**Game Design Documentation: Tower Defense Game**

**Game Idea & Overview:**

The idea for this game was to make a tower defense game, where the player has to strategically place down defensive towers or structures along a path or map to prevent waves of enemies from reaching the end of the path.

Each tower has a unique firing ability and a unique cost to build, where the goal of the game is to survive the waves of enemies and prevent them from reaching the end of the path. Players have to demonstrate quick thinking, efficient resource management over increasingly-difficult levels to achieve victory.

**Game Features and Mechanisms:**

**Characters / Towers:**

As for the characters of the game, they are the towers. In this case, there are 3 different towers that are possible to place down, with each tower having multiple upgrades.

All towers follow the same concept, where as the tower is upgraded, the appearance of the tower changes (As shown in source 1 as an example), and the firing rate lowers, which means the towers fire quicker.

The first row of towers, where each tower is a different upgrade level, with the levels increasing from left to right, is shown in source 1. The images for the sprites were found from the following sprite sheet: <https://zintoki.itch.io/ground-shaker> .



The second row of towers, is shown in source 2. This is the middle tower, costing X coins to place down. The images for the sprites can be found from the following sprite sheet: <https://zintoki.itch.io/ground-shaker> .

A screenshot of a video game

Description automatically generated

Finally, the third row of towers, is shown in source 3. This is the most expensive type of tower to place down, as it starts off with the fastest firing rate. The images for the sprites can be found from the following sprite sheet: <https://zintoki.itch.io/space-breaker> .

A red object with yellow arrows

Description automatically generated

**Enemies:**

The enemies of the game are the ones that traverse the path trying to get to the end. In this case, there are 3 types of enemies that can traverse the path. The difference between each of the enemies is their speed when traversing the path.

The first enemy which is shown in source 4, is the slowest enemy, as it only moves 1 tile per iteration.

A cartoon of a brown and yellow face

Description automatically generated with medium confidence

The second enemy which is shown in source 5, is the second slowest enemy. This enemy moves 2 tiles per iteration.

A cartoon character with a green band

Description automatically generated

The third and final enemy which is shown in source 6, is the fastest enemy. This enemy moves 3 tiles per iteration.

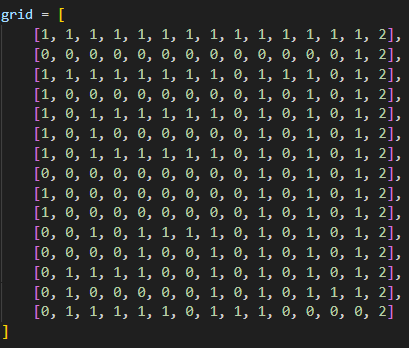
A cartoon character with a brown face

Description automatically generated

All three of these sprites were taken from the game Bloons TD Battles, more information about the game can be found here: <https://bloons.fandom.com/wiki/Bloons_TD_Battles> .

**Map:**

The map of this game is very simple, following a simple grid of 0s and 1s as shown in source 7.



Where 0s represent a grass block, and 1s represent a path block, as shown in source 8.

This method allows for control over each tile individually, giving the programmer more control over the game when coding.

**Menus:**

X

**Technical Requirements:**

The technical requirements for the game are quite insignificant, as the whole program is written in the Python programming language, and is using no game development engine. The game also requires no internet connection, as it is designed for single-player gameplay against the waves

Though the game is designed for desktop platforms, including Windows, MacOS and Linux. As for the hardware requirements,

**Minimum:**

* CPU: Dual-Core processor
* GPU: Integrated Graphics, or a dedicated GPU with DirectX 11 support
* RAM: 2 GB
* Storage: 200 MB available space (Excluding the installation of IDE and libraries)

**Recommended:**

* CPU: Quad-Core processor
* GPU: Dedicated GPU with DirectX 12 support
* RAM: 4 GB or more
* Storage: 500 MB available space (Excluding the installation of IDE and libraries)

**How to Run the Game:**

To run the game, the player needs to set up their environment, where they first need to;

1. Install Python: Ensure Python 3.x is installed onto system
2. Install PyGame: Use the ‘pip’ module that comes with Python to install PyGame. Can be done by running ‘pip install pygame’ in your Terminal or Command Prompt
3. Setup IDE: Choose an IDE (Integrated Development Environment) like Pycharm, Visual Studio Code, or even a simpler editor like Sublime Text

After having these features setup, the next thing needed is the ZIP File. Download the ZIP file from where accessible, such as from GitHub, and extract everything inside into a folder on your device.

Then import the folder with the files onto the IDE, where then when you run the ‘main.py’ file, it should execute the full game.

**User Interface and Controls:**

As for the controls of this game, tower defence has very minimal controls, where the only thing the player does is place & upgrade towers. For this, the only type of control needed is a mouse button click.

The first type of user interface is the main menu, shown in source 9:



Source 9: Main Menu

You have to navigate through this main menu, where if you click “play”, the game itself will start. Whereas if you click controls, a page explaining all the controls will appear, as shown in source 10:

A screenshot of a video game

Description automatically generated

Source 10: Controls Screen

Finally, if you click quit, the game will just quit.

When you do click “Play”, the game will start, where once you reach the map area, you can open the side menu, as shown in source 11, where you can select a tower and then place it on one of the unoccupied grass spots.

X

Other than this, you can also upgrade towers, where you can click on the tower you want to upgrade. If you have enough money to upgrade the tower, it will upgrade to the next level.

**The Different Screens in the Game:**

**Main Menu Screen;**



Source 12: Main Menu Screen

This is the screen the player first sees when they run the game, where there are three options; “Play”, “Controls” or “Quit”

The “play” button starts the game, where the player is then transferred to the level screen.

The “Controls” button opens the controls screen, as shown in source 13, where the controls screen is another screen that shows the player how to play the game.

A screenshot of a video game

Description automatically generated

Source 13: Controls Screen

Finally the “Quit” button, which quits the game.

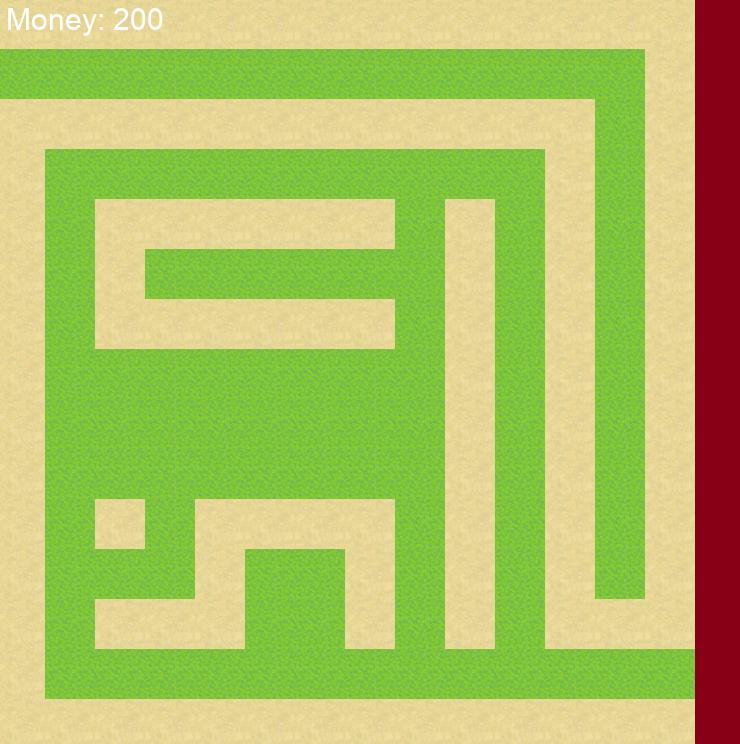
If the player clicks the “Play” button, the screen will change into a level screen, which shows the player the current level that they are on, as shown in source 14.

A black screen with yellow text

Description automatically generated

Source 14: Levels Screen

After a couple seconds delay, the screen changes again into the game screen, where the actual game is played, as shown in source 15. On this screen, the player is able to place towers and fight off the enemies.



Source 15: Game Screen

If the player does end up losing by the enemies getting to the end of the path, the game over screen is shown, which shows the player what level they lost on. An example shown in source 16.

A screenshot of a video game

Description automatically generated

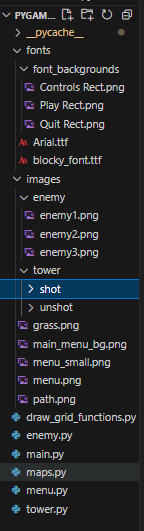
Source 16: Game Over Screen

**Answering the Project Requirements**

**2) Program Control Structures**

The game has appropriate control structures implemented, such as loops and conditionals, which manages the flow of the game.

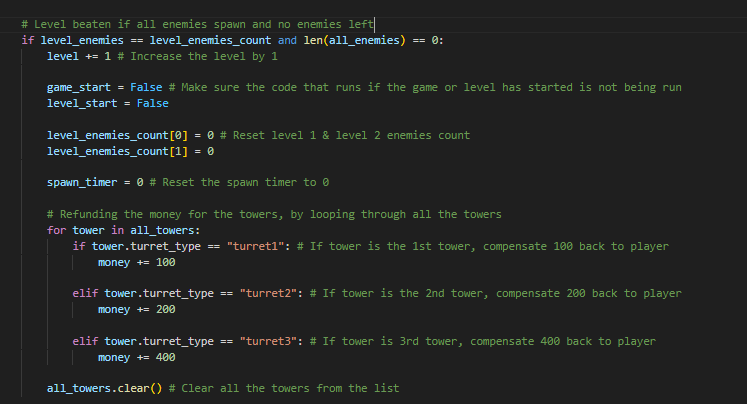
The code is also organised into multiple files, as shown in source 17, which shows that the code is organised into the different categories of the game.



Source 17: The multiple files for the game

The code also uses variables with the conditionals and loops to enhance the game flow, where such use can be seen when switching through the screens.

For example, the game checks if a player finishes a level by seeing if the enemies that the game was meant to spawn is equal to the amount of enemies actually spawned that round, and if there are no enemies currently alive in the game. If both terms are true, then that means the level has been beaten. The snippet of the code is shown in source 18.



Source 18: Code snippet for conditionals

The code also follows a logical structure, where even though it cannot be proven through code snippets, the code follows a structure where certain parts of the code are run if certain variables are true and false, this can stop certain blocks of code running when they are not meant to, enhancing the user experience.

**3) Data Types and Variables**

The game utilises a variety of data types including strings, integers, lists and dictionaries. Examples of the use of strings include displaying messages on the screen, such as for the main menu, and for the level screen. Example of a code snipped is shown in source 19.

A screen shot of a computer program

Description automatically generated

Source 19: Example of a code snippet where strings are used

As for the integers, there are multiple uses of integers in the game. Some include holding the amount of money the player has, holding the health of the player, and all the constants to work out the mathematics. An example of a code snippet is shown in source 20.

A black background with white text

Description automatically generated

Source 20: Example of a code snippet where integers are used

For the lists, the main example is the actual map, which is a 2-D grid. Every element in the 2-D list corresponds to a row in the actual map, where every sub-element in every sub-list corresponds to a singular tile in the game. An example of the grid is shown in source 21.

A screenshot of a computer screen

Description automatically generated

Source 21: Examples of where lists are used

Finally, I have used dictionaries to store the different pictures for the different upgrade levels of the towers. Dictionaries have also been used for the pricing of the different levels, which show how much it costs to upgrade a tower to its next level. An example of the use of the dictionary is shown in source 22.

A screenshot of a computer screen

Description automatically generated

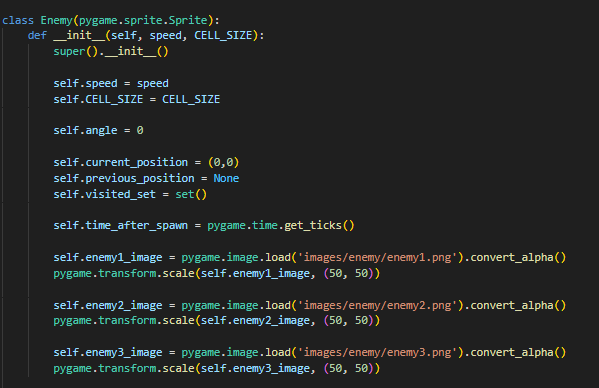
Source 22: Example of where dictionary is used

As shown in the images, the variables are also aptly named, with variable names describing exactly what the variable holds, and have been used to store game-related information.

**4) Modular Coding**

Modular coding describes the technique of separating the functionality of a code of a program into independent, interchangeable modules. The code of the game is organised into functions and classes, where each specific function and class has a specific task and purpose to the game.

An example of a class is the enemy class, which holds all the information related to the enemy. A code snippet of the enemy class is shown in source 23.

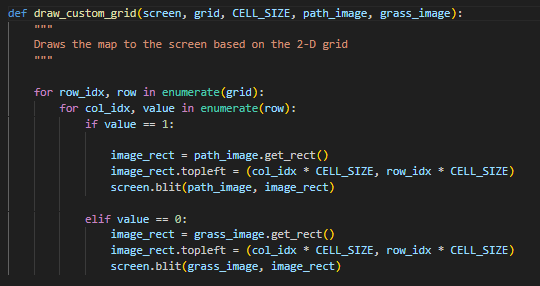


Source 23: A code snippet of the enemy class

As for functions, there are functions inside of the classes, which are called methods, and there are functions outside of the classes. An example of a function outside the class that the game uses is the “draw\_custom\_grid” function, which draws the map onto the screen based on the 2-D grid provided in source 21. This function takes in the input parameters of the following;

* screen: To draw the tiles to the screen so the player can see
* grid: To use the grid to draw the tiles to the screen
* CELL\_SIZE: To know how big each tile/cell is, so that they can draw in equal ratios on the screen
* path\_image: The image for when the cell value is 1 (Is a path block)
* grass\_image: The image for when the cell value is 0 (Is a grass block)

A code snippet of the function is shown in source 24.



Source 24: A code snippet of the “draw\_custom\_grid()” function

Some of the functions in the game have a return value, and others don’t. An example of a function used with a return value is “get\_firing\_rate” which is a method of the Tower class. This function/method gets the firing rate of the specific type of tower the tower is. This allows the different towers to shoot at different speeds. Finally, based on what tower it is, the function returns the firing value as an integer, which can be assigned to a variable by calling the function in the main loop. An example of the function with a return value is shown in source 25.

A computer screen shot of text

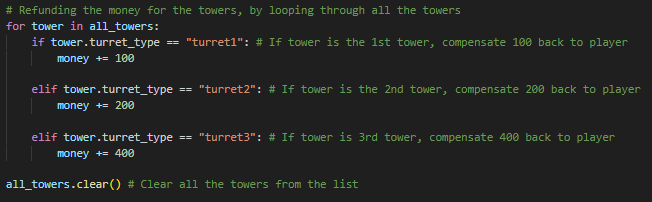
Description automatically generated

Source 25: A code snippet of the “get\_firing\_rate()” function

**5) Operators**

The game uses a combination of arithmetic, relational and logical operators within the code, to contribute to the game’s functionality and logic.

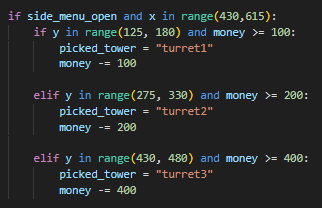
Arithmetic operators are performing basic mathematical operations on numerical data, such as addition, subtraction, multiplication, division, modulus, exponentiation, and floor division. Such arithmetic operations have been used in my code, to update the player’s money when they place down a tower, or to compensate them for their money at the end of a round. A code snippet is shown in source 26.



Source 26: A code snippet showing use of arithmetic operators

Relational operators, also called comparison operators, are used to compare two statements. The relation operators in Python are “equal to” (==), “not equal to” (!=), “greater than” (>), “greater than or equal to" (>=), “less than” (<) and “less than or equal to" (<=).

Such relational operators have been used in my code, where it has been used to check if the player has enough money to buy a tower. All relational operators have been used in the code, and an example is shown in source 27.



Source 27: A code snippet showing the use of relational operators

Logical operators are used to combine multiple conditions together and evaluate them as a single boolean expression. There are three types of logical operators in Python; “and”, “or” and “not”. These logical operators have been used in my code, where “and” has been used to check if the player has won the round or not based on if all the enemies have been spawned AND there are no enemies left, a code snippet is shown in source 28.

A screen shot of a computer code

Description automatically generated

Source 28: Example use of the “and” logical operator

“or” has been used for the random enemy generator, so that a level 1 enemy has a higher chance of spawning then a level 2 enemy. A code snippet is shown in source 29.

A computer screen shot of text

Description automatically generated

Source 29: Example use of the “or” logical operator

Finally the “not” operator, which has been used if the main menu is not open, so the player can select a turret, a code snippet is shown in source 30.

A black background with white text

Description automatically generated

Source 30: Example use of the “not” logical operator

**6) Data Structures**

In Python’s context, arrays are basically lists. The lists in the game code do store and manipulate game data, where for example, there is a list for holding all the enemies and all the towers, so that it can keep track of how many enemies and towers are currently in the game. This is especially useful when drawing the enemies and towers to the game, and the code can loop through this list to draw each sprite. An example of the lists is shown in source 31.



Source 31: Example of the use of lists that store & manipulate game data

As for the dictionaries, there are multiple dictionaries that store and manipulate game data, where for example, there is a dictionary that stores all the images for every upgrade level of the tower. This manipulates game data as it changes the image of the tower based on what level the tower is. An example of the dictionary is shown in source 32.

A black background with white text

Description automatically generated

Source 32: Example of the use of dictionaries that store & manipulate game data

**7) Development Processes and Good Programming Practices**

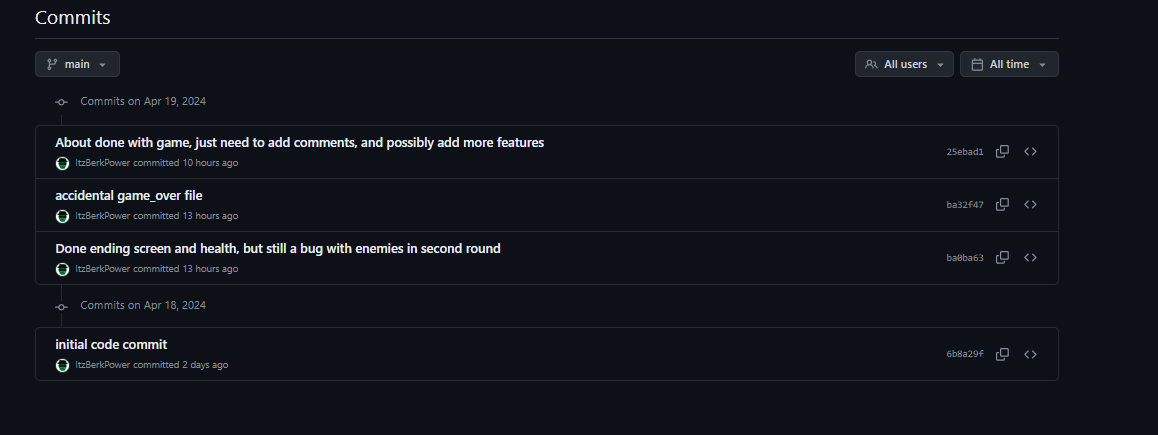
- Do mainline

- One task per subroutine

As for the use of stubs, stubs are code blocks that were meant to implemented if the programmer had more time to implement them in the code. They show evolution of what the product could’ve been if the programmer had more time

* Add the stubs

The version control was done through Github, where once any change was made to the code and was working, the code was pushed to Github to show the new version of the code. At the point in time writing this project, there are 4 versions of the code, each with a brief description of what was changed, an example is shown in source X.



Source X: Showing the different versions of the code

**Social and Ethical Issues Related to Game Development**

There are many social and ethical issues related to game development, some of the social issues include;

* **Representation and Diversity:** Where games have struggled with representing diverse characters & perspectives, often perpetuating stereotypes or even excluding marginalised groups
* **Toxicity and Harassment:** Online gaming communities can be breeding grounds for toxic behaviour, including harassment, bullying, and hate speech. This can drive away players from games, and contribute to broader online toxicity.
* **Addiction and Health Concerns:** Excessive gaming can lead to addiction and have negative impacts on physical and mental health.

Some of the ethical issues include;

* **Content and Themes:** Game developers must make ethical decisions regarding the content and themes depicted in their games. Includes considerations of violence, sexuality, drug use, and other sensitive topics.
* **Labour Practices:** Game development can lead to long hours, tight deadlines, and job instability. Ethical game development involves treating game developers fairly and ethically, ensuring reasonable work hours, fair compensation, and a healthy work-life balance.
* **Accessibility:** Ethical game development ensures games are accessible to players of all abilities.

**Responsible Coding Practices**

Responsible coding practices are ensured in this project through the requirements, though they also include:

* Writing clear, consistent, and self-explanatory code
* Conducting code reviews to identify bugs, improve code quality, and share knowledge among team members.
* Follow secure coding practices to ensure protection against threats such as injection attacks and data breaches. Can be done by validating user data.
* Writing code that is efficient, and optimised for performance. Consistently check the code to identify bottlenecks, and optimise critical sections.

**8) Algorithm Representation and Code Structure**

**9) Error Detection and Correction**

The code utilises robust error handling mechanisms for syntax, logic, and runtime errors.

**Syntax Errors:**

For the syntax errors the code has been thoroughly tested for errors and have been corrected. Sometimes the errors were runtime errors, which can be easily identifiable through the terminal, though other times it was logical errors.

In order to identify logical errors, print statements were used, to track the state of variables throughout the code. This allowed to get a general idea of how the variable was doing in comparison to what it should be doing in the game.

Though these print statements were later removed to improve the cleanliness and readability of the code, though an example of this is shown in source X

A black background with text and symbols

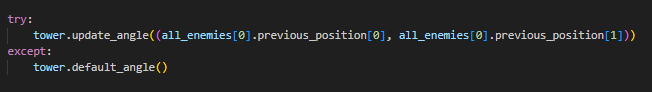
Description automatically generated with medium confidence

Source X: Use of a print statement to track the state of a variable through the game

**Runtime Errors:**

For runtime errors, try-catch blocks are used to make sure that if an error was to arise, the “catch” block would catch the error, and instead of throwing a runtime error, it would allow the code to keep going, and run the code inside of the “catch” block. This was used in the code where the tower tries to update its angle based on the enemies position, but if there is no enemy in the enemies list, it will throw an IndexError, as the code is trying to access an element that doesn’t exist. If it does throw this error, the code will catch it, and instead just keep it at its normal angle.

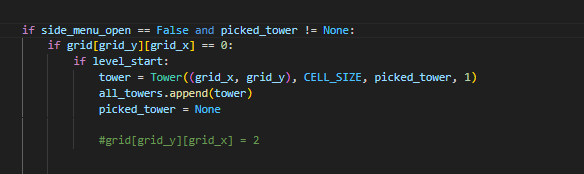
An example of this is shown in source X.



Source X: Example of use of “try-catch” block

**Logical Errors:**

As for the logical errors, these are errors where the code runs without any syntax or runtime errors, but produces incorrect results due to flawed logic in the code. Such error can occur through user errors, where input validation is important. User input can be validated through conditionals, where in my code, I have validated user input to check if the player tries placing a tower while the menu is still open, which can cause the tower to be accidentally placed on the menu. The same statement also includes if the player tries to place a tower without actually picking one. An example of this is shown in source X.



Source X: Example of the use of user input validation

**10) Testing With Data**

**Testing With Data**

A lot of data has been tested with the game, including appropriate test data that covers all pathways boundary conditions, and known outcomes.

Some examples of this data includes:

* X do test data

**Range and Type Checking**

Range and type checking have been used for all inputs in the game. There is limited checking as there is only one type of user input, which is the mouse button down. Examples of the range checking has been shown in source X, where it checks if the area the player clicked was actually inside of the button box.

A computer screen with text

Description automatically generated

Source X: Example of the use of range checking

An example of the type checking has been shown in source X, where it checks if grid\_y and grid\_x, which is the square the player clicks, is actually an integer so that it could be used to track the exact cell down in respect to the grid.

A computer screen shot of text

Description automatically generated

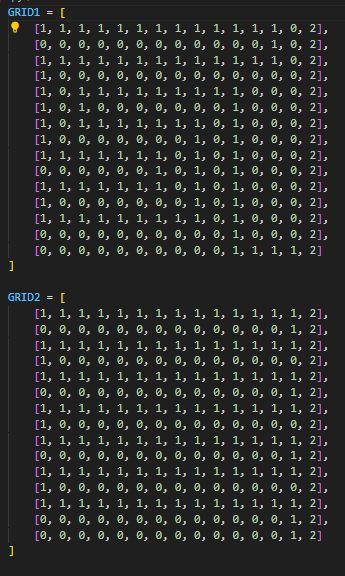
Source X: Example of the use of type checking

**Desk Checking:**

Desk checking has been used for the algorithms mentioned above in topic 8,

**DFS Algorithm:**

Algorithm has been tested with different maps, as shown in source X, where each map has a different layout, to see if the enemies are still able to traverse through the map with ease.



Source X: Different maps to test the DFS algorithm

Process:

The process was to first make the different maps, where the programmer’s imagination was the only limit. Once the maps were made, the actual in-game map was replaced with these maps to check if the enemy can still successfully traverse through the map.

Results:  
No matter what the map is, the enemies in level 1 & level 2 were able to successfully traverse the map.

**Reflecting on the Testing Process:**

Overall, the testing process was effective, as it helped uncover some bugs. As for the chosen test data, they were pretty ineffective, as the only type of user input was clicking, and on the screen there is only a certain amount of places the player can click, deeming the test data ineffective.

As for the desk checking, it allowed for the understanding where the logical errors where in the algorithms.

For the DFS algorithm, the error was where the cells in-front of the enemy were being added to the “visited” cell before they were actually visited, and since there is a validation stopping the enemy from going to a cell they already visited, the enemy was unable to go to the next cell. This was fixed by adding the cells to the visited set after they actually visited them, so that any cells in front could not be added.