for a game without a challenge	Timer resets when the player moves, kills the player when it reaches 0 seconds remaining	181
Have lava cells generated in their wake	Checking that lava cells are always generated behind the player, as they move	Without this, a player could backtrack, essentially spoiling the whole premise of this game	Lava cells are generated in any place where the user has visited, prevented them from backtracking	181-182

			through the maze	
Die (when either running into a lava cell or expending their allocated time)	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	
Win (upon traversing the maze correctly, without dying)	Checking that the player always wins upon reaching the goal	If this were to NOT happen, the player would be doomed to lose regardless	Upon reaching the goal, the player wins the game	182
Return to main menu or retry from the 'game over' screen	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	
Return to main menu or proceed to next level from the winning screen	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	[COVERED PREVIOUSLY]	

Test plan three – user testing

Again, this is similar to the previous stage of testing, only this time, the testing will be carried out by users, specifically my 5 designated stakeholders, to prevent any bias and to ensure that the system is usable for a novice user, someone who isn't me, the developer.

What the user will do:	**Justification:**	**Expected outcome:**	**Where this is carried out (page):**
Having stakeholders traverse through the maze, and ensure all lava cells generated are red, and are always fully visible, and that they kill the player upon contact.	This is to ensure that lava cells behave as expected and are visible, so that the player knows where they can and cannot traverse, as well as killing the player when touched.	All cells that are traversed become lava cells; they kill the player upon contact.	192
Having stakeholders play the game, and ensure the main character is always fully visible and is green, as well as being able to move.	This is to ensure the player always knows their exact location, so that they can play with ease and fairness.	While playing, the stakeholder is always able to see their character.	192-193
Checking that the same maze is never generated 3 times in a row hard difficulty, 4 times in a row for medium difficulty, and 5 times in a row for easy difficulty.	This is to ensure repeated mazes are not generated. Evidence will be shown using screenshots.	Unique mazes every time, due to random generations.	193-194
Checking that sound effects are played (as	This is to ensure the sound design always	Whenever a button is clicked to move to a	194-196

appropriate) every time a button is pressed, when the game starts, when the player wins and when the player loses.	works, and that sound prompts can give users information on what is happening in the game.	new state, a sound effect is played. (A table will be collated recording the sound effects for every button/event in the game, and whether or not they are working.)	
Repeatedly and rapidly traversing through different parts of the system, checking if any UI elements get 'mixed up' or if any menus are generated at an inappropriate time.	This is a robustness test, to try to 'catch' errors, or problems with UI elements that may 'lock' the user in place.	No matter how fast the user moves through the system, they are always presented with the correct screen at every juncture. (A table will be collated recording the states and their next state, and whether or not they are working.)	189-190
Repeatedly switching between control schemes and playing the game, checking if at any time, the non-selected controls become useable (e.g. while WASD is selected, checking if the arrow keys remain inactive, and vice-versa).	This is a robustness test, to try to see if there are ever any errors with the controls that can render the game unplayable.	No matter how many times the user switches between controls, ONLY one scheme should be operational – the selected control scheme	190

Test plan four – acceptance testing

Finally, the last stage of testing will be conducted to check how well the coded solution meets the given success criteria and requirements. This will be carried out by both me and the designated stakeholders, to collectively decide how far each success criteria is met.

This will be done in a table, analysing each success criteria, for each category (design, development, input/output, game). The tests from the start of 3.2.3a will be used to help aid testing, and post-development evaluation.

With the specification of the testing completed, the design section is concluded.

Section 3: Implementation

3.3.1 – Iterative development process

3.3.1a – Stages of development

Introduction to implementation

- I now have all the information on the design and approach to testing, thanks to the previous section. I am ready to start developing prototypes.
- As mentioned several times before, I will be following an iterative process. This means I will be taking prototypes and improving them, based on feedback and other factors, in order to achieve the optimal solution.
- As mentioned before, the entire system will be developed on Python, using the Pygame module.

Developing the maze generation algorithm

- As previously mentioned, I will be making use of the Growing Tree algorithm in order to generate my maze.
- I chose to develop this part of the program first, because it is one of the most integral parts and the entire game is based around the mazes derived from this algorithm.
- As previously mentioned, Growing Tree can be implemented in a variety of different ways in terms of cell selection method. For the initial prototype, I will be using the 'select from newest cell' method, which behaves similarly to recursive backtracker (mentioned in 3.2.2b).
- I will start by generating the maze for the 'easy' difficulty (10×10).

I initially chose to NOT generate the maze in Pygame, but simply as a large string value, similar to ASCII art. My reasoning behind this was because it was easier to develop the algorithm independently to the Pygame module, before taking the generated maze and using it to draw the maze in Pygame.

```
import random # Since the generation of this maze is random, I need the random module.

x_cells = 10 # This defines the number of cells in the x-direction.
y_cells = 10 # This defines the number of cells in the y-direction.
grid = [] # This allows me to build the grid from which the maze is generated.
```

My first action was to import the random module and initialise everything that will be needed for the generation of the maze. As established in design, knew I needed to start by defining 'x_cells', 'y_cells' and the 'grid' list. The purposes of these have all already been explained, and are explained again in the above screenshot.

I realised that, unlike my pseudocode implementation, I will need to also store the location of the walls, to prevent players from traversing into them and to ensure they can be imported into pygame and instantiated as physical objects. As a result, I will be using cells as walls in this implementation.

In this version of Growing Tree, I will be using random selection of adjacent cells, from the newest cell, in order to carve the maze from the grid. This will be repeated until it is no longer possible – there are no cells left in the grid that can be carved to, and hence the algorithm is finished.

A cell which is adjacent to a cell that has been carved to the maze is 'exposed'. Any of the exposed cells can be selected to carve to (randomly), and so on. This is how the 'tree grows'.

```
"""
KEY:
'|' indicates a WALL
' ' indicates a SPACE
'+' indicates an EXPOSED UNDETERMINED LOCATION
'?' indicates an UNDETERMINED LOCATION
"""
```

Before I start generating the grid, I need to come up with a 'key' that describes the values of the maze (as a string) and what they represent. This has been done on the left.

```
import random # Since the generation of this maze is random, I need the random module.

x_cells = 10 # This defines the number of cells in the x-direction.
y_cells = 10 # This defines the number of cells in the y-direction.
grid = [] # This allows me to build the grid from which the maze is generated.

"""
KEY:
'|' indicates a WALL
' ' indicates a SPACE
'+' indicates an EXPOSED UNDETERMINED LOCATION
'?' indicates an UNDETERMINED LOCATION
"""
for y in range(y_cells):
    row = [] # Creates a 'row' list for all y locations (10 rows)
    for x in range(x_cells):
        row.append('?') # Sets all cells to UNDETERMINED
    grid.append(row) # Fills the grid with these rows
```

After that, I added code to actually generate the entire grid, through the use of rows. The above loops generate rows of '?' (indicating undetermined cells) and joining these rows (lists) to complete the entire (blank) grid.

```
import random # Since the generation of this maze is random, I need the random module.

x_cells = 10 # This defines the number of cells in the x-direction.
y_cells = 10 # This defines the number of cells in the y-direction.
grid = [] # This allows me to build the grid from which the maze is generated.

"""
KEY:
'|' indicates a WALL
' ' indicates a SPACE
'+' indicates an EXPOSED UNDETERMINED LOCATION
'?' indicates an UNDETERMINED LOCATION
"""
for y in range(y_cells):
    row = [] # Creates a 'row' list for all y locations (10 rows)
    for x in range(x_cells):
        row.append('?') # Sets all cells to UNDETERMINED
    grid.append(row) # Fills the grid with these rows

exposed = [] # This is a list of EXPOSED UNDETERMINED LOCATIONS (+)
```

Finally, I added a list to hold all the exposed (undetermined) locations.

Next, I will need to create two functions for undetermined locations: one function to CARVE (make the location a space), and one to WALL (make the location a wall).

```
def carve(y, x): # This function makes the cell a space.
    grid[y][x] = ' ' # This takes the two parameters and updates location.


def wall(y, x): # This function make the cell a wall.
    grid[y][x] = '|' # This takes the two parameters and updates location.
```

These are the two functions in their most basic form. All they do is update a cell (given by parameters y and x) to be either a space or a wall.

However, for the carve function, I also need to update the surrounding adjacent cells to be exposed. I have to change their symbol to '+', and add them to the 'exposed' list.

```
def carve(y, x): # This function makes the cell a space.
    grid[y][x] = ' ' # This takes the two parameters and updates location.
    adjacent = [] # This stores any adjacent cells.

    if x > 0: # This checks if there is a cell behind the current cell (x direction).
        if grid[y][x-1] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y][x-1] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y,x-1)) # Add this cell to the 'adjacent' list.

    if x < x_cells - 1: # This checks if there is a cell in front of the current cell (x direction).
        if grid[y][x+1] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y][x+1] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y,x+1)) # Add this cell to the 'adjacent' list.

    if y > 0: # This checks if there is a cell behind the current cell (y direction).
        if grid[y-1][x] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y-1][x] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y-1,x)) # Add this cell to the 'adjacent' list.

    if y < y_cells - 1: # This checks if there is a cell in front of the current cell (y direction).
        if grid[y+1][x] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y+1][x] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y+1,x)) # Add this cell to the 'adjacent' list.

    random.shuffle(adjacent) # Shuffles the list of adjacent cells.
    exposed.extend(adjacent) # Adds the list of adjacent cells to the 'exposed' list.
```

I've split the code above into individual 'paragraphs'. Four of them are almost identical, except they account for different directions. The first of these ('if x > 0:…') checks if there is a cell on the left of the current cell. If there is, and it's an undetermined cell, it marks the cell as 'exposed'. The next three paragraphs do the same thing, for the right, top and bottom (respectively), while adding the cell to the 'adjacent' list. After this, the 'adjacent' list is shuffled and added to the 'exposed' list.

```
>>> carve(-1, 1000)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    carve(-1, 1000)
  File "C:\Users\Talhaa\Talhaa Hussain\School\A-Levels\Computer Science\Mr AbdulMajeed
\Computer Science A2 Project\code\MY GROWING TREE.py", line 26, in carve
    grid[y][x] = ' ' # This takes the two parameters and updates location.
IndexError: list assignment index out of range
>>> wall(100, -10)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    wall(100, -10)
  File "C:\Users\Talhaa\Talhaa Hussain\School\A-Levels\Computer Science\Mr AbdulMajeed
\Computer Science A2 Project\code\MY GROWING TREE.py", line 57, in wall
    grid[y][x] = '|' # This takes the two parameters and updates location.
IndexError: list index out of range
>>> wall(0.1, 1)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    wall(0.1, 1)
  File "C:\Users\Talhaa\Talhaa Hussain\School\A-Levels\Computer Science\Mr AbdulMajeed
\Computer Science A2 Project\code\MY GROWING TREE.py", line 57, in wall
    grid[y][x] = '|' # This takes the two parameters and updates location.
TypeError: list indices must be integers or slices, not float
>>>
```

I need a way of validating parameters for these functions. As evident from the test above, if x or y is smaller than 0 or greater than 10, (for the 10×10 'easy' grid/maze) or a non-integer value is given, this will cause problems – these values are completely out of range of the cell.

```
def carve(y, x): # This function makes the cell a space.
    if (0 > (x or y)) or ((x or y) > 10) or ((type(x) or type(y)) != 'integer'):
        return # This is for validation - 'if invalid data has been entered, ignore.'
```

```
def wall(y, x): # This function make the cell a wall.
    if (0 > (x or y)) or ((x or y) > 10) or ((type(x) or type(y)) != 'integer'):
        return # This is for validation - 'if invalid data has been entered, ignore.'
```

As shown above, I added two extra lines of code to both the 'wall' function and the 'carve' function. These are for validation – if a non-integer, negative or number greater than 10 is entered (for y or x), the input is ignored.

However, to improve this, I could make it suited for any size mazes – in which case, I replace the 10 with x_cells and y_cells.

```
def carve(y, x): # This function makes the cell a space.
    if (0 > (x or y)) or (x > x_cells) or (y > y_cells) or ((type(x) or type(y)) != 'integer'):
        return # This is for validation - 'if invalid data has been entered, ignore.'
```

```
def wall(y, x): # This function make the cell a wall.
    if (0 > (x or y)) or (x > x_cells) or (y > y_cells) or ((type(x) or type(y)) != 'integer'):
        return # This is for validation - 'if invalid data has been entered, ignore.'
```

I tested this and noticed a problem: it wouldn't carve.

```
>>> carve(2 , 2)
>>> print(grid)
[['?', '?', '?', '?', '?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '
?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?', '?', '?', '?', '?'], ['?
', '?', '?', '?', '?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?', '?', '?', '?', '?'], ['?', '
?', '?', '?', '?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?', '?',
 '?', '?', '?'], ['?', '?', '?', '?', '?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?', '?', '?', '?', '?']]
>>>
```

For some reason or another, the algorithm would now ignore ALL inputs. To try to diagnose where the problem was coming from, I added in two lines, 'print("CARVED!")' and 'print("REJECTED!")' to try to find out exactly where the function was choosing to not carve.

```
>>> carve(2, 2)
REJECTED!
>>>
```

As evident from the image on the left, the function had now started rejecting valid inputs. I checked that 'y' and 'x' were both integers – this was not the problem.

```
def carve(y, x): # This function makes the cell a space.
    if 0 > (x or y):
        print("REJECTED!")
        return # This is for validation - 'if invalid data has been entered, ignore.'
```

I revised the validation rules to ONLY check if the number was negative and ran the program again.

```
>>> carve(2, 2)
CARVED!
CARVED!
CARVED!
CARVED!
>>>
```

As evident from the image on the left, this was not the problem. I revised the validation rules again, splitting the 'if' statements into separate parts.

```
def carve(y, x): # This function makes the cell a space.
    if 0 > (x or y):
        print("REJECTED!")
        return # This is for validation - 'if invalid data has been entered, ignore.'
    if (x > x_cells) or (y > y_cells):
        print("REJECTED!")
        return
```

This didn't have any problems either.

```
>>> carve(3, 3)
CARVED!
CARVED!
>>>
```

I continued with the validation rules. The final rule was checking for integers.

I realised that I had earlier used the string 'integer' for checking instead of the 'int' function. I amended this and the system behaved as expected. From this, I amended the entire statement, merged the three statements back into one, and repeated this for the 'wall' function. I then tested them both, and both behaved as expected.

```
import random # Since the generation of this maze is random, I need the random module.

x_cells = 10 # This defines the number of cells in the x-direction.
y_cells = 10 # This defines the number of cells in the y-direction.
grid = [] # This allows me to build the grid from which the maze is generated.

"""
KEY:
'|' indicates a WALL
' ' indicates a SPACE
'+' indicates an EXPOSED UNDETERMINED LOCATION
'?' indicates an UNDETERMINED LOCATION
"""
for y in range(y_cells):
    row = [] # Creates a 'row' list for all y locations (10 rows)
    for x in range(x_cells):
        row.append('?') # Sets all cells to UNDETERMINED
    grid.append(row) # Fills the grid with these rows

exposed = [] # This is a list of EXPOSED UNDETERMINED LOCATIONS (+)

def carve(y, x): # This function makes the cell a space.
    if (0 > (x or y)) or ((x > x_cells) or (y > y_cells)) or (type(x) != int or type(y) != int):
        return # This is for validation - 'if invalid data has been entered, ignore.'

    grid[y][x] = ' ' # This takes the two parameters and updates location.
    adjacent = [] # This stores any adjacent cells.

    if x > 0: # This checks if there is a cell behind the current cell (x direction).
        if grid[y][x-1] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y][x-1] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y,x-1)) # Add this cell to the 'adjacent'

    if x < x_cells - 1: # This checks if there is a cell in front of the current cell (x direction).
        if grid[y][x+1] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y][x+1] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y,x+1)) # Add this cell to the 'adjacent' list.

    if y > 0: # This checks if there is a cell behind the current cell (y direction).
        if grid[y-1][x] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y-1][x] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y-1,x)) # Add this cell to the 'adjacent' list.

    if y < y_cells - 1: # This checks if there is a cell in front of the current cell (y direction).
        if grid[y+1][x] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y+1][x] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y+1,x)) # Add this cell to the 'adjacent' list.

    random.shuffle(adjacent) # Shuffles the list of adjacent cells.
    exposed.extend(adjacent) # Adds the list of adjacent cells to the 'exposed' list.
```

```
def wall(y, x): # This function make the cell a wall.
     if (0 > (x or y)) or ((x > x_cells) or (y > y_cells)) or (type(x) != int or type(y) != int):
        return # This is for validation - 'if invalid data has been entered, ignore.'

    grid[y][x] = '|' # This takes the two parameters and updates location.
```

This is the state of the program so far.

Next, I need a way to check if a cell at a given location can be carved or not. In other words, this decides if a given location will become a wall or a space.

```
'''
The following function determines if a cell should be a space or a wall.
If it returns True, it should become a space.
If it returns False, it should become a wall.
'''
def check(y, x, nodiagonals = True): # Takes the 'nodiagonals' parameter...
    # ...this parameter determines if a cell has NO diagonals (when True).
```

The 'check' function is responsible for this. It works slightly differently to the aforementioned functions – as well as taking the 'x' and 'y' parameters, it also takes a 'nodiagonals' parameter. This is a Boolean value that is true if a given cell has no spaces in its diagonals.

```
'''
The following function determines if a cell should be a space or a wall.
If it returns True, it should become a space.
If it returns False, it should become a wall.
'''
def check(y, x, nodiagonals = True): # Takes the 'nodiagonals' parameter...
    # ...this parameter determines if a cell has NO diagonals (when True).
    edgestate = 0 # This is a way of determining the behaviour of the edges around a  point.

    if x > 0: # If the cell has cells to the left...
        if grid[y][x-1] == ' ': # ...if the cell directly to the left is a space...
            edgestate += 1 # Increase edgestate by 1.

    if x < x_cells-1: # If the cell has cells to the right...
        if grid[y][x+1] == ' ': # ...if the cell directly to the right is a space...
            edgestate += 2 # Increase edgestate by 2.

    if y > 0: # If the cell has cells above...
        if grid[y-1][x] == ' ': # ...if the cell directly above is a space...
            edgestate += 4 # Increase edgestate by 4.

    if y < y_cells-1: # If the cell has cells below...
        if grid[y+1][x] == ' ': # ...if the cell directly below is a space...
            edgestate += 8 # Increase edgestate by 8.
```

The 'edgestate' variable is a way of determining the way the adjacent cells from a given cell behave (i.e. if they are spaces or walls). It is initialised at 0. The value of this variable changes depending on the state of the cells surrounding the current cell, as shown above. This is important because it allows me to assess the behaviour of a cell, and hence if I should carve to it or not. Remember, I cannot carve diagonally in my maze, hence this is needed.

The part of the function shown above does not check the diagonals. It merely looks at the cells directly top, bottom, left or/and right of the given cell, and sets 'edgestate' according to this.

This function is rather long, and is split over the following page:

```
if nodiagonals: # If 'nodiagonals' is True...
    if edgestate == 1: # ...if 'edgestate' is 1...
        if x < x_cells-1: # ...if there are cells to the right...
            if y > 0: # ...if there are cells above...
                if grid[y-1][x+1] == ' ': # ...if the cell directly top-right is a space...
                    return False # Return False.
            if y < y_cells-1: # ...if there are cells below...
                if grid[y+1][x+1] == ' ': # ...if the cell directly bottom-right is a space...
                    return False # Return False.
        return True # Return True.

    elif edgestate == 2: # ...else, if 'edgestate' is 2...
        if x > 0: # ...if there are cells to the left...
            if y > 0: # ...if there are cells above...
                if grid[y-1][x-1] == ' ': # ...if the cell directly top-left is a space...
                    return False # Return False.
            if y < y_cells-1: # ...if there are cells below...
                if grid[y+1][x-1] == ' ': # ...if the cell directly bottom-left is a space...
                    return False # Return False.
        return True # Return True.

    elif edgestate == 4: # ...else, if 'edgestate' is 4...
        if y < y_cells-1: # ...if there are cells below...
            if x > 0: # ...if there are cells to the left...
                if grid[y+1][x-1] == ' ': # ...if the cell directly bottom-left is a space...
                    return False # Return False.
            if x < x_cells-1: # ...if there are cells to the right...
                if grid[y+1][x+1] == ' ': # ...if the cell directly bottom-right is a space...
                    return False # Return False.
        return True # Return True.

    elif edgestate == 8: # ...if 'edgestate' is 8...
        if y > 0: # ...if there are cells above...
            if x > 0: # ...if there are cells to the left...
                if grid[y-1][x-1] == ' ': # ...if the cell directly top-left is a space...
                    return False # Return False.
            if x < x_cells-1: # ...if there are cells to the right...
                if grid[y-1][x+1] == ' ': # ...if the cell directly top-right is a space...
                    return False # Return False.

        return True # Return True.
    return False # Return False.

else: # ...if not... (hence 'nodiagonals' is False)
    if [1, 2, 4, 8].count(edgestate): # ...count the number if times that 'edgestate' appears in that array.
        # If it does appear in the array...
        return True # Return True.

    return False # Return
```

This part of the function actually looks at the diagonals, checks their state and returns true or false as appropriate. Remember, earlier I mentioned that this function returns ‘True’ if the current cell should become a space, and ‘False’ if it should become a wall.

This checks if ‘nodiagonals’ is true, and looks for spaces in any diagonals. If there are spaces, the current cell should become a wall, hence the function returns ‘False’. If not, then the current cell should become a space, hence the function returns ‘True’. If ‘edgestate’ is not 1, 2, 4 or 8, the current cell should become a wall… hence the function returns ‘False’.

However, if ‘nodiagonals’ is false, if ‘edgestate’ ever appears in the list [1, 2, 4, 8] (hence, if ‘edgestate’ is any one of these values), the current cell should become a space, and hence returns ‘True’. If not, it should be a wall, and hence ‘False’.

This function is needed to ensure that the previous functions behave as appropriate, and determines what a particular cell should be (a wall or a space), based on the behaviours of its surrounding, adjacent cells, including the diagonals.

```
x_random = random.randint(0, x_cells-1) # Choose a random point in the x-axis, on the grid.
y_random = random.randint(0, y_cells-1) # Choose a random point in the y-axis, on the grid.
carve(x_random, y_random) # Use these random points and carve.
```

The above code simply selects a random point to carve from. This is needed to ensure that the generation of the maze via this algorithm is fully randomised, and hence is always different.

```
while(len(exposed)): # While there are exposed cells...
    pos = random.random() # Select a random number.
    choice = exposed[int(pos*len(exposed))] # This randomly selects an exposed cell from the list.
    if check(*choice): # If the selected cell should become a space...
        carve(*choice) # ...make this cell a space.
    else: # If not...
        wall(*choice) # ...make this cell a wall.
    exposed.remove(choice) # Remove this cell from the 'exposed' list.
```

While there are exposed cells, the system needs to be either carving to these cells, or making these cells into walls. The above code chooses a random cell (out of the exposed cells, from the list), and checks if this cell should become a wall or a space (using the 'check' function). After this cell is carved/walled off, it is removed from the 'exposed' list, since it's no longer exposed. This continues as long as there are exposed cells.

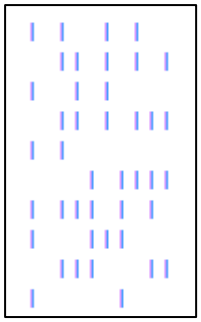
```
# The following changes any unexposed, unidentified cells to be walls.
for y in range(y_cells):
    for x in range(x_cells):
        if grid[y][x] == '?':
            grid[y][x] = '|'

# The following prints the entire maze.
for y in range(y_cells):
    maze = ' '
    for x in range(x_cells):
        maze += grid[y][x]
    print(maze)
```

The first of the above two paragraphs is used to simply 'sweep' the grid one last time, looking for any cells that are unassigned (are not walls or spaces). From here, it simply makes these cells into walls.

The second paragraph generates the maze as a string. The algorithm is now complete.

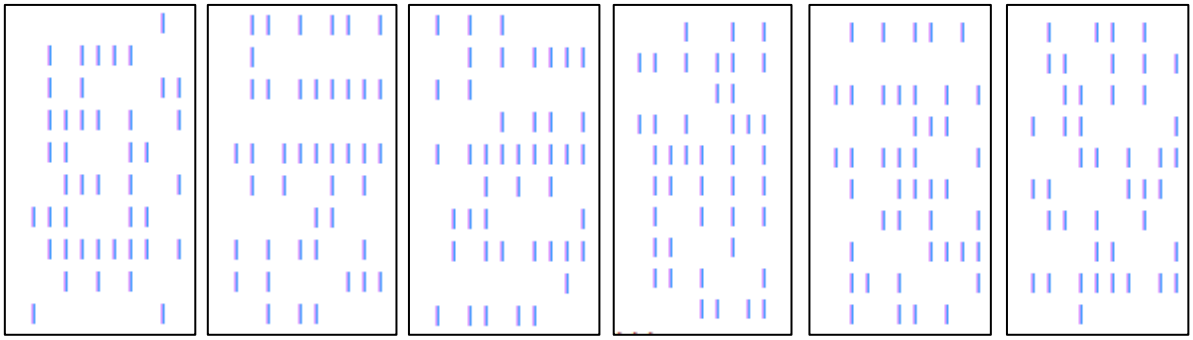
- Immediately upon testing the maze, I ran into an error. As it turns out, I missed a line in the 'check' function, meaning that cells that were outside of the maze (and hence out of range) were being referenced, causing errors. The line I missed was one which ensures a check is only carried out IF there are cells beneath the current cell. I fixed this, and this has been fixed in the screenshots shown previously.
- After this, there were no more errors, but there was a problem with cells occasionally being '?' cells – which shouldn't be allowed. I realised that I mistyped "grid[y][x] = '|'" as "grid[y][x] == '|'", meaning a comparison was being carried out, rather than an actual change of the current cell. I amended this, and there were no further problems.

Here are examples of mazes generated by this algorithm, with a 10×10 resolution.

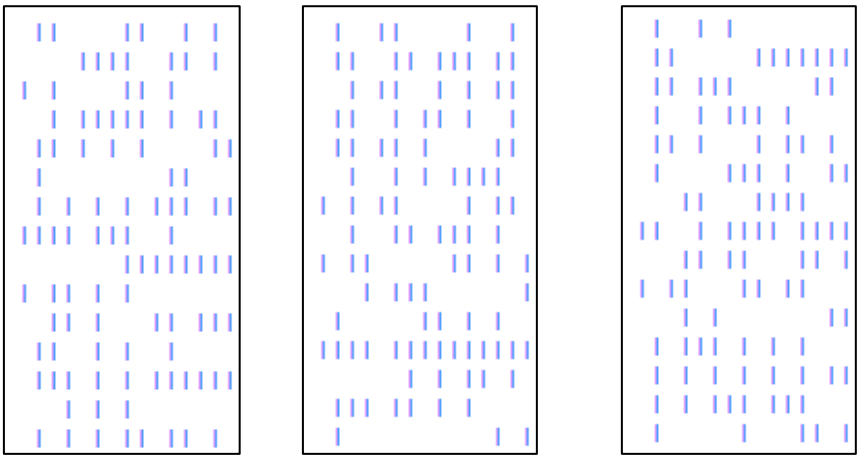


In these mazes, as previously mentioned, a space (' ') is a space in which a player can traverse. A vertical bar ('|') represents a wall.

As previously mentioned in design, a maze is 'solvable' if it's possible to get from one point to any other point on the maze. Any maze generated by this algorithm is solvable, and this is shown clearly in these mazes, where they are all solvable.

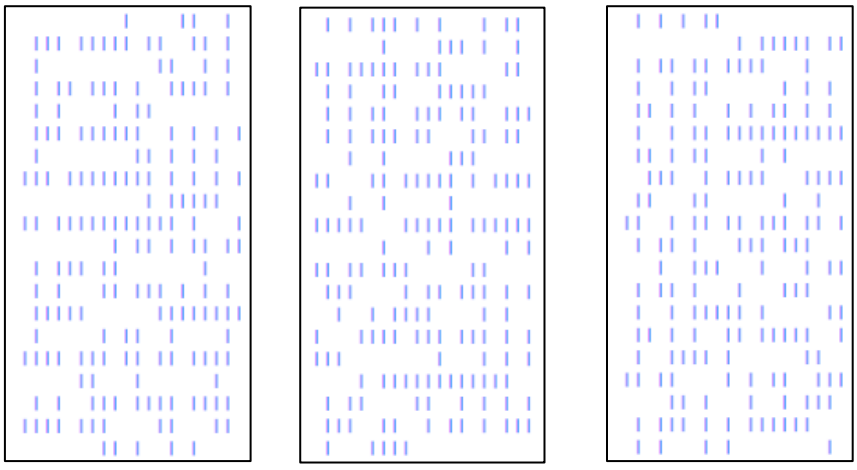


All of the mazes shown above are 10×10 mazes, for the 'easy' difficulty. I changed the resolution ('x_cells' and 'y_cells') to 15, in order to show generations of mazes for the 'medium' difficulty:



Again, these mazes are all fine and fully solvable – they don't have any cells which act abnormally (assuming either '?' or '+').

Finally, I checked if the algorithm behaves as expected for 20×20 mazes, for the 'hard' difficulty.



I showed my stakeholders all of these samples of every difficulty. I explained how the walls ('|') will be replaced by walls as shown in the diagram. Here are their responses:

- [redacted]: "Wow, these mazes are really good… I like that you've ensured that you can always solve every maze, no matter how large you make them."
- [redacted]: "It's good that these mazes are random, and provide such a variety – this means the game will never get boring."
- [redacted]: "These mazes look pretty challenging, while still being doable. Kudos for that."
- [redacted]: "It's a good thing these are all solvable, since if they weren't the game wouldn't be very playable. I also like how they generate literally immediately, meaning the user doesn't have to wait."
- [redacted]: "These mazes look super unbiased; I like that, because it keeps that element of randomness."

One final step I did was to change the end of the program into functions. The last set of instructions was named 'create()', and returns the 'maze_data' array. 'maze_data' is a 2 dimensional array that contains the locations of all walls (|), generated from the above algorithm. This is done so that the maze data can be used for the maze generation, rather than printed.

Below is all the code for the maze generation algorithm.

```
import random # Since the generation of this maze is random, I need the random module.

x_cells = 50 # This defines the number of cells in the x-direction.
y_cells = 50 # This defines the number of cells in the y-direction.
grid = [] # This allows me to build the grid from which the maze is generated.

"""
KEY:
'|' indicates a WALL
' ' indicates a SPACE
'+' indicates an EXPOSED UNDETERMINED LOCATION
'?' indicates an UNDETERMINED LOCATION
"""
for y in range(y_cells):
    row = [] # Creates a 'row' list for all y locations (10 rows)
    for x in range(x_cells):
        row.append('?') # Sets all cells to UNDETERMINED
    grid.append(row) # Fills the grid with these rows

exposed = [] # This is a list of EXPOSED UNDETERMINED LOCATIONS (+)

def carve(y, x): # This function makes the cell a space.
    if (0 > (x or y)) or ((x > x_cells) or (y > y_cells)) or (type(x) != int or type(y) != int):
        return # This is for validation - 'if invalid data has been entered, ignore.'

    grid[y][x] = ' ' # This takes the two parameters and updates location.
    adjacent = [] # This stores any adjacent cells.

    if x > 0: # This checks if there is a cell behind the current cell (x direction).
        if grid[y][x-1] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y][x-1] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y,x-1)) # Add this cell to the 'adjacent'

    if x < x_cells - 1: # This checks if there is a cell in front of the current cell (x direction).
        if grid[y][x+1] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y][x+1] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y,x+1)) # Add this cell to the 'adjacent' list.

    if y > 0: # This checks if there is a cell behind the current cell (y direction).
        if grid[y-1][x] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y-1][x] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y-1,x)) # Add this cell to the 'adjacent' list.

    if y < y_cells - 1: # This checks if there is a cell in front of the current cell (y direction).
        if grid[y+1][x] == '?': # If there is, and this adjacent cell is undetermined...
            grid[y+1][x] = '+' # ...Mark it as EXPOSED.
            adjacent.append((y+1,x)) # Add this cell to the 'adjacent' list.

    random.shuffle(adjacent) # Shuffles the list of adjacent cells.
    exposed.extend(adjacent) # Adds the list of adjacent cells to the 'exposed' list.
```

```
def wall(y, x): # This function make the cell a wall.
    if (0 > (x or y)) or ((x > x_cells) or (y > y_cells)) or (type(x) != int or type(y) != int):
        return # This is for validation - 'if invalid data has been entered, ignore.'
    grid[y][x] = '|' # This takes the two parameters and updates location.


'''
The following function determines if a cell should be a space or a wall.
If it returns True, it should become a space.
If it returns False, it should become a wall.
'''
def check(y, x, nodiagonals = True): # Takes the 'nodiagonals' parameter...
    # ...this parameter determines if a cell has NO diagonals (when True).
    edgestate = 0 # This is a way of determining the behaviour of the edges around a  point.

    if x > 0: # If the cell has cells to the left...
        if grid[y][x-1] == ' ': # ...if the cell directly to the left is a space...
            edgestate += 1 # Increase edgestate by 1.

    if x < x_cells-1: # If the cell has cells to the right...
        if grid[y][x+1] == ' ': # ...if the cell directly to the right is a space...
            edgestate += 2 # Increase edgestate by 2.

    if y > 0: # If the cell has cells above...
        if grid[y-1][x] == ' ': # ...if the cell directly above is a space...
            edgestate += 4 # Increase edgestate by 4.

    if y < y_cells-1: # If the cell has cells below...
        if grid[y+1][x] == ' ': # ...if the cell directly below is a space...
            edgestate += 8 # Increase edgestate by 8.

    if nodiagonals: # If 'nodiagonals' is True...
        if edgestate == 1: # ...if 'edgestate' is 1...
            if x < x_cells-1: # ...if there are cells to the right...
                if y > 0: # ...if there are cells above...
                    if grid[y-1][x+1] == ' ': # ...if the cell directly top-right is a space...
                        return False # Return False.
                if y < y_cells-1: # ...if there are cells below...
                    if grid[y+1][x+1] == ' ': # ...if the cell directly bottom-right is a space...
                        return False # Return False.
            return True # Return True.

        elif edgestate == 2: # ...else, if 'edgestate' is 2...
            if x > 0: # ...if there are cells to the left...
                if y > 0: # ...if there are cells above...
                    if grid[y-1][x-1] == ' ': # ...if the cell directly top-left is a space...
                        return False # Return False.
                if y < y_cells-1: # ...if there are cells below...
                    if grid[y+1][x-1] == ' ': # ...if the cell directly bottom-left is a space...
                        return False # Return False.
            return True # Return True.
```

```
        elif edgestate == 4: # ...else, if 'edgestate' is 4...
            if y < y_cells-1: # ...if there are cells below...
                if x > 0: # ...if there are cells to the left...
                    if grid[y+1][x-1] == ' ': # ...if the cell directly bottom-left is a space...
                        return False # Return False.
                if x < x_cells-1: # ...if there are cells to the right...
                    if grid[y+1][x+1] == ' ': # ...if the cell directly bottom-right is a space...
                        return False # Return False.
            return True # Return True.

        elif edgestate == 8: # ...if 'edgestate' is 8...
            if y > 0: # ...if there are cells above...
                if x > 0: # ...if there are cells to the left...
                    if grid[y-1][x-1] == ' ': # ...if the cell directly top-left is a space...
                        return False # Return False.
                if x < x_cells-1: # ...if there are cells to the right...
                    if grid[y-1][x+1] == ' ': # ...if the cell directly top-right is a space...
                        return False # Return False.

            return True # Return True.
        return False # Return False.

    else: # ...if not... (hence 'nodiagonals' is False)
        if [1, 2, 4, 8].count(edgestate): # ...count the number if times that 'edgestate' appears in that array.
            # If it does appear in the array...
            return True # Return True.

        return False # Return


x_random = random.randint(0, x_cells-1) # Choose a random point in the x-axis, on the grid.
y_random = random.randint(0, y_cells-1) # Choose a random point in the y-axis, on the grid.
carve(x_random, y_random) # Use these random points and carve.
```

```
def create():
    while(len(exposed)): # While there are exposed cells...
        pos = random.random() # Select a random number.
        choice = exposed[int(pos*len(exposed))] # This randomly selects an exposed cell from the list.
        if check(*choice): # If the selected cell should become a space...
            carve(*choice) # ...make this cell a space.
        else: # If not...
            wall(*choice) # ...make this cell a wall.
        exposed.remove(choice) # Remove this cell from the 'exposed' list.

# The following changes any unexposed, unidentified cells to be walls.
    for y in range(y_cells):
        for x in range(x_cells):
            if grid[y][x] == '?':
                grid[y][x] = '|'

# The following prints the entire maze.
    maze_data = []
    for y in range(y_cells):
        row = ' '
        for x in range(x_cells):
            row += grid[y][x]
        maze_data.append(row)
    return maze_data
```

I shall save the file for the algorithm independently, as a file called 'maze_algorithm.py'. This is done to make editing the editing of different parts of the code easier, and to avoid mistakes. The maze algorithm is now complete; I don't need to edit it until I wish to make changes.

The assets file

As mentioned before, each part of the program is being developed independently – as a result, I will have I need to make a file for the assets of the game. This file will contain the fonts, colours, sounds, and other settings for the game, and will be imported into all other files that require these.

```
import pygame
pygame.init()

screen_width = 1000
screen_height = 700
fps = 60
```

My first step was to import pygame (this is needed for the fonts and sounds) and to set the resolution of the window, and the framerate. These are needed for the main game file, since it will have the pygame window, which makes use of these variables.

```
import pygame
pygame.init()

screen_width = 1000
screen_height = 700
fps = 60

font_1 = pygame.font.SysFont("segoeui", 30)
font_2 = pygame.font.SysFont("segoeuiblack", 60)
```

Next, I imported the fonts and sound files.

For the fonts, 'font_1' is the primary font, while 'font_2' is the secondary font (Segoe UI and Segoe UI Black respectively.)

```
import pygame
pygame.init()

screen_width = 1000
screen_height = 700
fps = 60

font_1 = pygame.font.SysFont("segoeui", 30)
font_2 = pygame.font.SysFont("segoeuiblack", 60)

entry_sound = pygame.mixer.Sound("sounds/entry.wav")
exit_sound = pygame.mixer.Sound("sounds/exit.wav")
menu_back_sound = pygame.mixer.Sound("sounds/menu_back.wav")
menu_forward_sound = pygame.mixer.Sound("sounds/menu_forward.wav")
ticking_sound = pygame.mixer.Sound("sounds/ticking.ogg")
lose_sound = pygame.mixer.Sound("sounds/you_lose.wav")
win_sound = pygame.mixer.Sound("sounds/you_win.wav")
```

For the primary font, I set the size to 30px, while for the secondary, I set it to 60px. These values are estimates, and may change as implementation progresses, depending on how well sized the text is to the textboxes.

Before adding in the sound files, I checked that all my sound files were of the appropriate format. The pygame.mixer module only allows sounds that are either compressed .ogg files, or uncompressed .wav files. Fortunately, all my files were .wav files, except from 'ticking', which was an .ogg file. These had all been saved to a 'sounds' folder in the same directory as the main game code.

```
import pygame
pygame.init()

# screen setup
screen_width = 1000
screen_height = 700
fps = 60

# font setup
font_1 = pygame.font.SysFont("segoeui", 30)
font_2 = pygame.font.SysFont("segoeuiblack", 60)

# sound setup
entry_sound = pygame.mixer.Sound("sounds/entry.wav")
exit_sound = pygame.mixer.Sound("sounds/exit.wav")
menu_back_sound = pygame.mixer.Sound("sounds/menu_back.wav")
menu_forward_sound = pygame.mixer.Sound("sounds/menu_forward.wav")
ticking_sound = pygame.mixer.Sound("sounds/ticking.ogg")
lose_sound = pygame.mixer.Sound("sounds/you_lose.wav")
win_sound = pygame.mixer.Sound("sounds/you_win.wav")

# colour setup
black = (0, 0, 0)
white = (255, 255, 255)
red = (255, 0, 0)
orange = (255, 127, 0)
yellow = (255, 255, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
indigo = (46, 43, 95)
violet =(139, 0, 255)
```

Finally, I added the colours. These are all stored as tuples, so that they cannot be accidentally modified in the main file. I included all the elementary colours of the rainbow, for completeness and so that I can explore different options with my stakeholders, in case if they change their minds about any of the colours.

Linking the maze generation algorithm and Pygame

The main file that calls and links all of these different parts of the program will be called 'main.py'.

I need to start creating this part of the program, in order to be able to have the maze loaded on Pygame. For now, I will only have part of main.py developed; it will be expanded on further during development.

I also will need separate files for characters and UI elements.

I will create all of these files now in the same directory. Some won't have any code in them (for now), and I will work on these later.

```
''' Imports all needed modules and libraries.'''
import pygame
from pygame.locals import *
from maze_algorithm import *
from ui_elements import *
from characters import *

pygame.init() # Initialises pygame.
clock = pygame.time.Clock() # Renames the pygame clock so it can be used.
screen = pygame.display.set_mode((1000, 700)) # Creates a screen of the required resolution.

running = True # Creates a variable called 'running'.
while running: # While running is True...
    clock.tick(60) # Refresh the screen 60 times a second.
    for event in pygame.event.get():
        if event.type == QUIT: # If the user presses the [X] button...
            running = False # Set running to False (and hence break this loop).

    screen.fill((0, 0, 0)) # Fill the screen with black.

    pygame.display.update() # Update the screen.

pygame.quit() # Quit pygame.
```

In main.py, I initialised Pygame and imported all of these as modules. I also set up the clock, the game loop, and a plain black Pygame window of the appropriate resolution.

The game loop keeps the Pygame window open, and refreshes the screen 60 times a second (60 FPS).

Next, I need to create wall objects, and hence a 'wall' class. This will be done in characters.py.

```
import pygame
from assets import *

class Wall:
    def __init__(self, x, y, size, colour, screen):
        self.x = x
        self.y = y
        self.size = size
        self.colour = colour
        self.screen = screen

    def draw(self):
        pygame.draw.rect(self.screen, (self.colour), (self.x, self.y, self.size, self.size))
        pygame.display.update()
```

I imported the 'assets', so I can make use of colours easily. The code for the wall class is simple enough – the dimensions are passed in as arguments, as is the colour, and a rectangle is drawn and displayed from this.

I quickly checked that this code worked, using a game loop and instantiation of a wall object.

It worked as expected. All a wall is, in this context, is a single unit 'block' or square, which replaces all the '|' characters from the maze generation algorithm. When taken together, the maze is displayed on screen.

Finally, I have to link these two, in main.py, so that a generated maze is displayed.

Before doing this, I added a border around my maze. This was done in order to make it clear where the border is, and prevent players from travelling outside the mazes' boundaries.

```
for index, row in enumerate(maze_data):
    row = "|" + row[1:] + "|"
    maze_data[index] = row
maze_data.insert(0, "|" * 22)
maze_data.append("|" * 22)
```

I used this code to add borders to the maze (for a 20 by 20, 'hard' maze). As a result, the entire grid is now technically 22 by 22, if we include the borders, but this won't matter.

In order to make this code appropriate for any generation, I switched the upper limit for the insertion of walls for y_cells and x_cells, both plus 2 (to account for the new walls in every direction.)

```
for index, row in enumerate(maze_data):
    row = "|" + row[1:] + "|"
    maze_data[index] = row
maze_data.insert(0, "|" * (x_cells + 2)
maze_data.append("|" * (y_cells+  2))
```

Next, I instantiated walls for every '|' in 'maze_data', using the following code:

```
walls = []
for x, tiles in enumerate(maze_data):
    for y, tile in enumerate(tiles):
        if tile == "|":
            wall = Wall(50+(y*10), 10+(x*10), 10, white, screen)
            walls.append(wall)
```

I took every row in the arrays that make up 'maze_data', and went through these arrays. For every '|' in these rows, a wall object would be created (from 'characters.py'), taking the position, size, colour and screen that the wall (which is just a white square cell).

- The position is based on the position of a '|' in the 'maze_data' array.
- There is an x and a y position in the array, since this is a 2 dimensional array, as mentioned previously.
- Because there are two coordinates determining the location of a wall in 'maze_data', I can use this position to translate directly to a Pygame coordinate, for any and every wall. This is what the '50+(y*10)' and '10+(x*10)' does, it takes the coordinate in the array, and translates this to a pygame location.
- The constants, 50 and 10 are used to better centre the maze.
- The x and y coordinates are switched because using them the other way around will reverse the orientation of the maze (compared to the original maze data).
- This isn't an issue aesthetically, because the maze is still solvable, regardless of orientation, but this can cause problems when preventing players from traversing through the walls, using the maze data.

Finally, in the game loop, I included the following code:

```
for wall in walls:
    wall.draw()
```

This goes through the 'walls' array, and draws every single wall to the screen, at their specified location. This is essential if I want to be able to see the walls, and hence the maze composed of the walls.

Immediately, I ran into problems.

- The maze would flicker uncontrollably, for no apparent reason.
- I realised that, in the 'draw' method for the Walls class (from characters.py), I had left in the 'pygame.display.update()', method. This was causing problems with the game loop, causing the image to flicker uncontrollably.

Once this was resolved, I had a stable generation of a maze. I generated three mazes: easy, medium and hard, all according to the earlier specification.

I then presented these to the stakeholders.

- [redacted]: "I like how you've used black and white to keep it simple, and it's very clear where you can or can't go."
- [redacted]: "These mazes are great, they look complex enough, but still doable."
- [redacted]: "The colour contrast makes these great and easy to understand. Good job."

- [redacted]: “These are good, but are super small. With the small one, I could solve that even with the five second timer. These are two small to be challenging to anyone, I think you need to change the specification of the size of these mazes.”
- [redacted]: “These look great, but aren’t complicated enough… maybe you could expand them to give more of a challenge?”

I asked all the other stakeholders if they shared the same sentiment, and all of them agreed. We decided to remedy this by increasing the resolution of the maze for all difficulties. It was decided that the ‘hard’ difficulty was more appropriate for ‘easy’, and that ‘medium’ and ‘hard’ difficulty should be much higher resolutions.
Ultimately, it was decided that:

- Easy: 20 × 20
- Medium: 30 × 30
- Hard: 50 × 50

Finally, I needed a way of specifying immediately what maze needs to be generated, instead of hardcoding values for ‘y_cells’ and ‘x_cells’ in ‘maze_algorithm.py’.

- I put all of the functions and code in ‘maze_algorithm.py’ into one huge function, called ‘generate_maze’. This function takes ‘x_cells’ and ‘y_cells’ as it’s arguments, and from here, generates the maze.
- This ability to pass x_cells and y_cells in, and get a maze generation as an output, is why I had to modularise my program completely. Without it, I would not be able to generate different resolution mazes from ‘main.py.’

(The revised code will be shown at the end of this subsection.)

Finally, I tested the maze algorithm.

- I found that the algorithm does not always work when ‘x_cells’ and ‘y_cells’ are two different values (i.e. for non-square mazes), as shown below.

- I decided that this doesn't matter. My generations are all square shaped, and since the algorithm always works for square mazes, this isn't an issue. That being said, I cannot create rectangular mazes, so if the need ever arises, I would need to modify the algorithm.
- I tested progressively larger mazes. The table below shows how long every generation took.

Generation resolution (cells):	**Time taken (seconds):**
1×1 (1 cell)	≈ 1
10×10 (100 cells)	≈ 1
50×50 (2'500 cells)	≈ 1
100×100 (10'000 cells)	≈ 1
200×200 (40'000 cells)	≈ 2
250×250 (62'500 cells)	≈ 2
500×500 (250'000 cells)	≈ 7
1000×1000 (1'000'000 cells)	≈ 50

These results are very good.

- The largest generation required, 50 by 50, takes approximately one second to generate (slightly less). This is very fast, and as a result, the user will not have to wait for the game to load in, which is an advantage in terms of enjoyability.
- As the resolution grows, the time taken increases proportional to n^2, where n is the width and height. As a result, as n gets larger, the time taken accelerates massively. I can predict that, for example, 10'000×10'000 (100'000'000 cells, or 100 million) may take several minutes to generate, and that 1'000'000×1'000'000 (1'000'000'000'000 cells, or 1 trillion) would likely take hours to generate.
- Of course, this isn't relevant to the current game, since these resolutions are far more massive than any of the resolutions I use.

The algorithm guarantees that all mazes are solvable, so this is not an issue. As a result, the algorithm fulfils its original purpose: it generates solvable, random mazes, every time, for a specified resolution, within a reasonable, small amount of time. As a result, the algorithm has been perfected.

Its full code is shown here.

```
import random # Since the generation of this maze is random, I need the random module.

def generate_maze(x_cells, y_cells):
#x_cells = 20 # This defines the number of cells in the x-direction.
#y_cells = 20 # This defines the number of cells in the y-direction.
    grid = [] # This allows me to build the grid from which the maze is generated.

    for y in range(y_cells):
        row = [] # Creates a 'row' list for all y locations (10 rows)
        for x in range(x_cells):
            row.append('?') # Sets all cells to UNDETERMINED
        grid.append(row) # Fills the grid with these rows



    exposed = [] # This is a list of EXPOSED UNDETERMINED LOCATIONS (+)

    def carve(y, x): # This function makes the cell a space.
        if (0 > (x or y)) or ((x > x_cells) or (y > y_cells)) or (type(x) != int or type(y) != int):
            return # This is for validation - 'if invalid data has been entered, ignore.'

        grid[y][x] = ' ' # This takes the two parameters and updates location.
        adjacent = [] # This stores any adjacent cells.

        if x > 0: # This checks if there is a cell behind the current cell (x direction).
            if grid[y][x-1] == '?': # If there is, and this adjacent cell is undetermined...
                grid[y][x-1] = '+' # ...Mark it as EXPOSED.
                adjacent.append((y,x-1)) # Add this cell to the 'adjacent'

        if x < x_cells - 1: # This checks if there is a cell in front of the current cell (x direction).
            if grid[y][x+1] == '?': # If there is, and this adjacent cell is undetermined...
                grid[y][x+1] = '+' # ...Mark it as EXPOSED.
                adjacent.append((y,x+1)) # Add this cell to the 'adjacent' list.

        if y > 0: # This checks if there is a cell behind the current cell (y direction).
            if grid[y-1][x] == '?': # If there is, and this adjacent cell is undetermined...
                grid[y-1][x] = '+' # ...Mark it as EXPOSED.
                adjacent.append((y-1,x)) # Add this cell to the 'adjacent' list.

        if y < y_cells - 1: # This checks if there is a cell in front of the current cell (y direction).
            if grid[y+1][x] == '?': # If there is, and this adjacent cell is undetermined...
                grid[y+1][x] = '+' # ...Mark it as EXPOSED.
                adjacent.append((y+1,x)) # Add this cell to the 'adjacent' list.

        random.shuffle(adjacent) # Shuffles the list of adjacent cells.
        exposed.extend(adjacent) # Adds the list of adjacent cells to the 'exposed' list.
```

```
    def wall(y, x): # This function make the cell a wall.
        if (0 > (x or y)) or ((x > x_cells) or (y > y_cells)) or (type(x) != int or type(y) != int):
            return # This is for validation - 'if invalid data has been entered, ignore.'
        grid[y][x] = '|' # This takes the two parameters and updates location.



    def check(y, x, nodiagonals = True): # Takes the 'nodiagonals' parameter...
    # ...this parameter determines if a cell has NO diagonals (when True).
        edgestate = 0 # This is a way of determining the behaviour of the edges around a  point.

        if x > 0: # If the cell has cells to the left...
            if grid[y][x-1] == ' ': # ...if the cell directly to the left is a space...
                edgestate += 1 # Increase edgestate by 1.

        if x < x_cells-1: # If the cell has cells to the right...
            if grid[y][x+1] == ' ': # ...if the cell directly to the right is a space...
                edgestate += 2 # Increase edgestate by 2.

        if y > 0: # If the cell has cells above...
            if grid[y-1][x] == ' ': # ...if the cell directly above is a space...
                edgestate += 4 # Increase edgestate by 4.

        if y < y_cells-1: # If the cell has cells below...
            if grid[y+1][x] == ' ': # ...if the cell directly below is a space...
                edgestate += 8 # Increase edgestate by 8.

        if nodiagonals: # If 'nodiagonals' is True...
            if edgestate == 1: # ...if 'edgestate' is 1...
                if x < x_cells-1: # ...if there are cells to the right...
                    if y > 0: # ...if there are cells above...
                        if grid[y-1][x+1] == ' ': # ...if the cell directly top-right is a space...
                            return False # Return False.
                    if y < y_cells-1: # ...if there are cells below...
                        if grid[y+1][x+1] == ' ': # ...if the cell directly bottom-right is a space...
                            return False # Return False.
                return True # Return True.

            elif edgestate == 2: # ...else, if 'edgestate' is 2...
                if x > 0: # ...if there are cells to the left...
                    if y > 0: # ...if there are cells above...
                        if grid[y-1][x-1] == ' ': # ...if the cell directly top-left is a space...
                            return False # Return False.
                    if y < y_cells-1: # ...if there are cells below...
                        if grid[y+1][x-1] == ' ': # ...if the cell directly bottom-left is a space...
                            return False # Return False.
                return True # Return True.
```

```
            elif edgestate == 4: # ...else, if 'edgestate' is 4...
                if y < y_cells-1: # ...if there are cells below...
                    if x > 0: # ...if there are cells to the left...
                        if grid[y+1][x-1] == ' ': # ...if the cell directly bottom-left is a space...
                            return False # Return False.
                    if x < x_cells-1: # ...if there are cells to the right...
                        if grid[y+1][x+1] == ' ': # ...if the cell directly bottom-right is a space...
                            return False # Return False.
                return True # Return True.

            elif edgestate == 8: # ...if 'edgestate' is 8...
                if y > 0: # ...if there are cells above...
                    if x > 0: # ...if there are cells to the left...
                        if grid[y-1][x-1] == ' ': # ...if the cell directly top-left is a space...
                            return False # Return False.
                    if x < x_cells-1: # ...if there are cells to the right...
                        if grid[y-1][x+1] == ' ': # ...if the cell directly top-right is a space...
                            return False # Return False.

                return True # Return True.
            return False # Return False.

        else: # ...if not... (hence 'nodiagonals' is False)
            if [1, 2, 4, 8].count(edgestate): # ...count the number if times that 'edgestate' appears in that array.
                # If it does appear in the array...
                return True # Return True.

            return False # Return


    x_random = random.randint(0, x_cells-1) # Choose a random point in the x-axis, on the grid.
    y_random = random.randint(0, y_cells-1) # Choose a random point in the y-axis, on the grid.
    carve(x_random, y_random) # Use these random points and carve.
```

```
    def create():
        while(len(exposed)): # While there are exposed cells...
            pos = random.random() # Select a random number.
            choice = exposed[int(pos*len(exposed))] # This randomly selects an exposed cell from the list.
            if check(*choice): # If the selected cell should become a space...
                carve(*choice) # ...make this cell a space.
            else: # If not...
                wall(*choice) # ...make this cell a wall.
            exposed.remove(choice) # Remove this cell from the 'exposed' list.

    # The following changes any unexposed, unidentified cells to be walls.
        for y in range(y_cells):
            for x in range(x_cells):
                if grid[y][x] == '?':
                    grid[y][x] = '|'

    # The following prints the entire maze.
        maze_data = []
        for y in range(y_cells):
            row = ' '
            for x in range(x_cells):
                row += grid[y][x]
            maze_data.append(row)

        return maze_data

    maze_data = create()
    return maze_data

"""
KEY:
'|' indicates a WALL
' ' indicates a SPACE
'+' indicates an EXPOSED UNDETERMINED LOCATION
'?' indicates an UNDETERMINED LOCATION
"""

'''
The 'check' function determines if a cell should be a space or a wall.
If it returns True, it should become a space.
If it returns False, it should become a wall.
'''
```

Implementing UI elements

I also need to implement the two main UI elements: textboxes and buttons. These will be implemented in ui_elements.py.

I started by doing the basics – importing pygame and the assets file, and adding all the appropriate methods, with a 'pass' in every one.

```
import pygame
from assets import *

class Textbox:
    def __init__():
        pass
    def draw():
        pass

class Button:
    def __init__():
        pass
    def draw():
        pass
    def is_clicked():
        pass
```

Referencing the class diagram for the textbox, I can implement the arguments that will be passed into the textbox class, via the '__init__' method.

Next, I added everything to the '__init__' method, passing all the arguments in for an object, and assigning them to the correct 'self' variables for a particular instance.

For the text positioning, I wanted the text to sit directly in the middle of the box. As a result, I calculated the positioning of the text by taking the position, width and height of the box, as seen in the screenshot.

```
class Textbox:
    def __init__(x, y, colour, text, name, width, height, font):
        self.name = name # name of the textbox
        self.x_pos = x # x position of the textbox
        self.y_pos = y # y position of the textbox
        self.colour = colour # colour of the textbox
        self.text = str(text) # text on the textbox
        self.width = width # width of the textbox
        self.height = height # height of the textbox
        self.font = font # font used with the text
        self.txt_x = x + (width/2) # x position of the text
        self.txt_y = y + (height/2) # y position of the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
```

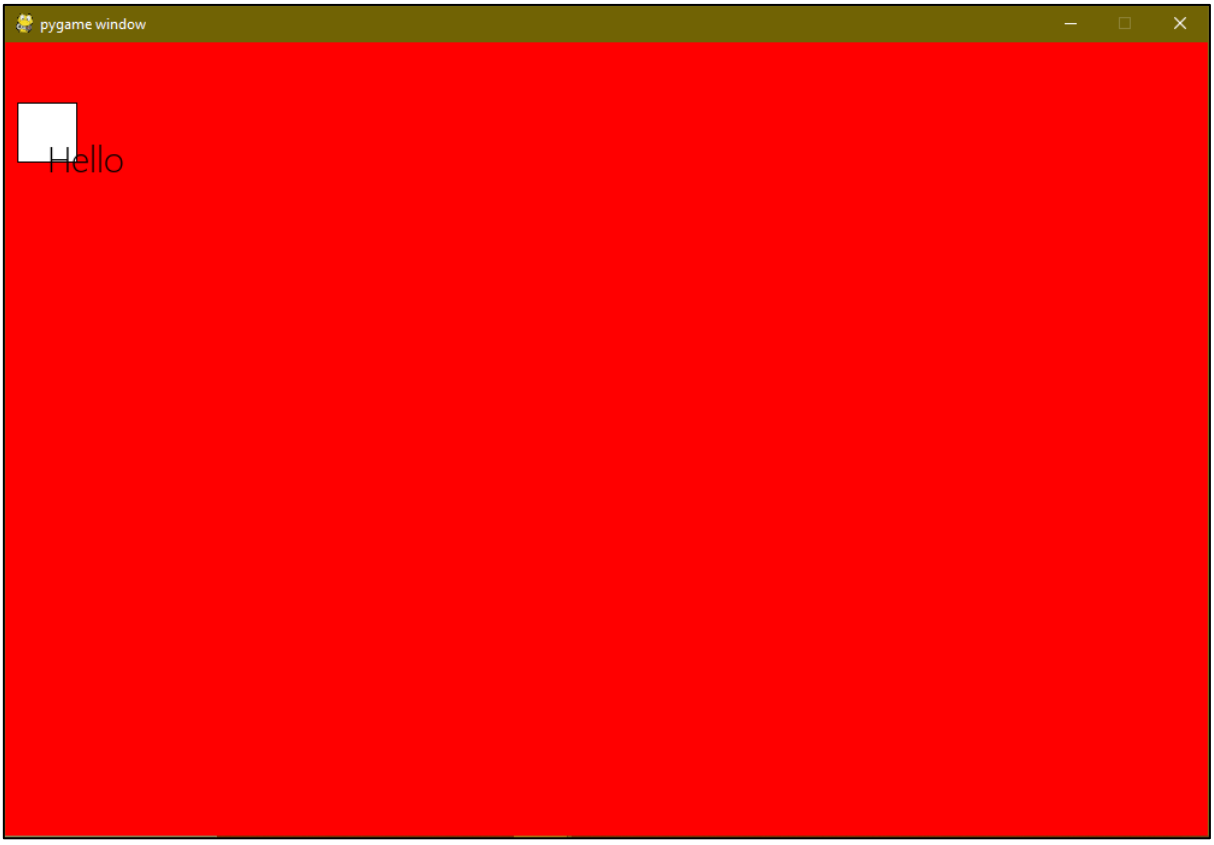
Finally, I need to draw the textbox to the screen. This will be done using the 'draw' method.

```
    def draw():
        pygame.draw.rect(screen, ((self.colour)), (self.x_pos, self.y_pos, self.width, self.height)) # draws the textbox
        pygame.draw.rect(screen, ((black)), (self.x_pos, self.y_pos, self.width, self.height), 1) # draws the outline
        screen.blit(self.txt, self.txt_x, self.txt_y) # draws the text
```

I tested this code quickly, using a primitive game loop, to see if I had made any mistakes. I ran into a few problems.

- I had forgotten to pass 'self' into all of the methods. I amended this.
- I had not ensured that 'self.txt_x' and 'self.txt_y' were only integers. I amended this by adding 'int()' to both.
- I had not placed brackets around 'self.txt_x, self.txt_y' in the draw method, and hence did not specify them as coordinates. I amended this.

After this, I tried again, and did not run into any more syntax errors. I did encounter a major logical error.



The issue here is that the text is not falling inside the space as it should. It isn't centred at all – my method for centring text has not worked. As a result, I need to find another way of doing this.

Pygame has a 'Rect' class for storing rectangular coordinates. Since I need the text to be centred the middle of the textbox, I can use a method from pygame.Rect: rect.center returns the centre of the rectangle. I can use this to replace 'txt_x' and 'txt_y' (the coordinates for the text), but in order to do this, I must change my '__init__' method, including a self.rect attribute, that stores the coordinates, width and height of the rectangle that is the textbox.

Because of this, I can get rid of 'txt_x' and 'txt_y'. I can also get rid of 'x_pos', 'y_pos', 'width' and 'height' and pass these directly into the 'self.rect' attribute.

I can also replace the coordinates, width and height arguments in the 'draw' method; since self.rect stores the coordinates and width and height, I can amend "pygame.draw.rect(screen, self.colour, (self.x_pos, self.y_pos, self.width, self.height))" to "pygame.draw.rect(screen, self.colour, self.rect)", and do the same for the outline box. Doing all of this is a good idea because it simplifies my code massively, and makes it a lot easier to understand in the future.

I tested this again, and got the expected results. I also tried this repeatedly for different sizes – the textbox always behaved as expected, with the text always being centered, no matter what size, the correct font and colour always being used.

The textbox class is now complete. It is shown above.

```
class Textbox:
    def __init__(self, x, y, colour, text, name, width, height, font):
        self.name = name # name of the textbox
        self.colour = colour # colour of the textbox
        self.text = str(text) # text on the textbox
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
        self.rect = pygame.Rect((x, y, width, height)) # stores the coordinates, width and height of the rectangle

    def draw(self):
        pygame.draw.rect(screen, self.colour, self.rect) # draws the textbox
        pygame.draw.rect(screen, black, self.rect, 1) # draws the outline
        txt_rect = self.txt.get_rect(center=self.rect.center) # draws the text to the center of the textbox
        screen.blit(self.txt, txt_rect) # draws the text
```

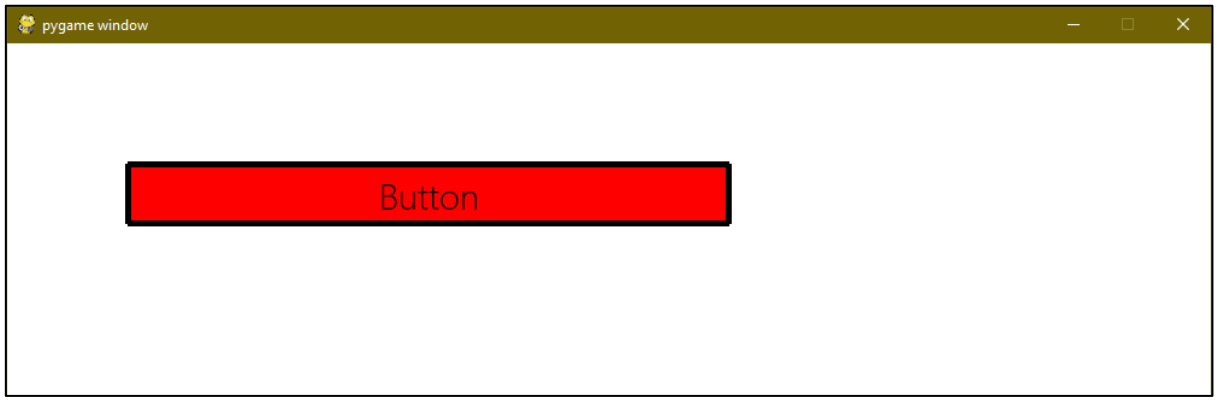
The button class is identical except:

a) It can be clicked
b) It has a thicker border around the box
c) It has a 'self.clicked' attribute

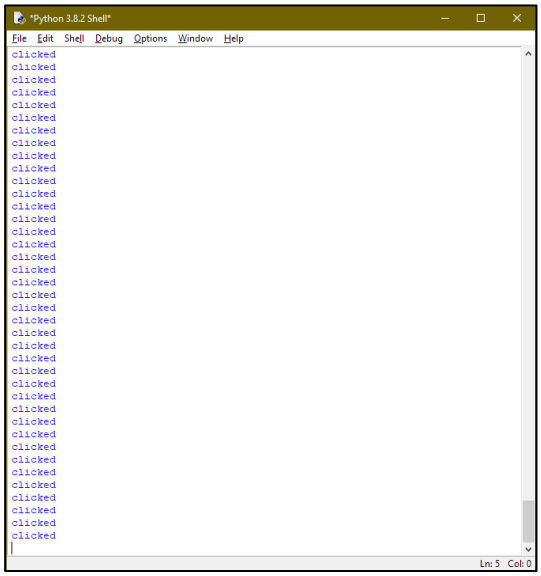
```
class Button:
    def __init__(self, x, y, colour, text, name, width, height, font):
        self.name = name # name of the button
        self.colour = colour # colour of the button
        self.text = str(text) # text on the button
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
        self.rect = pygame.Rect((x, y, width, height)) # stores the coordinates, width and height of the rectangle
        self.clicked = False

    def draw(self):
        pygame.draw.rect(screen, self.colour, self.rect) # draws the button
        pygame.draw.rect(screen, black, self.rect, 5) # draws the outline
        txt_rect = self.txt.get_rect(center=self.rect.center) # draws the text to the center of the button
        screen.blit(self.txt, txt_rect) # draws the text

    def is_clicked():
        pass
```

I immediately copied all of the methods from the textbox class to the button class, changing the thickness of the border and names/annotations as appropriate, as well as adding new the attribute.

```
class Button:
    def __init__(self, x, y, colour, text, name, width, height, font):
        self.name = name # name of the button
        self.colour = colour # colour of the button
        self.text = str(text) # text on the button
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
                                                           colour)))
                                                           ores the coordinates, width and height of the rectangle
    def
                                                           draws the button
                                                           ws the outline
                                                           r) # draws the text to the center of the button
    def
                                                           ct.collidepoint(pygame.mouse.get_pos()) == True:
    def
                                                           he button
                                                           # draws the outline
                                                           r) # draws the text to the center of the button
        print("clicked")
        self.clicked = False
```

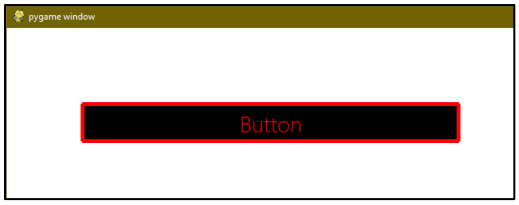


I quickly realised that it's better to have a method that checks if the button is clicked, which then calls the 'is_clicked' method. My reasoning behind this is that by splitting them, I can separate the checking, and the change of appearance of the button/the calling of the function that moves the game to the next required state. (In the above, the latter has been replaced with 'print("clicked")', for testing purposes. This will be altered soon.)



I tested this button after adding a game loop to the program, and found that, for the most part, it operated as expected. However, I had two major issues:

- The text would become invisible when the button is clicked, because it would blend in with the background (simple solution: make sure the text also changes colour, in the code above).
- Since the game refreshes 60 times a second, if I were



to hold down on the button for 1 second, the system would print ‘clicked’ 60 times.

I went about remedying the first issue immediately, by creating a new attribute in the __init__ method. I created a new attribute called ‘self.alt_txt’, which is identical to ‘self.txt’, except the colour is set to ‘self.colour’ instead of ‘black’ (and hence acting as the inverted version of the text.) I then modified the code in the ‘is_clicked’ method, referencing ‘self.alt_txt’ instead of ‘self.txt’. This proved effective immediately, as shown below:

The clicked button now has visible text. This issue has been solved.

I then went about solving the second problem. I decided to rewrite when ‘print(“clicked”)’ is used, to the point at which the user RELEASES the button, rather than as they’re clicking it. This is because, users may click buttons for varying lengths of time, which can result in a function being called more than once, which can cause issues with the program. Instead, by having the function called upon release, it guarantees that the function is only called once (for any button press).

I changed the code to behave like this.

Finally, I need the button to call some function when clicked. This function name will be passed into the button, and this will be called when the button is clicked (upon release), where ‘print(“clicked”’ was previously.

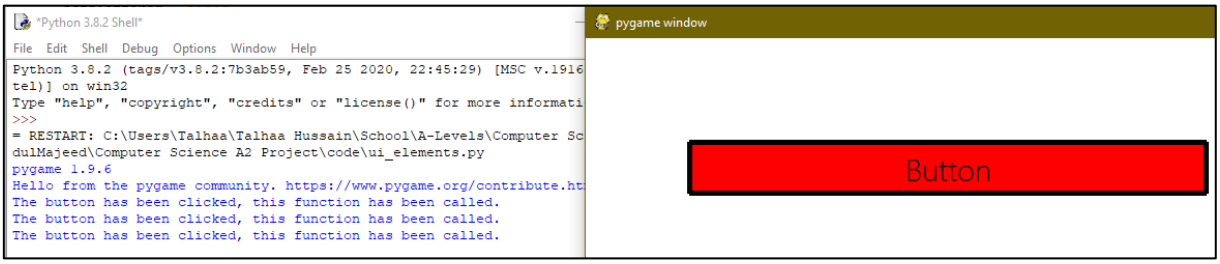
I implemented and tested this, using a function called ‘function_1’, which simply prints some text, and passing this into the button object.

```
    def check_clicked(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN:
            if self.rect.collidepoint(pygame.mouse.get_pos()) == True:
                self.clicked = True
                self.is_clicked()
        elif event.type == pygame.MOUSEBUTTONUP:
            if self.clicked:
                self.function()
            self.clicked = False


    def is_clicked(self):
        pygame.draw.rect(screen, black, self.rect) # draws the button
        pygame.draw.rect(screen, self.colour, self.rect, 5) # draws the outline
        txt_rect = self.alt_txt.get_rect(center=self.rect.center) # draws the text to the center of the button
        screen.blit(self.alt_txt, txt_rect) # draws the text



def function_1():
    print("The button has been clicked, this function has been called.")

button = Button(100, 100, red, "Button", "hi", 500, 50, font_1, function_1)
while 1:
    for event in pygame.event.get():
        if event.type == QUIT:
            exit()
        button.check_clicked(event)
    clock = pygame.time.Clock()
    clock.tick(144)
    screen.fill((white))

    button.draw()
    button.check_clicked(event)

    pygame.display.update()
```

I tested this code, ensuring that whenever the button is clicked, the function is called (and hence the text is printed.)

I also had to ensure that the function is only called when the button is RELEASED, so that the function is only called once per button press.

The button functioned exactly as expected.



The button and textbox objects are now fully functional and working correctly. As a result, I can remove all the extra code that was used for testing (the game loop and instantiation of objects), since this will be done in main.py.

```
class Textbox:
    def __init__(self, x, y, colour, text, name, width, height, font):
        self.name = name # name of the textbox
        self.colour = colour # colour of the textbox
        self.text = str(text) # text on the textbox
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
        self.alt_txt = font.render(self.text, True, ((self.colour)))
        self.rect = pygame.Rect((x, y, width, height)) # stores the coordinates, width and height of the rectangle

    def draw(self):
        pygame.draw.rect(screen, self.colour, self.rect) # draws the textbox
        pygame.draw.rect(screen, black, self.rect, 1) # draws the outline
        txt_rect = self.txt.get_rect(center=self.rect.center) # draws the text to the center of the textbox
        screen.blit(self.txt, txt_rect) # draws the text
```

Above is the final textbox class.

```
class Button:
    def __init__(self, x, y, colour, text, name, width, height, font, function):
        self.name = name # name of the button
        self.colour = colour # colour of the button
        self.text = str(text) # text on the button
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
        self.alt_txt = font.render(self.text, True, ((self.colour)))
        self.rect = pygame.Rect((x, y, width, height)) # stores the coordinates, width and height of the rectangle
        self.clicked = False # determines whether the button is currently clicked
        self.function = function # stores the function to be called when the button is clicked

    def draw(self):
        pygame.draw.rect(screen, self.colour, self.rect) # draws the button
        pygame.draw.rect(screen, black, self.rect, 5) # draws the outline
        txt_rect = self.txt.get_rect(center=self.rect.center) # draws the text to the center of the button
        screen.blit(self.txt, txt_rect) # draws the text

    def check_clicked(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN: # if the mouse is clicked...
            if self.rect.collidepoint(pygame.mouse.get_pos()) == True: # ...and the mouse is on the button...
                self.clicked = True # the button is clicked.
                self.is_clicked() # call 'is_clicked'
        elif event.type == pygame.MOUSEBUTTONUP: # when the button is released...
            if self.clicked: # if it WAS clicked...
                self.function() # calls the appropriate function
            self.clicked = False # sets the button to unclicked once again, as the button is released


    def is_clicked(self):
        pygame.draw.rect(screen, black, self.rect) # draws the button
        pygame.draw.rect(screen, self.colour, self.rect, 5) # draws the outline
        txt_rect = self.alt_txt.get_rect(center=self.rect.center) # draws the text to the center of the button
        screen.blit(self.alt_txt, txt_rect) # draws the text
```

Above is the final button class.

Implementing the character

From characters.py, I need to create the character class.

```
class Character:
    def __init__(self, x, y, width, height, colour, screen, controls):
        self.x_pos = x
        self.y_pos = y
        self.width = width
        self.height = height
        self.colour = colour
        self.screen = screen
        self.controls = controls
        self.alive = True

    def draw(self):
        pass

    def move_up(self):
        pass

    def move_down(self):
        pass

    def move_right(self):
        pass

    def move_left(self):
        pass

    def die(self):
        pass
```

I initialised the class as shown, using the class diagram from section 2 as a blueprint.

Obviously, 'alive' doesn't need to be passed into the class, because it's a Boolean value, according to the state of the character at any time.

Next, I implemented the 'draw', 'move_up', 'move_down',

'move_right' and 'move_left' methods. These methods all do as their name implies.

- I realised that, similar to the implementation of the button and textbox UI elements, I could use 'pygame.rect' to shorten and simplify my code.
- However, in this case, I need to keep 'self.x' and 'self.y' as attributes, because the square needs to be able to move, and as a result I need explicit variables to determine the location of the character, rather than implying the location using the 'x' and 'y' arguments.

```
class Character:
    def __init__(self, x, y, width, height, colour, screen, controls):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.colour = colour
        self.screen = screen
        self.controls = controls
        self.alive = True
        self.rect = pygame.Rect((self.x, self.y, width, height))

    def draw(self):
        pygame.draw.rect(self.screen, self.colour, self.rect)
```

I then implemented all the 'move' methods. I used an elementary game loop and 'pygame.key.get_pressed()' to detect movement, and passed 'controls' in as an array, containing the WASD controls referenced in pygame (this is what '[K_w, K_s, K_d, K_a]' means.)

```
    def move_up(self):
        if pressed_keys[self.controls[0]]:
            self.y -= 10

    def move_down(self):
        if pressed_keys[self.controls[1]]:
            self.y += 10

    def move_right(self):
        if pressed_keys[self.controls[2]]:
            self.x += 10

    def move_left(self):
        if pressed_keys[self.controls[3]]:
            self.x -= 10

    """def die(self):
        pass"""

character = Character(0, 0, 10, 10, green, screen, [K_w, K_s, K_d, K_a])

running = True
while running:
    clock.tick(60)
    for event in pygame.event.get():
        if event.type == QUIT:
            running = False



    pressed_keys = pygame.key.get_pressed()

    screen.fill(black)

    character.draw()
    character.move_up()
    character.move_down()
    character.move_left()
    character.move_right()

    pygame.display.update()

pygame.quit()
```

The move methods then move the character in the appropriate direction, when their keys are pressed.

I ran this code. Immediately, I ran into problems.

The square would not move. I remedied this by removing the 'self.rect' attribute – as it were, it turned out that this was interfering with the updating of 'self.x' and 'self.y', meaning that the object was not moving. Removing this code and replacing the code in the 'draw' method got rid of these problems.

I then realised that it was a better idea to put all of the 'draw' methods into a single 'draw' method. My reasoning behind this is that it makes the code much simpler, and means that I don't have to handle as many methods for the character class.

I also decided it was best to change the 'speed' of the character in every direction into a variable, instead of hard coding values. This is so that, later on in development, I could test and experiment

with what speeds work best, depending on the stakeholders' preferences. I had this passed into the class as an argument, so that I could choose speed upon instantiation. (In this case, I have defined 'speed' as the number of pixels moved up, down, left or right upon one press of any of the respective input keys.)

The updated methods are shown below.

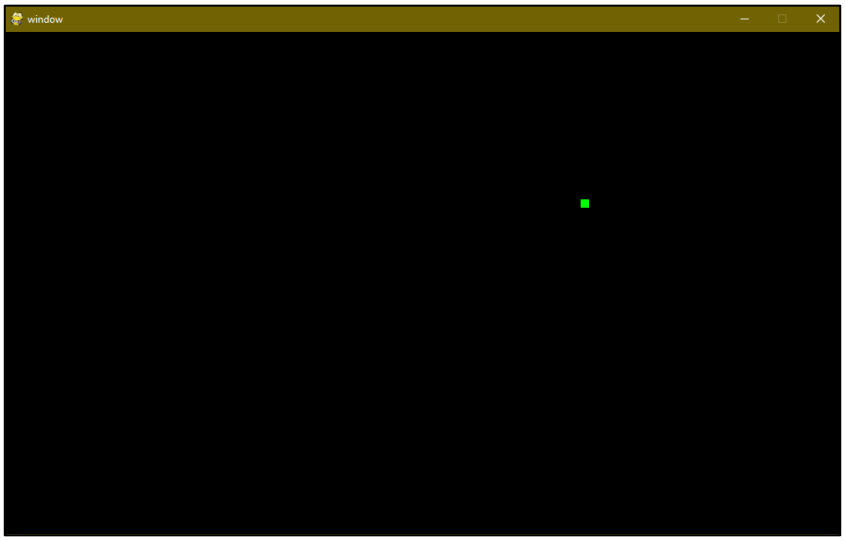
```
class Character:
    def __init__(self, x, y, width, height, colour, screen, controls, speed):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.colour = colour
        self.screen = screen
        self.controls = controls
        self.speed = speed
        self.alive = True

    def draw(self):
        pygame.draw.rect(self.screen, self.colour, pygame.Rect((self.x, self.y, self.width, self.height)))

    def move(self):
        if pressed_keys[self.controls[0]]:
            self.y -= self.speed

        if pressed_keys[self.controls[1]]:
            self.y += self.speed

        if pressed_keys[self.controls[2]]:
            self.x += self.speed

        if pressed_keys[self.controls[3]]:
            self.x -= self.speed
```

Finally, I introduced the 'die' method. For now, this is very basic, simply setting 'self.alive' to False. I also changed all the other methods to only operate if 'self.alive' is True, as shown.

```
    def draw(self):
        if self.alive == True:
            pygame.draw.rect(self.screen, self.colour, pygame.Rect((self.x, self.y, self.width, self.height)))

    def move(self):
        if self.alive == True:
            if pressed_keys[self.controls[0]]:
                self.y -= self.speed

            if pressed_keys[self.controls[1]]:
                self.y += self.speed

            if pressed_keys[self.controls[2]]:
                self.x += self.speed

            if pressed_keys[self.controls[3]]:
                self.x -= self.speed

    def die(self):
        self.alive = False
```

I tested out all of the methods.

- The green character would move in all directions correctly, from the corresponding inputs.



- As shown on the right, the green character would also appear on screen.
- Then, retesting using the 'die' method, I found that the character did not appear on screen – it worked exactly as expected, by not allowing any other methods to run.
- One mild concern was that the character was able to leave and return to the screen boundaries. That being said, this won't matter in the real game, as the character will be inside the walls of the maze, and unable to leave.
- However, I do need to come up with a way of ensuring that the character does not traverse through walls, whether this be the internal walls of the maze, or the walls on the outside.
- Finally, I moved 'pressed_keys = pygame.key.get_pressed()' into the 'move' method. This is so that it will check whether keys have been pressed every time the 'move' method is called.

Now, I finally have the character and the maze on the same screen.

I now must make sure that the character detects collisions with the walls, and cannot traverse through them.

I will do this in the 'move' method for the character, passing in the list of walls.

- I created a method that checks if the character has collided with a wall. This is called "wall_collision".

```
    def wall_collision(self, walls):
        for wall in walls:
            if wall.x == self.x and wall.y == self.y:
                return True
        return False
```

- I passed the 'walls' array from main.py into the 'move' method, and the 'wall_collision' method. This was done to ensure that I can check for any walls that have been collidededed with.

- From here, I modified the 'move' method to revert the player back to their original location if they traverse through a wall.
- Finally, I updated the player's spawning location in the character instantiation in main.py, to ensure that the character lands inside of the maze.

```
pressed_keys = pygame.key.get_pressed()
if self.alive == True:
    if pressed_keys[self.controls[0]]:
        self.y -= self.speed
        if self.wall_collision(walls):
            self.y += self.speed

    if pressed_keys[self.controls[1]]:
        self.y += self.speed
        if self.wall_collision(walls):
            self.y -= self.speed

    if pressed_keys[self.controls[2]]:
        self.x += self.speed
        if self.wall_collision(walls):
            self.x -= self.speed

    if pressed_keys[self.controls[3]]:
        self.x -= self.speed
        if self.wall_collision(walls):
            self.x += self.speed
```

I tested the code, and it worked well. The character was unable to move through walls, in any direction, and could not traverse outside the maze. However, the character was EXTREMELY fast and, as a result, uncontrollable at times.

Since I could not change the 'self.speed' attribute from the value '10', or else the character would partially traverse through walls, which would case separate issues.

Instead, I researched other methods to sort this out, and chose to add a movement delay variable, to slow the character down.

```
self.move_delay = 75
self.last_move = pygame.time.get_ticks()
```

```
def move(self, walls):
    now = pygame.time.get_ticks()

    if now - self.last_move > self.move_delay:
        self.last_move = now
```

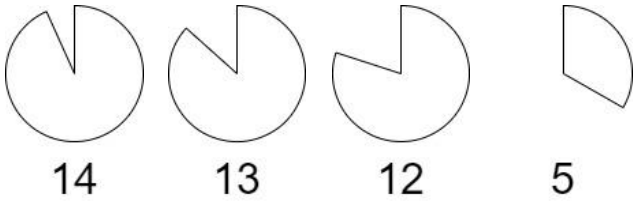
This worked, and slowed the character down, making the game playable.

The character is now complete.

I went to my stakeholders and asked them to review the character, and how they feel about its movement and behaviours.

- [redacted]: "The character moves fast enough, but not too fast, and is always in the right place. Good job."
- [redacted]: "The controls work, and you can't pass through walls, which is great. However, the mazes are a bit difficult to complete in real time, particularly the harder mazes; I think either changing the time allowed or making the mazes easier would help change this."
- [redacted]: "It works well, congrats."
- [redacted]: "I agree with [redacted], alternatively, you could give the player the option to backtrack a certain amount, seeing as there is a lot of room for them to make mistakes."
- [redacted]: "The character is a bit slow for my liking, but overall, it works well."

Addressing [redacted] and [redacted], I agreed – I decided that the game was overall too difficult at the current moment. However, because I had not yet implemented the timer and the lava, I decided that it was too early to pass a complete judgement on how the game should behave in terms of difficulty. As a result, I chose to leave it for now, and I would change and tweak the difficulty later on, so that the stakeholders and I could gauge better how the difficulty should behave.

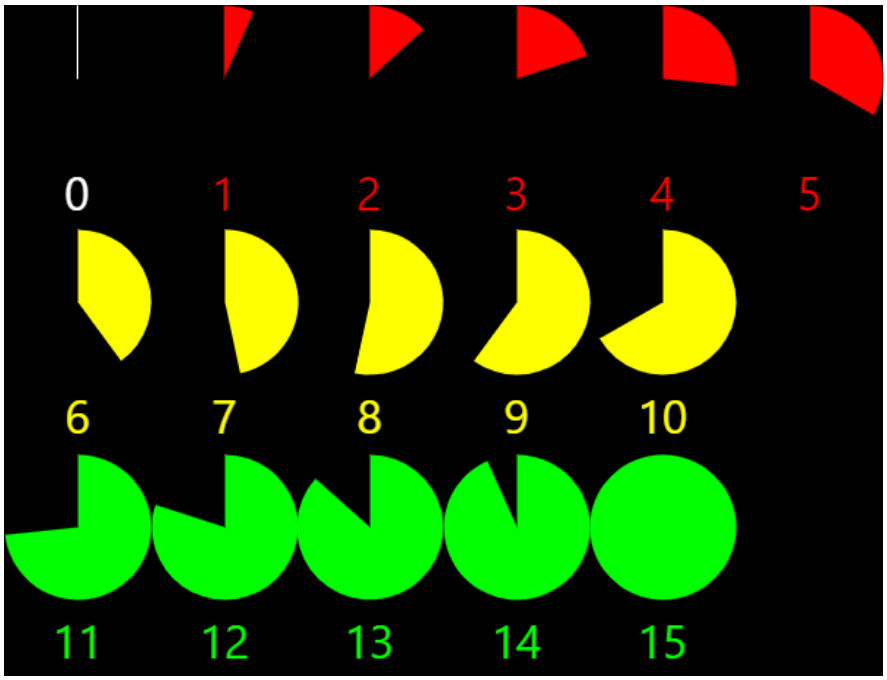


Implementing the timer

Next, I needed to implement the timer.

I imported the 'time' library, so I could count how much time is left.

A problem I ran into was that I realised that it was not possible to create semi-circles in pygame. To solve this, I opted to use sprites.



On the right are the sprites that I created, using draw.io.

I have displayed them on a black background to try to demonstrate their appearance in the final product in the most accurate way.

As the time counts down, the clock shall move towards 0.

As the clock ticks down, the 'tick' sound will be played.

I imported all 16 of these images into the 'assets.py' file.

I then created the timer class, with all the appropriate methods, as shown.

I used the 'time.sleep(1)' method to allow a one second delay for every time the timer ticks. This caused massive problems.

While the timer would tick normally, the entire game would be almost completely frozen. It was at this point that I realised that 'time.sleep(1)' renders the entire system inactive for that one second, meaning the user would not be able to move either. As a result, I cannot use this method, and hence, I deleted the library from my code, since it is effectively useless.

Instead, I chose to use the 'pygame.get_ticks()' method to gauge the current time, and update 'self.duration' by -1 every second. (pygame.get_ticks is measured in milliseconds, meaning every 1000 milliseconds, the clock will tick.) I

```
class Timer:
    def __init__(self, x, y, colour, width, height, duration, screen):
        self.x = x
        self.y = y
        self.colour = colour
        self.width = width
        self.height = height
        self.duration = int(duration)
        self.screen = screen

    def draw(self):
        if self.duration == 15:
            self.screen.blit(timer15_image, (self.x, self.y))
        elif self.duration == 14:
            self.screen.blit(timer14_image, (self.x, self.y))
        elif self.duration == 13:
            self.screen.blit(timer13_image, (self.x, self.y))
        elif self.duration == 12:
            self.screen.blit(timer12_image, (self.x, self.y))
        elif self.duration == 11:
            self.screen.blit(timer11_image, (self.x, self.y))
        elif self.duration == 10:
            self.screen.blit(timer10_image, (self.x, self.y))
        elif self.duration == 9:
            self.screen.blit(timer9_image, (self.x, self.y))
        elif self.duration == 8:
            self.screen.blit(timer8_image, (self.x, self.y))
        elif self.duration == 7:
            self.screen.blit(timer7_image, (self.x, self.y))
        elif self.duration == 6:
            self.screen.blit(timer6_image, (self.x, self.y))
        elif self.duration == 5:
            self.screen.blit(timer5_image, (self.x, self.y))
        elif self.duration == 4:
            self.screen.blit(timer4_image, (self.x, self.y))
        elif self.duration == 3:
            self.screen.blit(timer3_image, (self.x, self.y))
        elif self.duration == 2:
            self.screen.blit(timer2_image, (self.x, self.y))
        elif self.duration == 1:
            self.screen.blit(timer1_image, (self.x, self.y))
        else:
            self.screen.blit(timer0_image, (self.x, self.y))


    def countdown(self):
        time.sleep(1)
        self.duration -= 1

    def expired(self):
        pass
```

also added in the ticking sound effect, from the sounds folder in the main directory.

(I had to change out the ticking sound effect for a different one from 'opengameart.org', because this one was an extended clip rather than a distinct 'tick'. I replaced it with 'singlewind3.wav'.)

I then added an attribute and a corresponding method to cease the countdown when the timer reaches 0. This is the purpose of 'self.timer_expired' – it's set to 'True' when the method is called, and plays the losing sound. It also prevents the countdown from continuing.

```
        self.timer_expired = False
```

```
        else:
            self.screen.blit(timer0_image,
            self.expired()


    def countdown(self):
        if self.timer_expired == False:
            now = pygame.time.get_ticks()
            if now > self.last_tick + 1000:
                self.duration -= 1
                self.last_tick = now
                ticking_sound.play()

    def expired(self):
        self.timer_expired = True
        lose_sound.play()
```

I tested this code, and found that the timer worked perfectly. In fact, because of how I'd programmed the timer's 'costumes', I could instantiate a timer at any value (between 15 and 0), and the timer would simply pick up from there and run as normal. This will be particularly useful for the varying difficulties and their durations.

Below shows how the screen looks as the timer ticks during the game. In this case, the timer is at 7 seconds remaining.

(I do need to make adjustments to the location of the maze and the timer relative to each other and the window. This will be done later; right now I'm more focused on ensuring everything functions correctly.)

Finally, there was one more issue. When the timer expired, the 'losing sound' would be playing on loop, over and over again. To remedy this, I used a Boolean variable, which sets to 'True' once the sound is played. I called this attribute "self.sound_played"; by default, this is 'False'.

Next, I had to actually kill the character when the timer expires. I already had the attribute for the character class "self.alive", and a 'die' method which sets "self.alive" to 'False'.

Finally, I added methods to reset the timer. I implemented it such that every time the timer expires, the user dies, the sound plays, and then the player returns to life. This was done to ensure that this method works.

On the right, the code used in the game loop to reset the player whenever the timer expires is shown, as well as the corresponding method that exists in the timer class.

```
character.draw()
character.move(walls)
timer.draw()
timer.countdown()
if timer.timer_expired == True:
    character.die()
    timer.reset()
    character.reset()
```

As shown in the 'reset' method, I had to add in an attribute to hold the original duration of the timer, so that it would be able to reset to its original duration without any problems. This is the purpose of 'self.original_duration'.

```
def expired(self):
    self.timer_expired = True
    if self.sound_played == False:
        lose_sound.play()
        self.sound_played = True

def reset(self):
    self.timer_expired = False
    self.sound_played = False
    self.duration = self.original_duration
```

In the character class, I had similar methods, which simply killed and reset the character as needed.

```
def die(self):
    self.alive = False

def reset(self):
    self.alive = True
```

The timer class is now complete.

```
class Timer:
    def __init__(self, x, y, colour, width, height, duration, screen):
        self.x = x
        self.y = y
        self.colour = colour
        self.width = width
        self.height = height
        self.duration = int(duration)
        self.screen = screen
        self.last_tick = pygame.time.get_ticks()
        self.timer_expired = False
        self.sound_played = False
        self.original_duration = int(duration)

    def draw(self):
        if self.duration == 15:
            self.screen.blit(timer15_image, (self.x, self.y))
        elif self.duration == 14:
            self.screen.blit(timer14_image, (self.x, self.y))
        elif self.duration == 13:
            self.screen.blit(timer13_image, (self.x, self.y))
        elif self.duration == 12:
            self.screen.blit(timer12_image, (self.x, self.y))
        elif self.duration == 11:
            self.screen.blit(timer11_image, (self.x, self.y))
        elif self.duration == 10:
            self.screen.blit(timer10_image, (self.x, self.y))
        elif self.duration == 9:
            self.screen.blit(timer9_image, (self.x, self.y))
        elif self.duration == 8:
            self.screen.blit(timer8_image, (self.x, self.y))
        elif self.duration == 7:
            self.screen.blit(timer7_image, (self.x, self.y))
        elif self.duration == 6:
            self.screen.blit(timer6_image, (self.x, self.y))
        elif self.duration == 5:
            self.screen.blit(timer5_image, (self.x, self.y))
        elif self.duration == 4:
            self.screen.blit(timer4_image, (self.x, self.y))
        elif self.duration == 3:
            self.screen.blit(timer3_image, (self.x, self.y))
        elif self.duration == 2:
            self.screen.blit(timer2_image, (self.x, self.y))
        elif self.duration == 1:
            self.screen.blit(timer1_image, (self.x, self.y))
        else:
            self.screen.blit(timer0_image, (self.x, self.y))
            self.expired()
```

```
def countdown(self):
    if self.timer_expired == False:
        now = pygame.time.get_ticks()
        if now > self.last_tick + 1000:
            self.duration -= 1
            self.last_tick = now
            ticking_sound.play()

def expired(self):
    self.timer_expired = True
    if self.sound_played == False:
        lose_sound.play()
        self.sound_played = True

def reset(self):
    self.timer_expired = False
    self.sound_played = False
    self.duration = self.original_duration
```

Creating the lava

In order to know where lava cells need to be created, I needed a variable in main.py that stores the previous location of the character object. To do this, I'll use a variable that stores the current location of the character, and then check if the character has moved. If it has, the 'previous location' variable is updated to be the same as the 'current location' variable, and then the 'current location' is updated to be the character's ACTUAL current location.

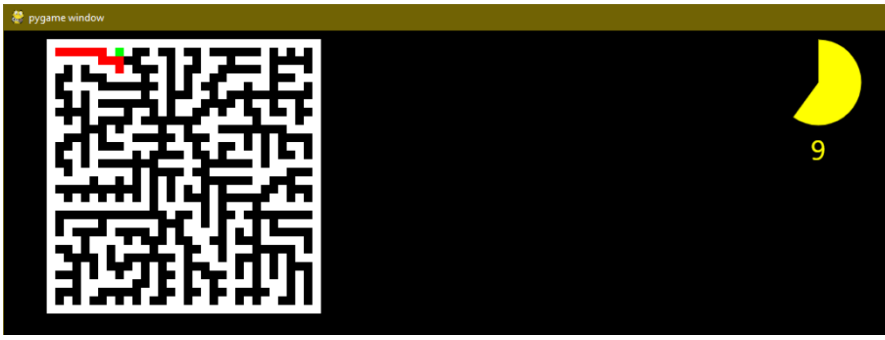
```
character_x = character.x
character_y = character.y
```

I then created the lava class. I chose to leave out the arguments for width, height and colour, instead of passing them in, because they won't need to change and it simplifies the code for instantiation. The

```
class Lava:
    def __init__(self, x, y, screen):
        self.x = x
        self.y = y
        self.width = 10
        self.height = 10
        self.colour = red
        self.touching = False
        self.screen = screen

    def draw(self):
        pygame.draw.rect(self.screen, self.colour, pygame.Rect((self.x, self.y, self.width, self.height)))

    def check_touching(self):
        pass

    def touching_player(self):
        pass
```

```
if (character.x != character_x) or (character.y != character_y):
    lava_cell = Lava(character_x, character_y, screen)
    lava_cells.append(lava_cell)

if (len(lava_cells)!= 0):
    for lava in lava_cells:
        lava.draw()
```

code for touching the player has not been included yet.



I made use of an array to store all the lava instances. This array was called 'lava_cells'. Every lava cell was instantiated at the previous x and y location, for every time the character moved (i.e. their location changed).

The code on the right shows how this worked. A new lava cell was instantiated every time the player moves, and this cell was added to the list of lava cells. Then, all cells in the list were drawn to the screen.

Above, you can see the lava following the player.

I did have an issue with the player disappearing into the lava, because the lava was 'drawn' AFTER the player object. To remedy this, I added in another 'character.draw()' method into the game loop. This solved the problem immediately.

I now had to add the methods into the lava class to check if the player was touching any cells in the 'lava_cells' list.

```
def check_touching(self, character):
    if (character.x == self.x) and (character.y == self.y):
        self.touching_player(character)

def touching_player(self, character):
    character.die()
```

```
if (len(lava_cells)!= 0):
    for lava in lava_cells:
        lava.draw()
        lava.check_touching(character)
```

```
def check_touching(self, character, timer):
    if (character.x == self.x) and (character.y == self.y):
        self.touching_player(character, timer)

def touching_player(self, character, timer):
    character.die()
    timer.duration = 0
```

I passed the character object into the 'check_touching' and 'touching_player' methods. This was done so that I could directly reference the current location of the player.

This worked. However, I also wanted the timer to expire upon the player's death, and hence play the sound. To do this, I also passed the timer into these two methods, and then set 'timer.duration = 0' from there.

I tested this, and found that it worked as expected. Whenever the player ran into the lava, they would die, the timer would immediately expire, the 'death sound' would be played, and the game would be rendered sterile, all simultaneously, all immediately.

The lava class is now complete.

```
class Lava:
    def __init__(self, x, y, screen):
        self.x = x
        self.y = y
        self.width = 10
        self.height = 10
        self.colour = red
        self.touching = False
        self.screen = screen

    def draw(self):
        pygame.draw.rect(self.screen, self.colour, pygame.Rect((self.x, self.y, self.width, self.height)))

    def check_touching(self, character, timer):
        if (character.x == self.x) and (character.y == self.y):
            self.touching_player(character, timer)

    def touching_player(self, character, timer):
        character.die()
        timer.duration = 0
```

```
    def move(self, walls, timer):
        if self.alive == True:
            now = pygame.time.get_ticks()

            if now - self.last_move > self.move_delay:
                self.last_move = now

                pressed_keys = pygame.key.get_pressed()
                if self.alive == True:
                    if pressed_keys[self.controls[0]]:
                        self.y -= self.speed
                        timer.reset()
                        if self.wall_collision(walls):
                            self.y += self.speed

                    if pressed_keys[self.controls[1]]:
                        self.y += self.speed
                        timer.reset()
                        if self.wall_collision(walls):
                            self.y -= self.speed

                    if pressed_keys[self.controls[2]]:
                        self.x += self.speed
                        timer.reset()
                        if self.wall_collision(walls):
                            self.x -= self.speed
```

Completing the timer

Next, I needed to program the timer to reset every time the player moved. I did this by passing the timer instance into the character class, and resetting the timer using it's 'reset' every time the player moved. I made a small change to the 'move' method of the character class, passing in the timer, so that every time the player moved, the timer would reset.

Creating the goal

A goal zone is required so that the player can actually win the game. However, the goal can change for different generations.

As a result, I had to find a way to check where is the closest available space near the bottom right corner. To do this, I checked the 'maze_data' 2D array, looking for the nearest available space which WASN'T a wall. I used the series of 'if statements shown on the right to do this.

```
if maze_data[x_cells][y_cells] == '|':
    if maze_data[x_cells - 1][y_cells] == '|':
        if maze_data[x_cells][y_cells - 1] == '|':
            if maze_data[x_cells - 1][y_cells - 1] == '|':
                pass
            else:
                goal_xpos = 340
                goal_ypos = 300
        else:
            goal_xpos = 350
            goal_ypos = 300
    else:
        goal_xpos = 340
        goal_ypos = 310
else:
    goal_xpos = 350
    goal_ypos = 310
```

I then created the goal class. I realised that I could make use of inheritance and polymorphism here, since the goal was literally a single lava cell, only a different colour, and instead of killing the player when touched, it would mean that the player has won the game. Because of this, I made the 'goal' class a subclass of the 'lava' class, and allowed it to inherit all methods and attributes.

I did need to override the method for checking if the player was being touched, and the method that was called upon touching the player. This is because I did not want to kill the player, nor did I need to 'expire' the timer.

```
class Goal(Lava):
    def __init__(self, x, y, screen):
        Lava.__init__(self, x, y, screen)
        self.colour = yellow

    def check_touching(self, character):
        if (character.x == self.x) and (character.y == self.y):
            self.touching_player(character)

    def touching_player(self, character):
        print("YOU WIN")
```

To make an improvement of my lava class (and hence, the 'goal' subclass), I decided to revert to passing 'colour' in as an attribute. By doing this, it disabled the need for me to hard code 'self.colour = yellow' in the lava class, and made better use of polymorphism.

For now, all that happens when the player does reach the goal is that the text "YOU WIN" is printed to the shell. This will change once I introduce a proper "You win!" screen.

Upon completing the goal object, I needed to check that it functions correctly and allows the player to win the game.

I initially tested for a 30 by 30 maze, and found this worked perfectly. A goal would always be provided in a location that was available (not a wall) and in the vicinity of the bottom right corner. Furthermore, upon contact, “YOU WIN” would be printed to the shell as required.

However, when I tested for mazes of different resolutions, I found that the goal would be places in locations that were nowhere near the bottom right corner, and (for lower resolutions) the goal would be completely outside the maze.

I found that this problem was caused by the hard-coded coordinates. I had hard coded coordinates for a 30 by 30 resolution maze, meaning that the goal would always stay at those coordinates, regardless of the size.

I needed to come up with some form of mathematical expression to translate the coordinates in the “maze_data” array to actual pygame coordinates, depending on the size of the maze.

Before doing this, I decided to talk to my stakeholders about the positioning of the maze on screen.

Every single stakeholder agreed that the maze was best positioned in the centre of the screen. Since the screen resolution was to be set to 1000 by 700 in the final product (although in the above testing screenshots, it has been set to a larger window to make it easier to work with), I figured that the centre of the screen has coordinates (500, 350). Generalising this for any screen resolution, the centre of the screen is (screen_x / 2, screen_y / 2), where screen_x and screen_y are the width and height of the screen respectively.

For a maze that is 30 by 30 cells, the top left hand corner (this is where pygame will place the maze from) can be found. If we assumed the cell width and height to be 10 pixels, we can find that the top right hand corner of the maze has an x-coordinate of 500 – (10 × 15) and a y-coordinate of 350 – (10 × 15). We can generalise this concept and derive mathematical formulae from this:

- $x\ position\ of\ the\ maze = \frac{screen\ width}{2} - \left(cell\ width\ \times \frac{number\ of\ cells\ in\ the\ x\ direction}{2}\right)$
- $y\ position\ of\ the\ maze = \frac{screen\ height}{2} - \left(cell\ height\ \times \frac{number\ of\ cells\ in\ the\ y\ direction}{2}\right)$

And hence, I implemented this to allow the maze to always be the centre of the screen.

I also made the resolution values for the screen and the size of the cells into variables, so that they can be used for arithmetic and can be passed into classes, for a more seamless game.

```
walls = []
cell_size = 10
maze_x = (screen_x/2)-(cell_size*(x_cells/2))
maze_y = ((screen_y/2)-(cell_size*(y_cells/2)))


for x, tiles in enumerate(maze_data):
    for y, tile in enumerate(tiles):
        if tile == "|":
            wall = Wall(maze_x + (y*10), maze_y + (x*10), cell_size, white, screen)
            walls.append(wall)
```

Finally, I returned to addressing the original issue, with the goal spawning in the wrong place. I looked to once again derive some kind of formula or mathematical expression that would always place the goal in a valid location, while still making use of the series of ‘if’ statements to check where the walls lie in the list.

I used the previous idea of finding the top left corner, applying this instead to the bottom right corner. Using my formula, I could find that using the same equation, except adding the bracketed part would lead to the bottom-rightmost corner that is NOT a wall by default (not part of the border of the maze). Using this, I then modified my if statements to instead assign the goal locations in that particular corner.

```
if maze_data[x_cells][y_cells] == '|':
    if maze_data[x_cells - 1][y_cells] == '|':
        if maze_data[x_cells][y_cells - 1] == '|':
            if maze_data[x_cells - 1][y_cells - 1] == '|':
                pass
            else:
                goal_xpos = (((screen_x/2)+(cell_size*(x_cells/2))) - cell_size)
                goal_ypos = (((screen_y/2)+(cell_size*(y_cells/2))) - cell_size)
        else:
            goal_xpos = ((screen_x/2)+(cell_size*(x_cells/2)))
            goal_ypos = (((screen_y/2)+(cell_size*(y_cells/2))) - cell_size)
    else:
        goal_xpos = (((screen_x/2)+(cell_size*(x_cells/2))) - cell_size)
        goal_ypos = ((screen_y/2)+(cell_size*(y_cells/2)))
else:
    goal_xpos = ((screen_x/2)+(cell_size*(x_cells/2))) # Originally 350
    goal_ypos = ((screen_y/2)+(cell_size*(y_cells/2))) # Originally 310
```

This code replaces the old version. Instead of giving hard-coded locations for each situation, it translates the location of the cell in the list to actual pygame coordinates, and interprets the appropriate location to place the player, if the previously optimal location (the location closest to the exact bottom right corner) is occupied by a wall.

By doing this, I have allowed the goal to always spawn in the correct location, regardless of screen size or maze resolution/variable difficulty.

```
if maze_data[1][1] == '|':
    if maze_data[1][2] == '|':
        if maze_data[2][1] == '|':
            if maze_data[2][2] == '|':
                pass
            else:
                default_xpos = (((screen_x/2)-(cell_size*(x_cells/2))) + cell_size) + cell_size
                default_ypos = (((screen_y/2)-(cell_size*(y_cells/2))) + cell_size) + cell_size
        else:
            default_xpos = (((screen_x/2)-(cell_size*(x_cells/2))) + cell_size)
            default_ypos = (((screen_y/2)-(cell_size*(y_cells/2))) + cell_size) + cell_size

    else:
        default_xpos = (((screen_x/2)-(cell_size*(x_cells/2))) + cell_size) + cell_size
        default_ypos = (((screen_y/2)-(cell_size*(y_cells/2))) + cell_size)
else:
    default_xpos = (((screen_x/2)-(cell_size*(x_cells/2))) + cell_size)
    default_ypos = (((screen_y/2)-(cell_size*(y_cells/2))) + cell_size)
```

I did similarly to the character, to ensure that they are always in the correct position, regardless of maze size or screen resolution, and to avoid walls when blocking the closest available cell to the top left corner. This operates under the exact same concept as the goal did, with the 'if' statements checking for places without walls, and deciding where to place the player upon spawn.

I also instantiated a title textbox to the top of the screen, to tell the player the current difficulty they are playing on.

<u>Correcting and organising my code</u>

By this point, my code was very disorganised (in main.py), particularly the game loop.

In order to ensure that I don't run into any problems in the future, I made sure to reorganise my code, starting with the game loop.

- For the game loop, I organised all methods being called into three categories: input, update and render.
- At this point in development, I don't have any input methods being called in the main game loop.
- All move methods, the timer countdown, checking for character death/timer expiry and updating the location of the character are all fall into the 'update' category.
- Drawing items to the screen and updating pygame fall into the 'render' category.

```
### GAMELOOP

running = True # Creates a variable called 'running'.
while running: # While running is True...
    clock.tick(60) # Refresh the screen 60 times a second.
    for event in pygame.event.get():
        if event.type == QUIT: # If the user presses the [X] button...
            running = False # Set running to False (and hence break this loop).

    ### UPDATE
    character_x = character.x
    character_y = character.y

    character.move(walls, timer)
    timer.countdown()
    if (character.x != character_x) or (character.y != character_y):
        lava_cell = Lava(character_x, character_y, cell_size, red, screen)
        lava_cells.append(lava_cell)

    for lava in lava_cells:
        lava.check_touching(character, timer)
    goal.check_touching(character, timer)
    if timer.timer_expired == True:
        character.die()


    ### RENDER
    screen.fill(black)
    for wall in walls:
        wall.draw()
    character.draw()
    goal.draw()
    timer.draw()
    title_textbox.draw(screen)
    for lava in lava_cells:
        lava.draw()

    pygame.display.update() # Update the screen.
```

Once I organised the game loop, I went about organising the rest of the program.

```
### Instantiates character, timer, textbox, goal
character = Character(default_xpos, default_ypos, cell_size, cell_size, green, screen, [K_w, K_s, K_d, K_a], cell_size)
timer = Timer(screen_x - 130, 30, red, 10, 10, 3, screen)
title_textbox = Textbox((screen_x/2)-100, (screen_y/25), white, str(x_cells) + " by " + str(y_cells), "test", 200, 50, font_1)
goal = Goal(goal_xpos, goal_ypos, cell_size, yellow, screen)
lava_cells = []
```

I kept all the major single-object instantiations (not including lava, which is instantiated as the character moves) in one place, so that I could easily change their arguments in the future.

```
### Finds the appropriate location for the character and goal
if maze_data[1][1] == '|':
    if maze_data[1][2] == '|':
        if maze_data[2][1] == '|':
            if maze_data[2][2] == '|':
                pass
            else:
                default_xpos = (((screen_x/2)-(cell_size*((x_cells+2)/2))) + cell_size) + cell_size
                default_ypos = (((screen_y/2)-(cell_size*((y_cells+2)/2))) + cell_size) + cell_size
        else:
            default_xpos = (((screen_x/2)-(cell_size*((x_cells+2)/2))) + cell_size)
            default_ypos = (((screen_y/2)-(cell_size*((y_cells+2)/2))) + cell_size) + cell_size

    else:
        default_xpos = (((screen_x/2)-(cell_size*((x_cells+2)/2))) + cell_size) + cell_size
        default_ypos = (((screen_y/2)-(cell_size*((y_cells+2)/2))) + cell_size)
else:
    default_xpos = (((screen_x/2)-(cell_size*((x_cells+2)/2))) + cell_size)
    default_ypos = (((screen_y/2)-(cell_size*((y_cells+2)/2))) + cell_size)

if maze_data[x_cells][y_cells] == '|':
    if maze_data[x_cells - 1][y_cells] == '|':
        if maze_data[x_cells][y_cells - 1] == '|':
            if maze_data[x_cells - 1][y_cells - 1] == '|':
                pass
            else:
                goal_xpos = (((screen_x/2)+(cell_size*((x_cells-2)/2))) - cell_size)
                goal_ypos = (((screen_y/2)+(cell_size*((y_cells-2)/2))) - cell_size)
        else:
            goal_xpos = ((screen_x/2)+(cell_size*((x_cells-2)/2)))
            goal_ypos = (((screen_y/2)+(cell_size*((y_cells-2)/2))) - cell_size)
    else:
        goal_xpos = (((screen_x/2)+(cell_size*((x_cells-2)/2))) - cell_size)
        goal_ypos = ((screen_y/2)+(cell_size*((y_cells-2)/2)))
else:
    goal_xpos = ((screen_x/2)+(cell_size*((x_cells-2)/2))) # Originally 350
    goal_ypos = ((screen_y/2)+(cell_size*((y_cells-2)/2))) # Originally 310
```

I also kept all the code that sets the location of the character and the goal in the same place, since these are related. Because these work perfectly, I didn't want to change anything about them, which could ruin their functionality.

```
### generates the maze and creates the walls
x_cells = 80
y_cells = 80
maze_data = generate_maze(x_cells, y_cells)
for index, row in enumerate(maze_data):
    row = "|" + row[1:] + "|"
    maze_data[index] = row
maze_data.insert(0, "|" * (y_cells + 2))
maze_data.append("|" * (x_cells +  2))
walls = []
cell_size = 10
maze_x = ((screen_x/2)-(cell_size*((x_cells+2)/2)))
maze_y = ((screen_y/2)-(cell_size*((y_cells+2)/2)))

for x, tiles in enumerate(maze_data):
    for y, tile in enumerate(tiles):
        if tile == "|":
            wall = Wall(maze_x + (y*10), maze_y + (x*10), cell_size, white, screen)
            walls.append(wall)
```

The above code relates to the generation of the maze. I kept these all together because they are all related, and initialise variables for the maze that are massively influential on the rest of the program. For example, as previously mentioned, 'x_cells' and 'y_cells' determine the dimensions of the maze (how many cells tall and how many cells high respectively, not including the borders.)

```
''' Imports all needed modules and libraries.'''
import pygame
from pygame.locals import *
from maze_algorithm import *
from ui_elements import *
from characters import *
from assets import *

### Initialises screen, pygame and clock
pygame.init() # Initialises pygame.
screen_x = 1440
screen_y = 900
clock = pygame.time.Clock() # Renames the pygame clock so it can be used.
screen = pygame.display.set_mode((screen_x, screen_y)) # Creates a screen of the required resolution
```

And, write at the top of the file, the initialising variables, and libraries.

Introducing state machines

Eventually, it was time to introduce the state machine. I researched online as to how to implement one, and based this off of my class diagrams from the design section.

I started with a generic class, called “states”. This class is a super class for all of the states mentioned in design. They will all inherit all methods and attributes of this class, which is why everything has been left blank.

```
class States:
    share = {}
    def __init__(self):
        self.running = True
        self.next = None
        self.quit = False

    def setup(self):
        pass

    def cleanup(self):
        pass

    def handle_events(self, event):
        pass

    def update(self):
        pass

    def render(self, screen):
        pass
```

An explanation of the attributes:

- ‘self.running’ is used to indicate that the state is currently running. When this is set to False, the game transitions into the next state.
- ‘self.next’ identifies the next state that the game should move to, once self.running is set to False.
- When ‘self.quit’ is set to True, the game is exited.

An explanation of the methods:

- ‘setup(self)’ is used to instantiate and initialise any important variables or methods for that particular state.
- ‘cleanup(self)’ is used to delete any objects or instantiations and literally ‘clean up’ to get ready for the next state.
- ‘handle_events(self, event)’ is used as the game loop. As a result, it has ‘event’ passed to it, so that it can use and reference the pygame ‘event’ functions. It also checks for certain events, such as a button being pressed. It acts similarly to the ‘input/update’ section of the original game loop.
- ‘update(self)’ updates items in the actual game, similar to the ‘update’ section of the game loop.
- ‘render(self, screen)’ is used to draw/render items to the screen.

‘share’ is a dictionary that carries any information that needs to be shared between states, for example, the controls settings or the difficulty selected.

```
class Control:
    def __init__(self):
        self.screen = pygame.display.set_mode((screen_x, screen_y))
        self.clock = pygame.time.Clock()
        self.running = True
        self.state_dict = None
        self.current_state = None

    def setup(self, state_dict, start_state):
        self.state_dict = state_dict
        self.current_state = self.state_dict[start_state]
        self.current_state.setup()

    def handle_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
            self.current_state.handle_events(event)

        if not(self.current_state.running):
            self.change_state()

        if self.current_state.quit == True:
            self.running = False

    def change_state(self):
        new_state = self.current_state.next
        self.current_state.cleanup()
        self.current_state = self.state_dict[new_state]
        self.current_state.setup()

    def main_loop(self):
        while self.running:
            self.clock.tick(fps)
            self.handle_events()
            self.current_state.update()
            self.current_state.render(self.screen)
            pygame.display.flip()
```

Next, I created a 'control' class. This class controls the entire game. This class is responsible for coordinating states, holding the real game loop, and transitioning between states when 'self.running = False' in any state.

- The 'self.screen' attribute creates the screen in pygame, according to the resolution. ('screen_x' and 'screen_y' is set in assets.py)
- 'self.clock' creates an object to track time, via pygame.
- 'self.running' is set to True, so that the game loop is active.
- 'self.state_dict' is a dictionary full of the states. This is needed so that I can point to the next state from a current state, and have all the states (and their instantiations) in one place.
- 'self.current_state' holds the name of the current state.
- The setup method takes in the state dictionary, so that the control class can make use of them, as well as the name of the start state, so that the game knows where to start from. This will be the main menu, because the game starts at this point.
- The current state is then set to the start state, the 'setup' method is called for this state, meaning all objects in that state are instantiated, and ready for use.
- The 'handle_events' method handles events, and quits the program if the [X] button is pressed on the window. It also checks if 'self.running' has been set to false for any state, and if it has, it transitions to the next state, calling the 'change_state' method.
- The 'change_state' method changes the state, cleaning up the current state (which deletes all the objects from the old state) and sets up the new state (which instantiates all the objects for the new state).
- The main loop state acts as the game loop. Just like the game loop, it refreshes the screen 60 times a second (since FPS = 60 in assets.py). It also calls 'handle_events', to check if the [X] button has been pressed, and hence, if the game should be closed. 'pygame.display.flip()' is a method used to refresh the screen.

```
pygame.init()
app = Control()
state_dict = {}
app.setup(state_dict, "main_menu")
app.main_loop()

pygame.quit()
```

In conjunction with the control class, at the bottom of the program, is the pygame initialisation method being called.

'app' is an instance of 'control', the state dictionary is there (it will hold the names of the states) and the setup method is called, passing in the dictionary of states and the first state (in this screenshot, this is 'main_menu'. The main loop is the called.

Implementing the game state

The first of the 'States' classes' children I chose to create was the game class. This was an obvious choice – I had already programmed the main game, and all I needed to do was move the code into

the class, making the variables attributes of the game class, by adding 'self.' to the start of all their identifiers.

- I moved the code for initialising x_cells and y_cells, the maze generation algorithm, building the maze via the algorithm, setting the spawning locations of goal/character, and the instantiation of all objects into the setup method. This was done so that the setup method would act as intended – when called, it would get everything needed for the game state ready.

```
def setup(self):
    self.running = True

    ### generates the maze and creates the walls
    self.x_cells = 50
    self.y_cells = 50

    self.maze_data = generate_maze(self.x_cells, self.y_cells)
    for index, row in enumerate(self.maze_data):
        row = "|" + row[1:] + "|"
        self.maze_data[index] = row
    self.maze_data.insert(0, "|" * (self.y_cells + 2))
    self.maze_data.append("|" * (self.x_cells +  2))
    self.walls = []
    self.cell_size = 10
    self.maze_x = ((screen_x/2)-(self.cell_size*((self.x_cells+2)/2)))
    self.maze_y = ((screen_y/2)-(self.cell_size*((self.y_cells+2)/2)))

    for x, tiles in enumerate(self.maze_data):
        for y, tile in enumerate(tiles):
            if tile == "|":
                wall = Wall(self.maze_x + (y*10), self.maze_y + (x*10), self.cell_size, white)
                self.walls.append(wall)

    ### Finds the appropriate location for the character and goal
    if self.maze_data[1][1] == '|':
        if self.maze_data[1][2] == '|':
            if self.maze_data[2][1] == '|':
                if self.maze_data[2][2] == '|':
                    pass
                else:
                    self.default_xpos = (((screen_x/2)-(self.cell_size*((self.x_cells+2)/2))) + self.cell_size) + self.cell_size
                    self.default_ypos = (((screen_y/2)-(self.cell_size*((self.y_cells+2)/2))) + self.cell_size) + self.cell_size
            else:
                self.default_xpos = (((screen_x/2)-(self.cell_size*((self.x_cells+2)/2))) + self.cell_size)
                self.default_ypos = (((screen_y/2)-(self.cell_size*((self.y_cells+2)/2))) + self.cell_size) + self.cell_size

        else:
            self.default_xpos = (((screen_x/2)-(self.cell_size*((self.x_cells+2)/2))) + self.cell_size) + self.cell_size
            self.default_ypos = (((screen_y/2)-(self.cell_size*((self.y_cells+2)/2))) + self.cell_size)
    else:
        self.default_xpos = (((screen_x/2)-(self.cell_size*((self.x_cells+2)/2))) + self.cell_size)
        self.default_ypos = (((screen_y/2)-(self.cell_size*((self.y_cells+2)/2))) + self.cell_size)

    if self.maze_data[self.x_cells][self.y_cells] == '|':
        if self.maze_data[self.x_cells - 1][self.y_cells] == '|':
            if self.maze_data[self.x_cells][self.y_cells - 1] == '|':
                if self.maze_data[self.x_cells - 1][self.y_cells - 1] == '|':
                    pass
                else:
                    self.goal_xpos = (((screen_x/2)+(self.cell_size*((self.x_cells-2)/2))) - self.cell_size)
                    self.goal_ypos = (((screen_y/2)+(self.cell_size*((self.y_cells-2)/2))) - self.cell_size)
            else:
                self.goal_xpos = ((screen_x/2)+(self.cell_size*((self.x_cells-2)/2)))
                self.goal_ypos = (((screen_y/2)+(self.cell_size*((self.y_cells-2)/2))) - self.cell_size)
        else:
            self.goal_xpos = (((screen_x/2)+(self.cell_size*((self.x_cells-2)/2))) - self.cell_size)
            self.goal_ypos = ((screen_y/2)+(self.cell_size*((self.y_cells-2)/2)))
    else:
        self.goal_xpos = ((screen_x/2)+(self.cell_size*((self.x_cells-2)/2))) # Originally 350
        self.goal_ypos = ((screen_y/2)+(self.cell_size*((self.y_cells-2)/2))) # Originally 310


    ### Instantiates character, timer, textbox, goal
    self.character = Character(self.default_xpos, self.default_ypos, self.cell_size, self.cell_size, green, self.share["keys"], self.cell_size)
    self.timer = Timer(screen_x - 130, 30, red, 10, 10, 15)
    self.title_textbox = Textbox((screen_x/2)-100, (screen_y/25), white, str(self.x_cells) + " by " + str(self.y_cells), "test", 200, 50, font_1)
    self.goal = Goal(self.goal_xpos, self.goal_ypos, self.cell_size, yellow)
    self.lava_cells = []
```

- I left the cleanup method empty for now. I did this because I wanted to experiment later, as to how objects would behave if I DIDN'T include the cleanup method.
- The update method held all the methods for updating and checking for objects, and checking for victory/death.

```
    def update(self):
        self.character_x = self.character.x
        self.character_y = self.character.y

        self.character.move(self.walls, self.timer)
        self.timer.countdown()
        if (self.character.x != self.character_x) or (self.character.y != self.character_y):
            lava_cell = Lava(self.character_x, self.character_y, self.cell_size, red)
            self.lava_cells.append(lava_cell)

        for lava in self.lava_cells:
            lava.check_touching(self.character, self.timer)

        self.goal.check_touching(self.character, self.timer)
        if self.timer.timer_expired == True:
            self.character.die()
            self.next = "game over"
            self.running = False

        if self.goal.win:
            self.next = "game won"
            self.running = False
```

- Lastly, the render method. This method simply draws the character object, goal, walls and any other onscreen objects.

```
    def render(self, screen):
        screen.fill(black)
        for wall in self.walls:
            wall.draw(screen)
        self.character.draw(screen)
        self.goal.draw(screen)
        self.timer.draw(screen)
        self.title_textbox.draw(screen)
        for lava in self.lava_cells:
            lava.draw(screen)
```

I had to make several changes in order to get this to work, a lot of the time, it was me simply forgetting to place 'self.' before some an attribute.

Finally, I defined, in the share dictionary, "keys" to be [K_w, K_s, K_d, K_a]. This is to ensure that the controls can be passed through states, so that the user can customise their controls. 'Keys' is then passed into the character class, instead of using the array on its own.

I ran the program, setting the initial state to be 'game' – the game state.

The program functioned perfectly, just as it did prior to the introduction of states.

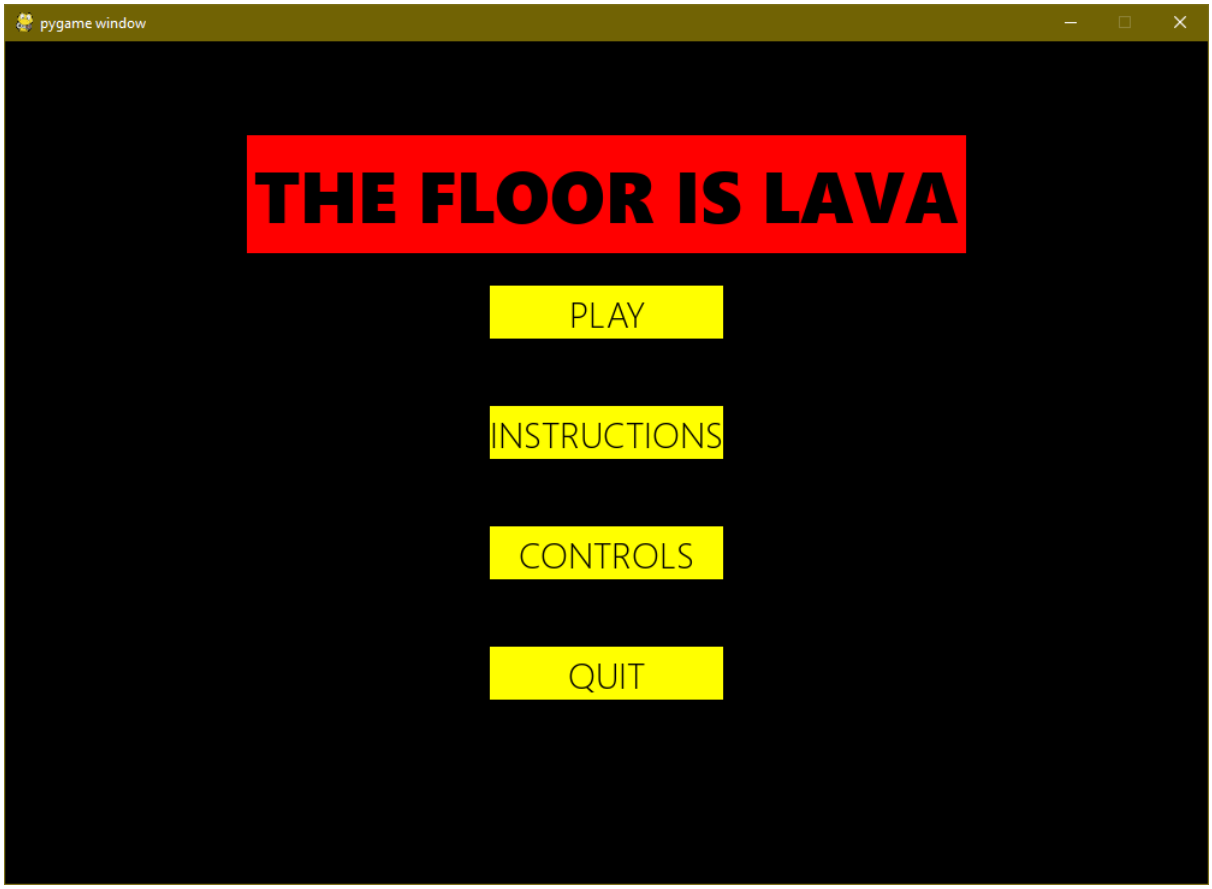
I then modified the program to point to the next state upon death/victory. With this modification, winning/losing would cause an error, because I had not yet implemented the next state. This will be done later.

<u>Creating the main menu</u>

I then created the main menu state.

```
class Main_Menu(States):
    def __init(self):
        States.__init__(self)

    def setup(self):
        self.running = True

        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "THE FLOOR IS LAVA", "title_textbox", 500, 100, font_2)
        self.play = Button((screen_x/2)-100, screen_y-500, yellow, "PLAY", "play_button", 200, 50, font_1, self.switch_play)
        self.instructions = Button((screen_x/2)-100, screen_y-400, yellow, "INSTRUCTIONS", "instructions_button", 200, 50, font_1, self.switch_instructions)
        self.controls = Button((screen_x/2)-100, screen_y-300, yellow, "CONTROLS", "controls_button", 200, 50, font_1, self.switch_controls)
        self.quit = Button((screen_x/2)-100, screen_y-200, yellow, "QUIT", "quit_button", 200, 50, font_1, self.quit_game)

    def cleanup(self):
        del self.title, self.play, self.instructions, self.quit

    def switch_play(self):
        self.next = "options_screen"
        self.running = False

    def switch_instructions(self):
        self.next = "instructions_screen"
        self.running = False

    def switch_controls(self):
        self.next = "controls_screen"
        self.running = False

    def quit_game(self):
        self.quit = True

    def handle_events(self, event):
        self.play.check_clicked(event)
        self.instructions.check_clicked(event)
        self.controls.check_clicked(event)
        self.quit.check_clicked(event)

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.title.draw(screen)
        self.play.draw(screen)
        self.instructions.draw(screen)
        self.controls.draw(screen)
        self.quit.draw(screen)
```

- In setup, I instantiated the textboxes and buttons, the only objects that appear in this class.
- In cleanup, these objects are deleted, to allow the next state to come in.
- ‘switch_play’, ‘switch_instructions’, ‘switch_controls’ and ‘quit_game’ are to enter the difficulty menu, the instructions menu, the controls menu, and quit the game, respectively. They all identify the next state (using the same name referenced in the state dictionary), and sets running to False. The control class then automatically cleans up and moves to the next state.
- ‘handle_events’ simply checks for buttons being pressed.
- There is nothing in the ‘update’ method. This is a static screen.
- ‘render’ simply draws everything to the screen.

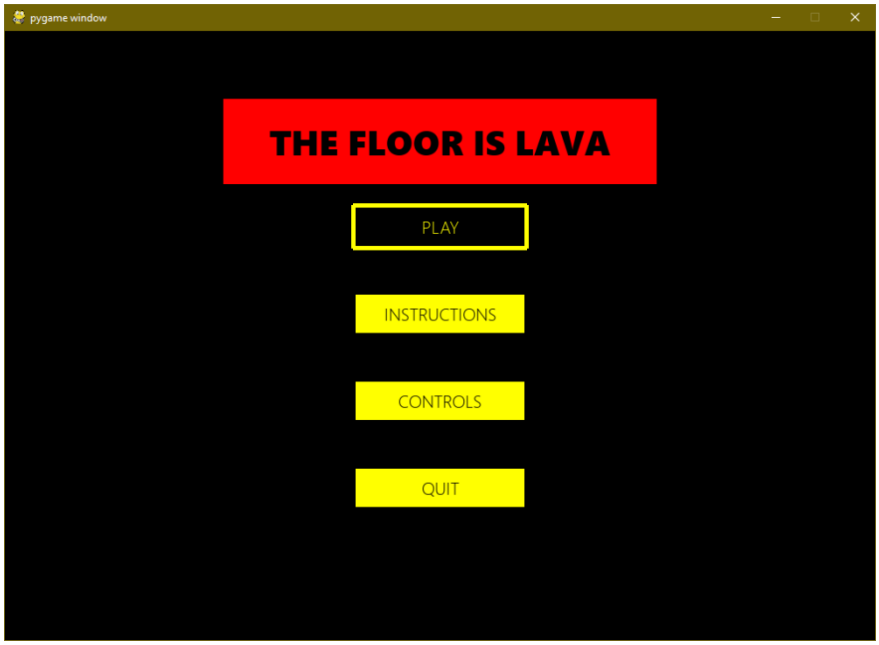


This was the title screen produced. I felt that the font size was too big, and asked my stakeholders for their sentiment on this. They agreed, so the font sizes were halved.

They preferred the font redesign much more, citing that it was more appropriate and less 'in your face'.

However, when I tried to click buttons, I got an error. I realised that I had to redesign my 'is_clicked' method in the button class, because it takes 'screen' as an argument – I cannot pass this in via the 'handle_events' method, only the 'render' method, and as a result, this would cause an argument error. The 'colour inversion' effect of the button (when it's clicked) is what was causing the problem.

I simply eliminated the 'is_clicked' method from the button class. I realised that all it does is draw the inversion effect; which can easily be done in the draw method instead. What's more, it's never called by the main program, only the class itself calls this method. This is great, because it makes my code more efficient, and also means I don't have to worry about being unable to pass 'screen' to the method.



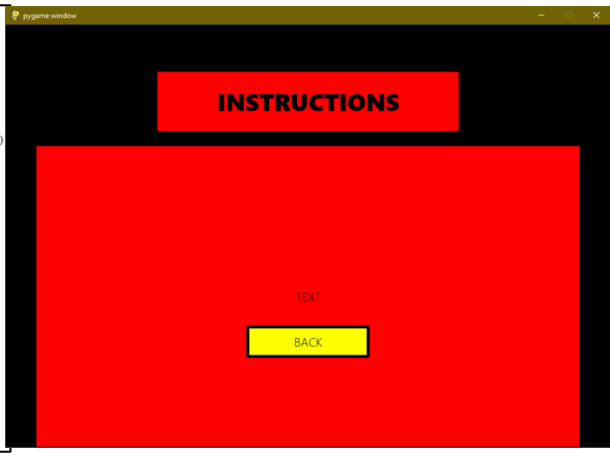
Instead, I used an 'if' statement to check if the button was clicked in the draw class, and draw the button appropriately. This worked, meaning the buttons are all now fully operational.

I then moved onto the instructions screen, because it was the easiest to implement (it's a simple textbox with a 'back' button).

Implementing the instructions state

- I started with the usual template, instantiating two textboxes and one button in the setup method. The two textboxes were for the title and information, and the one button for returning to the main menu.
- The cleanup method simply deleted all of these objects.
- I needed a new method here, to switch to the main menu (when the 'back' button is clicked). This method was called 'switch_main_menu', and simply set the next state to be 'main_menu' (this is the main menu's reference in the states dictionary). It then plays 'menu_back_sound', using the pygame mixer, and sets 'self.running' to False. This would switch the player back to the previous state.
- As previously established, the button class has a 'self.function' attribute, which takes a function in that the button will perform when clicked. It also has a 'check_clicked' method, which checks if the button has been clicked or not. In this instance, the 'self.function' attribute was passed the 'self.switch_main_menu' method, so that when the button is clicked, this method is called (and hence, switched to the main menu).
- The check_clicked method was called in 'handle_events', and had 'events' passed as an argument.
- The update method was empty; this is not a screen which needs it.
- The render method simply filled the screen with black, and drew the title, instructions and button onto the screen.
- For now, the text on the instructions button is left blank. Below shows the class, and the outcome:

```
class Instructions(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "INSTRUCTIONS", "title_textbox", 500, 100, font_2)
        self.instructions = Textbox((screen_x/2)-450, screen_y - 500, red, "TEXT", "instructions_textbox", 900, 500, font_1)
        self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.title, self.instructions, self.back

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def handle_events(self, event):
        self.back.check_clicked(event)

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.title.draw(screen)
        self.instructions.draw(screen)
        self.back.draw(screen)
```



As evident, the textbox is a bit too big.

I remedied this by adjusting the height of the box, reducing it significantly. I then went about writing the actual instructions that will be placed in the textbox.

First, I consulted my stakeholders on how they think would be the best way of writing the instructions. They agreed that listing out the instructions in bullet points would be the best way, each bullet point covering a major area of the game that the user should know. Because of this, I had to completely revise the previous instructions from the design section, in order to allow them to be in short bullet points. From this, I composed the following…

- Choose your difficulty and navigate your way through the maze, to the yellow goal.
- As you move through the maze, deadly lava will trail in your wake, so be careful – backtracking is not an option.
- You have to think fast and precisely – a timer will be counting down, and can only be reset if you move. If the timer expires, the lava will engulf you, and you will fail.
- Have fun and good luck!

I asked my stakeholders for their opinions on these instructions…

[redacted]: “Clear and concise. These are fine.”

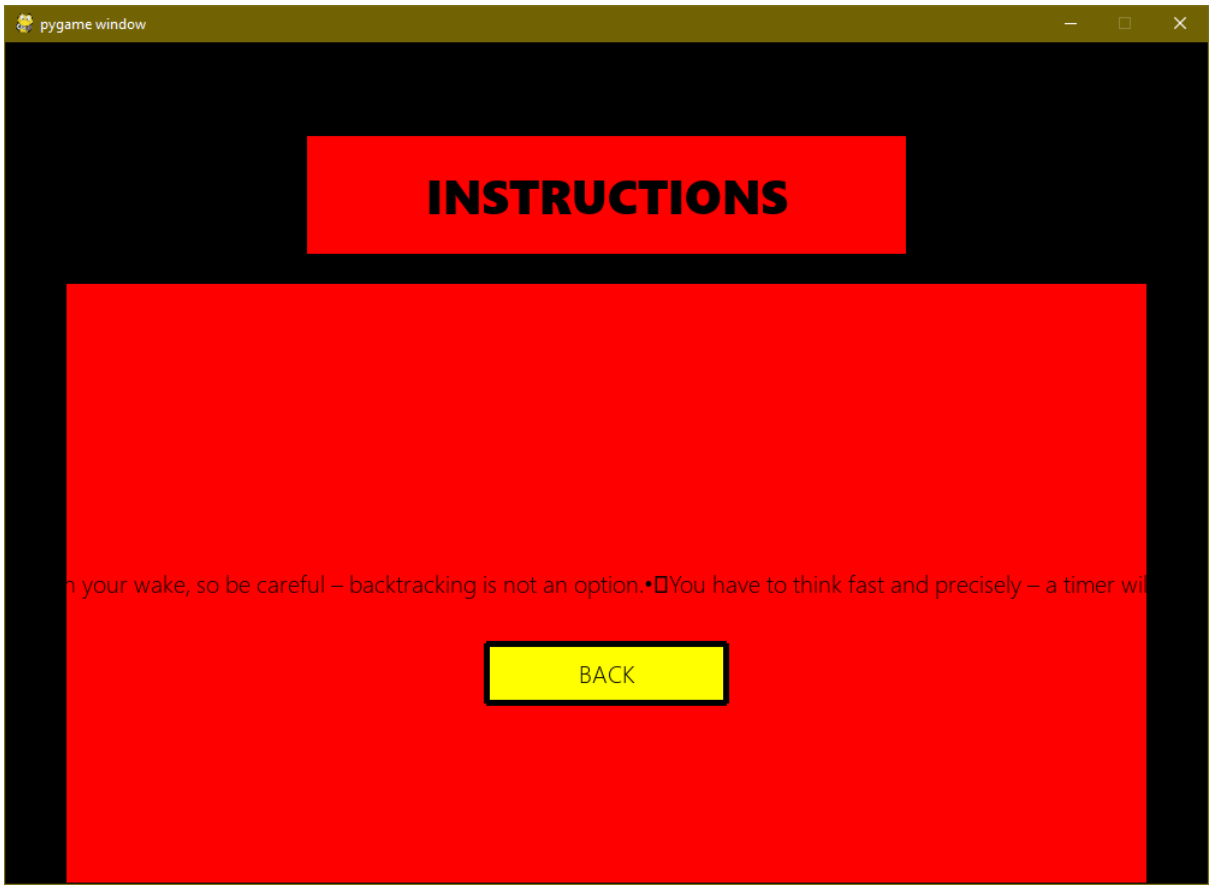
[redacted]: “I like how you’ve bullet-pointed them to make sure you cover everything, good job!”

[redacted]: “This is really detailed while still being easy to read. You’ve got everything down that we need to be able to play the game.”

[redacted]: “The use of bullet points is really good here.”

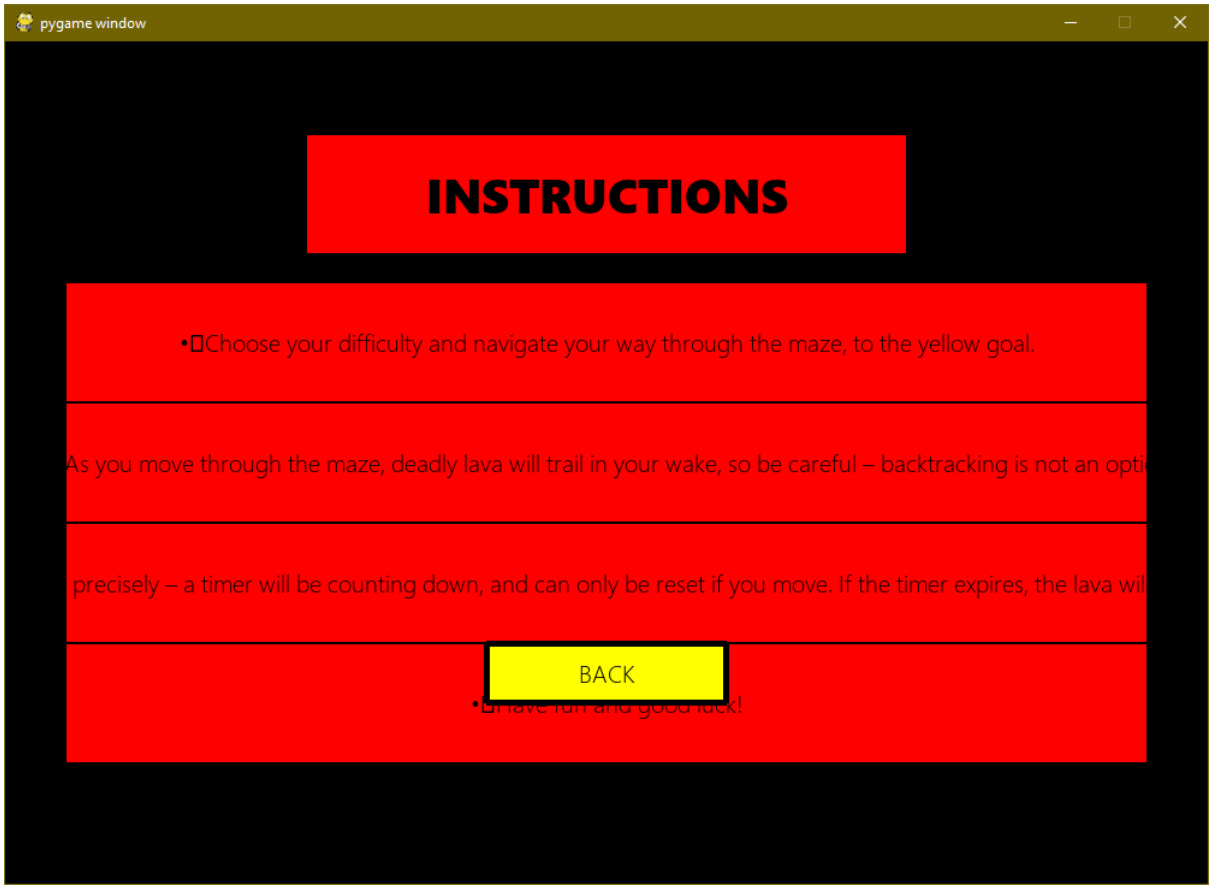
[redacted]: “A lot of information, while only keeping it necessary. This is fine, similar to the previous version.”

I added this information into the textbox.

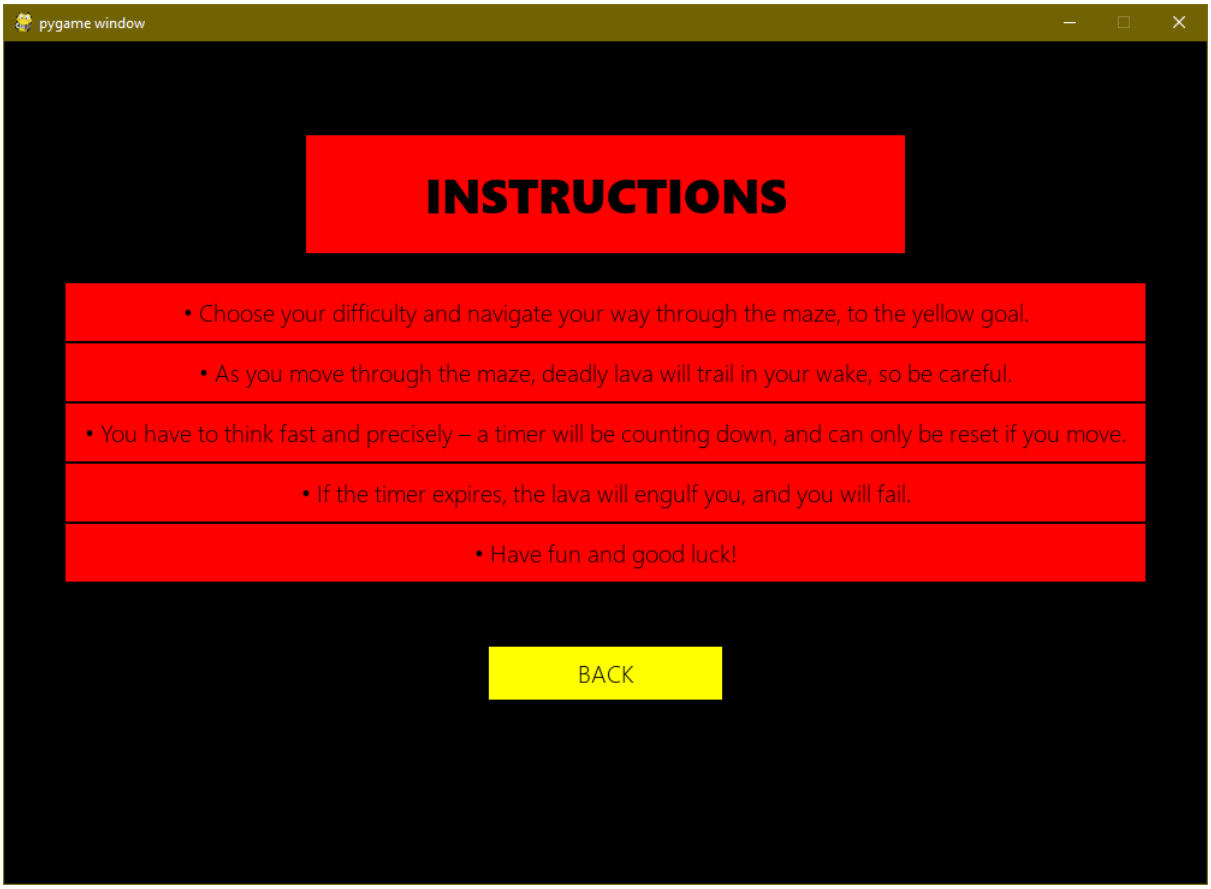


It did not work as I planned. I tried employing a ‘\n’, for a new line, but to no avail. As it turns out, pygame does not support new lines.

Instead, I chose to split the text across several textboxes instead.



This yielded better results, but still was far from perfect. I still needed to split the second bullet point into more lines of text.



This was pretty much perfect. I asked the stakeholders and they all agreed. The back button worked, navigating the user back to the main menu – the instructions state is now complete.

Implementing the controls menu state

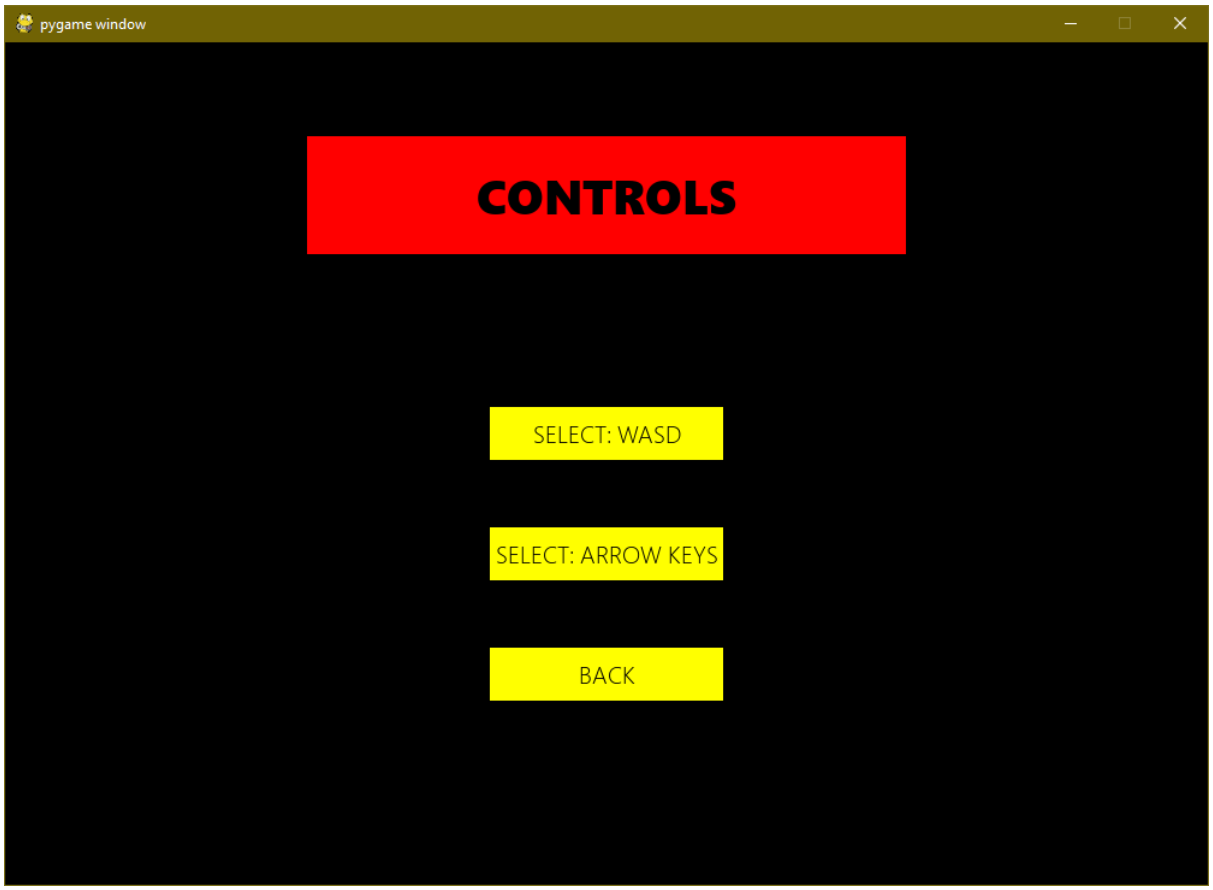
Going back to the main menu, the next button is the “controls” button.

- I instantiated a textbox and three buttons – the textbox was for the title, and the three buttons were for selecting WASD, selecting the arrow keys, and selecting the back button.
- As usual, ‘cleanup’ deleted all of these.
- An exclusive method to this class, “switch_to_wasd” is the method passed to one of the buttons. It simply plays a sound effect and changes ‘keys’ in the shared dictionary to be the array ‘[K_w, K_s, K_d, K_a]’. This is a list of the pygame references for the WASD controls, and this is passed to the character class in the game state, to allow the player to control the character using WASD.
- Similarly, there is another method that corresponds to the other button, called “switch_to_arrows”. This one changes ‘keys’ (in the shared dictionary) to ‘[K_UP, K_DOWN, K_RIGHT, K_LEFT]’, allowing the user to use the arrow keys to control their character.
- ‘handle_events’ simply checks if any buttons have been clicked.
- ‘update’ does nothing in this state (pass).
- ‘render’ fills the screen with black, and draws the four objects to the screen.

```
class Controls_Setup(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "CONTROLS", "title_textbox", 500, 100, font_2)
        self.wasd = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: WASD", "instructions_button", 200, 50, font_1, self.switch_to_wasd)
        self.arrows = Button((screen_x/2)-100, screen_y-300, yellow, "SELECT: ARROW KEYS", "instructions_button", 200, 50, font_1, self.switch_to_arrows)
        self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.title, self.wasd, self.arrows, self.back

    def switch_to_wasd(self):
        self.share.update({"keys": [K_w, K_s, K_d, K_a]})
        menu_forward_sound.play()

    def switch_to_arrows(self):
        self.share.update({"keys": [K_UP, K_DOWN, K_RIGHT, K_LEFT]})
        menu_forward_sound.play()

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def handle_events(self, event):
        self.wasd.check_clicked(event)
        self.arrows.check_clicked(event)
        self.back.check_clicked(event)

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.title.draw(screen)
        self.wasd.draw(screen)
        self.arrows.draw(screen)
        self.back.draw(screen)
```

```
share = {
    "keys": [K_w, K_s, K_d, K_a],
    }
```

(This is the share dictionary. By default, WASD controls are enabled.)



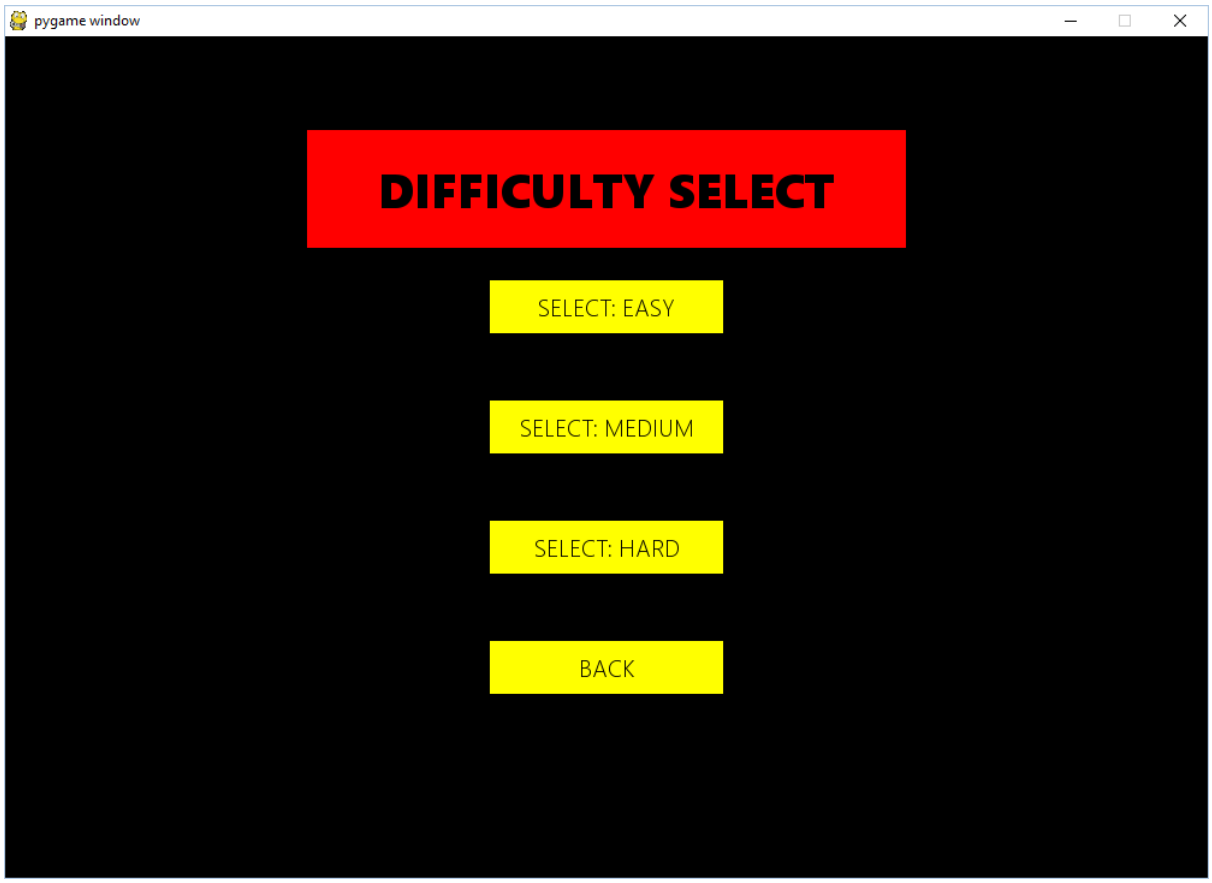
The controls class worked seemingly flawlessly with immediate effect, although (at this moment) it was not possible to tell if the selection of arrow keys/WASD had actually affected the game. This will be checked in the testing section.

Implementing the difficulty menu state

The difficulty menu is displayed after the user clicks 'PLAY' from the main menu. This menu allows the user to choose a difficulty, from which a timer and maze is generated (according to their selection).

- I instantiated a textbox and four buttons. The textbox was the title, and three of the buttons were assigned to selecting 'easy', 'medium' and 'hard' difficulties; the fourth button was for the 'back' button, which returns the user the main menu.
- The cleanup method deletes all of these.
- The 'handle_events' method checks if any of the four buttons have been pressed.
- As with previously, there is the 'switch_main_menu' method, which is passed into the 'back' button and takes the user back to the main menu (when clicked).
- The methods 'switch_easy', 'switch_medium' and 'switch_hard' all change the configuration of the game, using the shared dictionary.
  - 'switch_easy' would change 'x_cells' and 'y_cells' (in the shared dictionary) both to 20, and changes 'timer', another variable in the shared dictionary, to 15. These are then passed to the game, which uses 'x_cells' and 'y_cells' as the maze resolution, and uses 'timer' as the original duration of the instantiated timer.
  - 'switch_medium' sets 'x_cells' and 'y_cells' to 35, and 'timer' to 10.
  - 'switch_hard' sets 'x_cells' and 'y_cells' to 50, and 'timer' to 5.

- All of these methods assign the next state to ‘game’, and set self.running to False, to move the next state into the game state, which runs as specified by the selected button.
- ‘render’ simply draws all the items on the screen, and fills the screen with black.




```
class Difficulty(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "DIFFICULTY SELECT", "title_textbox", 500, 100, font_2)
        self.easy = Button((screen_x/2)-100, screen_y-500, yellow, "SELECT: EASY", "instructions_button", 200, 50, font_1, self.switch_easy)
        self.medium = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: MEDIUM", "instructions_button", 200, 50, font_1, self.switch_medium)
        self.hard = Button((screen_x/2)-100, screen_y-300, yellow, "SELECT: HARD", "instructions_button", 200, 50, font_1, self.switch_hard)
        self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.title, self.easy, self.medium, self.hard, self.back

    def handle_events(self, event):
        self.easy.check_clicked(event)
        self.medium.check_clicked(event)
        self.hard.check_clicked(event)
        self.back.check_clicked(event)

    def switch_easy(self):
        self.share.update({"x_cells": 20})
        self.share.update({"y_cells": 20})
        self.share.update({"timer": 15})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_medium(self):
        self.share.update({"x_cells": 35})
        self.share.update({"y_cells": 35})
        self.share.update({"timer": 10})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False
```

```
    def switch_hard(self):
        self.share.update({"x_cells": 50})
        self.share.update({"y_cells": 50})
        self.share.update({"timer": 5})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.title.draw(screen)
        self.easy.draw(screen)
        self.medium.draw(screen)
        self.hard.draw(screen)
        self.back.draw(screen)
```

The switch methods all played sound effects when called.

Linking and configuring states

Now that the states leading up to (and including) the game are implemented, it is a good idea to setup the state dictionary in a complete form, and get the game fully up and running.

```
pygame.init()
app = Control()
state_dict = {
    "main_menu": Main_Menu(),
    "difficulty_select": Difficulty(),
    "instructions_screen": Instructions(),
    "controls_screen" : Controls_Setup(),
    "game" : Game(),
    }
app.setup(state_dict, "main_menu")
app.main_loop()

pygame.quit()
```

'app.setup' takes the state dictionary as one argument, and the first state as the other. In this case, this is the main menu state.

I tested the program, with this configuration.

- I navigated from the main menu to the difficulty menu, the instructions menu and the controls menu. All of these worked just fine, deleting all objects from the screen when I backed out of every state.
- When in the controls menu, I changed the controls from WASD to the arrow keys. I did this to check if the arrow keys work as controls as well as WASD did previously.
- I entered the game on the hardest difficulty. The timer was capped at 5 seconds, with a 50 by 50 resolution. This worked excellently, without any problems.
- Upon death, predictably, the game crashed. I had not yet created the 'game over' state, and hence the game had nowhere to go after the player lost.

Hence, the next step would be to create and implement the winning/losing states, dependent on the outcome of the game.

Implementing the win/lose states

I created a class called 'Game_Over', a child of 'States'. This was to be the state that is called (and the screen that is displayed) when the player loses the game, either to the timer or by falling into the lava.

- In the setup method, I added a delay of 1.5 seconds, using the Python time library. This was done to prevent the player from immediately jumping states upon losing, to give them a delay instead, and allow the sound effect to play.
- 3 buttons and 1 textbox were instantiated, the textbox being the title, simply displaying "GAME OVER". The three buttons were to allow the player to retry at the current difficulty, change their difficulty level, or return to the main menu. Passed into these objects were the following methods (of the 'Game_Over' class):
    - 'switch_difficulty' – sets the next state to "difficulty_select", as a reference to the difficulty selection state in the states dictionary. It plays the 'menu_forward' sound, and sets 'self.running' to False, hence changing the state.
    - 'switch_game' – sets the next state to "game", hence referencing the previous state. The difficulty won't be changed, because 'x_cells', 'y_cells' and 'timer' from the share state have not been changed. As a result, these values are still used for the generation and timer. It plays the 'menu_forward' sound effect, and sets 'self.running' to False.
    - 'switch_main_menu' sets 'self.next' to "main_menu", pointing to the main menu in the states dictionary. It plays the 'menu_back' sound effect, and sets 'self.running' to False. This sound effect is played because the player is backing out of the game.
- 'render' draws these all to the screen.

```
class Game_Over(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        time.sleep(1.5)
        self.running = True
        self.you_lose = Textbox((screen_x/2)-250, screen_y/9, red, "GAME OVER", "title_textbox", 500, 100, font_2)
        self.retry = Button((screen_x/2)-100, screen_y-500, yellow, "RETRY", "retry_game", 200, 50, font_1, self.switch_game)
        self.change_difficulty = Button((screen_x/2)-100, screen_y-400, yellow, "CHANGE DIFFICULTY", "retry_game", 200, 50, font_1, self.switch_difficulty)
        self.main_menu = Button((screen_x/2)-100, screen_y-300, yellow, "RETURN TO MENU", "return_to_menu", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.you_lose, self.main_menu, self.retry, self.change_difficulty

    def handle_events(self, event):
        self.main_menu.check_clicked(event)
        self.retry.check_clicked(event)
        self.change_difficulty.check_clicked(event)

    def switch_difficulty(self):
        self.next = "difficulty_select"
        menu_forward_sound.play()
        self.running = False

    def switch_game(self):
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.you_lose.draw(screen)
        self.main_menu.draw(screen)
        self.change_difficulty.draw(screen)
        self.retry.draw(screen)
```

The buttons all worked as expected, I added this state to be referenced when the player dies in the game state:

```
        self.goal.check_touching(self.character, self.timer)
        if self.timer.timer_expired == True:
            self.character.die()
            self.next = "game_over"
            self.running = False

        if self.goal.win:
            self.next = "game_won"
            self.running = False
```

'game_over' is

the reference for the ‘Game_Over’ state in the state dictionary.

I also added a reference to ‘game_won’ in the state dictionary, when the player wins. This is related to the state called when the player wins the game.

I created this state next, as a class under the name of ‘You_Win’, inheriting the ‘States’ class. This class was practically identical to the ‘Game_Over’ class, apart from a few key differences:

- The title was ‘YOU WIN!’ instead of ‘GAME OVER’.
- The sound effect played prior to the calling of this state was the ‘you win’ sound effect.
- This state was called upon the user winning the game, instead of losing.

```
class You_Win(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        time.sleep(1.5)
        self.running = True
        self.you_win = Textbox((screen_x/2)-250, screen_y/9, red, "YOU WIN!", "title_textbox", 500, 100, font_2)
        self.retry = Button((screen_x/2)-100, screen_y-500, yellow, "RETRY", "retry_game", 200, 50, font_1, self.switch_game)
        self.change_difficulty = Button((screen_x/2)-100, screen_y-400, yellow, "CHANGE DIFFICULTY", "retry_game", 200, 50, font_1, self.switch_difficulty)
        self.main_menu = Button((screen_x/2)-100, screen_y-300, yellow, "RETURN TO MENU", "return_to_menu", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.you_win, self.main_menu, self.retry, self.change_difficulty

    def handle_events(self, event):
        self.main_menu.check_clicked(event)
        self.retry.check_clicked(event)
        self.change_difficulty.check_clicked(event)

    def switch_difficulty(self):
        self.next = "difficulty_select"
        menu_forward_sound.play()
        self.running = False

    def switch_game(self):
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.you_win.draw(screen)
        self.main_menu.draw(screen)
        self.change_difficulty.draw(screen)
        self.retry.draw(screen)
```

I entered this state into the states dictionary, under the name of “game_won”, set to be called when the player wins (as aforementioned).

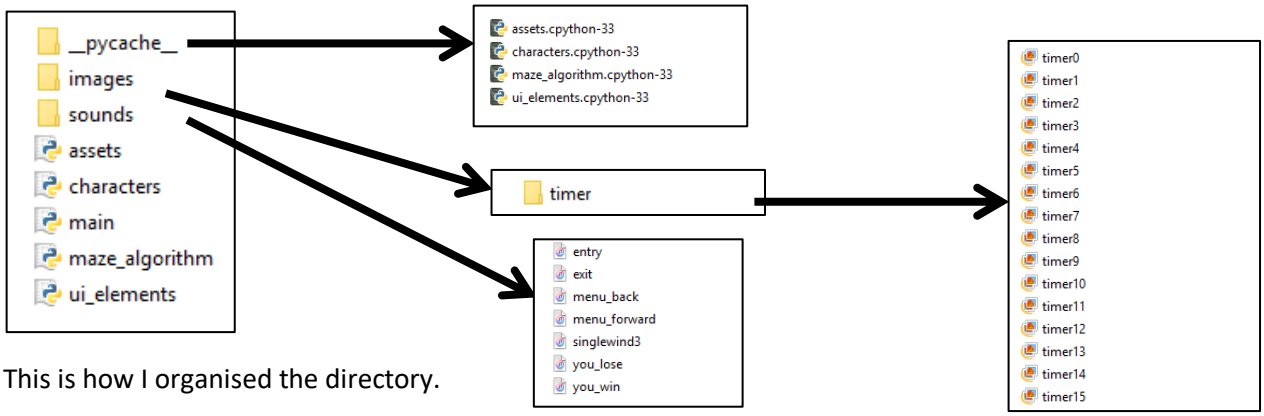
This state worked as expected, allowing the user to restart/navigate as planned.

And with this, the base program prototype is complete.

Future-proofing

Immediately after completing the program, my first steps were to future-proof the code and documentation, by organising the directory, and deleting any extra files that had been used during development and were no longer necessary to a final product.

I also looked to annotate my code further, detailing all of its purpose, and deleting obsolete code. Once this is done, I shall add all my classes into the documentation, separated by category. This is done to prepare for testing and debugging, as well as to ensure the program is understandable in the future, and feasible to further modification/updates and fixes.



This is how I organised the directory.

- '__pycache__' contained the compiled versions of assets.py, characters.py, maze_algorithm.py and ui_elements.py. This was done by Python automatically, upon running the program the first time (when main.py imported all of these files).
- 'images' contains a single folder, called 'timer'. This is a file of 16 images, representing the timer in its various stages, as it counts down.
- 'sounds' simply contains all the sound files for the game.

assets.py

```
import pygame
pygame.init()

# screen setup
screen_x = 1000
screen_y = 700
fps = 60

# font setup
font_1 = pygame.font.SysFont("segoeui", 20)
font_2 = pygame.font.SysFont("segoeuiblack", 40)

# sound setup
entry_sound = pygame.mixer.Sound("sounds/entry.wav")
exit_sound = pygame.mixer.Sound("sounds/exit.wav")
menu_back_sound = pygame.mixer.Sound("sounds/menu_back.wav")
menu_forward_sound = pygame.mixer.Sound("sounds/menu_forward.wav")
ticking_sound = pygame.mixer.Sound("sounds/singlewind3.wav")
lose_sound = pygame.mixer.Sound("sounds/you_lose.wav")
win_sound = pygame.mixer.Sound("sounds/you_win.wav")

#image setup
timer0_image = pygame.image.load("images/timer/timer0.png")#.convert_alpha()
timer1_image = pygame.image.load("images/timer/timer1.png")#.convert_alpha()
timer2_image = pygame.image.load("images/timer/timer2.png")#.convert_alpha()
timer3_image = pygame.image.load("images/timer/timer3.png")#.convert_alpha()
timer4_image = pygame.image.load("images/timer/timer4.png")#.convert_alpha()
timer5_image = pygame.image.load("images/timer/timer5.png")#.convert_alpha()
timer6_image = pygame.image.load("images/timer/timer6.png")#.convert_alpha()
timer7_image = pygame.image.load("images/timer/timer7.png")#.convert_alpha()
timer8_image = pygame.image.load("images/timer/timer8.png")#.convert_alpha()
timer9_image = pygame.image.load("images/timer/timer9.png")#.convert_alpha()
timer10_image = pygame.image.load("images/timer/timer10.png")#.convert_alpha()
timer11_image = pygame.image.load("images/timer/timer11.png")#.convert_alpha()
timer12_image = pygame.image.load("images/timer/timer12.png")#.convert_alpha()
timer13_image = pygame.image.load("images/timer/timer13.png")#.convert_alpha()
timer14_image = pygame.image.load("images/timer/timer14.png")#.convert_alpha()
timer15_image = pygame.image.load("images/timer/timer15.png")#.convert_alpha()

# colour setup
black = (0, 0, 0)
white = (255, 255, 255)
red = (255, 0, 0)
orange = (255, 127, 0)
yellow = (255, 255, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
indigo = (46, 43, 95)
violet =(139, 0, 255)
```

The entire code for assets.py is shown on the left.

- First, pygame is imported and initialised, to be used.
- The screen resolution and FPS (framerate) is then set.
- The fonts are then setup using pygame. This is done to be used in main.py.
- All the sounds are imported and assigned names using the pygame mixer module, to be used in the main program.
- The images are all imported and assigned names. There are only images for the timer in this game, as a replacement for Pygames lack of ability to allow 'shrinking circles'.
- Finally, the colour tuples are setup. This is done so that the colours can be referenced and implemented by name, rather than worrying about RGB values (in main.py). Furthermore, I can easily modify colours from assets.py, rather than having to change every individual colour in main.py.

characters.py

characters.py contains all the character classes that are used in main.py. This includes the wall, the character, the timer, the lava and the goal classes.

```
import pygame
from assets import *
```

This was used to initialise and setup pygame. It imports the assets file.

```
# Wall class - used for the maze.
class Wall:
    def __init__(self, x, y, size, colour):
        self.x = x
        self.y = y
        self.size = size
        self.colour = colour

    def draw(self, screen):
        pygame.draw.rect(screen, (self.colour), (self.x, self.y, self.size, self.size))
```

The wall class is shown above, with two simple methods.

```
# Character class - used for the player to control in the main game.
class Character:
    def __init__(self, x, y, width, height, colour, controls, speed):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.colour = colour
        self.controls = controls
        self.speed = speed
        self.alive = True
        self.move_delay = 75
        self.last_move = pygame.time.get_ticks()


    def draw(self, screen):
        if self.alive == True:
            pygame.draw.rect(screen, self.colour, pygame.Rect((self.x, self.y, self.width, self.height)))

    def wall_collision(self, walls):
        for wall in walls:
            if wall.x == self.x and wall.y == self.y:
                return True
        return False
```

```
    def move(self, walls, timer):
        if self.alive == True:
            now = pygame.time.get_ticks()

            if now - self.last_move > self.move_delay: # implements a slight movement delay
                self.last_move = now

                pressed_keys = pygame.key.get_pressed()
                if self.alive == True: # the character can only move if they're alive
                    """
                        The following first checks what key has been pressed, and then moves the character in that direction once.
                        It then checks if the character has collided with a wall.
                        If yes, they're moved back to where they were previously.
                    """
                    if pressed_keys[self.controls[0]]:
                        self.y -= self.speed
                        timer.reset()
                        if self.wall_collision(walls):
                            self.y += self.speed

                    if pressed_keys[self.controls[1]]:
                        self.y += self.speed
                        timer.reset()
                        if self.wall_collision(walls):
                            self.y -= self.speed

                    if pressed_keys[self.controls[2]]:
                        self.x += self.speed
                        timer.reset()
                        if self.wall_collision(walls):
                            self.x -= self.speed

                    if pressed_keys[self.controls[3]]:
                        self.x -= self.speed
                        timer.reset()
                        if self.wall_collision(walls):
                            self.x += self.speed
```

```
    def die(self):
        self.alive = False

    def reset(self):
        self.alive = True
```

The character class. I added annotations for the more complicated and ambiguous code in the move method, to explain what it does. This is done in case I ever need to make changes in the future.

```
# Timer class - counts how much more time the player can remain idle before death.
class Timer:
    def __init__(self, x, y, colour, width, height, duration):
        self.x = x
        self.y = y
        self.colour = colour
        self.width = width
        self.height = height
        self.duration = int(duration)
        self.last_tick = pygame.time.get_ticks() # Used to allow the timer to count down
        self.timer_expired = False
        self.sound_played = False
        self.paused = False
        self.original_duration = int(duration)
```

```
    def draw(self, screen):
        if self.paused == False: # Only counts down if the timer isn't paused
            """
            if the timer is at 'n' duration, display the image with n seconds left.
            """
            if self.duration == 15:
                screen.blit(timer15_image, (self.x, self.y))
            elif self.duration == 14:
                screen.blit(timer14_image, (self.x, self.y))
            elif self.duration == 13:
                screen.blit(timer13_image, (self.x, self.y))
            elif self.duration == 12:
                screen.blit(timer12_image, (self.x, self.y))
            elif self.duration == 11:
                screen.blit(timer11_image, (self.x, self.y))
            elif self.duration == 10:
                screen.blit(timer10_image, (self.x, self.y))
            elif self.duration == 9:
                screen.blit(timer9_image, (self.x, self.y))
            elif self.duration == 8:
                screen.blit(timer8_image, (self.x, self.y))
            elif self.duration == 7:
                screen.blit(timer7_image, (self.x, self.y))
            elif self.duration == 6:
                screen.blit(timer6_image, (self.x, self.y))
            elif self.duration == 5:
                screen.blit(timer5_image, (self.x, self.y))
            elif self.duration == 4:
                screen.blit(timer4_image, (self.x, self.y))
            elif self.duration == 3:
                screen.blit(timer3_image, (self.x, self.y))
            elif self.duration == 2:
                screen.blit(timer2_image, (self.x, self.y))
            elif self.duration == 1:
                screen.blit(timer1_image, (self.x, self.y))
            else:
                screen.blit(timer0_image, (self.x, self.y))
                self.expired()


    def countdown(self):
        if self.paused == False:
            if self.timer_expired == False:
                now = pygame.time.get_ticks()
                if now > self.last_tick + 1000:
                    self.duration -= 1
                    self.last_tick = now
                    ticking_sound.play()
```

```
    def expired(self):
        if self.paused == False:
            self.timer_expired = True
            if self.sound_played == False:
                lose_sound.play()
                self.sound_played = True

    def pause(self):
        self.paused = True

    def reset(self):
        self.timer_expired = False
        self.sound_played = False
        self.duration = self.original_duration
```

The timer class. I added annotations to the draw method which specifies how the timer counts down.

```
# Lava class - used to trail the player in the main game.
class Lava:
    def __init__(self, x, y, size, colour):
        self.x = x
        self.y = y
        self.width = size
        self.height = size
        self.colour = colour
        self.touching = False

    def draw(self, screen):
        pygame.draw.rect(screen, self.colour, pygame.Rect((self.x, self.y, self.width, self.height)))

    def check_touching(self, character, timer):
        if (character.x == self.x) and (character.y == self.y):
            self.touching_player(character, timer)

    def touching_player(self, character, timer):
        character.die()
        timer.duration = 0 # Sets the duration to 0, killing the player
```

The lava class is shown above.

```
# Goal class (child of lava) - a single cell that the player must reach to win.
# Operates similarly to the lava, except triggers victory.
class Goal(Lava):
    def __init__(self, x, y, size, colour):
        Lava.__init__(self, x, y, size, colour) # Initialises as a lava cell...
        # ...but has two more attributes
        self.sound_played = False
        self.win = False

    def check_touching(self, character, timer):
        if (character.x == self.x) and (character.y == self.y):
            self.touching_player(character, timer)

    def touching_player(self, character, timer):
        if self.sound_played == False:
            win_sound.play()
            self.sound_played = True
            self.won(timer, character)

    def won(self, timer, character):
        self.win = True
        timer.pause()
        character.die()
```

The goal class is shown above.

(maze_algorithm.py has already been fully explained and annotated.)

ui_elements.py

```
import pygame
from assets import *

from pygame.locals import *
pygame.init()
```

```
class Textbox:
    def __init__(self, x, y, colour, text, name, width, height, font):
        self.name = name # name of the textbox
        self.colour = colour # colour of the textbox
        self.text = str(text) # text on the textbox
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
        self.alt_txt = font.render(self.text, True, ((self.colour)))
        self.rect = pygame.Rect((x, y, width, height)) # stores the coordinates, width and height of the rectangle

    def draw(self, screen):
        pygame.draw.rect(screen, self.colour, self.rect) # draws the textbox
        pygame.draw.rect(screen, black, self.rect, 1) # draws the outline
        txt_rect = self.txt.get_rect(center=self.rect.center) # draws the text to the center of the textbox
        screen.blit(self.txt, txt_rect) # draws the text
```

```
class Button:
    def __init__(self, x, y, colour, text, name, width, height, font, function):
        self.name = name # name of the button
        self.colour = colour # colour of the button
        self.text = str(text) # text on the button
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
        self.alt_txt = font.render(self.text, True, ((self.colour)))
        self.rect = pygame.Rect((x, y, width, height)) # stores the coordinates, width and height of the rectangle
        self.clicked = False # determines whether the button is currently clicked
        self.function = function # stores the function to be called when the button is clicked

    def draw(self, screen):
        if self.clicked == False:
            pygame.draw.rect(screen, self.colour, self.rect) # draws the button
            pygame.draw.rect(screen, black, self.rect, 5) # draws the outline
            txt_rect = self.txt.get_rect(center=self.rect.center) # draws the text to the center of the button
            screen.blit(self.txt, txt_rect) # draws the text
        else:
            pygame.draw.rect(screen, black, self.rect) # draws the button
            pygame.draw.rect(screen, self.colour, self.rect, 5) # draws the outline
            txt_rect = self.alt_txt.get_rect(center=self.rect.center) # draws the text to the center of the button
            screen.blit(self.alt_txt, txt_rect) # draws the text

    def check_clicked(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN: # if the mouse is clicked...
            if self.rect.collidepoint(pygame.mouse.get_pos()) == True: # ...and the mouse is on the button...
                self.clicked = True # the button is clicked.
                #self.is_clicked() # call 'is_clicked'
        elif event.type == pygame.MOUSEBUTTONUP: # when the button is released...
            if self.clicked: # if it WAS clicked...
                self.function() # calls the appropriate function
            self.clicked = False # sets the button to unclicked once again, as the button is released
```

The textbox and button classes, fully annotated and explained.

main.py

This is the most important file of the lot, this is the file which brings together all of the other modules.

```
1  ''' Imports all needed modules and libraries.'''
2  import time
3  import pygame
4  from pygame.locals import *
5  from maze_algorithm import *
6  from ui_elements import *
7  from characters import *
8  from assets import *
9  entry_sound.play()
```

This is the first part of the code. It merely initialises all the modules, importing the other files, and plays the sound effect for entry into the game.

```
"""
The states class.
This is simply a template, which every state shall inherit methods and attributes from.
The subclasses may also add their own methods and attributes according to their needs.
"""
class States:
    # The share dictionary holds information to be shared between states.
    share = {
        "keys": [K_w, K_s, K_d, K_a],
        "x_cells": 20,
        "y_cells": 20,
        "timer": 15,
        }
    def __init__(self):
        self.running = True
        self.next = None
        self.quit = False

    def setup(self):
        pass

    def cleanup(self):
        pass

    def handle_events(self, event):
        pass

    def update(self):
        pass

    def render(self, screen):
        pass
```

This is the ‘States’ superclass, from which all other states inherit their methods and attributes. All the methods are self-explanatory, and don’t contain anything, apart from the constructor. The ‘share’ dictionary contains data that is to be shared between states.

```
class Main_Menu(States): # The state for the main menu.
    def __init(self):
        States.__init__(self) # Initialises the state using the States superclass constructor.

    def setup(self):
        self.running = True
        """
        Instantiates all buttons and textboxes.
        """
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "THE FLOOR IS LAVA", "title_textbox", 500, 100, font_2)
        self.play = Button((screen_x/2)-100, screen_y-500, yellow, "PLAY", "play_button", 200, 50, font_1, self.switch_play)
        self.instructions = Button((screen_x/2)-100, screen_y-400, yellow, "INSTRUCTIONS", "instructions_button", 200, 50, font_1, self.switch_instructions)
        self.controls = Button((screen_x/2)-100, screen_y-300, yellow, "CONTROLS", "controls_button", 200, 50, font_1, self.switch_controls)
        self.quit_button = Button((screen_x/2)-100, screen_y-200, yellow, "QUIT", "quit_button", 200, 50, font_1, self.quit_game)

    def cleanup(self):
        """
        Deletes all buttons and textboxes.
        """
        del self.title, self.play, self.instructions, self.quit_button

    def switch_play(self):
        """
        Directs to the difficulty select state.
        """
        self.next = "difficulty_select"
        menu_forward_sound.play()
        self.running = False

    def switch_instructions(self):
        """
        Directs to the instructions state.
        """
        self.next = "instructions_screen"
        menu_forward_sound.play()
        self.running = False

    def switch_controls(self):
        """
        Directs to the controls select state.
        """
        self.next = "controls_screen"
        menu_forward_sound.play()
        self.running = False
```

```
    def quit_game(self):
        """
        Exits the game.
        """
        exit_sound.play()
        self.quit = True

    def handle_events(self, event):
        """
        Checks if any buttons have been clicked.
        """
        self.play.check_clicked(event)
        self.instructions.check_clicked(event)
        self.controls.check_clicked(event)
        self.quit_button.check_clicked(event)

    def update(self):
        pass

    def render(self, screen):
        """
        Fills the screen with black and draws buttons and textboxes.
        """
        screen.fill(black)
        self.title.draw(screen)
        self.play.draw(screen)
        self.instructions.draw(screen)
        self.controls.draw(screen)
        self.quit_button.draw(screen)
```

The main menu state is shown above. I annotated all methods to explain their purpose.

```
class Instructions(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        """
        Instantiates all buttons and textboxes.
        """
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "INSTRUCTIONS", "title_textbox", 500, 100, font_2)
        """
        The following 5 objects are textboxes that contain the instructions.
        """
        self.instructions_1 = Textbox((screen_x/2)-450, screen_y - 500, red,
                                      "• Choose your difficulty and navigate your way through the maze, to the yellow goal.",
                                      "instructions_textbox", 900, 50, font_1)
        self.instructions_2 = Textbox((screen_x/2)-450, screen_y - 450, red,
                                      "• As you move through the maze, deadly lava will trail in your wake, so be careful.",
                                      "instructions_textbox", 900, 50, font_1)
        self.instructions_3 = Textbox((screen_x/2)-450, screen_y - 400, red,
                                      "• You have to think fast and precisely - a timer will be counting down, and can only be reset if you move.",
                                      "instructions_textbox", 900, 50, font_1)
        self.instructions_4 = Textbox((screen_x/2)-450, screen_y - 350, red,
                                      "• If the timer expires, the lava will engulf you, and you will fail.",
                                      "instructions_textbox", 900, 50, font_1)
        self.instructions_5 = Textbox((screen_x/2)-450, screen_y - 300, red,
                                      "• Have fun and good luck!",
                                      "instructions_textbox", 900, 50, font_1)
        """
        The back button, to return the player to the main menu.
        """
        self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        """
        Deletes all buttons and textboxes.
        """
        del self.title, self.instructions_1, self.instructions_2, self.instructions_3, self.instructions_4, self.instructions_5, self.back
```

```
    def switch_main_menu(self):
        """
        Directs to the main menu state.
        """
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def handle_events(self, event):
        """
        Checks if the back button has been clicked.
        """
        self.back.check_clicked(event)

    def update(self):
        pass

    def render(self, screen):
        """
        Draws all UI elements to the screen.
        """
        screen.fill(black)
        self.title.draw(screen)
        self.instructions_1.draw(screen)
        self.instructions_2.draw(screen)
        self.instructions_3.draw(screen)
        self.instructions_4.draw(screen)
        self.instructions_5.draw(screen)
        self.back.draw(screen)
```

The instructions class is shown over the previous two images.

```
class Controls_Setup(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "CONTROLS", "title_textbox", 500, 100, font_2)
        # The following button takes 'switch_to_wasd' as a method. It calls this method when it's clicked.
        self.wasd = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: WASD", "instructions_button", 200, 50, font_1, self.switch_to_wasd)
        # The following button takes 'switch_to_arrows' as a method. It calls this method when it's clicked.
        self.arrows = Button((screen_x/2)-100, screen_y-300, yellow, "SELECT: ARROW KEYS", "instructions_button", 200, 50, font_1, self.switch_to_arrows)
        self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.title, self.wasd, self.arrows, self.back

    def switch_to_wasd(self):
        """
        This modifies the value referenced by the key 'keys' in the share dictionary.
        This changes the controls to "WASD" to be used in the game.
        """
        self.share.update({"keys": [K_w, K_s, K_d, K_a]})
        menu_forward_sound.play()

    def switch_to_arrows(self):
        """
        This modifies the value referenced by the key 'keys' in the share dictionary.
        This changes the controls to the arrow keys to be used in the game.
        """
        self.share.update({"keys": [K_UP, K_DOWN, K_RIGHT, K_LEFT]})
        menu_forward_sound.play()

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False
```

```
    def handle_events(self, event):
        self.wasd.check_clicked(event)
        self.arrows.check_clicked(event)
        self.back.check_clicked(event)

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.title.draw(screen)
        self.wasd.draw(screen)
        self.arrows.draw(screen)
        self.back.draw(screen)
```

Above is the controls setup class, the screen from which the player can choose their controls layout.

```
class Difficulty(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "DIFFICULTY SELECT", "title_textbox", 500, 100, font_2)
        self.easy = Button((screen_x/2)-100, screen_y-500, yellow, "SELECT: EASY", "instructions_button", 200, 50, font_1, self.switch_easy)
        self.medium = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: MEDIUM", "instructions_button", 200, 50, font_1, self.switch_medium)
        self.hard = Button((screen_x/2)-100, screen_y-300, yellow, "SELECT: HARD", "instructions_button", 200, 50, font_1, self.switch_hard)
        self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.title, self.easy, self.medium, self.hard, self.back

    def handle_events(self, event):
        self.easy.check_clicked(event)
        self.medium.check_clicked(event)
        self.hard.check_clicked(event)
        self.back.check_clicked(event)

    def switch_easy(self):
        """
        If 'easy' difficulty is selected, 'x_cells', 'y_cells' and 'timer' are modified accordingly.
        This is done in the 'share' dictionary, and is carried to the game state.
        """
        self.share.update({"x_cells": 20})
        self.share.update({"y_cells": 20})
        self.share.update({"timer": 15})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_medium(self):
        """
        If 'medium' difficulty is selected, 'x_cells', 'y_cells' and 'timer' are modified accordingly.
        This is done in the 'share' dictionary, and is carried to the game state.
        """
        self.share.update({"x_cells": 35})
        self.share.update({"y_cells": 35})
        self.share.update({"timer": 10})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False
```

```
    def switch_hard(self):
        """
        If 'hard' difficulty is selected, 'x_cells', 'y_cells' and 'timer' are modified accordingly.
        This is done in the 'share' dictionary, and is carried to the game state.
        """
        self.share.update({"x_cells": 50})
        self.share.update({"y_cells": 50})
        self.share.update({"timer": 5})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.title.draw(screen)
        self.easy.draw(screen)
        self.medium.draw(screen)
        self.hard.draw(screen)
        self.back.draw(screen)
```

Above is the class for the difficulty selection screen.

Below is the most complicated part of the entire system: the game state.

```
class Game(States):
    def __init__(self):
        States.__init__(self) # Initialises via the constructor of the superclass.

    def setup(self):
        self.running = True
        ### generates the maze and creates the walls
        """
        Uses the maze algorithm, passing in the width and height of the grid (x_cells and y_cells).
        """
        self.maze_data = generate_maze(self.share["x_cells"], self.share["y_cells"])
        for index, row in enumerate(self.maze_data):
            row = "|" + row[1:] + "|"
            self.maze_data[index] = row
        self.maze_data.insert(0, "|" * (self.share["y_cells"] + 2))
        self.maze_data.append("|" * (self.share["x_cells"] +  2))
        self.walls = []
        self.cell_size = 10
        self.maze_x = ((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2)))
        self.maze_y = ((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2)))

        for x, tiles in enumerate(self.maze_data):
            for y, tile in enumerate(tiles):
                if tile == "|":
                    wall = Wall(self.maze_x + (y*10), self.maze_y + (x*10), self.cell_size, white)
                    self.walls.append(wall)
```

The setup method is very expansive, and his hence divided into smaller subsections, all with a purpose. These are denoted by '###', followed by a brief explanation.

```
        ### Finds the appropriate location for the character and goal
        if self.maze_data[1][1] == '|':
            if self.maze_data[1][2] == '|':
                if self.maze_data[2][1] == '|':
                    if self.maze_data[2][2] == '|':
                        pass
                    else:
                        self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size
                        self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size
                else:
                    self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)
                    self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size

            else:
                self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size
                self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)
        else:
            self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)
            self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)

        if self.maze_data[self.share["x_cells"]][self.share["x_cells"]] == '|':
            if self.maze_data[self.share["x_cells"] - 1][self.share["y_cells"]] == '|':
                if self.maze_data[self.share["x_cells"]][self.share["y_cells"] - 1] == '|':
                    if self.maze_data[self.share["x_cells"] - 1][self.share["y_cells"] - 1] == '|':
                        pass
                    else:
                        self.goal_xpos = (((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2))) - self.cell_size)
                        self.goal_ypos = (((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2))) - self.cell_size)
                else:
                    self.goal_xpos = ((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2)))
                    self.goal_ypos = (((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2))) - self.cell_size)
            else:
                self.goal_xpos = (((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2))) - self.cell_size)
                self.goal_ypos = ((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2)))
        else:
            self.goal_xpos = ((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2))) # Originally 350
            self.goal_ypos = ((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2))) # Originally 310
```

```
        ### Instantiates character, timer, textbox, goal
        self.character = Character(self.default_xpos, self.default_ypos, self.cell_size, self.cell_size, green, self.share["keys"], self.cell_size)
        self.timer = Timer(screen_x - 130, 30, red, 10, 10, self.share["timer"])
        self.title_textbox = Textbox((screen_x/2)-100, (screen_y/25), white, str(self.share["x_cells"]) + " by " + str(self.share["y_cells"]), "test", 200, 50, font_1)
        self.goal = Goal(self.goal_xpos, self.goal_ypos, self.cell_size, yellow)
        self.lava_cells = []
```

```
    def cleanup(self):
        """
        Deletes all objects on screen, including walls and lava.
        """
        del self.character, self.title_textbox, self.timer, self.goal
        for wall in self.walls:
            del wall
        for lava in self.lava_cells:
            del lava

    def update(self):
        """
        The following two variables are used to keep track of the character location,
        and to check when the character moves.
        """
        self.character_x = self.character.x
        self.character_y = self.character.y

        self.character.move(self.walls, self.timer) # Allows the character to move.
        self.timer.countdown() # The timer counts down.

        if (self.character.x != self.character_x) or (self.character.y != self.character_y): # If the character has moved at all...
            lava_cell = Lava(self.character_x, self.character_y, self.cell_size, red) # Instantiate a lava cell at the character's previous location.
            self.lava_cells.append(lava_cell) # Append this cell to the array of lava cells.

        for lava in self.lava_cells: # For all the lava cells in the array...
            lava.check_touching(self.character, self.timer) # Check if the character is touching any of the cells.

        self.goal.check_touching(self.character, self.timer) # Check if the character has reached the goal.
        if self.timer.timer_expired == True: # If the timer has expired...
            self.character.die() # Kill the character.
            self.next = "game_over" # Set the next state to the game over screen.
            self.running = False # Move to the next state.

        if self.goal.win: # If the character wins...
            self.next = "game_won" # Set the next state to the winning screen.
            self.running = False # Move to the next state.
```

```
    def render(self, screen):
        screen.fill(black)
        for wall in self.walls: # Draws all the walls to screen.
            wall.draw(screen)
        self.character.draw(screen)
        self.goal.draw(screen)
        self.timer.draw(screen)
        self.title_textbox.draw(screen)
        for lava in self.lava_cells: # Draws all the lava cells to screen.
            lava.draw(screen)
```

A lot of the game state concerns how the lava cells, walls, character and timer interact, as shown above, unlike the other states, which are more oriented around UI elements (textboxes and buttons).

```
class You_Win(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        time.sleep(1.5)
        self.running = True
        self.you_win = Textbox((screen_x/2)-250, screen_y/9, red, "YOU WIN!", "title_textbox", 500, 100, font_2)
        self.retry = Button((screen_x/2)-100, screen_y-500, yellow, "RETRY", "retry_game", 200, 50, font_1, self.switch_game)
        self.change_difficulty = Button((screen_x/2)-100, screen_y-400, yellow, "CHANGE DIFFICULTY", "retry_game", 200, 50, font_1, self.switch_difficulty)
        self.main_menu = Button((screen_x/2)-100, screen_y-300, yellow, "RETURN TO MENU", "return_to_menu", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.you_win, self.main_menu, self.retry, self.change_difficulty

    def handle_events(self, event):
        self.main_menu.check_clicked(event)
        self.retry.check_clicked(event)
        self.change_difficulty.check_clicked(event)

    def switch_difficulty(self):
        self.next = "difficulty_select"
        menu_forward_sound.play()
        self.running = False

    def switch_game(self):
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.you_win.draw(screen)
        self.main_menu.draw(screen)
        self.change_difficulty.draw(screen)
        self.retry.draw(screen)
```

```
class Game_Over(States):
    def __init__(self):
        States.__init__(self)

    def setup(self):
        time.sleep(1.5) # Delay is used to stop the state jumping immediately (from the game state).
        self.running = True
        self.you_lose = Textbox((screen_x/2)-250, screen_y/9, red, "GAME OVER", "title_textbox", 500, 100, font_2)
        self.retry = Button((screen_x/2)-100, screen_y-500, yellow, "RETRY", "retry_game", 200, 50, font_1, self.switch_game)
        self.change_difficulty = Button((screen_x/2)-100, screen_y-400, yellow, "CHANGE DIFFICULTY", "retry_game", 200, 50, font_1, self.switch_difficulty)
        self.main_menu = Button((screen_x/2)-100, screen_y-300, yellow, "RETURN TO MENU", "return_to_menu", 200, 50, font_1, self.switch_main_menu)

    def cleanup(self):
        del self.you_lose, self.main_menu, self.retry, self.change_difficulty

    def handle_events(self, event):
        self.main_menu.check_clicked(event)
        self.retry.check_clicked(event)
        self.change_difficulty.check_clicked(event)

    def switch_difficulty(self):
        self.next = "difficulty_select"
        menu_forward_sound.play()
        self.running = False

    def switch_game(self):
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.you_lose.draw(screen)
        self.main_menu.draw(screen)
        self.change_difficulty.draw(screen)
        self.retry.draw(screen)
```

Above are the ‘Game_Over’ and ‘You_Win’ states, called as appropriate.

```
class Control: # This class controls the entire program, handling all the states.
    def __init__(self):
        self.screen = pygame.display.set_mode((screen_x, screen_y)) # Sets up the screen according to the resolution (from assets.py).
        self.clock = pygame.time.Clock() # Sets up the pygame clock.
        self.running = True # Starts running the current state.
        self.state_dict = None # Initialises the state dictionary as None.
        self.current_state = None # Initialises the current state as None.

    def setup(self, state_dict, start_state): # This takes in the state dictionary and the start state, as a starting point for the program.
        # 'start_state' represents the key for the instance of the state in 'state_dict'.
        self.state_dict = state_dict
        self.current_state = self.state_dict[start_state] # Sets the current state to an instance of the start state, referencing the state dictionary.
        self.current_state.setup() # Calls the setup method for the current state, instantiating all objects needed.

    def handle_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT: # Checks if the [X] button has been pressed.
                self.running = False
            self.current_state.handle_events(event)

        if not(self.current_state.running): # If the current state is no longer running...
            self.change_state() # ...move to the next state.

        if self.current_state.quit == True:
            self.running = False # Quits the game.

    def change_state(self):
        new_state = self.current_state.next # Sets up the key for the new state, to reference in 'state_dict'.
        self.current_state.cleanup() # Deletes all objects from the current state.
        self.current_state = self.state_dict[new_state] # Sets the current state to the object referenced by the key 'new_state', from 'state_dict'.
        self.current_state.setup() # Sets up the new current state.

    def main_loop(self): # Acts as a game loop.
        while self.running:
            self.clock.tick(fps)
            self.handle_events()
            self.current_state.update()
            self.current_state.render(self.screen)
            pygame.display.flip() # Refreshes display.
```

The control class is shown above. I added a heavy amount of annotation, because this is a particularly difficult class to understand.

```
pygame.init() # Initialises pygame.
app = Control() # Creates an instance of 'Control', named 'app'.
"""
Below is the state dictionary.
It contains instances of all the states.
"""
state_dict = {
    "main_menu": Main_Menu(),
    "difficulty_select": Difficulty(),
    "instructions_screen": Instructions(),
    "controls_screen" : Controls_Setup(),
    "game" : Game(),
    "game_won": You_Win(),
    "game_over": Game_Over(),
    }
"""
Below calls the setup method for app (instance of Control),
and passes in the state dictionary and the main menu as the starting state.
"""
app.setup(state_dict, "main_menu")
app.main_loop()
pygame.quit() # Quits pygame. Only happens when [X] is pressed.
```

This is the end of the program. The control class is instantiated, and the state dictionary resides here.

I had now deleted all useless code, and organised classes, as well as adding annotations, explaining certain parts of the program. This was done to ensure that the program would have minimum possible bugs/problems, and to make the code easy to understand in the future, where I may want to further modify the game, and make it easy to add more classes.

3.3.1b – Prototype solutions

With the completion of the previous section, the first iteration of the program is now complete. Now that we have a working system, we can make use of iterative development, taking feedback for the current prototype.

This section will explore and evaluate all major parts of the project, and allow the stakeholders to explore every part and make their own assessments and opinions on each part. From this, I can look to develop and improve the product through iterative development.

<u>The button and textbox (UI elements)</u>

This section looks specifically at the buttons and textboxes; this includes information textboxes and title textboxes. I showed the user the following sample images of a title textbox, an information textbox and a button.

I asked for their individual opinions on a variety of categories: colour, shape/size and text.

Colour:

NAME:	COMMENT:
[redacted]	"I think there should be more distinction between the title textboxes and the information textboxes, in terms of colour."
[redacted]	"I agree with [redacted]."
[redacted]	"I like how you've kept it all themed around lava; it looks great and very thematically appropriate."
[redacted]	"The colours are a sharp contrast from the background, which makes it easy to read and notice."
[redacted]	"I don't like that the title and the information textboxes are both red, other than that, these are fine."

Shape/size:

NAME:	COMMENT:
[redacted]	"The title is big, blocky and bold, which is appropriate."
[redacted]	"I think the shapes are all perfect, keeping the 'blocky' aesthetic."
[redacted]	"I like how more important UI elements are bigger, while others are smaller."
[redacted]	"The buttons are smaller than the title, but not too small, so that they're still easy to click and navigate with."
[redacted]	"All the UI elements are clear and easy to see."

Text:

NAME:	COMMENT:
[redacted]	"I like that the text on the title boxes is bold and black, it really makes it pop."
[redacted]	"The text is clear and easy to read for all the UI elements."
[redacted]	"The bullet points on the information textboxes

	are great, because they help the user to differentiate them from each other, good job!"
[redacted]	"All of the text is clear, and you can clearly tell which box is intended to be a title."
[redacted]	"The varying text sizes allow some parts, e.g. the title, to be more eye-catching than others."

I met with the stakeholders to discuss their comments among ourselves. I then asked the stakeholders what improvements I should make upon the next iteration, based on their previous comments. Their responses are recorded below.

[redacted]: "Add a clear contrast between the title boxes and the information boxes. You could use different colours to let this happen."

[redacted]: "Keep the blocky theme, but try to make elements more distinct from one another."

[redacted]: "Making the contrast between title and information textboxes is an important one. Keep with the 'fiery' theme when using colours – perhaps use another shade of red/yellow, or use a colour like orange."

[redacted]: "Other than what the others have mentioned, these are great."

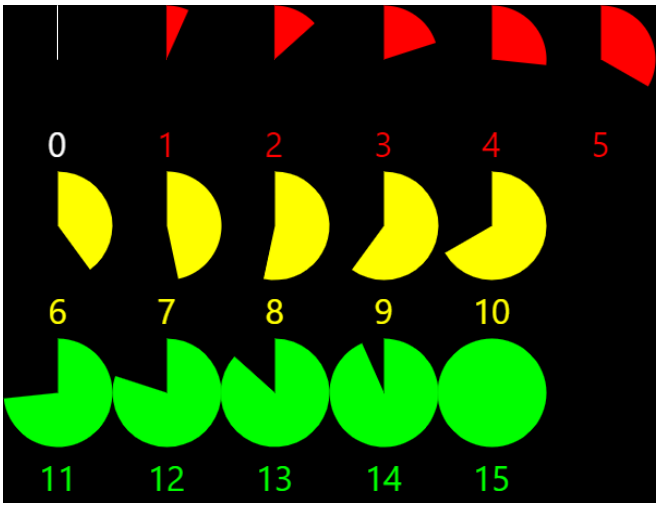
[redacted]: "I agree with [redacted] – you should use colour to make the distinction between title and information textboxes more apparent."

They collective agreed with [redacted] that the best option is orange, because it fits the theme. Hence, this will be done in the next iteration.

<u>The timer</u>

The timer is the second major part of the program that my stakeholders must review in-depth.

Upon completion of the timer, I showed my stakeholders the timer designs, how they 'tick' and their appearance in the main game.



I asked for their feedback, if they had any praise or criticisms to give.

[redacted]: “These look really good – they contrast the black background, and they change colour as the remaining time gets lower and lower. I really like these, well done.”

[redacted]: “The way you included individual images to give the ‘ticking’ illusion look very convincing, as does the different colours.”

[redacted]: “I love how you ensured that the timer reduces by exactly one-fifteenth of the circle every time, that’s a really cool and detailed artistic effect.”

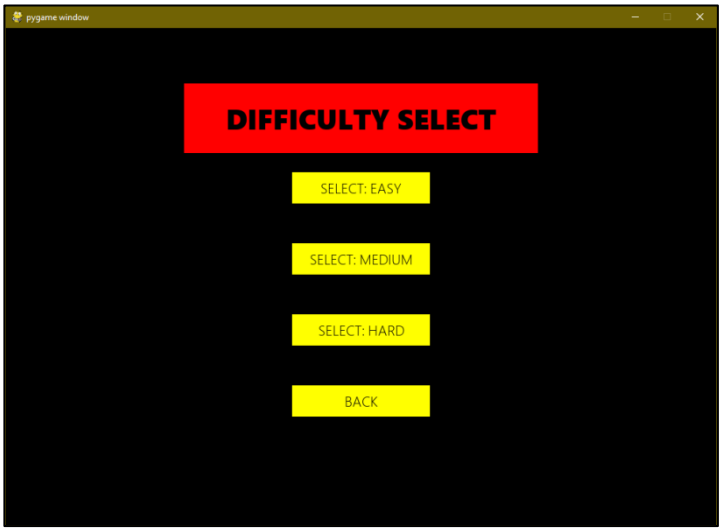
[redacted]: “I like the use of colours here. I have no criticisms to give.”

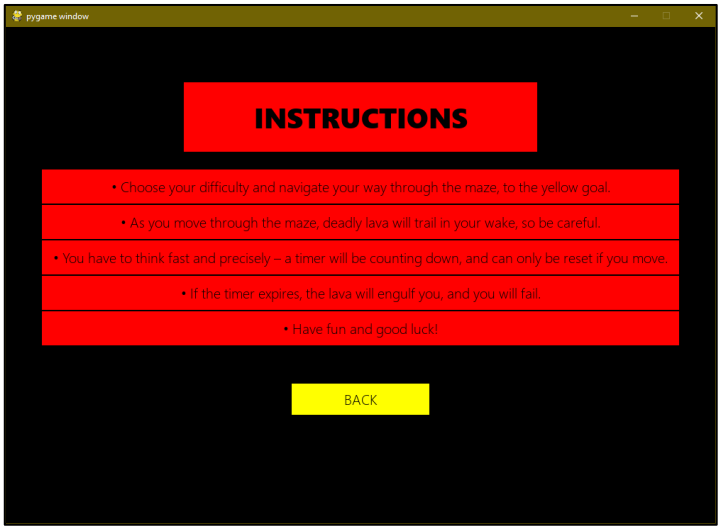
[redacted]: “The colours look great and are thematically appropriate.”

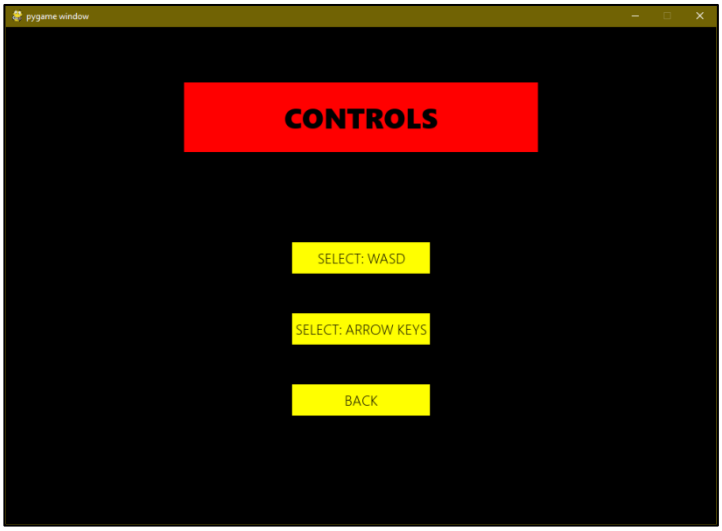
The stakeholders all agreed that the timer didn’t need any further changes or improvements upon subsequent iterations.

<u>The individual state screens</u>

Next, I showed them the individual state screens.







I asked if they had any criticisms/improvements of these screens, for future iterations.

[redacted]: “I think the buttons on the ‘controls’ screen can be centred much better. Right now, the gap between them and the title looks very awkward.”

[redacted]: “I do think you should add the option to exit the game directly from the winning and losing screens.”

[redacted]: “It would be really great if you had some way of actually tracking a user’s progress across the games – maybe a points or leader board system. Right now, it’s impossible for a user to tell how well they’re performing.”

[redacted]: “I agree with [redacted], and think you should shift the buttons upwards so that they match the layout of the “GAME OVER” and “YOU WIN” screens.”

[redacted]: “As aforementioned, change the colours of the instruction textboxes, to separate them from the title textbox.”

We were all particularly interested in [redacted]’s point, seeing as nothing of the sort had been implemented yet. We had an in-depth discussion as to how would be the best way to add a feature which shows a user how well they are doing.

[redacted] proposed a system where information textboxes show the user how many games they have won (on each difficulty) during the duration for which the game has been active. The other stakeholders really liked this idea.

I agreed with this idea, because it can be easily done – all I’d need to do is instantiate three textboxes, and use a variable, likely in the ‘share’ dictionary that keeps track of the number of wins a player has in any difficulty.

As a result, we collectively decided that (in the next iteration) there will be the addition of textboxes that show the number of wins the player has in each difficulty. A basic design for these is provided below:

The stakeholders all agreed with this design idea, and hence it would be implemented.

<u>The main game</u>

Next, I asked the stakeholders to look at the main game. I showed them the gameplay as it was in its current state, in all three difficulty levels.

I then asked for feedback on what could be improved aesthetically, gameplay-wise, and if there are any other features I need to add into the main game.

Their responses are recorded below.

Aesthetics:

NAME:	COMMENT:
[redacted]	"Honestly, I really like the aesthetics of the game. I like that the textbox is white instead of orange, red or yellow, it helps it stand out a lot."
[redacted]	"I think you should probably put the name of the difficulty level in the top textbox."
[redacted]	"I think everything's fine here, the character, lava and goal are all distinct from the maze. Good job."
[redacted]	"Everything is clearly visible off of the background, no changes from me."
[redacted]	"I like that you've kept the maze always centred."

Gameplay:

NAME:	COMMENT:
[redacted]	"The gameplay is great."
[redacted]	"I do feel like the gameplay could be a bit smoother – right now, the characters move a bit 'stiffly'."
[redacted]	"As I mentioned before, the game is easy to understand what is what, I have no complaints."
[redacted]	"The custom controls make the game a lot more tailored to the user's own experience, and allows them to play comfortably."
[redacted]	"I partially agree with [redacted] – at times, there character felt slightly unresponsive."

Other features:

NAME:	COMMENT:
[redacted]	"I think this is fine overall, I don't think there's anything major that you need to change."
[redacted]	"Apart from my two previous criticisms, I don't think there's anything else you need to add."
[redacted]	"You could add an option to forfeit the game from the game itself, rather than having to wait to die/kill yourself."
[redacted]	"I second [redacted], I think it'll make the game a lot more streamlined, and take out waiting time."
[redacted]	"I also agree with [redacted] and [redacted]."

We discussed [redacted]'s and [redacted]'s major points, and found that everyone agreed, especially with [redacted]. It was frustrating to have to wait for the character to die if, for example, the wrong difficulty was selected.

This was to be implemented as a 'FORFEIT' yellow button directly beneath the maze. Upon clicking this button, the game would end, and the user would be sent to the 'GAME OVER' screen. Below is a basic design for this button.

FORFEIT

The stakeholders were all satisfied by this design idea, and hence, it would be implemented.

The second iteration

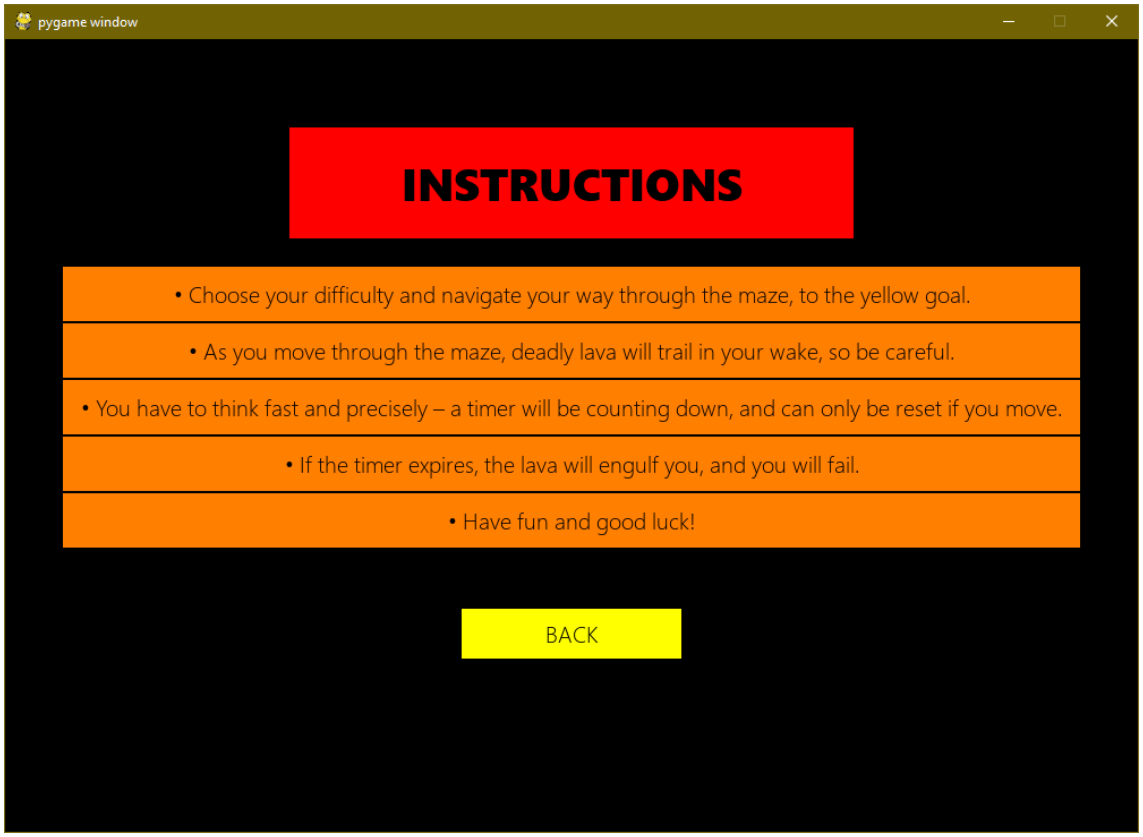
The stakeholders had now explored the entirety of the game, and made points of their desired changes to be made to the system. These are all compiled below, along with their justification:

1. Change the information textboxes to be orange, so that they are more distinguishable from title textboxes.
2. Shift the appearance of the 'controls' menu, such that the buttons are slightly higher, the gap between the title and the buttons looks very out of place.
3. Add the option to exit the game (a 'QUIT' button) from the winning/losing screens, rather than having to back out to the main menu screen.
4. Add textboxes to the title screen that show how many wins the user has achieved on each difficulty, in order to allow them to gain some sense of progression throughout the game.
5. Add the name of the difficulty to the top textbox of the main game screen, because the resolution alone is too ambiguous for a user.
6. Improve the movement of the character to be less 'stiff' and more 'fluid'.
7. Implement a 'FORFEIT' button in the main game screen, to allow the user to give up (and immediately lose).

I implemented these one by one.

Firstly, I went to the 'Instructions' class in main.py, and went to the 'setup' method. I changed the colour from 'red' to 'orange'.

```
120 class Instructions(States):
121     def __init__(self):
122         States.__init__(self)
123 
124     def setup(self):
125         """
126         Instantiates all buttons and textboxes.
127         """
128         self.running = True
129         self.title = Textbox((screen_x/2)-250, screen_y/9, red, "INSTRUCTIONS", "title_textbox", 500, 100, font_2)
130         """
131         The following 5 objects are textboxes that contain the instructions.
132         """
133         self.instructions_1 = Textbox((screen_x/2)-450, screen_y - 500, orange,
134                                       "* Choose your difficulty and navigate your way through the maze, to the yellow goal.",
135                                       "instructions_textbox", 900, 50, font_1)
136         self.instructions_2 = Textbox((screen_x/2)-450, screen_y - 450, orange,
137                                       "* As you move through the maze, deadly lava will trail in your wake, so be careful.",
138                                       "instructions_textbox", 900, 50, font_1)
139         self.instructions_3 = Textbox((screen_x/2)-450, screen_y - 400, orange,
140                                       "* You have to think fast and precisely - a timer will be counting down, and can only be reset if you move.",
141                                       "instructions_textbox", 900, 50, font_1)
142         self.instructions_4 = Textbox((screen_x/2)-450, screen_y - 350, orange,
143                                       "* If the timer expires, the lava will engulf you, and you will fail.",
144                                       "instructions_textbox", 900, 50, font_1)
145         self.instructions_5 = Textbox((screen_x/2)-450, screen_y - 300, orange,
146                                       "* Have fun and good luck!",
```



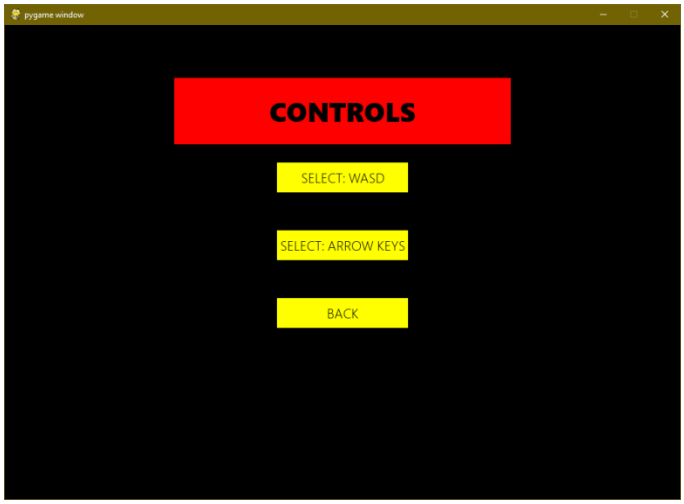
I then checked that this worked on the program:

It did just fine. This step is now complete.

I then went about remedying the 'awkward' gap between the buttons and the title on the controls menu.

```
190 class Controls_Setup(States):
191     def __init__(self):
192         States.__init__(self)
193 
194     def setup(self):
195         self.running = True
196         self.title = Textbox((screen_x/2)-250, screen_y/9, red, "CONTROLS", "title_textbox", 500, 100, font_2)
197         # The following button takes 'switch_to_wasd' as a method. It calls this method when it's clicked.
198         self.wasd = Button((screen_x/2)-100, screen_y-500, yellow, "SELECT: WASD", "instructions_button", 200, 50, font_1, self.switch_to_wasd)
199         # The following button takes 'switch_to_arrows' as a method. It calls this method when it's clicked.
200         self.arrows = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: ARROW KEYS", "instructions_button", 200, 50, font_1, self.switch_to_arrows)
201         self.back = Button((screen_x/2)-100, screen_y-300, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)
```

I changed the value of the y-position of the 'WASD', 'ARROW KEYS' and 'BACK' buttons to the values shown on screen ('screen_y-500', 'screen_y-400' and 'screen_y-300' respectively). This moved the buttons up from the bottom of the screen, eliminating the gap. The results are shown below:



I went back to the stakeholders to review my results so far. They all agreed that this was a major improvement over the previous product.

I then continued to develop the program, moving onto adding a 'QUIT' button to the bottom of the 'YOU WIN!' and 'GAME OVER' screens.

To start with, I instantiated a button in their setup methods. This button was called 'self.quit_button', and had a new method, 'quit_game' passed into it as the function. All this does is exit the game.

```
478     def quit_game(self):
479         """
480         Exits the game.
481         """
482         exit_sound.play()
```

This was done in both state classes, so that the user can quit the game regardless of whether they won or lost their previous game.

I made sure to include the 'self.quit_button' object in the cleanup methods, the render methods and the handle events methods, so that the button would appear on screen and be fully functional and clickable.

I tested out the code for both the winning and losing screens…

The buttons appeared and functioned correctly on both screens. Upon clicking, I was immediately booted out of the game.

Next, I needed to add textboxes to the main menu, which show the amount of wins a user has achieved on every difficulty setting.

I started by adding variables to the 'share' dictionary. The key names were 'easy_wins', 'medium_wins' and 'hard_wins'. They were all initialised at a value of 0.

```
16  class States:
17      # The share dictionary holds information to be shared between states.
18      share = {
19          "keys": [K_w, K_s, K_d, K_a],
20          "x_cells": 20,
21          "y_cells": 20,
22          "timer": 15,
23          "easy_wins": 0,
24          "medium_wins": 0,
25          "hard_wins": 0,
26          }
```

These variables track the number of wins a user has achieved in one playtime.

Next, I added code to the 'Game' state, to update these values upon a win.

```
435         if self.goal.win: # If the character wins...
436             """
437             The following 'if' statements check what difficulty the player was playing at.
438             """
439             if (self.share["x_cells"] == 20) and (self.share["y_cells"] == 20) and (self.share["timer"] == 15):
440                 self.share["easy_wins"] = self.share["easy_wins"] + 1
441             if (self.share["x_cells"] == 35) and (self.share["y_cells"] == 35) and (self.share["timer"] == 10):
442                 self.share["medium_wins"] = self.share["medium_wins"] + 1
443             if (self.share["x_cells"] == 50) and (self.share["y_cells"] == 50) and (self.share["timer"] == 5):
444                 self.share["hard_wins"] = self.share["hard_wins"] + 1
445 
446             self.next = "game_won" # Set the next state to the winning screen.
447             self.running = False # Move to the next state.
```

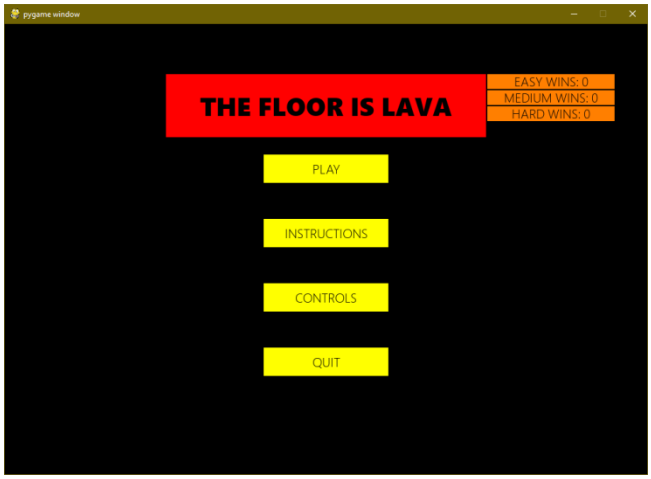
The above simply checks the difficult the player was playing at, and updates the according values.

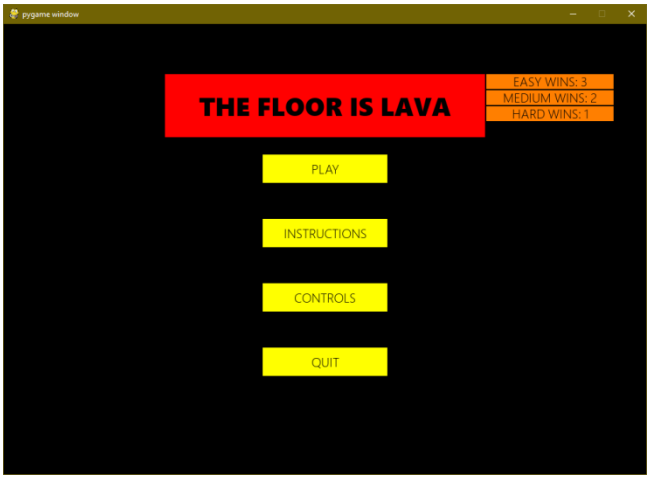
I then created the textboxes on the main menu, in the 'Main_Menu' state.

```
47  class Main_Menu(States): # The state for the main menu.
48      def __init(self):
49          States.__init__(self) # Initialises the state using the States superclass constructor.
50  
51      def setup(self):
52          self.running = True
53          """
54          Instantiates all buttons and textboxes.
55          """
56          self.title = Textbox((screen_x/2)-250, screen_y/9, red, "THE FLOOR IS LAVA", "title_textbox", 500, 100, font_2)
57          self.play = Button((screen_x/2)-100, screen_y-500, yellow, "PLAY", "play_button", 200, 50, font_1, self.switch_play)
58          self.instructions = Button((screen_x/2)-100, screen_y-400, yellow, "INSTRUCTIONS", "instructions_button", 200, 50, font_1, self.switch_instructions)
59          self.controls = Button((screen_x/2)-100, screen_y-300, yellow, "CONTROLS", "controls_button", 200, 50, font_1, self.switch_controls)
60          self.quit_button = Button((screen_x/2)-100, screen_y-200, yellow, "QUIT", "quit_button", 200, 50, font_1, self.quit_game)
61          self.easy_wins_textbox = Textbox((screen_x-250), screen_y/9, orange, "EASY WINS: " + str(self.share["easy_wins"]), "easy_wins", 200, 25, font_1)
62          self.medium_wins_textbox = Textbox((screen_x-250), screen_y/9 + 25, orange, "MEDIUM WINS: " + str(self.share["medium_wins"]), "medium_wins", 200, 25, font_1)
63          self.hard_wins_textbox = Textbox((screen_x-250), screen_y/9 + 50, orange, "HARD WINS: " + str(self.share["hard_wins"]), "hard_wins", 200, 25, font_1)
```

'self.easy_wins_textbox', 'self.medium_wins_textbox' and 'self.hard_wins_textbox' are the three textboxes that are to be drawn to the screen. They take the values from the 'share' dictionary, and uses them as text to show how many wins the player has on each level. The textboxes are rendered in the render method and deleted in the cleanup method.

I then showed the stakeholders the results:





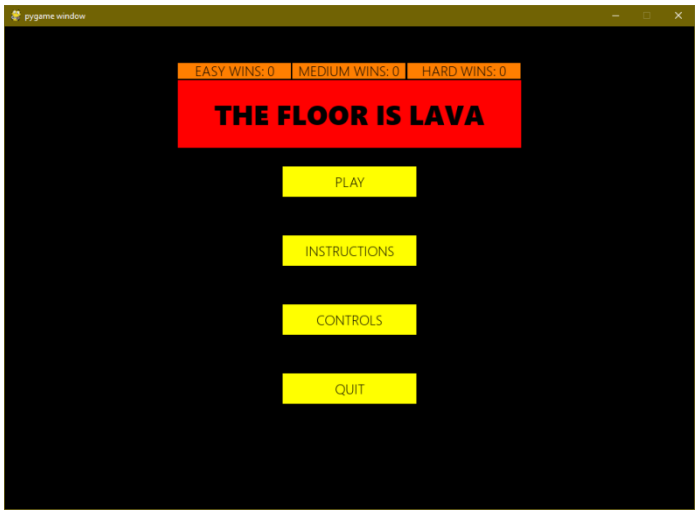
While all of them liked the appearance and functionality of the individual boxes, they were divided on the positioning of the boxes.

Many of the stakeholders disliked how the UI was no longer symmetrical.

- [redacted] proposed that the boxes should be stacked at the bottom of the screen.
- [redacted] suggested that the boxes be at the bottom of the screen, but in a single horizontal line, rather than stacked one on top of the other.
- [redacted] suggested that the boxes be in a line, as described by [redacted], except directly above the title box. Upon this, all the other stakeholders agreed this was the best idea – the design was very compact and symmetrical, and easy to read.

I then implemented this, by changing how the textboxes are instantiated.

```
self.easy_wins_textbox = Textbox((screen_x/2)-250, screen_y/9 - 25, orange, "EASY WINS: " + str(self.share["easy_wins"]), "easy_wins", 500/3, 25, font_1)
self.medium_wins_textbox = Textbox((screen_x/2)-(250/3), screen_y/9 - 25, orange, "MEDIUM WINS: " + str(self.share["medium_wins"]), "medium_wins", 500/3, 25, font_1)
self.hard_wins_textbox = Textbox((screen_x/2)+(250/3), screen_y/9 - 25, orange, "HARD WINS: " + str(self.share["hard_wins"]), "hard_wins", 500/3, 25, font_1)
```



I showed the stakeholders the new design. They all agreed that this version looked a lot more natural and aesthetically pleasing, while still being clear and easy to read.

I then looked to move onto the next step: adding the difficulty title to the top of the textbox in the game state.

To start with, I decided to add an attribute called ‘current_difficulty’ into the share state.

```
class States:
    # The share dictionary holds information to be shared between states.
    share = {
        "keys": [K_w, K_s, K_d, K_a],
        "x_cells": 20,
        "y_cells": 20,
        "timer": 15,
        "easy_wins": 0,
        "medium_wins": 0,
        "hard_wins": 0,
        "current_difficulty": None,
        }
```

Initialised at ‘None’, this is modified according to the difficult selected at the ‘Difficulty’ state. From this, I decided I could improve my code for checking the difficulty level (when updating the number of wins), by simply checking for this variable, rather than ‘x_cells’, ‘y_cells’ and ‘timer’.

```
        if self.goal.win: # If the character wins...
            """
            The following 'if' statements check what difficulty the player was playing at.
            """
            if self.share["current_difficulty"] == "EASY":
                self.share["easy_wins"] = self.share["easy_wins"] + 1
            if self.share["current_difficulty"] == "MEDIUM":
                self.share["medium_wins"] = self.share["medium_wins"] + 1
            if self.share["current_difficulty"] == "HARD":
                self.share["hard_wins"] = self.share["hard_wins"] + 1

            self.next = "game_won" # Set the next state to the winning screen.
            self.running = False # Move to the next state.
```

I then changed the title textbox for the game state to include ‘self.share[“current_difficulty”]’ as part of the title.

```
self.title_textbox = Textbox((screen_x/2)-100, (screen_y/25), white, str(self.share["current_difficulty"]) + ": " + str(self.share["x_cells"]) + " by " + str(self.share["y_cells"]), "test", 200, 50, font_1)
```

I concatenated the text together in order to get a title, and showed my stakeholders the results.

They were all happy with the result, stating that it was much more clear what difficulty level the user was playing on. I moved onto the next point: making the game 'smoother'.

I tested out the effect of changing the framerate (the 'FPS' variable in assets.py) on the main game.

I found that, as I increased the value of FPS, the game would seem to become steadily smoother. I conversed about this with my stakeholders who agreed; the higher the FPS value, the smoother the game will become. We tested for massive numbers, and found that this trend continued. I also noted how this did not affect the timer, or any other elements of the game – only the user's movement would be affected. As a result, we made the collective decision to turn the FPS from 60 to 144, in order to increase the fluidity of movement for the character. (This was done in assets.py).

I also decreased the movement delay in the character class from 75 to 65. This was done to make the character more responsive and move more smoothly, while not making the character too fast. Upon showing the stakeholders, they agreed the character felt far more responsive and controllable.

```
4   # screen setup
5   screen_x = 1000
6   screen_y = 700
7   fps = 144
```

Thus, this issue was solved, allowing us to move onto implementing 'forfeit' buttons.

I instantiated a button called 'self.forfeit_button', and added a method associated with it to the game class. This method was simply called 'forfeit' and all it did was set the timer's remaining duration to 0, hence immediately killing the player. This method was then passed into the button, allowing the button to immediately kill the player and end the game when pressed.

```
403         self.forfeit_button = Button((screen_x/2)-100, screen_y-200, yellow, "FORFEIT", "forfeit_button", 200, 50, font_1, self.forfeit)
404
405
406     def cleanup(self):
407         """
408         Deletes all objects on screen, including walls and lava.
409         """
410         del self.character, self.title_textbox, self.timer, self.goal, self.forfeit_button
411         for wall in self.walls:
412             del wall
413         for lava in self.lava_cells:
414             del lava
415
416     def handle_events(self, event):
417         self.forfeit_button.check_clicked(event)
```

```
458     def forfeit(self):
459         self.timer.duration = 0
460
461     def render(self, screen):
462         screen.fill(black)
463         for wall in self.walls: # Draws all the walls to screen.
464             wall.draw(screen)
465         self.character.draw(screen)
466         self.goal.draw(screen)
467         self.timer.draw(screen)
468         self.title_textbox.draw(screen)
469         for lava in self.lava_cells: # Draws all the lava cells to screen.
470             lava.draw(screen)
471         self.forfeit_button.draw(screen)
```

I also needed to add a 'handle_events' method to check if the button had been clicked, and add the new button to the render method.

I then tested this out.

The button functioned correctly, however, the positioning of the button proved to be an issue. The button would interfere with the maze on the medium and harder difficulties.

I spoke with the stakeholders about remedying this. They collectively agreed that the best idea was to put the button on the top-left corner, instead of beneath the maze, to avoid any problems with the button conflicting with the main game.

The stakeholders all agreed that this worked much better, since the maze would not interfere with the button.

[redacted] noticed that the 'exit' sound effect wouldn't get played – the game would be exited before it had a chance to play this sound effect. I tried to remedy this by 'sleeping' the system (using time.sleep(2.5)) while the sound plays, but this left the button 'inverted', and the screen would freeze. The stakeholders collectively decided that this sound effect wasn't necessary, and it was removed from the game entirely. I updated assets.py, commenting out the code.

The second iteration is now complete. I have remedied all the initial problems that my stakeholders had with the first iteration, and added their features.

<u>The third iteration</u>

After the completion of the previous iteration, I met with my stakeholders again, and asked them if they had any other suggestions/problems with the program. This time, instead of asking about specific areas of the program, I asked about the program in general.

[redacted]: “The game doesn’t really have any difficulties between easy, medium and hard. Also, on the controls select screen, it’s very unclear as to which settings have been currently chosen. If you could add a textbox that tells what controls are currently in use, that would be great.”

[redacted]: “I don’t really see any big problems with the game right now, I think its fine.”

[redacted]: “The game has an issue that it becomes boring very quickly. Once you beat the three maze difficulties, there really isn’t much to do at all.”

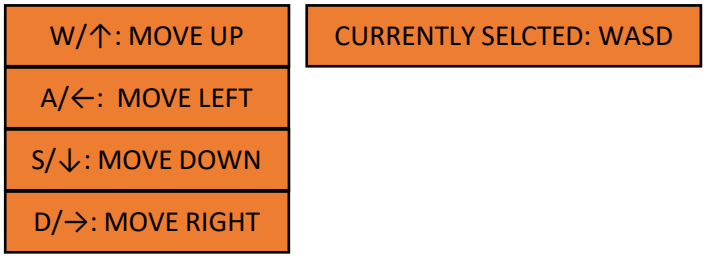
[redacted]: “Maybe you could include a small textbox on the controls menu explaining how the controls work. Other than that, there isn’t much to change.”

[redacted]: “The game does stagnate really fast – if you could add some more game modes or something, it’ll increase its replayability value.”

Looking at the points raised by [redacted], [redacted] and [redacted], I asked my stakeholders how I would make a new game mode. [redacted] suggested that I include a version which allows the user to customise their difficulty level, by choosing the maze resolution and timer duration. The other stakeholders agreed with this ‘custom’ mode, and hence, we collectively decided it would be implemented.

They also agreed with [redacted]’s other idea of a textbox telling the user what difficulty is current in selection, and [redacted]’s explanation of controls.

I started by designing solutions for the controls screen.



I proposed these textbox designs for the users. The ‘CURRENTLY SELECTED’ textbox changes according to the user selection for controls.

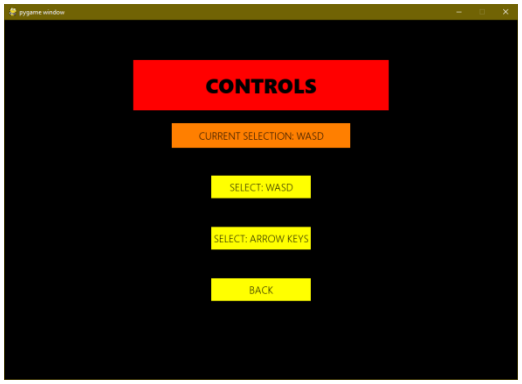
First, I added a value to the shared dictionary, called ‘current_controls’. By default, this was set to “WASD”. This value simply represents the current control configuration as a string. This was needed to add to the text on the textbox showing the current button selection.

For the ‘CURRENTLY SELECTED’ textbox, I instantiated a textbox in the setup method, and shifted all the other buttons down. I had to make sure the textbox was wider than the buttons, because it contains more letters than them. I also made sure this was updated in the share state when the user selects a different controls scheme.

```
206     def setup(self):
207         self.running = True
208         self.title = Textbox((screen_x/2)-250, screen_y/9, red, "CONTROLS", "title_textbox", 500, 100, font_2)
209         # The following button takes 'switch_to_wasd' as a method. It calls this method when it's clicked.
210         self.wasd = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: WASD", "instructions_button", 200, 50, font_1, self.switch_to_wasd)
211         # The following button takes 'switch_to_arrows' as a method. It calls this method when it's clicked.
212         self.arrows = Button((screen_x/2)-100, screen_y-300, yellow, "SELECT: ARROW KEYS", "instructions_button", 200, 50, font_1, self.switch_to_arrows)
213         self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)
214         self.current_selection = Textbox((screen_x/2)-175, screen_y-500, orange, "CURRENT SELECTION: " + str(self.share["current_controls"]), "current_selection", 350, 50, font_1)
```

I added this textbox into the render method, drawing it to the screen.

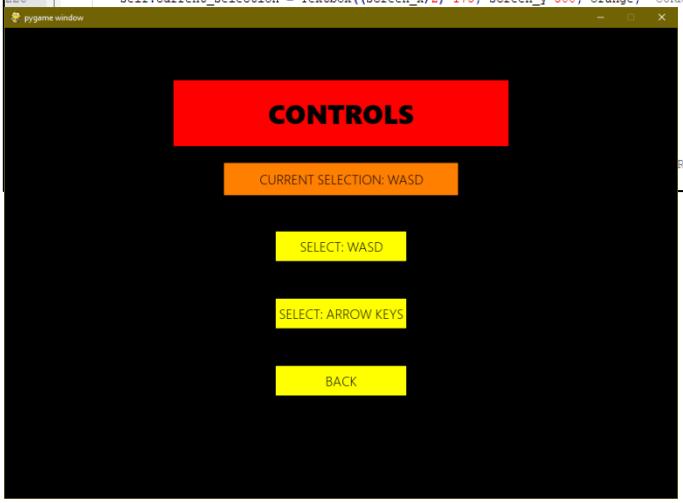
I tested this code.

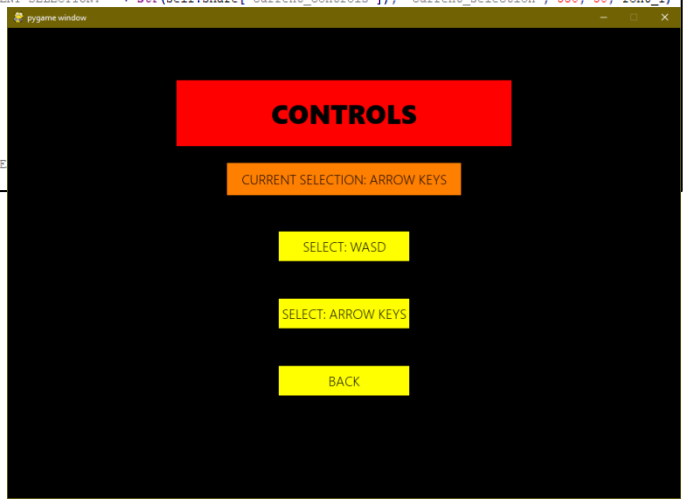


It worked fine in showing the initial selection, but would not update when I selected a new control scheme. I realised that this was because I had not used the update method – prior to this textbox, this was a completely static screen (other than the effects upon button presses).

I realised that, rather than using the update method and going through the long complicated process of passing a lot of variables and constants into this method, I could simply instantiate a new textbox over the old one, whenever the player clicks on a new control scheme.

```
219     def switch_to_wasd(self):
220         """
221         This modifies the value referenced by the key 'keys' in the share dictionary.
222         This changes the controls to "WASD" to be used in the game.
223         """
224         self.share.update({"keys": [K_w, K_s, K_d, K_a]})
225         self.share.update({"current_controls": "WASD"})
226         self.current_selection = Textbox((screen_x/2)-175, screen_y-500, orange, "CURRENT SELECTION: " + str(self.share["current_controls"]), "current_selection", 350, 50, font_1)
```





Every time the player clicks a button, a new textbox appears over the old one, indicating their selection. If they were to leave and return to the state, it would still show the correct selection, because the 'setup' method still uses the current selection.

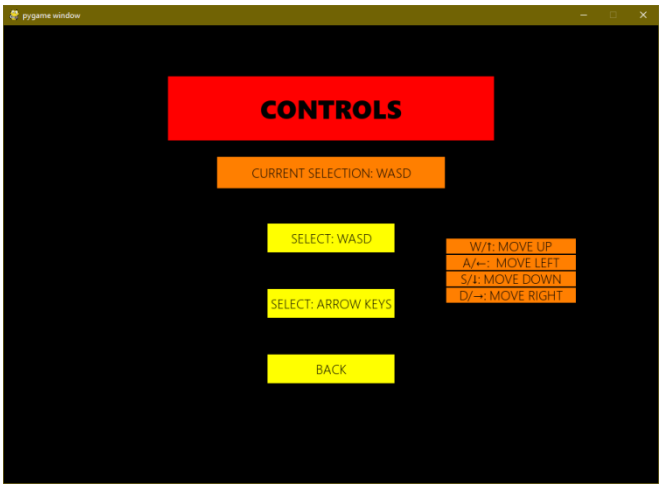
Finally, to make the code more efficient, I figured that it was wise to delete the old instantiation when a new one is created. I did this using 'del self.current_selection' in the control switching methods.

```
219     def switch_to_wasd(self):
220         """
221         This modifies the value referenced by the key 'keys' in the share dictionary.
222         This changes the controls to "WASD" to be used in the game.
223         """
224         self.share.update({"keys": [K_w, K_s, K_d, K_a]})
225         self.share.update({"current_controls": "WASD"})
226         del self.current_selection
227         self.current_selection = Textbox((screen_x/2)-175, screen_y-500, orange, "CURRENT SELECTION: " + str(self.share["current_controls"]), "current_selection", 350, 50, font_1)
228         menu_forward_sound.play()
229
230     def switch_to_arrows(self):
231         """
232         This modifies the value referenced by the key 'keys' in the share dictionary.
233         This changes the controls to the arrow keys to be used in the game.
234         """
235         self.share.update({"keys": [K_UP, K_DOWN, K_RIGHT, K_LEFT]})
236         self.share.update({"current_controls": "ARROW KEYS"})
237         del self.current_selection
238         self.current_selection = Textbox((screen_x/2)-175, screen_y-500, orange, "CURRENT SELECTION: " + str(self.share["current_controls"]), "current_selection", 350, 50, font_1)
239         menu_forward_sound.play()
```

I checked that the system still worked fine. I chose to delete old objects because it was more efficient, and would prevent issues with too many objects being drawn to the screen.

I then created some textboxes to explain how the controls work. This was simple, I just instantiated more textboxes in the 'setup' method.

```
206     def setup(self):
207         self.running = True
208         self.title = Textbox((screen_x/2)-250, screen_y/9, red, "CONTROLS", "title_textbox", 500, 100, font_2)
209         # The following button takes 'switch_to_wasd' as a method. It calls this method when it's clicked.
210         self.wasd = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: WASD", "instructions_button", 200, 50, font_1, self.swit
211         # The following button takes 'switch_to_arrows' as a method. It calls this method when it's clicked.
212         self.arrows = Button((screen_x/2)-100, screen_y-300, yellow, "SELECT: ARROW KEYS", "instructions_button", 200, 50, font_1, 
213         self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)
214         self.current_selection = Textbox((screen_x/2)-175, screen_y-500, orange, "CURRENT SELECTION: " + str(self.share["current_co
215         self.instructions_1 = Textbox(screen_x-325, screen_y-375, orange, "W/↑: MOVE UP", "current_selection", 200, 25, font_1)
216         self.instructions_2 = Textbox(screen_x-325, screen_y-350, orange, "A/←:  MOVE LEFT", "current_selection", 200, 25, font_1)
217         self.instructions_3 = Textbox(screen_x-325, screen_y-325, orange, "S/↓: MOVE DOWN", "current_selection", 200, 25, font_1)
218         self.instructions_4 = Textbox(screen_x-325, screen_y-300, orange, "D/→: MOVE RIGHT", "current_selection", 200, 25, font_1)
```



The resultant screen is shown on the left.

I asked my stakeholders how they felt about it, and they were generally pleased. They felt that it was concise and not "in the way", while still delivering all the required information.

Next, I moved onto implementing the 'custom' mode, where the user can choose the resolution of the maze.

First thing I did was to check the maximum feasible resolution of the maze (the dimensions where the maze would not interfere with the UI elements).

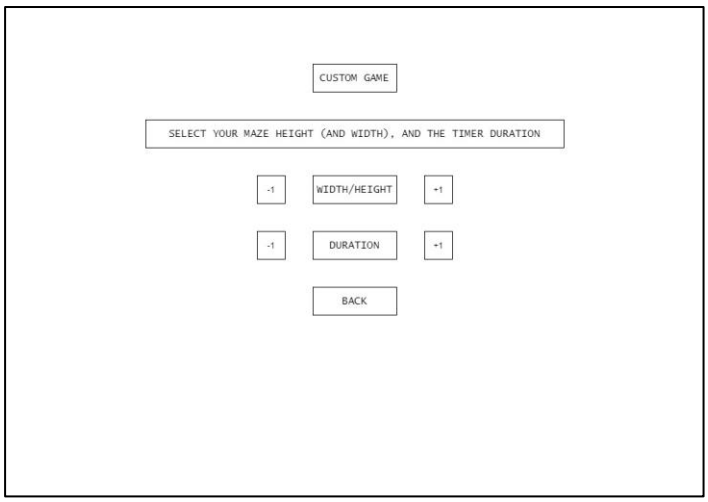
52 by 52 was the largest possible resolution. As a result, when implementing the 'custom' mode, the largest possible resolution I could allow is a 52 by 52 maze; any larger, and parts of the maze would be obscured by the UI elements.

My next step was to create a brand new state: 'Custom_Menu', inheriting the 'States' superclass and initialising it via the constructor. I needed this new state to allow the user to customise their settings for the custom game.

```
348 class Custom_Menu(States):
349     def __init__(self):
350         States.__init__(self) # Initialises via the constructor of the superclass.
351 
352     def setup(self):
353         pass
354 
355     def cleanup(self):
356         pass
357 
358     def handle_events(self, event):
359         pass
360 
361     def update(self):
362         pass
363 
364     def render(self, screen):
365         pass
```

I started off with the classic class template as shown above. I also added this state to the 'state_dict', under the name 'custom_menu'.

I then met with the stakeholders to come up with suitable designs for the screen of this new state.



Eventually, we came up with the design shown on the left.

The 'CUSTOM GAME' box is a title, and the rest of the boxes are textboxes, apart from the '+1', '-1' and 'BACK' boxes. These are buttons, with the '+1' and '-1' boxes being buttons that adjust the duration or size of the maze.

I did not give the user the option to independently modify the width and height of the maze. This was done to avoid the user attempting to create mazes that were non-square – the algorithm is only designed to work with mazes that are square (have the same width and height).

I also will enforce restrictions on the values a user can select for duration and size.

- Width/height can only be between 1 and 52, inclusive.
- Duration can only be between 1 and 15, inclusive.

This is done to prevent the program from breaking when encountering values out of range.

We chose to use buttons instead of an input field to prevent this issue – with buttons, I can enforce strict values, while with an input field it's harder to do.

I started by instantiating a whole set of buttons and textboxes in the setup method (after setting 'self.running' to True):

```
    def setup(self):
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "CUSTOM GAME", "custom_title", 500, 100, font_2)
        self.instructions = Textbox((screen_x/2)-450, screen_y - 500, orange,
                                    "SELECT YOUR MAZE WIDTH/HEIGHT AND TIMER DURATION. ONLY SQUARE MAZES ARE SUPPORTED.",
                                    "instructions_textbox", 900, 50, font_1)
        self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange, "WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)
        self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange, "DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)
        self.back = Button((screen_x/2)-100, screen_y-100, yellow, "BACK", "back_button", 200, 50, font_1, self.switch_main_menu)
        self.size_increase = Button((screen_x/2)+110, screen_y-400, yellow, "+1", "back_button", 50, 50, font_1, self.increase_size)
        self.size_decrease = Button((screen_x/2)-160, screen_y-400, yellow, "-1", "back_button", 50, 50, font_1, self.decrease_size)
        self.duration_increase = Button((screen_x/2)+110, screen_y-300, yellow, "+1", "back_button", 50, 50, font_1, self.increase_duration)
        self.duration_decrease = Button((screen_x/2)-160, screen_y-300, yellow, "-1", "back_button", 50, 50, font_1, self.decrease_duration)
        self.play = Button((screen_x/2)-100, screen_y-200, yellow, "PLAY", "play_button", 200, 50, font_1, self.switch_game)
```

I then created methods to be associated with each button to be clicked.

I started with the adjustment buttons for size of the maze and duration of the timer.

```
    def increase_size(self):
        if (self.share["x_cells"] == 52) or (self.share["y_cells"] == 52):
            self.share.update({"x_cells": 1})
            self.share.update({"y_cells": 1})
            del self.size_textbox
            self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange, "WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)
        else:
            self.share["x_cells"] = self.share["x_cells"] + 1
            self.share["y_cells"] = self.share["y_cells"] + 1
            del self.size_textbox
            self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange, "WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)

    def decrease_size(self):
        if (self.share["x_cells"] == 1) or (self.share["y_cells"] == 1):
            self.share.update({"x_cells": 52})
            self.share.update({"y_cells": 52})
            del self.size_textbox
            self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange, "WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)
        else:
            self.share["x_cells"] = self.share["x_cells"] - 1
            self.share["y_cells"] = self.share["y_cells"] - 1
            del self.size_textbox
            self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange, "WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)

    def increase_duration(self):
        if (self.share["timer"] == 15):
            self.share.update({"timer": 1})
            del self.duration_textbox
            self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange, "DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)
        else:
            self.share["timer"] = self.share["timer"] + 1
            del self.duration_textbox
            self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange, "DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)

    def decrease_duration(self):
        if (self.share["timer"] == 1):
            self.share.update({"timer": 15})
            del self.duration_textbox
            self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange, "DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)
        else:
            self.share["timer"] = self.share["timer"] - 1
            del self.duration_textbox
            self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange, "DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)
```

These methods all look very complicated, but in reality, they all do the same thing.

- First, they check if the user is about to increase/decrease the value beyond/beneath the limit. (For size, the upper limit is 52, the lower limit is 1; for duration, the upper limit is 15, the lower limit is 1). If they are, it simply loops the number round to the lower limit, to prevent values that are negative and/or out of range.
- If not, it increases/decreases the value as by 1 (as appropriate).
- Regardless of this, the textbox is deleted and instantiated again. This is a similar technique I used previously in this iteration, to make the ‘current controls selection’ textbox update whenever the user chooses a new control scheme.
- All four methods do the above, except all according to their needs, operating on their variables and only increasing or decreasing according to their purpose.
- These methods were each passed to their respective buttons, and called every time the button was pressed.

```
    def cleanup(self):
        del self.title, self.instructions, self.size_textbox, self.duration_textbox, self.back, self.size_increase, self.size_decrease, self.duration_increase, self.duration_decrease

    def handle_events(self, event):
        self.back.check_clicked(event)
        self.size_increase.check_clicked(event)
        self.size_decrease.check_clicked(event)
        self.duration_increase.check_clicked(event)
        self.duration_decrease.check_clicked(event)
        self.play.check_clicked(event)
```

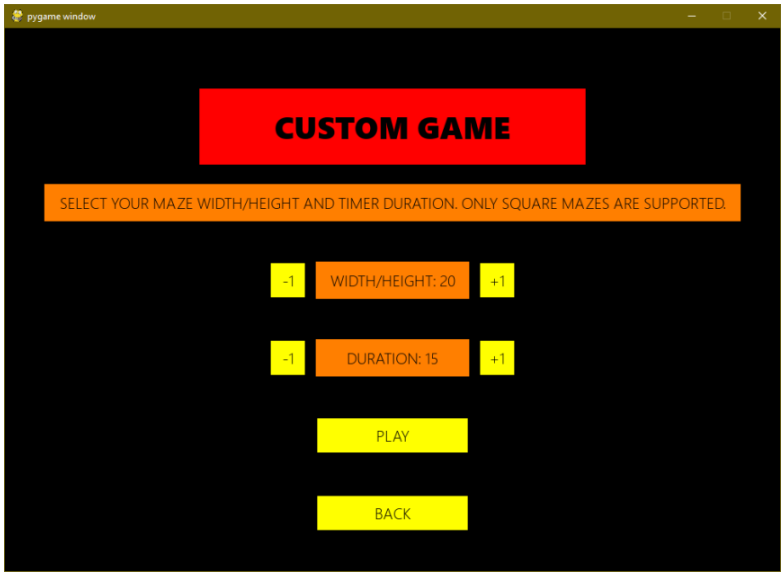
The above are the 'cleanup' and 'handle_events' methods. They operate as expected, deleting all objects (when the state is exited) and checking for the clicking of any buttons.

```
434     def switch_game(self):
435         self.share.update({"current_difficulty": "CUSTOM"})
436         self.next = "game"
437         menu_forward_sound.play()
438         self.running = False
439
440     def switch_main_menu(self):
441         self.next = "main_menu"
442         menu_back_sound.play()
443         self.running = False
444
445     def update(self):
446         pass
447
448     def render(self, screen):
449         screen.fill(black)
450         self.title.draw(screen)
451         self.instructions.draw(screen)
452         self.size_textbox.draw(screen)
453         self.duration_textbox.draw(screen)
454         self.back.draw(screen)
455         self.size_increase.draw(screen)
456         self.size_decrease.draw(screen)
457         self.duration_increase.draw(screen)
458         self.duration_decrease.draw(screen)
459         self.play.draw(screen)
```

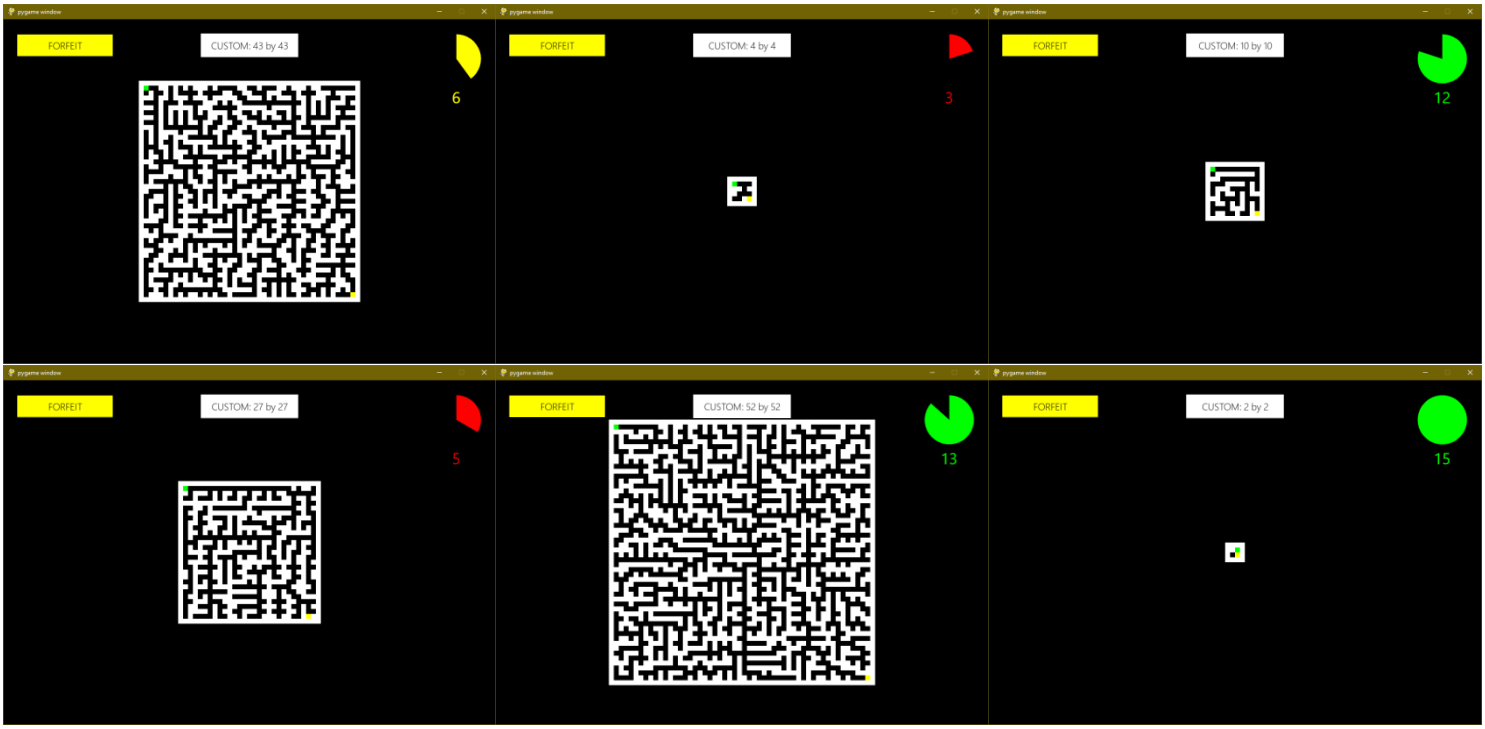
The methods shown on the left simply do the following:

- 'switch_game' switches to the game state. It also sets 'current_difficulty' (in the shared dictionary) to custom, so that this can be included in the title of the game state.
- 'switch_main_menu' switches to the main menu.
- 'update' does nothing here.
- 'render' simply fills the screen with black, and draws all the UI elements to the screen.

I tested out this code, navigating to the custom menu.



- I was able to adjust all the variables, and this would be shown in the orange textboxes.
- When I reached the upper and lower limits for the variables, it would simply 'flip' around to the other limit.
- If the 'WIDTH/HEIGHT' variable was at 52, and I were to '+1', it would simply return to 1, and continue. The same would happen if I were to subtract 1 from 1, the value would become 52.
- This would also happen for duration, between 1 and 15 instead.
- The moment 'PLAY' was pressed, a maze of the desired size would be immediately generated. This is owed to the speed of the algorithm I created – it is remarkably fast for smaller mazes (less than 500 by 500), and since generations cannot exceed 52 by 52, it was near instantaneous.
- All generations were fully solvable. In the title of the game, the difficulty was marked as 'CUSTOM'.



Above are some custom generated games.

- All of these are solvable; the algorithm guarantees this.
- None of them interfere with/are obscured by the UI elements; the 52 by 52 resolution cap guarantees this.
- All of these are perfectly centred on the screen; the formula guarantees this.

I presented this product to the stakeholders, and they were all very pleased and impressed.

[redacted]: “Wow, this is really good! I like that you’ve restricted the options for size and duration – this’ll stop problems happening in the future.”

[redacted]: “I think this is great because it allows the player to make the game much more ‘tailored’ around them.”

[redacted]: “I like how you also kept the option to choose the timer duration, rather than just the maze size, it allows for a much larger range of customisation.”

[redacted]: “This mode will allow the players to have a lot of fun, exploring maze combinations as well as playing and completing the main game.”

[redacted]: “The fact that you’ve made sure your maze is solvable no matter what size is used is really useful here – well done.”

The custom game mode is now complete.

<u>Further propositions</u>

The stakeholders and I discussed adding the ‘custom’ difficulty to the main screen as a difficulty level with the number of wins tracked. After extensive discussion, we rejected this idea.

The reason behind this was because users could simply use the 1 by 1 game (a square where the goal and player spawn on the same location, meaning the player immediately wins) to ‘spam’ wins over and over again. This will also cause problems with the text leaving the textbox, particularly as the number of wins pushes into three digits. While this isn’t a problem with the other difficulties – they’re difficult enough that a user is highly unlikely to win games in three digits in one playing session – this is a big problem in the custom game mode, because players can win about 40-50 custom games in the space of one minute, simply by replaying the 1 by 1 maze over and over again.

Furthermore, adding a textbox tracking the number of wins on the custom mode would not distinguish between how hard the maze was – a 52 by 52 maze win and a 1 by 1 maze win would appear as the same thing, despite the former being drastically more difficult than the latter.

A theoretical solution to this would be to have 52 different textboxes for all the difficulties… but this is ridiculous, the screen doesn’t have enough space for 52 more textboxes, not including all the UI elements that are currently on the main menu. Furthermore, this doesn’t take into account all the different timer durations, in order to create textboxes for every single possible difficulty combination, I would need 780 different textboxes, along with all the UI elements currently on screen. This just isn’t feasible, and as a result of all of this, the stakeholders and I decided that the number of ‘custom wins’ will not be tracked on the main menu.

I asked my stakeholders once again, if there was anything they were interested in me adding to the game.

[redacted] proposed the idea of a pause button for the main game. We considered this as a group, and rejected this idea. Our reasoning was that a pause button in a game with a timer would give an easy way to cheat – one could simply solve the maze while the game was paused.

Even if the paused game would remove the maze from the screen, one could screenshot and solve the maze while the game is paused, before unpausing. Furthermore, implementing a pause button could cause problems with the game state, since the game state is ran according to the timer. Allowing the user to manipulate game time could cause serious problems with the timer, which would break the game. As a result, we collectively rejected this idea.

Once again, I asked my stakeholders if there was anything wrong with the current system, or if there was anything they wished me to fix. They collectively agreed that the game was complete; there was nothing else to add/implement or modify.

3.3.2 – Testing to inform development

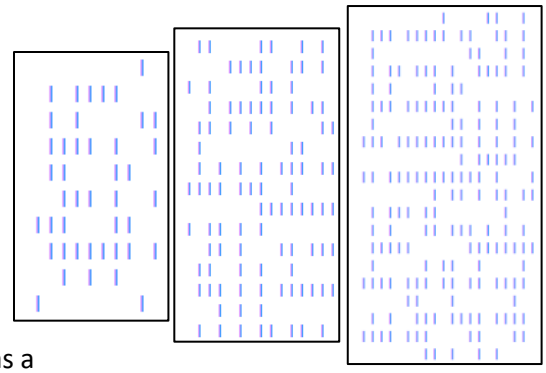
As aforementioned, I needed to carry out tests on the product during development.

3.3.2a – Testing at each stage

Test plan one (white box testing) takes effect here.

Checking that the algorithm always generates solvable mazes

The first of the white box tests is checking that the maze algorithm is always solvable, no matter what size is used.

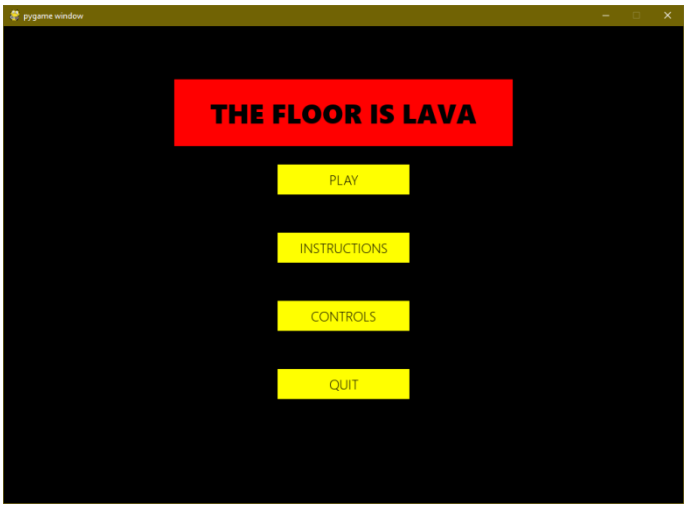


Shown in the three images to the right are some sample maze generations, in a form similar to ASCII art. Just as earlier, a single vertical line '|' represents a wall, while a blank space represents a path upon which the player can traverse.

All of the mazes shown on the right are solvable, as a result of the algorithm. This means that any single space can be reached by taking a path from any other space – the spaces (as a collective) form a tree.

Because the algorithm operates identically, regardless of size or resolution, this means that we can conclude that mazes generated will always be solvable.

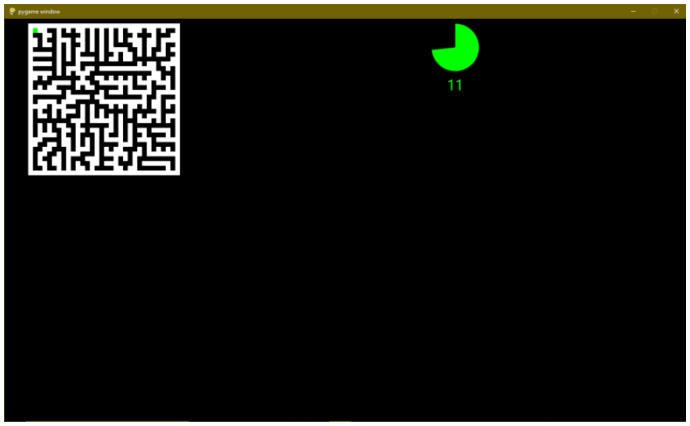
Checking that the screen window is at a resolution of 1000 by 700




```
# screen setup
screen_x = 1000
screen_y = 700
```

The above code specifies the resolution of the screen as 1000×700. The window produced from this, as a result, is this resolution, with a 10:7 aspect ratio. This cannot be changed by the player, and hence, the program passes this test.

Checking that the timer counts down correctly, and resets whenever the player moves



This test was conducted very early in development. The screenshot on the left depicts a timer, which started at 15, counting down while the player remains stationary.

Upon moving, the timer would reset. This would be done via the timer's method to reset it's duration to the original duration, when it moves. I also checked for varying durations, e.g. 5 seconds (on hard mode), and found that, as expected, the timer would reset to 5 seconds (the original duration used here) rather than 15.

As a result, the timer is working as expected.

Checking that buttons transition between states as needed

I tested every single button in the game during development, and recorded these results in the table below. They are sorted by the order of each iteration.

STATE:	BUTTON:	EFFECT:	SOUND EFFECT PLAYED (IF APPLICABLE):	WORKING?
ITERATION 1				
Main Menu	"PLAY"	Navigates to difficulty selection menu when clicked	Menu forward sound	Y
Main Menu	"INSTRUCTIONS"	Navigates to the instructions state when clicked	Menu forward sound	Y
Main Menu	"CONTROLS"	Navigates to the controls menu state when clicked	Menu forward sound	Y
Main Menu	"QUIT"	Quits the game when clicked	None	Y
Instructions	"BACK"	Navigates back to the main menu when clicked	Menu back sound	Y
Controls	"SELECT: WASD"	Sets the current control scheme to "WASD" when clicked; changes the textbox of this	Menu forward sound	Y

		state to read "CURRENT SELECTION: WASD"		
Controls	"SELECT: ARROW KEYS"	Sets the current control scheme to the arrow keys when clicked; changes the textbox of this state to read "CURRENT SELECTION: ARROW KEYS"	Menu forward sound	Y
Controls	"BACK"	Navigates back to the main menu when clicked	Menu back sound	Y
Difficulty selection	"SELECT: EASY"	Changes 'x_cells' and 'y_cells' to 20, and 'timer' to 15 when clicked; generates a maze of this resolution; generates a timer of this duration; moves to the game state	Menu forward sound	Y
Difficulty selection	"SELECT: MEDIUM"	Changes 'x_cells' and 'y_cells' to 35, and 'timer' to 10 when clicked; generates a maze of this resolution; generates a timer of this duration; moves to the game state	Menu forward sound	Y
Difficulty selection	"SELECT: HARD"	Changes 'x_cells' and 'y_cells' to 50, and 'timer' to 5 when clicked; generates a maze of this resolution; generates a timer of this duration; moves to the game state	Menu forward sound	Y
Difficulty selection	"BACK"	Navigates back to the main menu when clicked	Menu back sound	Y
Game over	"RETRY"	Regenerates a maze of the same	Menu forward sound	Y

		difficulty level when clicked; moves to the game state		
Game over	"CHANGE DIFFICULTY"	Navigates to difficulty selection menu when clicked	Menu forward sound	Y
Game over	"RETURN TO MENU"	Navigates back to the main menu when clicked	Menu back sound	Y
You win	"RETRY"	Regenerates a maze of the same difficulty level when clicked; moves to the game state	Menu forward sound	Y
You win	"CHANGE DIFFICULTY"	Navigates to difficulty selection menu when clicked	Menu forward sound	Y
You win	"RETURN TO MENU"	Navigates back to the main menu when clicked	Menu back sound	Y
ITERATION 2				
Game	"FORFEIT"	Immediately ends the current game when clicked; navigates to the game over screen	Lose sound	Y
Game over	"QUIT"	Quits the game when clicked	None	Y
You win	"QUIT"	Quits the game when clicked	None	Y
ITERATION 3				
Difficulty selection	"SELECT: CUSTOM"	Navigates to the custom menu when clicked	Menu forward sound	Y
Custom menu	"+1" (Assigned to width/height)	Increases 'x_cells' and 'y_cells' by 1 when clicked; changes the associated textbox to read accordingly (if x_cells and y_cells are 52, this will change them to 1)	None	Y
Custom menu	"-1" (Assigned to width/height)	Decreases 'x_cells' and 'y_cells' by 1 when clicked; changes the	None	Y

		associated textbox to read accordingly (if x_cells and y_cells are 1, this will change them to 52)		
Custom menu	"+1" (Assigned to duration)	Increases 'timer' by 1 when clicked; changes the associated textbox to read accordingly (if timer is 15, this will change it to 1)	None	Y
Custom menu	"-1" (Assigned to duration)	Decreases 'timer' by 1 when clicked; changes the associated textbox to read accordingly (if timer is 1, this will change it to 15)	None	Y
Custom menu	"PLAY"	When clicked, generates a maze of this resolution; generates a timer of this duration; moves to the game state	Menu forward sound	Y
Custom menu	"BACK"	Navigates back to difficulty selection menu when clicked	Menu back sound	Y

Checking UI elements are drawn to the screen in the correct position, and in the correct colour

Once again, I checked every single UI element against their original designs, and recorded them all in the table below.

TYPE:	NAME:	STATE:	COLOUR:	PURPOSE	WORKING?
Main menu state					
Textbox	Main menu title	Main menu	Red	Textbox for the title of the game	Y
Textbox	Easy wins	Main menu	Orange	Textbox displaying the number of easy wins in the current session	Y
Textbox	Medium wins	Main menu	Orange	Textbox displaying the number of	Y

				medium wins in the current session	
Textbox	Hard wins	Main menu	Orange	Textbox displaying the number of hard wins in the current session	Y
Button	“PLAY”	Main menu	Yellow	Button to navigate user to difficulty state.	Y
Button	“INSTRUCTIONS”	Main menu	Yellow	Button to navigate user to instructions state.	Y
Button	“CONTROLS”	Main menu	Yellow	Button to navigate user to controls setup state.	Y
Button	“QUIT”	Main Menu	Yellow	Button to quit the game	Y
Instructions state					
Textbox	Instructions title	Instructions	Red	Textbox for the title of the instructions state	
Textbox	Instructions 1	Instructions	Orange	Provide the player with instructions on how to play the game	Y
Textbox	Instructions 2	Instructions	Orange	[See above]	Y
Textbox	Instructions 3	Instructions	Orange	[See above]	Y
Textbox	Instructions 4	Instructions	Orange	[See above]	Y
Textbox	Instructions 5	Instructions	Orange	[See above]	Y
Button	“BACK”	Instructions	Yellow	To navigate the user back out to the main menu	Y
Controls state					
Textbox	Controls title	Controls	Red	Textbox for the title of the controls state	Y
Textbox	Current selection	Controls	Orange	Shows the player their currently selected control scheme	Y
Textbox	Controls guide 1	Controls	Orange	Provide the	Y

				player with instructions on how the controls work	
Textbox	Controls guide 2	Controls	Orange	[See above]	Y
Textbox	Controls guide 3	Controls	Orange	[See above]	Y
Textbox	Controls guide 4	Controls	Orange	[See above]	Y
Button	"SELECT: WASD"	Controls	Yellow	Selects WASD and the control scheme	Y
Button	"SELECT: ARROW KEYS"	Controls	Yellow	Selects the arrow keys as the control scheme	Y
Button	"BACK"	Controls	Yellow	To navigate the user back out to the main menu	Y
Difficulty selection state					
Textbox	Difficulty selection title	Difficulty selection	Red	Textbox for the title of the difficulty selection state	Y
Button	"SELECT: EASY"	Difficulty selection	Yellow	Button to navigate the user to an easy game	Y
Button	"SELECT: MEDIUM"	Difficulty selection	Yellow	Button to navigate the user to a medium game	Y
Button	"SELECT: HARD"	Difficulty selection	Yellow	Button to navigate the user to a hard game	Y
Button	"SELECT: CUSTOM"	Difficulty selection	Yellow	Button to navigate the user to the custom menu	Y
Button	"BACK"	Difficulty selection	Yellow	To navigate the user back out to the main menu	Y
Custom menu state					
Textbox	Custom state title	Custom menu	Red	Textbox for the title of the custom menu state	y
Textbox	Custom menu information	Custom menu	Orange	Textbox that provides	Y

				information on how to use the custom menu	
Textbox	Width/height textbox	Custom menu	Orange	Textbox that tells the user the current value for width and height of the maze	Y
Textbox	Duration textbox	Custom menu	Orange	Textbox that tells the user the current value for the original timer duration	Y
Button	“+1” (Assigned to width/height)	Custom menu	Yellow	Button that increases width/height by 1	Y
Button	“-1” (Assigned to width/height)	Custom menu	Yellow	Button that decreases width/height by 1	Y
Button	“+1” (Assigned to duration)	Custom menu	Yellow	Button that increases timer duration by 1	Y
Button	“-1” (Assigned to duration)	Custom menu	Yellow	Button that decreases timer by duration by 1	Y
Button	“PLAY”	Custom menu	Yellow	Button to navigate the user to the game state (with the custom settings used)	Y
Button	“BACK”	Custom menu	Yellow	To navigate the user back out to the difficulty selection state	Y
Game state					
Textbox	Game state title	Game	White	Textbox that informs the player of the current difficulty configuration	Y
Button	“FORFEIT”	Game	Yellow	Button to allow the user to	Y

				forfeit the current game	
Game over state					
Textbox	Game over title	Game over	Red	Textbox for the title of the game over state	Y
Button	“RETRY”	Game over	Yellow	Button to navigate the user back to the game state, regenerating the maze	Y
Button	“CHANGE DIFFICULTY”	Game over	Yellow	Button to navigate the user to the main menu	Y
Button	“RETURN TO MENU”	Game over	Yellow	Button to navigate the user to the difficult selection menu	Y
Button	“QUIT”	Game over	Yellow	Button to quit the game	Y
You win state					
Textbox	You win title	You win	Red	Textbox for the title of the you win state	Y
Button	“RETRY”	You win	Yellow	Button to navigate the user back to the game state, regenerating the maze	Y
Button	“CHANGE DIFFICULTY”	You win	Yellow	Button to navigate the user to the difficulty selection menu	Y
Button	“RETURN TO MENU”	You win	Yellow	Button to navigate the user to the main menu	Y
Button	“QUIT”	You win	Yellow	Button to quit the game	Y

Checking that any lava cell kills the player upon contact

Finally, I conducted the final test – checking that the lava cells kill the player on contact.

Analysing my code, there is an list 'lava_cells', which holds every single lava cell instance. For all cells in this list, the 'check_touching' method is called, to check if the player is touching the any cells. If this is true, the player is killed via the 'die' method.

```
for lava in self.lava_cells: # For all the lava cells in the array...
    lava.check_touching(self.character, self.timer) # Check if the character is touching any of the cells.
```

I went about checking if this actually worked in the game.

The above shows me as the player (left) backtracking into the lava cells, which kills me (right). The game over screen was then loaded, after the death sound effect was played.

Since the lava has killed me, I know that this will work for all lava cells; I do not need to test every single cell to verify this. This is because of the 'for lava in lava_cells' loop, which checks ALL lava cells in the array, and the fact that all cells are instances of the same class… they all operate exactly identically. Hence, if one lava cell kills the character upon contact, all of them will. Because I have proven this true for one particular lava cell in the list, it works for all lava cells in the list, and since any lava cells created are added to this list, it works for all lava cells in the game.

As a result, I know that ANY lava cell will kill a player upon contact, as I have proven this above. This test has been passed.

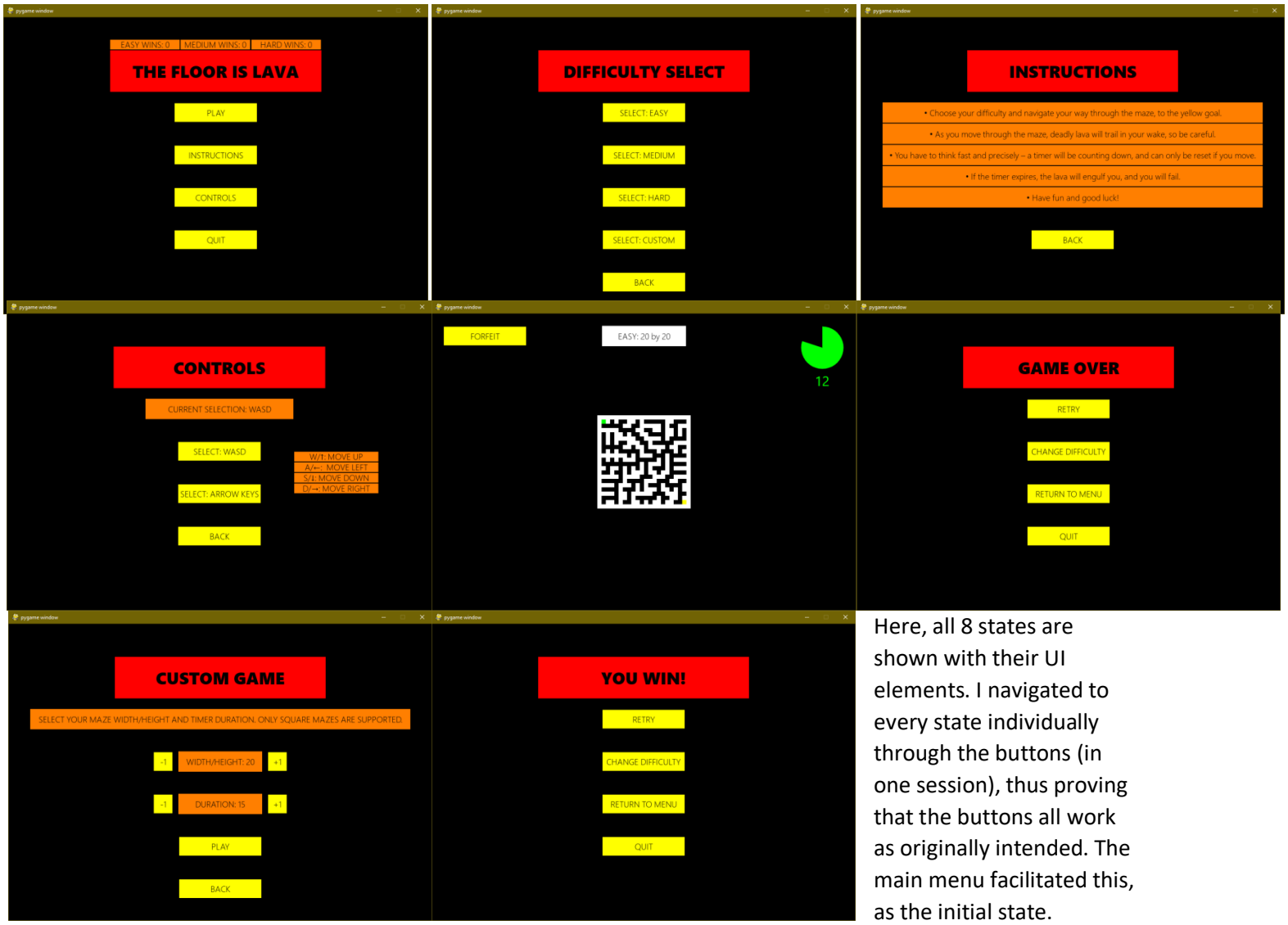
With the completion of test plan one, we can move onto test plan two (black box testing).

Test plan two (black box testing) takes effect here. This takes place after development has been completed.

Requirement ("The user must be able to…")	**Test/data used:**
Access different parts of the game via the main menu, as well as being able to return to the main menu.	Navigating to all the different parts of the system via the main menu, using the buttons
Exit the game from the main menu.	Pressing the 'QUIT/EXIT' button, and checking if I leave the game.
Select their control scheme (WASD or arrow keys)	Selecting each control scheme and checking they both work
Select difficulty	Checking that all difficulties generate different mazes as appropriate, screenshots as evidence
Have a maze generated based on their selected difficulty	[COVERED PREVIOUSLY]
Move using their selected control scheme	[COVERED PREVIOUSLY]

Be disallowed from traversing through walls of the maze	Attempting to traverse through walls, recording results
Have a timer that tracks how much time they have left before death	Testing that the timer actually kills the player, testing that it resets upon movement
Have lava cells generated in their wake	Checking that lava cells are always generated behind the player, as they move
Die (when either running into a lava cell or expending their allocated time)	[COVERED PREVIOUSLY]
Win (upon traversing the maze correctly, without dying)	Checking that the player always wins upon reaching the goal
Return to main menu or retry from the 'game over' screen	[COVERED PREVIOUSLY]
Return to main menu or proceed to next level from the winning screen	[COVERED PREVIOUSLY]

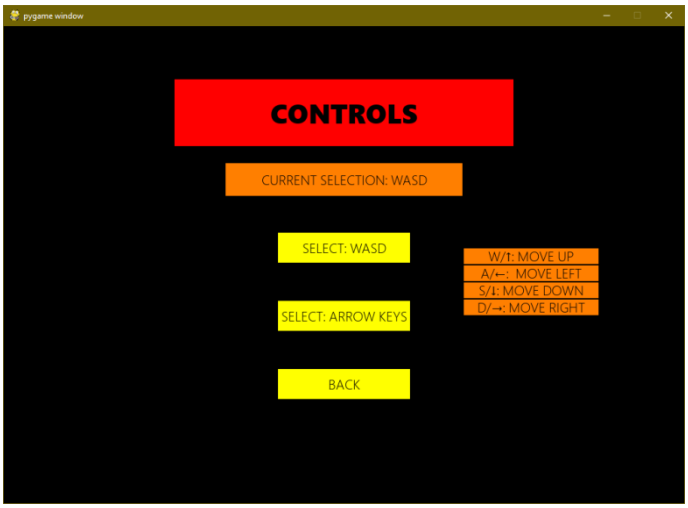
Testing states and the system as a whole

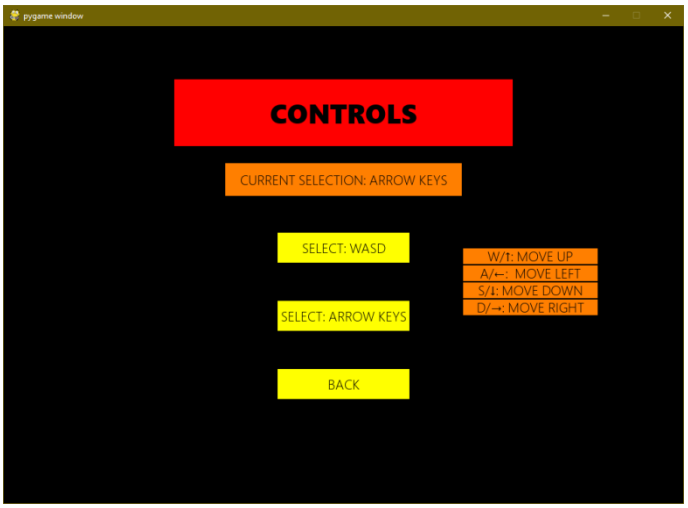


Here, all 8 states are shown with their UI elements. I navigated to every state individually through the buttons (in one session), thus proving that the buttons all work as originally intended. The main menu facilitated this, as the initial state.

- I was also able to exit the game upon clicking the 'QUIT' button. This was tested across three states: the main menu, the game over screen and the victory screen.

Testing controls settings

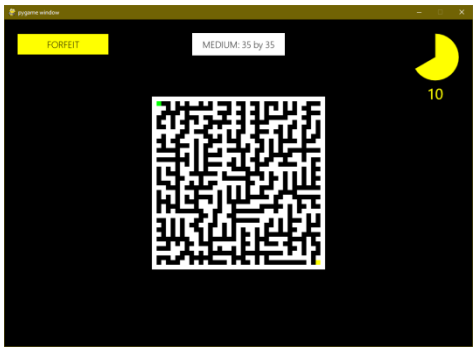


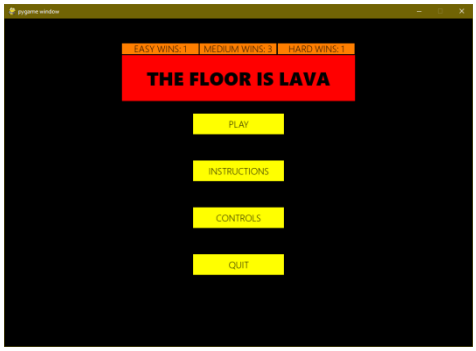


Upon entering the game, the controls are set to WASD. I played five games with these controls, they were all responsive and functioning, and didn't allow me to use the arrow keys to move the character.

I went back to the controls state and checked if my current selction was still set to WASD. I then switched to the arrow keys and played five more games of varying difficulty, and found that the arrow keys were just as responsive as the WASD controls, and they also did not accept inputs from the WASD controls.

If I tried to move the character with a controls scheme that wasn't selected, nothing would happen. This meant that the controls were all functioning as expected.





Testing difficulty settings

The three screenshots above show randomly generated mazes for all three standard difficulties (easy, medium, hard). Their timers are also shown in ther initial states, at 15, 10 and 5 seconds respectively. Hence, the mazes and timers are initialised correctly for all difficulties.

The textboxes on the main menu showing the number of wins is also shown on the left, shown to be correctly recording the number of wins I accumulated during that single playing session. This was wiped when I closed the program, and hence functions correctly.

Above shows a custom game being initialised (left) and generated (right).

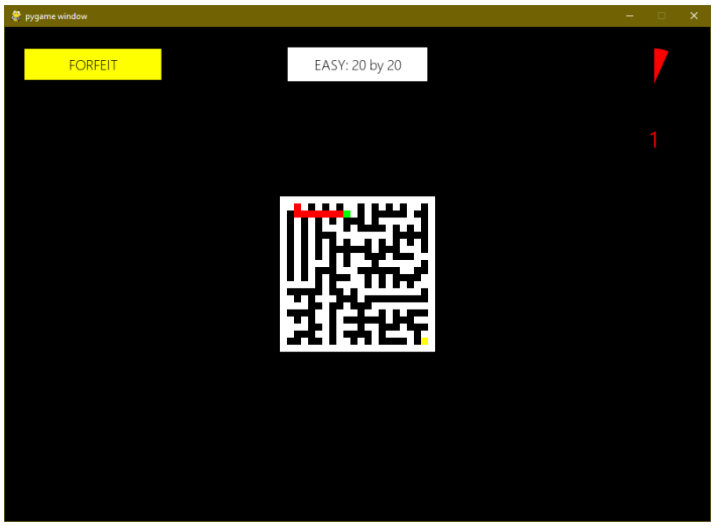
I chose the width and height of the maze to be 27 cells, and the duration on the timer to be 8 seconds. Upon clicking play, a (solvable) maze and timer were both generated according to this specification. Whenever I moved, the timer would reset to its initial duration (8 seconds), hence showing that the custom mode also worked perfectly.

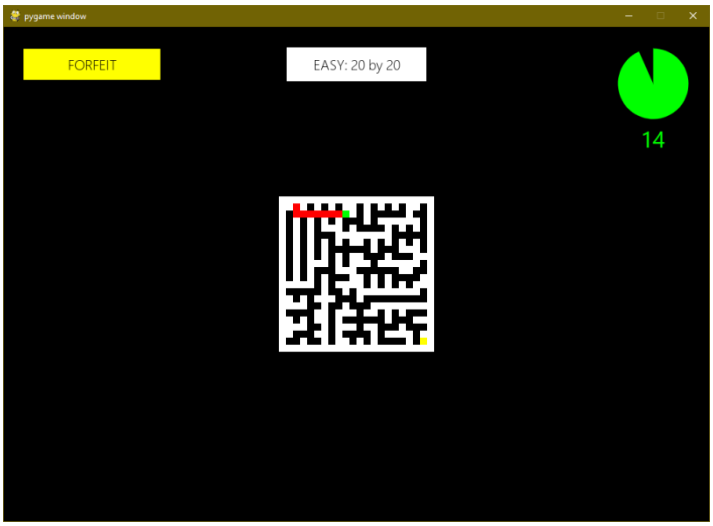
- Having a maze generated based on the selected difficulty and moving using the selected control scheme have already been covered.

<u>Testing walls</u>

I booted up a game, and tried repeatedly to make the character move through a wall. This didn't work; the character simply remained in the same place until I chose to move in another direction. Since, like the lava cells, all the walls are identical (all instances of the same class) and all belong to the same list, the list of walls, I can conclude that the character cannot traverse through ANY wall. Hence, this test is complete.

However, I noticed a rather huge issue. Even though the character wasn't moving when I attempted to traverse through a wall, the timer would reset whenever I attempted to move.





This was an issue because it meant the player could simply ‘spam’ timer resets by moving in the direction of a wall, while planning a route. This basically rendered the timer useless, and needs to be fixed. This will be remedied in the next section.

Testing the timer

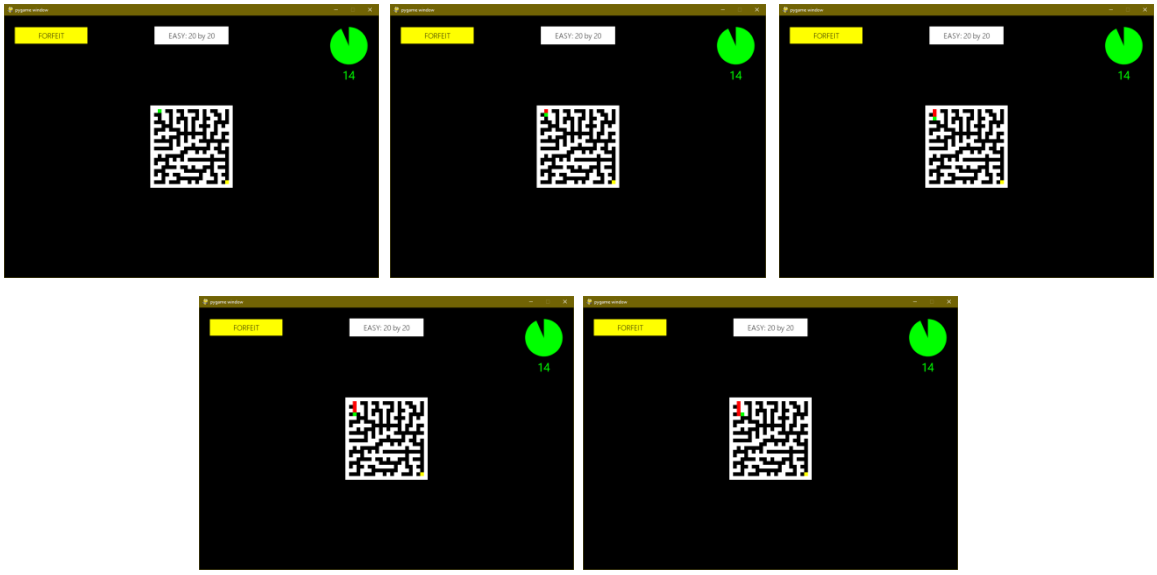
I carried out some basic tests to ensure that the timer behaved as specified.

- I checked that, for all values between 15 and 0 inclusive, that the timer would begin to countdown from that value, and return to the value upon expiring, after playing the sound and killing the player.
- I found that the timer never had any issues.
- In the screenshot above, I instantiated the timer with a duration of 11, and allowed it to count down and expire upon reaching zero, playing the sound and killing the player.
- Upon the reset, the timer started with a duration of 11, and the player was revived in their previous position.
- I repeated this test for all allowed values of the timer. Every time, the timer worked correctly.
- I also checked for values greater than 15, and found that the timer would simply remain at zero. While this isn’t an expected outcome, this doesn’t really matter, because the timer never will have to handle values greater than 15, and the user does not have the ability to modify the duration without accessing the source code and changing the values.
- As a result, I left this alone. I checked for float values, and found that the ‘int’ function worked correctly to eliminate any errors.

Testing lava cell generation

This was a simple test to conduct. I simply loaded up a game, and moved the character through the maze 4 times, checking that lava always precedes them.

The screenshots recorded below are for this purpose, showing the progress of the character through the maze, and how lava cells are continually spawned in their wake.

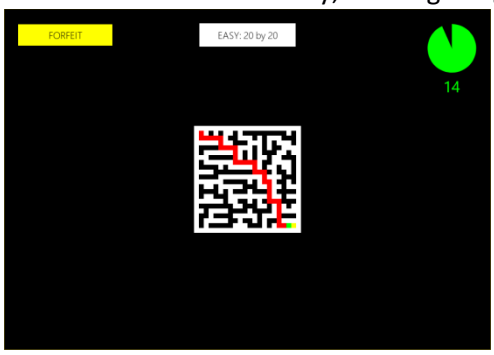


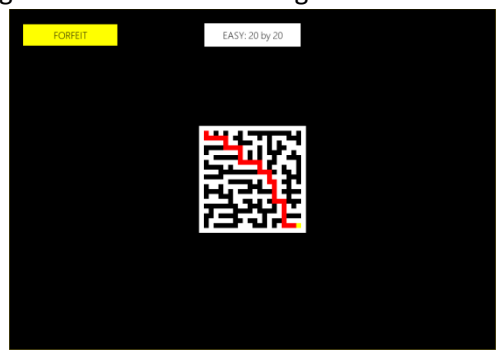
The lava cells always spawn behind the player, and hence, this part of the game is working.

Testing goal completion

I needed to test if the player could actually win the game.

I tested on easy, winning the game to ensure that the goal would work.

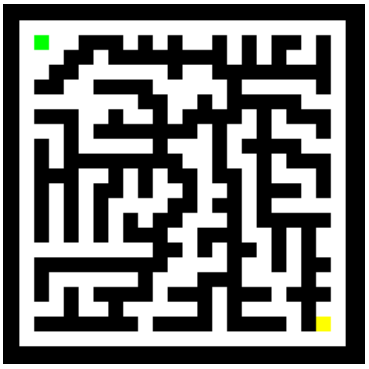




Since the goal object works on easy mode, it will work on all other difficulties and mazes. The goal object is always instantiated in the exact same way, derived from the exact same class every time. As a result, if it works for this difficulty setting and maze, it will work for all difficulties and mazes.

Hence, the goal works as expected. This test has been passed.

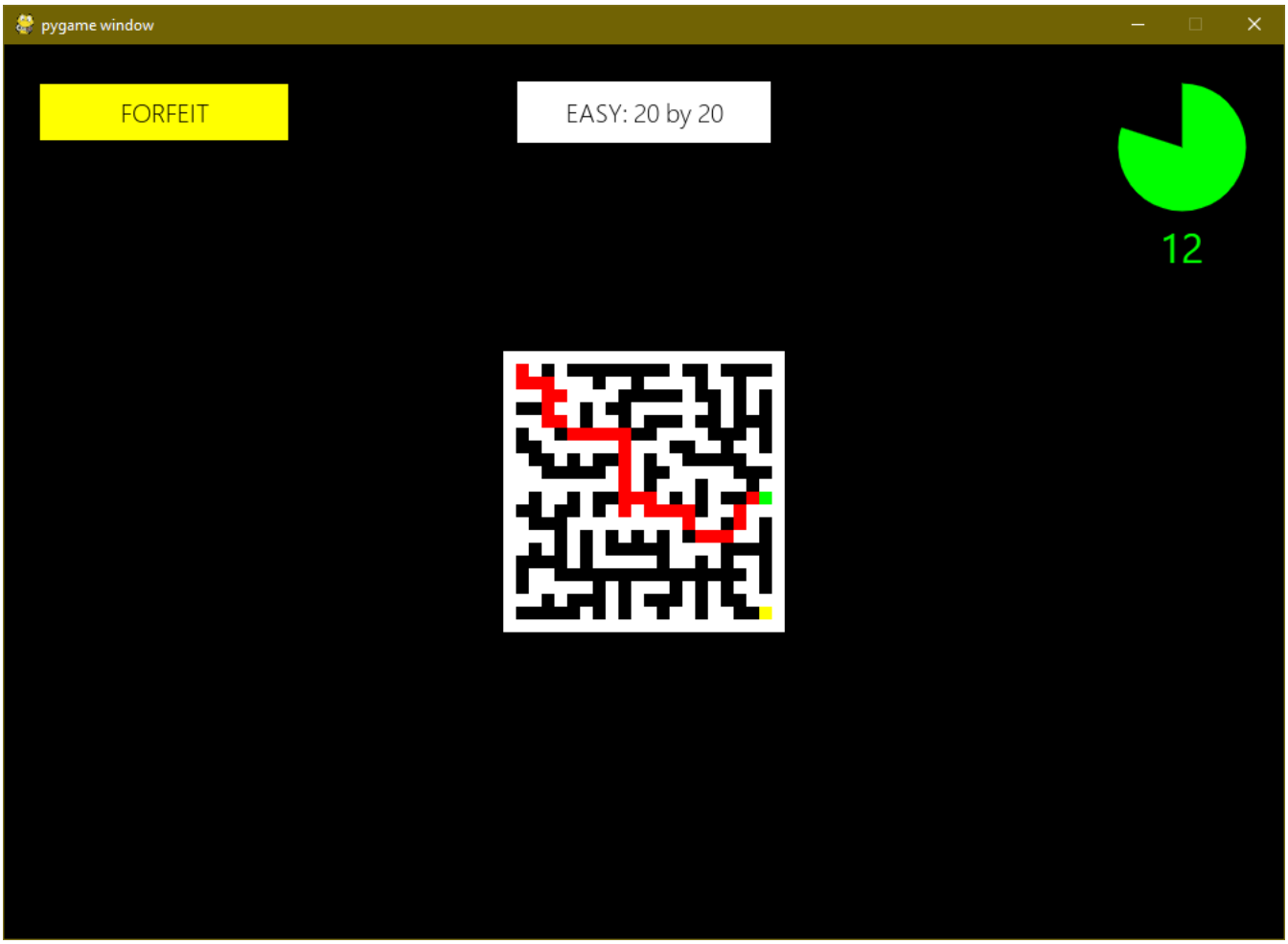
While conducting this test, I found an extremely problematic glitch with the game – the following maze was generated:



With this generation I was unable to move. This is a severe problem that needs to be solved, since the player can no longer move when this takes place.

A final bug I encountered was that I was able to ‘cut corners’ in the main game.

This is shown below.



As shown in the above screenshot, the player can 'skip' corners by holding two directional buttons at once (e.g. DOWN and RIGHT), moving them directly diagonally, instead of down and right, one at a time. By doing this, the player could skip spaces and escape corners that would otherwise kill them. This is a third huge issue which must be addressed – if not, the player can cheat the game and evade death despite warranting it.

3.3.2b – Remedial actions taken

With this, I need to go about solving the three major problems:

1. Forcing timer resets unfairly
2. Broken spawn locations
3. Cutting corners

Fixing forced timer resets

With this issue, the player can repeatedly spam a movement in the direction of a wall (and hence not move at all) while still letting the timer reset. This exploit allows the player to play the game with no consequence, as they can plan their route comfortably and win the game, regardless of how much time is on the timer.

As a result, this has to be fixed, because it gives the player an unfair advantage. This would need to be remedied from characters.py.

I navigated to the move method for the character class – this is where the timer resets were being called from and the movement was being managed.

The issue was that the 'timer.reset()' method was being called every time a valid movement keystroke (WASD or arrow keys) was detected by the system. It did not take into account whether the player had actually moved. The original code is displayed below.

```
40      def move(self, walls, timer):
41          if self.alive == True:
42              now = pygame.time.get_ticks()
43
44              if now - self.last_move > self.move_delay: # implements a slight movement delay
45                  self.last_move = now
46
47                  pressed_keys = pygame.key.get_pressed()
48                  if self.alive == True: # the character can only move if they're alive
49                      """
50                          The following first checks what key has been pressed, and then moves the character in that direction once.
51                          It then checks if the character has collided with a wall.
52                          If yes, they're moved back to where they were previously.
53                      """
54                      if pressed_keys[self.controls[0]]:
55                          self.y -= self.speed
56                          timer.reset()
57                          if self.wall_collision(walls):
58                              self.y += self.speed
59
60                      if pressed_keys[self.controls[1]]:
61                          self.y += self.speed
62                          timer.reset()
63                          if self.wall_collision(walls):
64                              self.y -= self.speed
65
66                      if pressed_keys[self.controls[2]]:
67                          self.x += self.speed
68                          timer.reset()
69                          if self.wall_collision(walls):
70                              self.x -= self.speed
71
72                      if pressed_keys[self.controls[3]]:
73                          self.x -= self.speed
74                          timer.reset()
75                          if self.wall_collision(walls):
76                              self.x += self.speed
```

I created a Boolean attribute for the class called 'self.moved', and set it to False. This attribute detects whether the player has ACTUALLY moved, rather than if a valid keystroke has been entered.

I then deleted all the 'timer.reset()' calls from each conditional statement, and added in 'self.moved = False' if the character has not moved despite a valid keystroke being detected (i.e. if they've ran into a wall). If the player hasn't ran into a wall, then self.moved is set to True; they can move unencumbered if there is no wall in their way.

```
55  if pressed_keys[self.controls[0]]:
56      self.y -= self.speed
57      if self.wall_collision(walls):
58          self.y += self.speed
59          self.moved = False
60      else:
61          self.moved = True
```

I did this for all four conditional statements (for all four of the directions), meaning self.moved could only be set to True if the player actually moved.

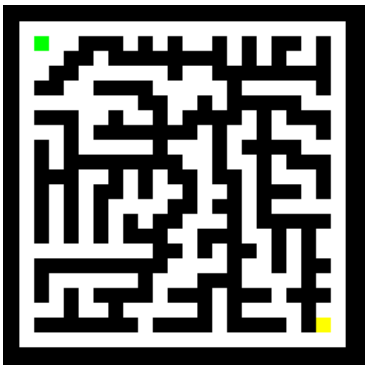
Then, I added a fifth conditional statement at the bottom of the previous four.

```
if self.moved == True:
    timer.reset()
    self.moved = False
```

This statement allowed the timer to reset ONLY if the player had moved, by calling the method to reset the timer every time 'self.moved' was set to True. It then sets it back to False, to ensure that the timer doesn't reset itself over and over again.

The code here is now complete. I tested it to check if it remedies the problem, by loading up multiple games and repeatedly trying to traverse in the direction of a wall, to see if the timer would reset. The timer did not change. However, when I moved into a space, the timer immediately reset, as expected. As a result, this change was successful – it remedied the problem while still allowing the timer to reset when it needs to.

Fixing broken spawn locations



My next step was to fix the issue with spawning locations.

In the image shown on the right, the player has spawned on top of walls, meaning they can't go anywhere and just have to die.

This is caused by the way the program finds the best possible spot to spawn the player, in the event that the immediate top right spot isn't available.

Essentially, a series of conditional statements check if the adjacent cells are available, going further and further out, but there are only a few of these statements, before the player is simply spawned in the default location.

```
### Finds the appropriate location for the character and goal
if self.maze_data[1][1] == '|':
    if self.maze_data[1][2] == '|':
        if self.maze_data[2][1] == '|':
            if self.maze_data[2][2] == '|':
                pass
            else:
                self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size
                self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size
        else:
            self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)
            self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size

    else:
        self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size
        self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)
else:
    self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)
    self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)
```

When searching beyond two spaces out from the original corner, there is a 'pass' – this means, if the program can't find an appropriate space in this range, it will simply drop the player at the top right corner of the maze, which causes the issue seen in the previous screenshot from the game.

I initially wanted to expand the range for which the system checks for available locations, but I cannot infinitely program 'if' statements. Instead, [redacted] proposed that, if an available location cannot be found, the program simply regenerates the maze. I preferred this solution, because it guarantees that the player will never be spawned in a location where they cannot move.

To do this, I replaced the 'pass' in the code above with code that simply restarts the state.

```
### Finds the appropriate location for the character and goal
if self.maze_data[1][1] == '|':
    if self.maze_data[1][2] == '|':
        if self.maze_data[2][1] == '|':
            if self.maze_data[2][2] == '|':
                self.next = "game"
                self.running = False
            else:
                self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size
                self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size
        else:
            self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)
            self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size

    else:
        self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size
        self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)
else:
    self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)
    self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)
```

The code 'self.next = "game"' and 'self.running = False' will simply restart the current state if the game is unable to find an available space, hence regenerating the maze. The chance of this being called in the first place is very, very low, and this will guarantee that the user will be given a maze where the character spawns in the correct position, giving a playable game.

In order to test if this actually took effect, I temporarily swapped out all the other 'else' statements with the 'self.next = "game"' and 'self.running = False' lines of code. These meant that, in essence, when I ran the program, I was only allowing mazes to that had the top left corner empty to be used. I tested this ten times, and I was only given the specified type of mazes. As a result, I know that these lines of code work and remedy this problem.

I also adapted the goal spawning location to behave in the same way – if no cell is available within a distance of 2 from the bottom right corner, restart the state and regenerate the maze. I did this to ensure that the user could never get a maze where the goal was unreachable. The chances of either of these happening (a broken character spawn or an unreachable goal) are very low, but in order to ensure my product is fully consumer ready, I must guarantee perfection and no bugs.

```
        ### Finds the appropriate location for the character and goal
        if self.maze_data[1][1] == '|':
            if self.maze_data[1][2] == '|':
                if self.maze_data[2][1] == '|':
                    if self.maze_data[2][2] == '|':
                        self.next = "game"
                        self.running = False
                    else:
                        self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size
                        self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size
                else:
                    self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)
                    self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size

            else:
                self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size
                self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)
        else:
            self.default_xpos = (((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)
            self.default_ypos = (((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)

        if self.maze_data[self.share["x_cells"]][self.share["x_cells"]] == '|':
            if self.maze_data[self.share["x_cells"] - 1][self.share["y_cells"]] == '|':
                if self.maze_data[self.share["x_cells"]][self.share["y_cells"] - 1] == '|':
                    if self.maze_data[self.share["x_cells"] - 1][self.share["y_cells"] - 1] == '|':
                        self.next = "game"
                        self.running = False
                    else:
                        self.goal_xpos = (((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2))) - self.cell_size)
                        self.goal_ypos = (((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2))) - self.cell_size)
                else:
                    self.goal_xpos = ((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2)))
                    self.goal_ypos = (((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2))) - self.cell_size)
            else:
                self.goal_xpos = (((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2))) - self.cell_size)
                self.goal_ypos = ((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2)))
        else:
            self.goal_xpos = ((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2))) # Originally 350
            self.goal_ypos = ((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2))) # Originally 310
```

This problem is now solved.

Fixing cutting corners

Finally, I need to fix the issue where the character can jump over corners (i.e. move diagonally), because it allows the player to unfairly escape/cheat death.

I took a similar approach to the one used to solve the first problem, by introducing Boolean attributes that are updated when the player moves.

```
29              self.moved_x = False
30              self.moved_y = False
```

These attributes are to check if the player has moved in the x-direction or the y-direction.

The issue stems from the player being able to SIMULTANEOUSLY move in the x and y directions, hence, moving diagonally. As a result, I figured that, if I can force the player to ONLY be able to move in the x-direction OR y-direction, rather than both, this issue would be resolved.

```
51                  if self.alive == True: # the character can only move if they're alive
52                      """
53                          The following first checks what key has been pressed, and then moves the character in that direction once.
54                          It then checks if the character has collided with a wall.
55                          If yes, they're moved back to where they were previously.
56                      """
57                      if pressed_keys[self.controls[0]] and (self.moved_x == False):
58                          self.y -= self.speed
59                          if self.wall_collision(walls):
60                              self.y += self.speed
61                              self.moved = False
62                          else:
63                              self.moved = True
64                              self.moved_y = True
65
66
67                      if pressed_keys[self.controls[1]] and (self.moved_x == False):
68                          self.y += self.speed
69                          if self.wall_collision(walls):
70                              self.y -= self.speed
71                              self.moved = False
72                          else:
73                              self.moved = True
74                              self.moved_y = True
75
76                      if pressed_keys[self.controls[2]] and (self.moved_y == False):
77                          self.x += self.speed
78                          if self.wall_collision(walls):
79                              self.x -= self.speed
80                              self.moved = False
81                          else:
82                              self.moved = True
83                              self.moved_x = True
84
85                      if pressed_keys[self.controls[3]] and (self.moved_y == False):
86                          self.x -= self.speed
87                          if self.wall_collision(walls):
88                              self.x += self.speed
89                              self.moved = False
90                          else:
91                              self.moved = True
92                              self.moved_x = True
```

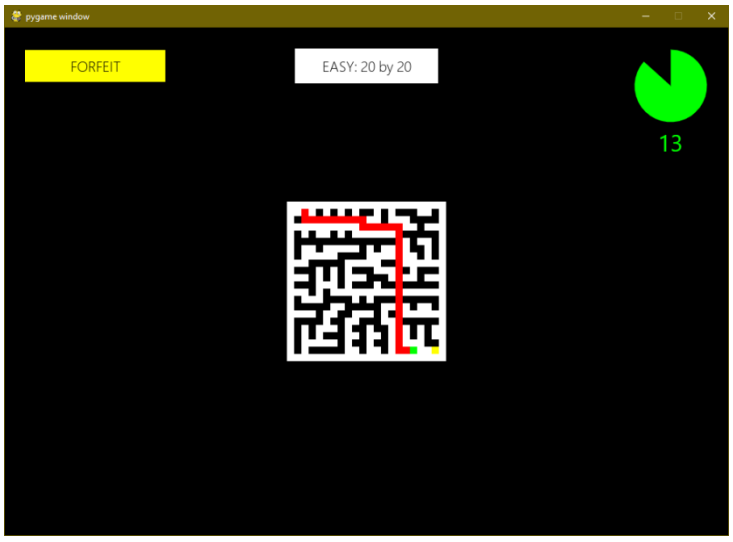
I then modified the conditional statements so that, as well as checking for whether or not a movement key was pressed, they also check that the player is not trying to move in the perpendicular direction. (For example, when moving up/down, the computer checks that the player isn't moving left/right, and vice versa.)

```
94                      if self.moved == True:
95                          timer.reset()
96                          self.moved = False
97
98                      if self.moved_x == True:
99                          self.moved_x = False
100
101                     if self.moved_y == True:
102                         self.moved_y = False
```

I then made sure to set 'self.moved_x' to True or 'self.moved_y' to True upon movement, dependent on the direction the character had moved.

I then simply included conditional statements to reset 'self.moved_x' and 'self.moved_y' when the player stopped moving, in order to allow them to move once again.

I tested this out, attempting to move diagonally in the maze. The screenshot below shows the result of my attempt.



As evident, instead of 'jumping' the corner and moving directly diagonally (like the game used to do), the player is forced to move to the corner and then to the adjacent space. Because of this, backtracking is now impossible, as there are no free spaces preceding the player where they can jump to, and the player cannot skip over corners without traversing them first.

I tested this out several times, and found that it was now impossible to

jump over a corner; the player could no longer move diagonally. As a result, the third and final issue behind this program has been solved.

Because there are no more issue flagged from testing to inform development, and there are no more iterations of the product to complete, the product is officially finished.

Section 4: Evaluation

3.4.1 – Testing to inform evaluation

3.4.1a – Robustness

Introduction to evaluation

This section will evaluate the success of the final product, against the stakeholders and the original success criteria/requirements. It will also look at potential future improvements.

Robustness testing

This program has extremely controlled inputs – i.e. the user cannot really make any text/data inputs, other than clicking buttons or moving the character. As result, robustness testing is fairly easy. The last two tests of test plan three will be conducted here, by the user.

What the user will do:	Justification:
Repeatedly and rapidly traversing through different parts of the system, checking if any UI elements get 'mixed up' or if any menus are generated at an inappropriate time.	This is a robustness test, to try to 'catch' errors, or problems with UI elements that may 'lock' the user in place.
Repeatedly switching between control schemes and playing the game, checking if at any time, the non-selected controls become useable (e.g. while WASD is selected, checking if the arrow keys remain inactive, and vice-versa).	This is a robustness test, to try to see if there are ever any errors with the controls that can render the game unplayable.

Checking UI elements (robustness)

The objective off this test is to try to "catch the system out" – i.e. find a point where a UI element has not been rendered, or find a point where the game breaks. This will be done by the stakeholders, they will randomly and rapidly move through all of the menus, looking to find if there is ever a point where a button or textbox is missing/not operational.

A table has been collated below, showing each state, their next possible states and whether or not it is possible to reach all of these states from the current state.

State:	Next state(s):	Working?
Main Menu	Difficulty selection; Instructions; Controls	Y
Difficulty selection	Main Menu; Custom game setup; Game	Y
Custom game setup	Game; Main Menu	Y

Instructions	Main Menu	Y
Controls	Main Menu	Y
Game	Game over; You win	Y
Game over	Game; Difficulty selection; Main Menu	Y
You win	Game; Difficulty selection; Main Menu	Y

All the states always functioned correctly, no matter how my stakeholders chose to travel between them. Whenever a state was loaded, the UI elements would always be loaded, with full functionality available, and no issues. As a result, this test is a success.

Checking control schemes (robustness)

The objective of this test is to check that, even after switching between control schemes and playing games that…

1) The textbox indicating the current controls setup (in the controls state) remains accurate at all times
2) ONLY the currently selected controls scheme can be used in the main game

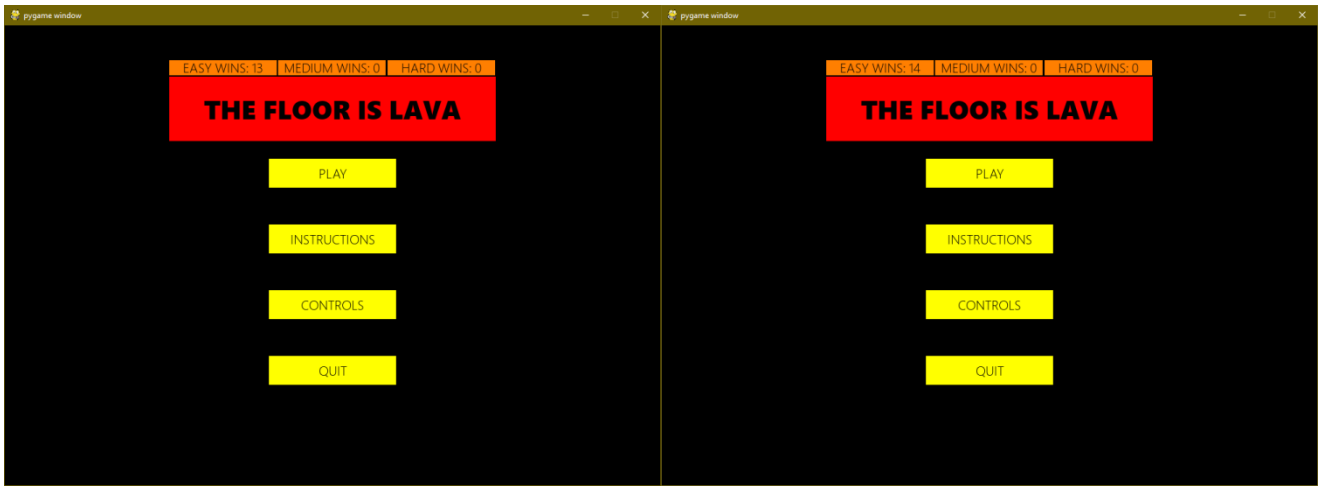
All the stakeholders conducted this test. They verified the first point by entering the controls state, randomly choosing a controls scheme, selecting it, and then leaving and re-entering the state, to check if the textbox displayed the correct information. They all found that it always did, no matter how many times they left and re-entered the menu. As a result, this first test is a success.

Next, they actually played the game using the different controls schemes. They found that, when any single control scheme was selected, the other was immediately ineffective, and did not have any influence over the game whatsoever. Even when rapidly switching between controls schemes, in order to try to get the system 'muddled up', the controls setup would remain consistent and true. As a result, this test is also a success.

Checking the textboxes showing the number of wins (robustness)

This test wasn't included in test plan two or three, but will be conducted anyway, because it is important to the textboxes showing the number of wins the player has achieved (from the main menu state).

In this, I will be checking whether or not these textboxes count up wins correctly, and how they deal with double-digit numbers – does all the text still fit in the box?



- From the above screenshots, it's clear that the textbox can hold double digit numbers without the text leaving the box.
- It's also evident that, with every win, the textbox increases the number by one.

As a result, it's evident that these textboxes are working correctly.

Because of these robustness tests, we can conclude that the controls scheme is always consistent and functioning, where only the chosen controls are active. This part of the system works properly.

3.4.1b – User feedback

User testing

Test plan three takes effect here.

During this stage, user testing will take place. As well as conducting the specified tests, they will play through the game freely, and be able to comment on improvements, or if they are dissatisfied with any parts of the game, as well as providing feedback on what they like.

What the user will do:	**Justification:**
Having stakeholders traverse through the maze, and ensure all lava cells generated are red, and are always fully visible, and that they kill the player upon contact.	This is to ensure that lava cells behave as expected and are visible, so that the player knows where they can and cannot traverse, as well as killing the player when touched.
Having stakeholders play the game, and ensure the main character is always fully visible and is green, as well as being able to move.	This is to ensure the player always knows their exact location, so that they can play with ease and fairness.
Checking that the same maze is never generated 3 times in a row hard difficulty, 4 times in a row for medium difficulty, and 5 times in a row for easy difficulty.	This is to ensure repeated mazes are not generated. Evidence will be shown using screenshots.
Checking that sound effects are played (as appropriate) every time a button is pressed, when the game starts, when the player wins and when the player loses.	This is to ensure the sound design always works, and that sound prompts can give users information on what is happening in the game.

Traversing the maze/testing lava cells

I asked each stakeholder to traverse through the maze, to ensure there are no bugs or issues with character movement (e.g. corner jumping, erratic movement, etc.). Their responses and feedback is recorded below.

[redacted]: “The controls were all very responsive and easy to use, and the lava would follow my character everywhere. Everything seems to be working correctly.”

[redacted]: “The movement is a lot more smoother compared to when I first observed it. The character is more responsive, and, unlike last time, keystrokes are never ignored. The lava killed me whenever I tried to go back, as expected.”

[redacted]: “Now that the issue with timer reset spamming and corner-jumping are gone, the movement feels really refined and precise, regardless of which control scheme used. This is a huge improvement over the original. Lava cells all worked as expected, killing me whenever I ran into them.”

[redacted]: “I occasionally overshot moves, but other than that, this is really smooth, while not being too fast or uncontrollable. Whenever I did make a wrong move, lava cells blocked me from backtracking, meaning I had to forfeit or get killed.”

[redacted]: “By tapping buttons when moving, you can ensure that you’re always where you want to be. Keystrokes are never missed, the game consistently detects when a key is being tapped. I like the movement a lot and am very satisfied with how this turned out.”

I asked the stakeholders if there were any improvements to be made to the movement or the lava cells, they all said no.

Checking character visibility

I asked every stakeholder if it were every ambiguous or unclear where the character was, if the character ever left the maze, or if at any point (besides victory or death) the character was no longer located on the maze.

[redacted]: “The colour green provides a great contrast against the black background, meaning it was always clear where the character was. I couldn’t go through any walls, and hence couldn’t leave the maze.”

[redacted]: “The character was always clear, visible and responsive. I never ran into any issues as to locating or using the character.”

[redacted]: “Because of the surrounding walls, it was not possible to leave the maze. I always knew where the character was, and always had control over it while playing.”

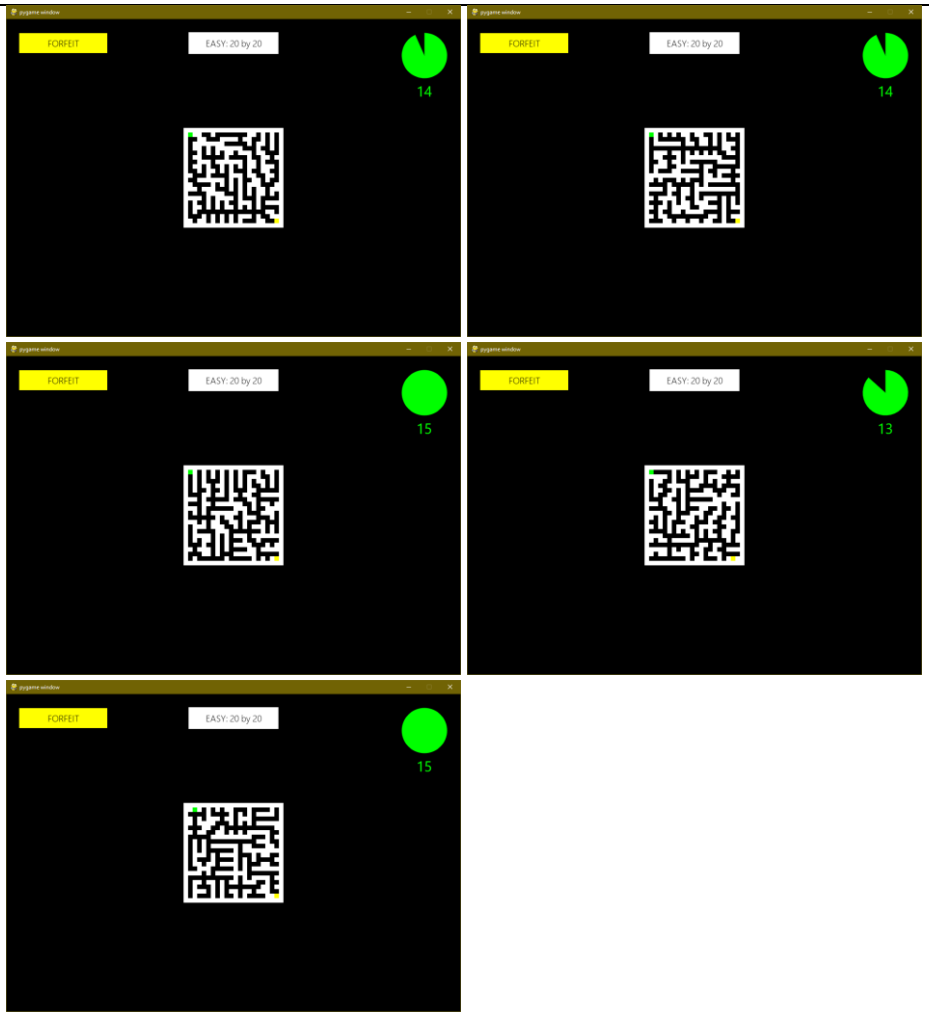
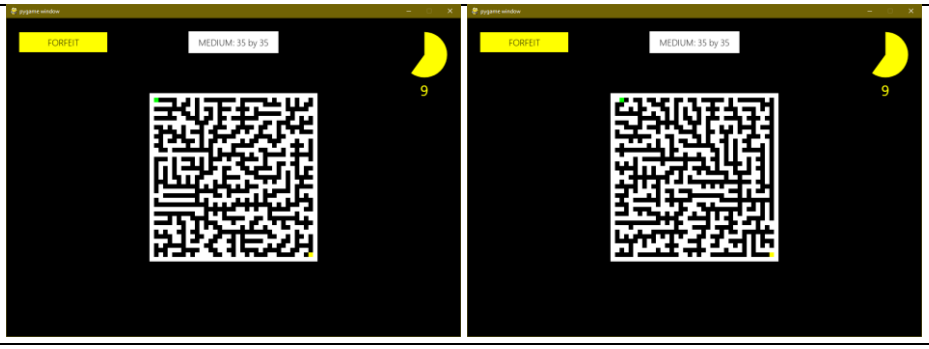
[redacted]: “I could always see my character, except from when the game was over. I never had any issues with character visibility.”

[redacted]: “I never had any issues with being able to see the character; the colour really pops out from its surroundings.”

As a result, we can verify that the character object is not an issue at all – it's responsive, visually distinguishable and follows the rules of the game.

Checking unique mazes

As a group, we conducted checks that, for the easy, medium and hard difficulties, we do not get the same maze generations for a number of repetitions. This evidence is compiled in the table below.

Difficulty level:	**Number of consecutive attempts:**	**Screenshots as evidence:**
Easy	5	
Medium	4	

		 
Hard	3	  

As evident from the table above, every single maze generated is unique, while still falling under the specification for resolution. As a result, we can conclude that the algorithm will always generate random mazes, citing the above table as evidence.

Checking sound effects

I also had the stakeholders check every sound effect in the game, to make sure they all work. The results are compiled in the table below.

Sound effect:	**Trigger:**	**State:**	**Working?**
Entry sound	Starting up the game	N/A	Y
Menu forward sound	Clicking the ‘PLAY’ button	Main Menu	Y
Menu forward sound	Clicking the ‘INSTRUCTIONS’ button	Main Menu	Y

Menu forward sound	Clicking the 'CONTROLS' button	Main Menu	Y
Menu forward sound	Clicking the 'SELECT: EASY' button	Difficulty selection	Y
Menu forward sound	Clicking the 'SELECT: MEDIUM' button	Difficulty selection	Y
Menu forward sound	Clicking the 'SELECT: HARD' button	Difficulty selection	Y
Menu forward sound	Clicking the 'SELECT: CUSTOM' button	Difficulty selection	Y
Menu back sound	Clicking the 'BACK' button	Difficulty selection	Y
Menu back sound	Clicking the 'BACK' button	Instructions	Y
Menu forward sound	Clicking the 'SELECT: WASD' button	Controls	Y
Menu forward sound	Clicking the 'SELECT: ARROW KEYS' button	Controls	Y
Menu back sound	Clicking the 'BACK' button	Controls	Y
Menu forward sound	Clicking the 'PLAY' button	Custom game setup	Y
Menu back sound	Clicking the 'BACK' button	Custom game setup	Y
Clock ticking sound	Every second that passes while the timer is active	Game	Y
Losing sound	Clicking the 'FORFEIT' button	Game	Y
Losing sound	When the clock expires	Game	Y
Losing sound	When the player runs into lava	Game	Y
Winning sound	When the player	Game	Y

	reaches the goal		
Menu forward sound	Clicking the 'RETRY' button	Game over	Y
Menu forward sound	Clicking the 'CHANGE DIFFICULTY' button	Game over	Y
Menu back sound	Clicking the 'RETURN TO MENU' button	Game over	Y
Menu forward sound	Clicking the 'RETRY' button	You win	Y
Menu forward sound	Clicking the 'CHANGE DIFFICULTY' button	You win	Y
Menu back sound	Clicking the 'RETURN TO MENU' button	You win	Y

All sounds in the game have been accounted for and are all functioning correctly, every time. As a result, the sound design in this game is fully complete and behaving as expected.

Playing the game

Now that the stakeholders have conducted and completed test plan three, I will ask them to play the game freely for a duration of 5 minutes each, and collate their feedback and opinions in the table below.

Stakeholder:	**Feedback:**
[redacted]	"I had a lot of fun playing this game. It turned out really well, and I literally couldn't find any problems with the main game, although it was very challenging at times. I really loved the custom mode, because it was really cool to see all the different mazes you could make – the possibilities seemed endless. I'm happy with this product."
[redacted]	"The ability to select your difficulty level came in really handy, because it let me play the game more casually, rather than trying super hard. It was really cool to see the number of wins that had been achieved. I really had a lot of fun with this game and I look forward to playing it more."
[redacted]	"This turned out to be quite remarkable. I couldn't find any bugs or problems/unexpected outcomes in the main game, it seems you covered and patched those really well. The inclusion of sound effects really added to the immersion of the game, although it would've been cool to see more sound effects in the main game, other than the winning and losing sounds."

[redacted]	"While the game can be quite difficult, I can play it a lot before getting bored. The custom mode was easily my favourite; it was fun messing around with different mazes. As I said previously, the game was hard, and sometimes I'd overshoot while moving, meaning that I had to forfeit. Overall, really solid."
[redacted]	"The unique mazes made every game different from the last, and with the ability to choose your own difficulty, the game felt really customisable and unique for everyone. I think one improvement that would've been cool to see is the ability to customise the character colour, but in terms of gameplay, this is complete."

Based on all of their feedback, it is pretty evident that the stakeholders are all very happy with the final product, and they all collectively agree on this. Since the program has completed test plan three, we can move to the next stage.

3.4.2 – Success of solution

3.4.2a – Evaluation of the solution

Test plan four takes effect here.

Here I will discuss the success of the final product, in comparison to the original requirements specification.

DESIGN CRITERIA:

Criteria:	How to evidence:	Test:	Succeeded?
The final product must be a 2D puzzle game.	Screenshot of main gameplay	N/A	Highly successful; shown throughout documentation.
The window for the game must be 1000×700.	Screenshot, and confirmation from stakeholders	Playing through the game and ensuring the window never changes in size.	Highly successful; proven during test plan one.
The game background must be black or white.	Screenshot of main gameplay	N/A	Highly successful
The primary threat (lava) in the game must be coloured red.	Screenshot of main gameplay	Having stakeholders traverse through the maze, and ensure all lava cells generated are red, and are always fully visible.	Highly successful
The main character/object of the game must be coloured green.	Screenshot of main gameplay	Having stakeholders play the game, and ensure the main character is always fully visible and is green.	Highly successful
The menu must feature	Screenshot of main menu	Checking that buttons	Highly successful

buttons for the player to click on to make decisions.		are all fully operational and carry out their job.	

DEVELOPMENT CRITERIA:

Criteria:	How to evidence:	Test:	Succeeded?
The game must be developed on Python, using the Pygame module.	Screenshot of code in Python, using the Pygame module	N/A	Highly successful
Development must feature prototypes, and improve these prototypes through iteration.	Screenshots and reviews of every iteration, with criticisms from the stakeholders	N/A	Highly successful; three iterations of prototypes used.
Development must take feedback from the selected stakeholders, and make changes to iterations based on this feedback.	Screenshots and reviews showing how each iteration has been improved based on previous feedback	N/A	Highly successful; feedback from stakeholders led to development of further iterations.

INPUT/OUTPUT CRITERIA:

Criteria:	How to evidence:	Test:	Succeeded?
The player must be able to choose between WASD and the arrow keys as their input controls.	Screenshots of control selection menu, stakeholder reviews ensuring controls work	Ensuring that the system only allows the player to move when the correct inputs are entered, for the selected control scheme, and checking that inputs only affect their designated direction.	Highly successful; proven successful during test plan two.
The game must output to a screen.	Picture of game outputting to screen	N/A	Highly successful
The game must run in windowed form.	Screenshot of the game window	Checking that full-screen cannot be toggled.	Highly successful
The game must feature some audio, in the form of basic sound effects.	Stakeholder review confirming the presence of audio for some events	Checking audio effects only playing at designated point, with the correct sound effects.	Highly successful; proven successful during test plan three.

GAME CRITERIA:

Criteria:	How to evidence:	Test:	
The game must feature a title screen.	Screenshot of title screen	Checking all elements of the title screen	Highly successful;

		function correctly.	shown throughout document.
The game must feature an introductory screen.	Screenshot of introductory screen	Checking all elements of the introductory screen function correctly.	Highly successful; shown throughout document.
The player must be able to choose their difficulty level.	Screenshot of difficulty selection screen, with stakeholder review to ensure that it works	Checking that every difficulty selection generates the appropriate game (maze and timer).	Highly successful; proven successful during test plans two and three.
The introductory screen must allow the user to view a tutorial screen with basic instructions.	Screenshot of tutorial screen with instructions, clarifying with stakeholders that instructions are understandable	Checking that instructions are displayed on the introductory screen, and are fully readable.	Highly successful; shown throughout document.
The introductory screen must allow the user to exit the application completely.	Screenshot of 'EXIT' button and stakeholders confirming button's functionality	Checking that the exit button allows the user to fully exit and close the game immediately.	Highly successful; proven successful during test plan two.
The game must feature a main game screen, where the user can play.	Screenshot of game screen, stakeholders confirming that the game is playable	Checking that the game is fully playable; that all inputs are always registered.	Highly successful; shown throughout document.
The game must feature a game over/winning screen.	Screenshot of the game over screen; screenshot of the winning screen	Checking that the appropriate screen is shown as a result of the game (winning screen for winning; game over screen for losing).	Highly successful; proven successful during test plans two and three.
The algorithm must generate mazes that are always solvable.	Screenshots of all difficulty mazes to show they're all fully solvable	Checking that the chosen algorithm always generates fully solvable mazes.	Highly successful; proven successful during test plan one.
Any tiles/spaces/cells that the player traverses must become red.	Screenshots of a player as they traverse through the maze, and how lava is left in their wake	Checking that all traversed cells become lava cells.	Highly successful; shown throughout document.
The player must not be allowed to traverse lava	Stakeholder review confirming that they die	Checking that the player dies when	Highly successful;

without dying.	whenever they attempt to touch lava cells	they attempt to traverse the lava.	proven successful during test plan one.
The game must feature a timer to force players to make moves.	Stakeholder review confirming that they die if the timer reaches 0; confirming that the timer resets with every move	Checking that the correct timer is generated; checking that the timer counts down as appropriate, and starts at the correct time; checking that the player dies when the timer expires (for all difficulties); checking that the timer resets with every move; checking that the reset timer functions correctly and kills the player.	Highly successful; proven successful during test plan one.
Harder difficulty mazes must be a higher resolution (more detailed, with more cells) than their lower difficulty counterparts.	Screenshots of all maze difficulties [covered above], showing increased resolution for higher difficulty settings	Checking that appropriate mazes are generated; checking that mazes disallow players from traversing through their walls.	Highly successful; shown throughout document.
Harder difficulties must feature a shorter timer than their lower difficulty counterparts.	Screenshots and confirmation from stakeholders that higher difficulty mazes have a shorter timer	Checking that the correct timer is generated for all difficulties.	Highly successful; shown throughout document.

The final product managed to meet every single requirement set out by the original success criteria, all to a high standard. Because of this, there is no need to address any criteria that could've been improved/completed to a better standard. I asked the stakeholders if they agreed that all the success criteria had been met to a high standard.

[redacted]: "I really think that every single point has been met – there isn't much more you could've done, as far as the success criteria goes."

[redacted]: "You've done everything listed, and shown them. I think the success criteria have all been completed."

[redacted]: "I agree with [redacted]. Every single point on this list has been met to a really good standard, and there isn't much more to add at all."

[redacted]: “There isn’t anything you’ve missed, everything has been considered and included in the final product. The project is finished.”

[redacted]: “All success criteria have been shown to be highly successful.”

Since the stakeholders all agree that all success criteria have been met and evidenced to a high standard, the project can be evaluated as highly successful.

3.4.3 – Describe the final solution

3.4.3a – Usability features

Describing the final product

- The final product is a 2D puzzle videogame in which the player has to move through a maze, towards a goal, using four-directional keyboard controls.
- While moving through the maze, the player will be unable to backtrack – all of their previous locations will be covered by red lava.
- The player cannot remind idle for more than a given amount of time. A timer tracks how much of this time is left, and if the timer expires, the player dies (and loses). This timer resets every time the player moves.
- The player can choose their controls between WASD and the arrow keys.
- The player can also choose their difficulty between easy (a 20 by 20 maze, 15 second timer), medium (a 35 by 35 maze, 10 second timer) or hard (a 50 by 50 maze, 5 second timer).
- They also have the option to choose a custom game mode, where they select the size of the maze and the duration on the timer.
- All mazes are randomly generated by a maze generation algorithm, which always provides solvable mazes for the user.
- There is an instructions screen for the player to view instructions if unsure.
- The game contains sound effects, which are played at various points throughout the game.

Describing the usability features

In 3.2.2c, I identified a series of usability features. These are features implemented deliberately to make the system easier to use and more accessible for my user. They are listed in the table below.

Usability feature:	**Description:**	**Justification:**
The tutorial screen (instructions)	This is a screen that displays the instructions for the player on how to play the main game.	Not all users will understand what is meant to be done in the game, or how the controls may work; the inclusion of an instructions screen will allow them to play the game fairly.
Fonts used for text	The fonts ‘Segoe UI Black’ and ‘Segoe UI’ will be used	These are two fonts that are very clear and easy to read, and are

	for the title textboxes and standard textboxes/buttons respectively.	distinct from each other (Segoe UI Black is more bold and thick than its counterpart), while still looking similar enough to carry a similar theme.
Colour scheme	The background will be black, with buttons being yellow and (standard) textboxes being orange. The title textboxes will be red. The character item will be green; the maze will be white. The timer will feature colours that change, from green to yellow to red, depending on how much time is left.	All of these colours contrast sharply from the black background making them clearly visible for the user. The timer's adjusting colours bring about a sense of urgency, prompting the player to make their move before it expires, and making the game feel more intense. The clear, consistent colour theme for the buttons, textboxes and title textboxes means that the player can easily identify what the buttons are and click on them, as well as the other UI elements.
Details of textboxes and buttons	The textboxes (title and standard) and buttons are all uniquely detailed to be identifiable from one another. They all use different colours, and titles use different fonts to be further distinguishable.	This is done to make sure the user knows what can be clicked, and what cannot. I can't have the user thinking that, for example, a button is just a standard textbox, because this would mean they won't click on it, meaning they miss some important information. By making all of the UI elements unique from each other, this is less likely to happen.
Adjustable controls	The user has the ability to choose between WASD or the arrow keys for their controls settings (for the character in the main game).	This is done to allow the user to use controls they're comfortable with. As shown in the analysis section, these are the two most used and well-known control schemes for four-directional movement (when using the keyboard as input). In fact, these are pretty much the only two control schemes used for four-directional movement. As a

		result, the user is guaranteed to be familiar and comfortable using one of these two control schemes.
Adjustable difficulty	The user has the ability to choose between easy, medium or hard difficulty. Alternatively, they can choose their own specifications from which a maze generated.	By allowing the user to choose their own difficulty, I can let them play the game the way they want to play it, either more casually or more competitively. The custom mode takes this even further, making the game a lot more enjoyable for the user (rather than being frustrating from being too easy/difficult).

Below, I will show evidence that each of these usability features have been implemented into the final program.

Tutorial screen (instructions):

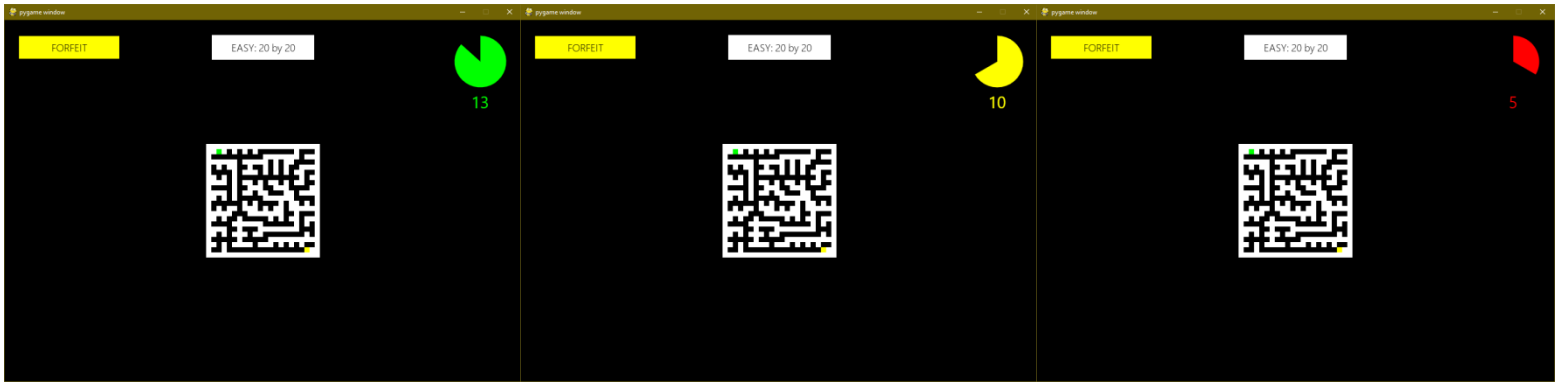
- The above, left screenshot shows the instructions screen, with textboxes which explain how the game should be played. This evidences the instructions screen.
- As well as this instructions screen, the top-right screenshot shows the controls state, which also contains a guide on how the controls move the character, on the right hand of the screen.
- Hence, the user is provided with a clear guide on how the game is to be played.

Fonts used for text:

- Looking at the previous screenshots, it's clear that the title textboxes and the buttons/standard textboxes use different fonts. The title textboxes use Segoe UI Black, while the rest of the UI elements use Segoe UI. The title is significantly bolder, identifying it as a title.

Colour scheme:

- Once again, looking at the previous screenshots, the background is black. The title textbox is red, standard textboxes are orange, and the buttons are all yellow. Because of this, they're all clearly identifiable.
- As shown in the screenshots below, the character is green, the goal is yellow, the maze is white, and the timer changes colour dependent on the remaining duration.



- Because of these colour choices, all elements of the screen are clearly identifiable, and their purpose can be understood by the user very easily.

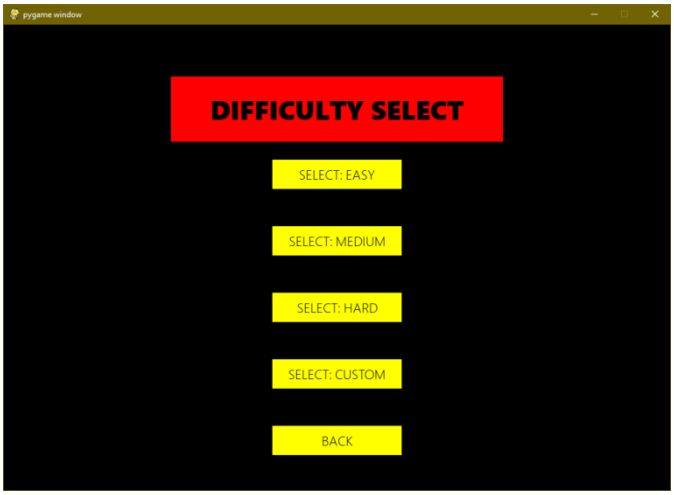
Details of textboxes and buttons:

- As evidenced previously, all buttons, textboxes and title textboxes are clearly identifiable, and maintain a consistent theme throughout the game, so that the user always knows what every UI element does.

Adjustable controls:

- As shown from the screenshots above, the user can choose their own controls, between WASD and the arrow keys. Hence, this usability feature has been successfully implemented.

Adjustable difficulty:



- As shown from the screenshot to the right, the player can choose their difficulty level, and can even customise the specifics of the game (size of the maze and duration of the timer). Hence, this usability feature has been successfully implemented.

Hence, all of the usability features from the design section have successfully been implemented. They are described and shown above, with justification.

<u>Evaluating the usability features</u>

Here, I will discuss the usability features with the stakeholders, and how successful they each were.

Usability feature	**Stakeholder feedback**	**How successful?**
The tutorial screen (instructions)	"This was very important for us, as it explained the entire game and how to play it, without leaving any ambiguity at all."	Highly successful (9/10)
Fonts used for text	"This was fairly important; the big font of the title made it very clear what each screen was showing, and made the system very easily to navigate."	Fairly successful (7/10)
Colour scheme	"The colour scheme was consistent and the timer changing colour really instilled a sense of urgency."	Very successful (8/10)
Details of textboxes and buttons	"The buttons and textboxes were clearly identifiable from each other, thanks to both their positioning and their physical	Successful (6/10)

	locations on the screen. It was never confusing as to which one was which."	
Adjustable controls	"Being able to choose controls meant that everyone felt familiar with the controls, and it was never confusing as to how the buttons work."	Highly successful (9/10)
Adjustable difficulty	"This was probably the most important part of the entire game. The ability to choose your own difficulty made the game much more interesting and playable for everyone, and being able to customise the difficulty settings meant that players could play their own way."	Extremely successful (10/10)

All of the usability features of the final product had a positive effect on the game experience, as evidenced by user feedback shown above. As a result, they were a huge success.

3.4.4 – Maintenance and development

3.4.4a – Maintainability of the solution

Maintaining the system in the future

I also must look at how I will maintain the system in the future.

Maintenance is where software is revised and improved/fixed over time, in order to ensure that it continues to function in the future.

The game can function in its entirety with or without a network connection – it is completely independent of whether or not the computer is connected to the internet, because it is a standalone, offline game. Since it is a complete product, there is no need to update or maintain the system, it does not use any data that needs updating or maintaining as time passes by.

Regardless, if I ever did need to make changes to the product, I have kept the source code and documentation, with annotations on the original code to help me when I revisit it. If I were to update/improve the game, I would have to rerelease the entire system, because it's not possible for me to remotely update it; nowhere in implementation did I include code that would facilitate this. Redistributing the software would be quite a task; I would likely have to make the product open to the public, to download from a website or a videogame distribution service, such as Steam or Origin.

In its current state, there is no need for routine maintenance of the system. I asked my stakeholders to verify this, and they agreed. Regardless, the code has been organised with annotations (as shown throughout implementation), and into smaller sub-programs, all named appropriately, so that, in the event that an update needs to be made, the programmer can easily understand what is going on.

3.4.4b – Further development

Future improvement

I asked the stakeholders what improvements could be made to the program in a potential future version.

[redacted]: “The addition of a local competitive mode, where players race to reach an objective, would be quite fun and interesting. Alternatively, an online play mode would be great!”

[redacted]: “I think that time should a be a bigger factor in the game. Maybe, you could include a stopwatch that records how much time it takes you to win the game, and (on the main menu) the best times for every game mode is displayed. From this, you could even add a specific game mode which gives the user times to beat, dependent on the size/difficulty of the maze.”

[redacted]: “I think it would be great if the spawn locations of the character and goal were randomised to be one of the four corners every time. This would add a lot more variety and challenge to the game, and force the player to explore new areas of the maze (i.e. the player would be forced to traverse through a different quadrant than usual).”

[redacted]: “There could be an option for a ‘Zen mode’, where the timer is disabled, and the user can simply traverse the maze in their own time. This would allow for more relaxed players to play at a slower, more casual pace”

[redacted]: “Allowing character customisation, or the ability to choose different colour schemes would be really cool. Also, powerups could really take the game further – you could add a powerup which freezes the timer, or a powerup that allows the player to traverse over lava cells temporarily. Adding powerups would give a lot more variety and excitement to the game.”

To sum everything up, the following were included as potential for future improvement:

- Local multiplayer
- Online multiplayer
- Timed modes
- Saving times as personal high scores
- Varying spawn locations of the goal and player in each corner/quadrant
- A zen mode without the timer enabled
- Character customisation
- Selectable colour schemes for the game theme
- Powerups providing different abilities in the main game

With this, the documentation is complete.

Appendix

The following contains all code from all files as raw text.

main.py

```
''' Imports all needed modules and libraries.'''

import time

import pygame

from pygame.locals import *

from maze_algorithm import *

from ui_elements import *

from characters import *

from assets import *

entry_sound.play()


"""

The states class.

This is simply a template, which every state shall inherit methods and attributes from.

The subclasses may also add their own methods and attributes according to their needs.

"""

class States:

    # The share dictionary holds information to be shared between states.

    share = {

        "keys": [K_w, K_s, K_d, K_a],

        "x_cells": 20,

        "y_cells": 20,

        "timer": 15,

        "easy_wins": 0,

        "medium_wins": 0,

        "hard_wins": 0,

        "current_difficulty": None,

        "current_controls": "WASD",
```

```
        }
    def __init__(self):
        self.running = True
        self.next = None
        self.quit = False

    def setup(self):
        pass

    def cleanup(self):
        pass

    def handle_events(self, event):
        pass

    def update(self):
        pass

    def render(self, screen):
        pass

class Main_Menu(States): # The state for the main menu.
    def __init(self):
        States.__init__(self) # Initialises the state using the States superclass
constructor.

    def setup(self):
        self.running = True
        """
        Instantiates all buttons and textboxes.
        """
```

```
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "THE FLOOR IS LAVA",
"title_textbox", 500, 100, font_2)

        self.play = Button((screen_x/2)-100, screen_y-500, yellow, "PLAY", "play_button",
200, 50, font_1, self.switch_play)

        self.instructions = Button((screen_x/2)-100, screen_y-400, yellow, "INSTRUCTIONS",
"instructions_button", 200, 50, font_1, self.switch_instructions)

        self.controls = Button((screen_x/2)-100, screen_y-300, yellow, "CONTROLS",
"controls_button", 200, 50, font_1, self.switch_controls)

        self.quit_button = Button((screen_x/2)-100, screen_y-200, yellow, "QUIT",
"quit_button", 200, 50, font_1, self.quit_game)

        self.easy_wins_textbox = Textbox((screen_x/2)-250, screen_y/9 - 25, orange, "EASY
WINS: " + str(self.share["easy_wins"]), "easy_wins", 500/3, 25, font_1)

        self.medium_wins_textbox = Textbox((screen_x/2)-(250/3), screen_y/9 - 25, orange,
"MEDIUM WINS: " + str(self.share["medium_wins"]), "medium_wins", 500/3, 25, font_1)

        self.hard_wins_textbox = Textbox((screen_x/2)+(250/3), screen_y/9 - 25, orange,
"HARD WINS: " + str(self.share["hard_wins"]), "hard_wins", 500/3, 25, font_1)




    def cleanup(self):

        """

        Deletes all buttons and textboxes.

        """

        del self.title, self.play, self.instructions, self.quit_button,
self.easy_wins_textbox, self.medium_wins_textbox, self.hard_wins_textbox



    def switch_play(self):

        """

        Directs to the difficulty select state.

        """

        self.next = "difficulty_select"

        menu_forward_sound.play()

        self.running = False



    def switch_instructions(self):

        """
```

```
        Directs to the instructions state.
        """
        self.next = "instructions_screen"
        menu_forward_sound.play()
        self.running = False

    def switch_controls(self):
        """
        Directs to the controls select state.
        """
        self.next = "controls_screen"
        menu_forward_sound.play()
        self.running = False

    def quit_game(self):
        """
        Exits the game.
        """
        exit_sound.play()
        self.quit = True

    def handle_events(self, event):
        """
        Checks if any buttons have been clicked.
        """
        self.play.check_clicked(event)
        self.instructions.check_clicked(event)
        self.controls.check_clicked(event)
        self.quit_button.check_clicked(event)

    def update(self):
```

```
        pass


    def render(self, screen):
        """
        Fills the screen with black and draws buttons and textboxes.
        """
        screen.fill(black)
        self.title.draw(screen)
        self.play.draw(screen)
        self.instructions.draw(screen)
        self.controls.draw(screen)
        self.quit_button.draw(screen)
        self.easy_wins_textbox.draw(screen)
        self.medium_wins_textbox.draw(screen)
        self.hard_wins_textbox.draw(screen)



class Instructions(States):
    def __init__(self):
        States.__init__(self)


    def setup(self):
        """
        Instantiates all buttons and textboxes.
        """
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "INSTRUCTIONS",
"title_textbox", 500, 100, font_2)
        """
        The following 5 objects are textboxes that contain the instructions.
        """
        self.instructions_1 = Textbox((screen_x/2)-450, screen_y - 500, orange,
```

```
                                    "• Choose your difficulty and navigate your way through
the maze, to the yellow goal.",
                                    "instructions_textbox", 900, 50, font_1)
        self.instructions_2 = Textbox((screen_x/2)-450, screen_y - 450, orange,
                                    "• As you move through the maze, deadly lava will trail
in your wake, so be careful.",
                                    "instructions_textbox", 900, 50, font_1)
        self.instructions_3 = Textbox((screen_x/2)-450, screen_y - 400, orange,
                                    "• You have to think fast and precisely – a timer will
be counting down, and can only be reset if you move.",
                                    "instructions_textbox", 900, 50, font_1)
        self.instructions_4 = Textbox((screen_x/2)-450, screen_y - 350, orange,
                                    "• If the timer expires, the lava will engulf you, and
you will fail.",
                                    "instructions_textbox", 900, 50, font_1)
        self.instructions_5 = Textbox((screen_x/2)-450, screen_y - 300, orange,
                                    "• Have fun and good luck!",
                                    "instructions_textbox", 900, 50, font_1)
        """
        The back button, to return the player to the main menu.
        """
        self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button",
200, 50, font_1, self.switch_main_menu)


    def cleanup(self):
        """
        Deletes all buttons and textboxes.
        """
        del self.title, self.instructions_1, self.instructions_2, self.instructions_3,
self.instructions_4, self.instructions_5, self.back


    def switch_main_menu(self):
        """
        Directs to the main menu state.
```

```
        """
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def handle_events(self, event):
        """
        Checks if the back button has been clicked.
        """
        self.back.check_clicked(event)

    def update(self):
        pass

    def render(self, screen):
        """
        Draws all UI elements to the screen.
        """
        screen.fill(black)
        self.title.draw(screen)
        self.instructions_1.draw(screen)
        self.instructions_2.draw(screen)
        self.instructions_3.draw(screen)
        self.instructions_4.draw(screen)
        self.instructions_5.draw(screen)
        self.back.draw(screen)


class Controls_Setup(States):
    def __init__(self):
        States.__init__(self)
```

```
    def setup(self):

        self.running = True

        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "CONTROLS",
"title_textbox", 500, 100, font_2)

        # The following button takes 'switch_to_wasd' as a method. It calls this method
when it's clicked.

        self.wasd = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: WASD",
"instructions_button", 200, 50, font_1, self.switch_to_wasd)

        # The following button takes 'switch_to_arrows' as a method. It calls this method
when it's clicked.

        self.arrows = Button((screen_x/2)-100, screen_y-300, yellow, "SELECT: ARROW KEYS",
"instructions_button", 200, 50, font_1, self.switch_to_arrows)

        self.back = Button((screen_x/2)-100, screen_y-200, yellow, "BACK", "back_button",
200, 50, font_1, self.switch_main_menu)

        self.current_selection = Textbox((screen_x/2)-175, screen_y-500, orange, "CURRENT
SELECTION: " + str(self.share["current_controls"]), "current_selection", 350, 50, font_1)

        self.instructions_1 = Textbox(screen_x-325, screen_y-375, orange, "W/↑: MOVE UP",
"current_selection", 200, 25, font_1)

        self.instructions_2 = Textbox(screen_x-325, screen_y-350, orange, "A/←:  MOVE
LEFT", "current_selection", 200, 25, font_1)

        self.instructions_3 = Textbox(screen_x-325, screen_y-325, orange, "S/↓: MOVE DOWN",
"current_selection", 200, 25, font_1)

        self.instructions_4 = Textbox(screen_x-325, screen_y-300, orange, "D/→: MOVE
RIGHT", "current_selection", 200, 25, font_1)



    def cleanup(self):

        del self.title, self.wasd, self.arrows, self.back, self.current_selection



    def switch_to_wasd(self):

        """

        This modifies the value referenced by the key 'keys' in the share dictionary.

        This changes the controls to "WASD" to be used in the game.

        """

        self.share.update({"keys": [K_w, K_s, K_d, K_a]})

        self.share.update({"current_controls": "WASD"})
```

```
        del self.current_selection

        self.current_selection = Textbox((screen_x/2)-175, screen_y-500, orange, "CURRENT
SELECTION: " + str(self.share["current_controls"]), "current_selection", 350, 50, font_1)

        menu_forward_sound.play()


    def switch_to_arrows(self):

        """

        This modifies the value referenced by the key 'keys' in the share dictionary.

        This changes the controls to the arrow keys to be used in the game.

        """

        self.share.update({"keys": [K_UP, K_DOWN, K_RIGHT, K_LEFT]})

        self.share.update({"current_controls": "ARROW KEYS"})

        del self.current_selection

        self.current_selection = Textbox((screen_x/2)-175, screen_y-500, orange, "CURRENT
SELECTION: " + str(self.share["current_controls"]), "current_selection", 350, 50, font_1)

        menu_forward_sound.play()


    def switch_main_menu(self):

        self.next = "main_menu"

        menu_back_sound.play()

        self.running = False


    def handle_events(self, event):

        self.wasd.check_clicked(event)

        self.arrows.check_clicked(event)

        self.back.check_clicked(event)


    def update(self):

        pass


    def render(self, screen):

        screen.fill(black)
```

```
        self.title.draw(screen)

        self.wasd.draw(screen)

        self.arrows.draw(screen)

        self.back.draw(screen)

        self.current_selection.draw(screen)

        self.instructions_1.draw(screen)

        self.instructions_2.draw(screen)

        self.instructions_3.draw(screen)

        self.instructions_4.draw(screen)
```

```
class Difficulty(States):

    def __init__(self):

        States.__init__(self)


    def setup(self):

        self.running = True

        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "DIFFICULTY SELECT",
"title_textbox", 500, 100, font_2)

        self.easy = Button((screen_x/2)-100, screen_y-500, yellow, "SELECT: EASY",
"easy_button", 200, 50, font_1, self.switch_easy)

        self.medium = Button((screen_x/2)-100, screen_y-400, yellow, "SELECT: MEDIUM",
"medium_button", 200, 50, font_1, self.switch_medium)

        self.hard = Button((screen_x/2)-100, screen_y-300, yellow, "SELECT: HARD",
"hard_button", 200, 50, font_1, self.switch_hard)

        self.custom = Button((screen_x/2)-100, screen_y-200, yellow, "SELECT: CUSTOM",
"custom_button", 200, 50, font_1, self.switch_custom_menu)

        self.back = Button((screen_x/2)-100, screen_y-100, yellow, "BACK", "back_button",
200, 50, font_1, self.switch_main_menu)


    def cleanup(self):

        del self.title, self.easy, self.medium, self.hard, self.back
```

```
    def handle_events(self, event):
        self.easy.check_clicked(event)
        self.medium.check_clicked(event)
        self.hard.check_clicked(event)
        self.custom.check_clicked(event)
        self.back.check_clicked(event)

    def switch_easy(self):
        """
        If 'easy' difficulty is selected, 'x_cells', 'y_cells' and 'timer' are modified
accordingly.
        This is done in the 'share' dictionary, and is carried to the game state.
        """
        self.share.update({"x_cells": 20})
        self.share.update({"y_cells": 20})
        self.share.update({"timer": 15})
        self.share.update({"current_difficulty": "EASY"})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_medium(self):
        """
        If 'medium' difficulty is selected, 'x_cells', 'y_cells' and 'timer' are modified
accordingly.
        This is done in the 'share' dictionary, and is carried to the game state.
        """
        self.share.update({"x_cells": 35})
        self.share.update({"y_cells": 35})
        self.share.update({"timer": 10})
        self.share.update({"current_difficulty": "MEDIUM"})
        self.next = "game"
```

```
        menu_forward_sound.play()
        self.running = False

    def switch_hard(self):
        """
        If 'hard' difficulty is selected, 'x_cells', 'y_cells' and 'timer' are modified
accordingly.
        This is done in the 'share' dictionary, and is carried to the game state.
        """
        self.share.update({"x_cells": 50})
        self.share.update({"y_cells": 50})
        self.share.update({"timer": 5})
        self.share.update({"current_difficulty": "HARD"})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_custom_menu(self):
        """
        If 'custom' difficulty is selected, the custom menu needs to be loaded.
        """
        self.next = "custom_menu"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def update(self):
        pass
```

```
    def render(self, screen):
        screen.fill(black)
        self.title.draw(screen)
        self.easy.draw(screen)
        self.medium.draw(screen)
        self.hard.draw(screen)
        self.custom.draw(screen)
        self.back.draw(screen)


class Custom_Menu(States):
    def __init__(self):
        States.__init__(self) # Initialises via the constructor of the superclass.


    def setup(self):
        self.running = True
        self.title = Textbox((screen_x/2)-250, screen_y/9, red, "CUSTOM GAME",
"custom_title", 500, 100, font_2)
        self.instructions = Textbox((screen_x/2)-450, screen_y - 500, orange,
                                    "SELECT YOUR MAZE WIDTH/HEIGHT AND TIMER DURATION. ONLY
SQUARE MAZES ARE SUPPORTED.",
                                    "instructions_textbox", 900, 50, font_1)
        self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange, "WIDTH/HEIGHT:
" + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)
        self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange, "DURATION:
" + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)
        self.back = Button((screen_x/2)-100, screen_y-100, yellow, "BACK", "back_button",
200, 50, font_1, self.switch_main_menu)
        self.size_increase = Button((screen_x/2)+110, screen_y-400, yellow, "+1",
"back_button", 50, 50, font_1, self.increase_size)
        self.size_decrease = Button((screen_x/2)-160, screen_y-400, yellow, "-1",
"back_button", 50, 50, font_1, self.decrease_size)
        self.duration_increase = Button((screen_x/2)+110, screen_y-300, yellow, "+1",
"back_button", 50, 50, font_1, self.increase_duration)
```

```
        self.duration_decrease = Button((screen_x/2)-160, screen_y-300, yellow, "-1",
"back_button", 50, 50, font_1, self.decrease_duration)

        self.play = Button((screen_x/2)-100, screen_y-200, yellow, "PLAY", "play_button",
200, 50, font_1, self.switch_game)




    def cleanup(self):

        del self.title, self.instructions, self.size_textbox, self.duration_textbox,
self.back, self.size_increase, self.size_decrease, self.duration_increase,
self.duration_decrease



    def handle_events(self, event):

        self.back.check_clicked(event)

        self.size_increase.check_clicked(event)

        self.size_decrease.check_clicked(event)

        self.duration_increase.check_clicked(event)

        self.duration_decrease.check_clicked(event)

        self.play.check_clicked(event)



    def increase_size(self):

        if (self.share["x_cells"] == 52) or (self.share["y_cells"] == 52):

            self.share.update({"x_cells": 1})

            self.share.update({"y_cells": 1})

            del self.size_textbox

            self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange,
"WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)

        else:

            self.share["x_cells"] = self.share["x_cells"] + 1

            self.share["y_cells"] = self.share["y_cells"] + 1

            del self.size_textbox

            self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange,
"WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)



    def decrease_size(self):
```

```
        if (self.share["x_cells"] == 1) or (self.share["y_cells"] == 1):

            self.share.update({"x_cells": 52})

            self.share.update({"y_cells": 52})

            del self.size_textbox

            self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange,
"WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)

        else:

            self.share["x_cells"] = self.share["x_cells"] - 1

            self.share["y_cells"] = self.share["y_cells"] - 1

            del self.size_textbox

            self.size_textbox = Textbox((screen_x/2)-100, screen_y-400, orange,
"WIDTH/HEIGHT: " + str(self.share["x_cells"]), "size_textbox", 200, 50, font_1)



    def increase_duration(self):

        if (self.share["timer"] == 15):

            self.share.update({"timer": 1})

            del self.duration_textbox

            self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange,
"DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)

        else:

            self.share["timer"] = self.share["timer"] + 1

            del self.duration_textbox

            self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange,
"DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)



    def decrease_duration(self):

        if (self.share["timer"] == 1):

            self.share.update({"timer": 15})

            del self.duration_textbox

            self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange,
"DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)

        else:

            self.share["timer"] = self.share["timer"] - 1

            del self.duration_textbox
```

```
            self.duration_textbox = Textbox((screen_x/2)-100, screen_y-300, orange,
"DURATION: " + str(self.share["timer"]), "duration_textbox", 200, 50, font_1)


    def switch_game(self):
        self.share.update({"current_difficulty": "CUSTOM"})
        self.next = "game"
        menu_forward_sound.play()
        self.running = False


    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False


    def update(self):
        pass


    def render(self, screen):
        screen.fill(black)
        self.title.draw(screen)
        self.instructions.draw(screen)
        self.size_textbox.draw(screen)
        self.duration_textbox.draw(screen)
        self.back.draw(screen)
        self.size_increase.draw(screen)
        self.size_decrease.draw(screen)
        self.duration_increase.draw(screen)
        self.duration_decrease.draw(screen)
        self.play.draw(screen)



class Game(States):
```

```
    def __init__(self):
        States.__init__(self) # Initialises via the constructor of the superclass.


    def setup(self):
        self.running = True
        ### generates the maze and creates the walls
        """
        Uses the maze algorithm, passing in the width and height of the grid (x_cells and
y_cells).
        """
        self.maze_data = generate_maze(self.share["x_cells"], self.share["y_cells"])
        for index, row in enumerate(self.maze_data):
            row = "|" + row[1:] + "|"
            self.maze_data[index] = row
        self.maze_data.insert(0, "|" * (self.share["y_cells"] + 2))
        self.maze_data.append("|" * (self.share["x_cells"] +  2))
        self.walls = []
        self.cell_size = 10
        self.maze_x = ((screen_x/2)-(self.cell_size*((self.share["x_cells"]+2)/2)))
        self.maze_y = ((screen_y/2)-(self.cell_size*((self.share["y_cells"]+2)/2)))


        for x, tiles in enumerate(self.maze_data):
            for y, tile in enumerate(tiles):
                if tile == "|":
                    wall = Wall(self.maze_x + (y*10), self.maze_y + (x*10), self.cell_size,
white)
                    self.walls.append(wall)


        ### Finds the appropriate location for the character and goal
        if self.maze_data[1][1] == '|':
            if self.maze_data[1][2] == '|':
                if self.maze_data[2][1] == '|':
```

```
                    if self.maze_data[2][2] == '|':

                        self.next = "game"

                        self.running = False

                    else:

                        self.default_xpos = (((screen_x/2)-
(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size

                        self.default_ypos = (((screen_y/2)-
(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size

                else:

                    self.default_xpos = (((screen_x/2)-
(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)

                    self.default_ypos = (((screen_y/2)-
(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size) + self.cell_size



            else:

                self.default_xpos = (((screen_x/2)-
(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size) + self.cell_size

                self.default_ypos = (((screen_y/2)-
(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)

        else:

            self.default_xpos = (((screen_x/2)-
(self.cell_size*((self.share["x_cells"]+2)/2))) + self.cell_size)

            self.default_ypos = (((screen_y/2)-
(self.cell_size*((self.share["y_cells"]+2)/2))) + self.cell_size)



        if self.maze_data[self.share["x_cells"]][self.share["x_cells"]] == '|':

            if self.maze_data[self.share["x_cells"] - 1][self.share["y_cells"]] == '|':

                if self.maze_data[self.share["x_cells"]][self.share["y_cells"] - 1] == '|':

                    if self.maze_data[self.share["x_cells"] - 1][self.share["y_cells"] - 1]
== '|':

                        self.next = "game"

                        self.running = False

                    else:

                        self.goal_xpos =
(((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2))) - self.cell_size)
```

```
                        self.goal_ypos =
(((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2))) - self.cell_size)

                else:

                    self.goal_xpos = ((screen_x/2)+(self.cell_size*((self.share["x_cells"]-
2)/2)))

                    self.goal_ypos =
(((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2))) - self.cell_size)

            else:

                self.goal_xpos = (((screen_x/2)+(self.cell_size*((self.share["x_cells"]-
2)/2))) - self.cell_size)

                self.goal_ypos = ((screen_y/2)+(self.cell_size*((self.share["y_cells"]-
2)/2)))

        else:

            self.goal_xpos = ((screen_x/2)+(self.cell_size*((self.share["x_cells"]-2)/2)))
# Originally 350

            self.goal_ypos = ((screen_y/2)+(self.cell_size*((self.share["y_cells"]-2)/2)))
# Originally 310




        ### Instantiates character, timer, textbox, goal

        self.character = Character(self.default_xpos, self.default_ypos, self.cell_size,
self.cell_size, green, self.share["keys"], self.cell_size)

        self.timer = Timer(screen_x - 130, 30, red, 10, 10, self.share["timer"])

        self.title_textbox = Textbox((screen_x/2)-100, (screen_y/25), white,
str(self.share["current_difficulty"]) + ": " + str(self.share["x_cells"]) + " by " +
str(self.share["y_cells"]), "test", 200, 50, font_1)

        self.goal = Goal(self.goal_xpos, self.goal_ypos, self.cell_size, yellow)

        self.lava_cells = []

        self.forfeit_button = Button(25, screen_y/25, yellow, "FORFEIT", "forfeit_button",
200, 50, font_1, self.forfeit)




    def cleanup(self):

        """

        Deletes all objects on screen, including walls and lava.

        """
```

```
        del self.character, self.title_textbox, self.timer, self.goal, self.forfeit_button
        for wall in self.walls:
            del wall
        for lava in self.lava_cells:
            del lava

    def handle_events(self, event):
        self.forfeit_button.check_clicked(event)



    def update(self):
        """
        The following two variables are used to keep track of the character location,
        and to check when the character moves.
        """
        self.character_x = self.character.x
        self.character_y = self.character.y

        self.character.move(self.walls, self.timer) # Allows the character to move.
        self.timer.countdown() # The timer counts down.

        if (self.character.x != self.character_x) or (self.character.y !=
self.character_y): # If the character has moved at all...
            lava_cell = Lava(self.character_x, self.character_y, self.cell_size, red) #
Instantiate a lava cell at the character's previous location.
            self.lava_cells.append(lava_cell) # Append this cell to the array of lava
cells.

        for lava in self.lava_cells: # For all the lava cells in the array...
            lava.check_touching(self.character, self.timer) # Check if the character is
touching any of the cells.

        self.goal.check_touching(self.character, self.timer) # Check if the character has
reached the goal.
```

```
        if self.timer.timer_expired == True: # If the timer has expired...
            self.character.die() # Kill the character.
            self.next = "game_over" # Set the next state to the game over screen.
            self.running = False # Move to the next state.

        if self.goal.win: # If the character wins...
            """
            The following 'if' statements check what difficulty the player was playing at.
            """
            if self.share["current_difficulty"] == "EASY":
                self.share["easy_wins"] = self.share["easy_wins"] + 1
            if self.share["current_difficulty"] == "MEDIUM":
                self.share["medium_wins"] = self.share["medium_wins"] + 1
            if self.share["current_difficulty"] == "HARD":
                self.share["hard_wins"] = self.share["hard_wins"] + 1

            self.next = "game_won" # Set the next state to the winning screen.
            self.running = False # Move to the next state.

    def forfeit(self):
        self.timer.duration = 0

    def render(self, screen):
        screen.fill(black)
        for wall in self.walls: # Draws all the walls to screen.
            wall.draw(screen)
        self.character.draw(screen)
        self.goal.draw(screen)
        self.timer.draw(screen)
        self.title_textbox.draw(screen)
        for lava in self.lava_cells: # Draws all the lava cells to screen.
```

```
            lava.draw(screen)

        self.forfeit_button.draw(screen)



class Game_Over(States):

    def __init__(self):

        States.__init__(self)


    def setup(self):

        time.sleep(1.5) # Delay is used to stop the state jumping immediately (from the
game state).

        self.running = True

        self.you_lose = Textbox((screen_x/2)-250, screen_y/9, red, "GAME OVER",
"title_textbox", 500, 100, font_2)

        self.retry = Button((screen_x/2)-100, screen_y-500, yellow, "RETRY", "retry_game",
200, 50, font_1, self.switch_game)

        self.change_difficulty = Button((screen_x/2)-100, screen_y-400, yellow, "CHANGE
DIFFICULTY", "retry_game", 200, 50, font_1, self.switch_difficulty)

        self.main_menu = Button((screen_x/2)-100, screen_y-300, yellow, "RETURN TO MENU",
"return_to_menu", 200, 50, font_1, self.switch_main_menu)

        self.quit_button = Button((screen_x/2)-100, screen_y-200, yellow, "QUIT",
"quit_button", 200, 50, font_1, self.quit_game)


    def cleanup(self):

        del self.you_lose, self.main_menu, self.retry, self.change_difficulty,
self.quit_button


    def handle_events(self, event):

        self.main_menu.check_clicked(event)

        self.retry.check_clicked(event)

        self.change_difficulty.check_clicked(event)

        self.quit_button.check_clicked(event)


    def switch_difficulty(self):
```

```
        self.next = "difficulty_select"
        menu_forward_sound.play()
        self.running = False

    def switch_game(self):
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def quit_game(self):
        """
        Exits the game.
        """
        exit_sound.play()
        self.quit = True

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.you_lose.draw(screen)
        self.main_menu.draw(screen)
        self.change_difficulty.draw(screen)
        self.retry.draw(screen)
        self.quit_button.draw(screen)
```

```
class You_Win(States):

    def __init__(self):

        States.__init__(self)


    def setup(self):

        time.sleep(1.5)

        self.running = True

        self.you_win = Textbox((screen_x/2)-250, screen_y/9, red, "YOU WIN!",
"title_textbox", 500, 100, font_2)

        self.retry = Button((screen_x/2)-100, screen_y-500, yellow, "RETRY", "retry_game",
200, 50, font_1, self.switch_game)

        self.change_difficulty = Button((screen_x/2)-100, screen_y-400, yellow, "CHANGE
DIFFICULTY", "retry_game", 200, 50, font_1, self.switch_difficulty)

        self.main_menu = Button((screen_x/2)-100, screen_y-300, yellow, "RETURN TO MENU",
"return_to_menu", 200, 50, font_1, self.switch_main_menu)

        self.quit_button = Button((screen_x/2)-100, screen_y-200, yellow, "QUIT",
"quit_button", 200, 50, font_1, self.quit_game)


    def cleanup(self):

        del self.you_win, self.main_menu, self.retry, self.change_difficulty,
self.quit_button


    def handle_events(self, event):

        self.main_menu.check_clicked(event)

        self.retry.check_clicked(event)

        self.change_difficulty.check_clicked(event)

        self.quit_button.check_clicked(event)


    def switch_difficulty(self):

        self.next = "difficulty_select"

        menu_forward_sound.play()

        self.running = False
```

```
    def switch_game(self):
        self.next = "game"
        menu_forward_sound.play()
        self.running = False

    def switch_main_menu(self):
        self.next = "main_menu"
        menu_back_sound.play()
        self.running = False

    def quit_game(self):
        """
        Exits the game.
        """
        exit_sound.play()
        self.quit = True

    def update(self):
        pass

    def render(self, screen):
        screen.fill(black)
        self.you_win.draw(screen)
        self.main_menu.draw(screen)
        self.change_difficulty.draw(screen)
        self.retry.draw(screen)
        self.quit_button.draw(screen)


class Control: # This class controls the entire program, handling all the states.
    def __init__(self):
```

```
        self.screen = pygame.display.set_mode((screen_x, screen_y)) # Sets up the screen
according to the resolution (from assets.py).

        self.clock = pygame.time.Clock() # Sets up the pygame clock.

        self.running = True # Starts running the current state.

        self.state_dict = None # Initialises the state dictionary as None.

        self.current_state = None # Initialises the current state as None.


    def setup(self, state_dict, start_state): # This takes in the state dictionary and the
start state, as a starting point for the program.

        # 'start_state' represents the key for the instance of the state in 'state_dict'.

        self.state_dict = state_dict

        self.current_state = self.state_dict[start_state] # Sets the current state to an
instance of the start state, referencing the state dictionary.

        self.current_state.setup() # Calls the setup method for the current state,
instantiating all objects needed.


    def handle_events(self):

        for event in pygame.event.get():

            if event.type == pygame.QUIT: # Checks if the [X] button has been pressed.

                self.running = False

            self.current_state.handle_events(event)


        if not(self.current_state.running): # If the current state is no longer running...

            self.change_state() # ...move to the next state.


        if self.current_state.quit == True:

            self.running = False # Quits the game.


    def change_state(self):

        new_state = self.current_state.next # Sets up the key for the new state, to
reference in 'state_dict'.

        self.current_state.cleanup() # Deletes all objects from the current state.

        self.current_state = self.state_dict[new_state] # Sets the current state to the
object referenced by the key 'new_state', from 'state_dict'.
```

```
        self.current_state.setup() # Sets up the new current state.


    def main_loop(self): # Acts as a game loop.
        while self.running:
            self.clock.tick(fps)
            self.handle_events()
            self.current_state.update()
            self.current_state.render(self.screen)
            pygame.display.flip() # Refreshes display.




pygame.init() # Initialises pygame.
app = Control() # Creates an instance of 'Control', named 'app'.
"""
Below is the state dictionary.
It contains instances of all the states.
"""
state_dict = {
    "main_menu": Main_Menu(),
    "difficulty_select": Difficulty(),
    "instructions_screen": Instructions(),
    "controls_screen" : Controls_Setup(),
    "custom_menu": Custom_Menu(),
    "game" : Game(),
    "game_won": You_Win(),
    "game_over": Game_Over(),
    }
"""
Below calls the setup method for app (instance of Control),
and passes in the state dictionary and the main menu as the starting state.
```

```
"""
app.setup(state_dict, "main_menu")
app.main_loop()
pygame.quit() # Quits pygame. Only happens when [X] is pressed.
```

characters.py

```
import pygame
from assets import *

# Wall class - used for the maze.
class Wall:
    def __init__(self, x, y, size, colour):
        self.x = x
        self.y = y
        self.size = size
        self.colour = colour

    def draw(self, screen):
        pygame.draw.rect(screen, (self.colour), (self.x, self.y, self.size, self.size))

# Character class - used for the player to control in the main game.
class Character:
    def __init__(self, x, y, width, height, colour, controls, speed):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.colour = colour
        self.controls = controls
        self.speed = speed
        self.alive = True
        self.move_delay = 65
        self.last_move = pygame.time.get_ticks()
        self.moved = False
        self.moved_x = False
        self.moved_y = False
```

```
    def draw(self, screen):
        if self.alive == True:
            pygame.draw.rect(screen, self.colour, pygame.Rect((self.x, self.y, self.width,
self.height)))

    def wall_collision(self, walls):
        for wall in walls:
            if wall.x == self.x and wall.y == self.y:
                return True # returns True if the character is touching a wall
        return False # if not, returns False

    def move(self, walls, timer):
        if self.alive == True:
            now = pygame.time.get_ticks()

            if now - self.last_move > self.move_delay: # implements a slight movement delay
                self.last_move = now

                pressed_keys = pygame.key.get_pressed()
                if self.alive == True: # the character can only move if they're alive
                    """
                        The following first checks what key has been pressed, and then
moves the character in that direction once.
                        It then checks if the character has collided with a wall.
                        If yes, they're moved back to where they were previously.
                    """
                    if pressed_keys[self.controls[0]] and (self.moved_x == False):
                        self.y -= self.speed
                        if self.wall_collision(walls):
                            self.y += self.speed
                            self.moved = False
                        else:
                            self.moved = True
                            self.moved_y = True
```

```
if pressed_keys[self.controls[1]] and (self.moved_x == False):
    self.y += self.speed
    if self.wall_collision(walls):
        self.y -= self.speed
        self.moved = False
    else:
        self.moved = True
        self.moved_y = True

if pressed_keys[self.controls[2]] and (self.moved_y == False):
    self.x += self.speed
    if self.wall_collision(walls):
        self.x -= self.speed
        self.moved = False
    else:
        self.moved = True
        self.moved_x = True

if pressed_keys[self.controls[3]] and (self.moved_y == False):
    self.x -= self.speed
    if self.wall_collision(walls):
        self.x += self.speed
        self.moved = False
    else:
        self.moved = True
        self.moved_x = True

if self.moved == True:
    timer.reset()
    self.moved = False

if self.moved_x == True:
    self.moved_x = False
```

```
                    if self.moved_y == True:
                        self.moved_y = False
    def die(self):
        self.alive = False

    def reset(self):
        self.alive = True


# Timer class - counts how much more time the player can remain idle before death.
class Timer:
    def __init__(self, x, y, colour, width, height, duration):
        self.x = x
        self.y = y
        self.colour = colour
        self.width = width
        self.height = height
        self.duration = int(duration)
        self.last_tick = pygame.time.get_ticks() # Used to allow the timer to count down
        self.timer_expired = False
        self.sound_played = False
        self.paused = False
        self.original_duration = int(duration)

    def draw(self, screen):
        if self.paused == False: # Only counts down if the timer isn't paused
            """
            if the timer is at 'n' duration, display the image with n seconds left.
            """
            if self.duration == 15:
                screen.blit(timer15_image, (self.x, self.y))
            elif self.duration == 14:
                screen.blit(timer14_image, (self.x, self.y))
            elif self.duration == 13:
                screen.blit(timer13_image, (self.x, self.y))
            elif self.duration == 12:
```

```
            screen.blit(timer12_image, (self.x, self.y))
        elif self.duration == 11:
            screen.blit(timer11_image, (self.x, self.y))
        elif self.duration == 10:
            screen.blit(timer10_image, (self.x, self.y))
        elif self.duration == 9:
            screen.blit(timer9_image, (self.x, self.y))
        elif self.duration == 8:
            screen.blit(timer8_image, (self.x, self.y))
        elif self.duration == 7:
            screen.blit(timer7_image, (self.x, self.y))
        elif self.duration == 6:
            screen.blit(timer6_image, (self.x, self.y))
        elif self.duration == 5:
            screen.blit(timer5_image, (self.x, self.y))
        elif self.duration == 4:
            screen.blit(timer4_image, (self.x, self.y))
        elif self.duration == 3:
            screen.blit(timer3_image, (self.x, self.y))
        elif self.duration == 2:
            screen.blit(timer2_image, (self.x, self.y))
        elif self.duration == 1:
            screen.blit(timer1_image, (self.x, self.y))
        else:
            screen.blit(timer0_image, (self.x, self.y))
            self.expired()


def countdown(self):
    if self.paused == False:
        if self.timer_expired == False:
            now = pygame.time.get_ticks()
            if now > self.last_tick + 1000:
                self.duration -= 1
                self.last_tick = now
                ticking_sound.play()
```

```
    def expired(self):
        if self.paused == False:
            self.timer_expired = True
            if self.sound_played == False:
                lose_sound.play()
                self.sound_played = True

    def pause(self):
        self.paused = True

    def reset(self):
        self.timer_expired = False
        self.sound_played = False
        self.duration = self.original_duration


# Lava class - used to trail the player in the main game.
class Lava:
    def __init__(self, x, y, size, colour):
        self.x = x
        self.y = y
        self.width = size
        self.height = size
        self.colour = colour
        self.touching = False

    def draw(self, screen):
        pygame.draw.rect(screen, self.colour, pygame.Rect((self.x, self.y, self.width,
self.height)))

    def check_touching(self, character, timer):
        if (character.x == self.x) and (character.y == self.y):
            self.touching_player(character, timer)

    def touching_player(self, character, timer):
```

```
        character.die()
        timer.duration = 0 # Sets the duration to 0, killing the player


# Goal class (child of lava) - a single cell that the player must reach to win.
# Operates similarly to the lava, except triggers victory.
class Goal(Lava):
    def __init__(self, x, y, size, colour):
        Lava.__init__(self, x, y, size, colour) # Initialises as a lava cell...
        # ...but has two more attributes
        self.sound_played = False
        self.win = False

    def check_touching(self, character, timer):
        if (character.x == self.x) and (character.y == self.y):
            self.touching_player(character, timer)

    def touching_player(self, character, timer):
        if self.sound_played == False:
            win_sound.play()
            self.sound_played = True
            self.won(timer, character)

    def won(self, timer, character):
        self.win = True
        timer.pause()
        character.die()
```

assets.py

```
import pygame
pygame.init()

# screen setup
screen_x = 1000
screen_y = 700
fps = 144
```

```
# font setup
font_1 = pygame.font.SysFont("segoeui", 20)
font_2 = pygame.font.SysFont("segoeuiblack", 40)

# sound setup
entry_sound = pygame.mixer.Sound("sounds/entry.wav")
exit_sound = pygame.mixer.Sound("sounds/exit.wav")
menu_back_sound = pygame.mixer.Sound("sounds/menu_back.wav")
menu_forward_sound = pygame.mixer.Sound("sounds/menu_forward.wav")
ticking_sound = pygame.mixer.Sound("sounds/singlewind3.wav")
lose_sound = pygame.mixer.Sound("sounds/you_lose.wav")
win_sound = pygame.mixer.Sound("sounds/you_win.wav")

# image setup
timer0_image = pygame.image.load("images/timer/timer0.png")#.convert_alpha()
timer1_image = pygame.image.load("images/timer/timer1.png")#.convert_alpha()
timer2_image = pygame.image.load("images/timer/timer2.png")#.convert_alpha()
timer3_image = pygame.image.load("images/timer/timer3.png")#.convert_alpha()
timer4_image = pygame.image.load("images/timer/timer4.png")#.convert_alpha()
timer5_image = pygame.image.load("images/timer/timer5.png")#.convert_alpha()
timer6_image = pygame.image.load("images/timer/timer6.png")#.convert_alpha()
timer7_image = pygame.image.load("images/timer/timer7.png")#.convert_alpha()
timer8_image = pygame.image.load("images/timer/timer8.png")#.convert_alpha()
timer9_image = pygame.image.load("images/timer/timer9.png")#.convert_alpha()
timer10_image = pygame.image.load("images/timer/timer10.png")#.convert_alpha()
timer11_image = pygame.image.load("images/timer/timer11.png")#.convert_alpha()
timer12_image = pygame.image.load("images/timer/timer12.png")#.convert_alpha()
timer13_image = pygame.image.load("images/timer/timer13.png")#.convert_alpha()
timer14_image = pygame.image.load("images/timer/timer14.png")#.convert_alpha()
timer15_image = pygame.image.load("images/timer/timer15.png")#.convert_alpha()

# colour setup
black = (0, 0, 0)
white = (255, 255, 255)
red = (255, 0, 0)
```

```
orange = (255, 127, 0)
yellow = (255, 255, 0)
green = (0, 255, 0)
blue = (0, 0, 255)
indigo = (46, 43, 95)
violet =(139, 0, 255)
```

maze_algorithm.py

```
import random # Since the generation of this maze is random, I need the random module.

def generate_maze(x_cells, y_cells):
#x_cells = 20 # This defines the number of cells in the x-direction.
#y_cells = 20 # This defines the number of cells in the y-direction.
    grid = [] # This allows me to build the grid from which the maze is generated.

    for y in range(y_cells):
        row = [] # Creates a 'row' list for all y locations (10 rows)
        for x in range(x_cells):
            row.append('?') # Sets all cells to UNDETERMINED
        grid.append(row) # Fills the grid with these rows




    exposed = [] # This is a list of EXPOSED UNDETERMINED LOCATIONS (+)

    def carve(y, x): # This function makes the cell a space.
        if (0 > (x or y)) or ((x > x_cells) or (y > y_cells)) or (type(x) != int or type(y)
!= int):
            return # This is for validation - 'if invalid data has been entered, ignore.'

        grid[y][x] = ' ' # This takes the two parameters and updates location.
        adjacent = [] # This stores any adjacent cells.

        if x > 0: # This checks if there is a cell behind the current cell (x direction).
            if grid[y][x-1] == '?': # If there is, and this adjacent cell is
undetermined...
```

```
                grid[y][x-1] = '+' # ...Mark it as EXPOSED.
                adjacent.append((y,x-1)) # Add this cell to the 'adjacent'


        if x < x_cells - 1: # This checks if there is a cell in front of the current cell
(x direction).
            if grid[y][x+1] == '?': # If there is, and this adjacent cell is
undetermined...
                grid[y][x+1] = '+' # ...Mark it as EXPOSED.
                adjacent.append((y,x+1)) # Add this cell to the 'adjacent' list.


        if y > 0: # This checks if there is a cell behind the current cell (y direction).
            if grid[y-1][x] == '?': # If there is, and this adjacent cell is
undetermined...
                grid[y-1][x] = '+' # ...Mark it as EXPOSED.
                adjacent.append((y-1,x)) # Add this cell to the 'adjacent' list.


        if y < y_cells - 1: # This checks if there is a cell in front of the current cell
(y direction).
            if grid[y+1][x] == '?': # If there is, and this adjacent cell is
undetermined...
                grid[y+1][x] = '+' # ...Mark it as EXPOSED.
                adjacent.append((y+1,x)) # Add this cell to the 'adjacent' list.


        random.shuffle(adjacent) # Shuffles the list of adjacent cells.
        exposed.extend(adjacent) # Adds the list of adjacent cells to the 'exposed' list.



    def wall(y, x): # This function make the cell a wall.
        if (0 > (x or y)) or ((x > x_cells) or (y > y_cells)) or (type(x) != int or type(y)
!= int):
            return # This is for validation - 'if invalid data has been entered, ignore.'
        grid[y][x] = '|' # This takes the two parameters and updates location.



    def check(y, x, nodiagonals = True): # Takes the 'nodiagonals' parameter...
    # ...this parameter determines if a cell has NO diagonals (when True).
```

```
        edgestate = 0 # This is a way of determining the behaviour of the edges around a
point.

        if x > 0: # If the cell has cells to the left...
            if grid[y][x-1] == ' ': # ...if the cell directly to the left is a space...
                edgestate += 1 # Increase edgestate by 1.

        if x < x_cells-1: # If the cell has cells to the right...
            if grid[y][x+1] == ' ': # ...if the cell directly to the right is a space...
                edgestate += 2 # Increase edgestate by 2.

        if y > 0: # If the cell has cells above...
            if grid[y-1][x] == ' ': # ...if the cell directly above is a space...
                edgestate += 4 # Increase edgestate by 4.

        if y < y_cells-1: # If the cell has cells below...
            if grid[y+1][x] == ' ': # ...if the cell directly below is a space...
                edgestate += 8 # Increase edgestate by 8.

        if nodiagonals: # If 'nodiagonals' is True...
            if edgestate == 1: # ...if 'edgestate' is 1...
                if x < x_cells-1: # ...if there are cells to the right...
                    if y > 0: # ...if there are cells above...
                        if grid[y-1][x+1] == ' ': # ...if the cell directly top-right is a
space...
                            return False # Return False.
                    if y < y_cells-1: # ...if there are cells below...
                        if grid[y+1][x+1] == ' ': # ...if the cell directly bottom-right is
a space...
                            return False # Return False.
                return True # Return True.

            elif edgestate == 2: # ...else, if 'edgestate' is 2...
                if x > 0: # ...if there are cells to the left...
                    if y > 0: # ...if there are cells above...
                        if grid[y-1][x-1] == ' ': # ...if the cell directly top-left is a
space...
```

```
                            return False # Return False.
                    if y < y_cells-1: # ...if there are cells below...
                        if grid[y+1][x-1] == ' ': # ...if the cell directly bottom-left is
a space...
                            return False # Return False.
                return True # Return True.

            elif edgestate == 4: # ...else, if 'edgestate' is 4...
                if y < y_cells-1: # ...if there are cells below...
                    if x > 0: # ...if there are cells to the left...
                        if grid[y+1][x-1] == ' ': # ...if the cell directly bottom-left is
a space...
                            return False # Return False.
                    if x < x_cells-1: # ...if there are cells to the right...
                        if grid[y+1][x+1] == ' ': # ...if the cell directly bottom-right is
a space...
                            return False # Return False.
                return True # Return True.

            elif edgestate == 8: # ...if 'edgestate' is 8...
                if y > 0: # ...if there are cells above...
                    if x > 0: # ...if there are cells to the left...
                        if grid[y-1][x-1] == ' ': # ...if the cell directly top-left is a
space...
                            return False # Return False.
                    if x < x_cells-1: # ...if there are cells to the right...
                        if grid[y-1][x+1] == ' ': # ...if the cell directly top-right is a
space...
                            return False # Return False.

                return True # Return True.
            return False # Return False.

        else: # ...if not... (hence 'nodiagonals' is False)
            if [1, 2, 4, 8].count(edgestate): # ...count the number if times that
'edgestate' appears in that array.
                # If it does appear in the array...
                return True # Return True.
```

```
            return False # Return


    x_random = random.randint(0, x_cells-1) # Choose a random point in the x-axis, on the
grid.
    y_random = random.randint(0, y_cells-1) # Choose a random point in the y-axis, on the
grid.
    carve(x_random, y_random) # Use these random points and carve.



    def create():
        while(len(exposed)): # While there are exposed cells...
            pos = random.random() # Select a random number.
            choice = exposed[int(pos*len(exposed))] # This randomly selects an exposed cell
from the list.
            if check(*choice): # If the selected cell should become a space...
                carve(*choice) # ...make this cell a space.
            else: # If not...
                wall(*choice) # ...make this cell a wall.
            exposed.remove(choice) # Remove this cell from the 'exposed' list.


    # The following changes any unexposed, unidentified cells to be walls.
        for y in range(y_cells):
            for x in range(x_cells):
                if grid[y][x] == '?':
                    grid[y][x] = '|'


    # The following prints the entire maze.
        maze_data = []
        for y in range(y_cells):
            row = ' '
            for x in range(x_cells):
                row += grid[y][x]
            maze_data.append(row)


        return maze_data
```

```
    maze_data = create()
    return maze_data

"""
KEY:
'|' indicates a WALL
' ' indicates a SPACE
'+' indicates an EXPOSED UNDETERMINED LOCATION
'?' indicates an UNDETERMINED LOCATION
"""

'''
The 'check' function determines if a cell should be a space or a wall.
If it returns True, it should become a space.
If it returns False, it should become a wall.
'''
```

ui_elements.py

```
import pygame
from assets import *

from pygame.locals import *
pygame.init()



class Textbox:
    def __init__(self, x, y, colour, text, name, width, height, font):
        self.name = name # name of the textbox
        self.colour = colour # colour of the textbox
        self.text = str(text) # text on the textbox
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
        self.alt_txt = font.render(self.text, True, ((self.colour)))
        self.rect = pygame.Rect((x, y, width, height)) # stores the coordinates, width and
height of the rectangle
```

```
    def draw(self, screen):
        pygame.draw.rect(screen, self.colour, self.rect) # draws the textbox
        pygame.draw.rect(screen, black, self.rect, 1) # draws the outline
        txt_rect = self.txt.get_rect(center=self.rect.center) # draws the text to the
center of the textbox
        screen.blit(self.txt, txt_rect) # draws the text



class Button:
    def __init__(self, x, y, colour, text, name, width, height, font, function):
        self.name = name # name of the button
        self.colour = colour # colour of the button
        self.text = str(text) # text on the button
        self.font = font # font used with the text
        self.txt = font.render(self.text, True, ((black))) # renders the text in pygame
        self.alt_txt = font.render(self.text, True, ((self.colour)))
        self.rect = pygame.Rect((x, y, width, height)) # stores the coordinates, width and
height of the rectangle
        self.clicked = False # determines whether the button is currently clicked
        self.function = function # stores the function to be called when the button is
clicked


    def draw(self, screen):
        if self.clicked == False:
            pygame.draw.rect(screen, self.colour, self.rect) # draws the button
            pygame.draw.rect(screen, black, self.rect, 5) # draws the outline
            txt_rect = self.txt.get_rect(center=self.rect.center) # draws the text to the
center of the button
            screen.blit(self.txt, txt_rect) # draws the text
        else:
            pygame.draw.rect(screen, black, self.rect) # draws the button
            pygame.draw.rect(screen, self.colour, self.rect, 5) # draws the outline
            txt_rect = self.alt_txt.get_rect(center=self.rect.center) # draws the text to
the center of the button
            screen.blit(self.alt_txt, txt_rect) # draws the text


    def check_clicked(self, event):
        if event.type == pygame.MOUSEBUTTONDOWN: # if the mouse is clicked...
```

```
            if self.rect.collidepoint(pygame.mouse.get_pos()) == True: # ...and the mouse
is on the button...
                self.clicked = True # the button is clicked.
                #self.is_clicked() # call 'is_clicked'
        elif event.type == pygame.MOUSEBUTTONUP: # when the button is released...
            if self.clicked: # if it WAS clicked...
                self.function() # calls the appropriate function
            self.clicked = False # sets the button to unclicked once again, as the button
is released
```

[redacted]

([redacted].