Table of Contents

[Section 2 – Design: 1](#_Toc452564653)

[Problem decomposition: 1](#_Toc449098961)

[Modular overview: 1](#_Toc1262054784)

[Main Menu Flow Diagram: 1](#_Toc1329578954)

[Main Game Loop Flow Diagram: 1](#_Toc830479775)

[Game Features Flow Diagram: 2](#_Toc1985130731)

[Procedural Generated Map Flow Diagram: 2](#_Toc739130157)

[Randomly Generated Maze Flow Diagram: 3](#_Toc585788746)

[Description of components: 4](#_Toc73880348)

[Main Menu: 5](#_Toc290755414)

[Game Features: 6](#_Toc1061463483)

[Game Loop: 7](#_Toc732545268)

[Procedural Generation Level: 9](#_Toc1343216539)

[Random Generation Maze: 11](#_Toc912791061)

[Algorithms: 12](#_Toc47133993)

[Collision Detection Algorithm: 13](#_Toc439366135)

[Movement Algorithm: 13](#_Toc390771467)

[Shooting Mechanic: 15](#_Toc2002108495)

[Interface Design 17](#_Toc619028641)

[Screen Designs 20](#_Toc1834036278)

[Main Menu 20](#_Toc2018241228)

[Difficulty Select: 21](#_Toc1247588814)

[Load Game: 22](#_Toc2048145591)

[Settings: 22](#_Toc952177678)

[Graphics: 23](#_Toc1824458828)

[Sound: 24](#_Toc727404249)

[Keybind: 25](#_Toc766962380)

[About: 26](#_Toc1883921681)

[Data Requirements 27](#_Toc566311161)

[Test Strategy 32](#_Toc1955536953)

[Test Strategy 32](#_Toc1564854229)

[Test Plan 32](#_Toc328063584)

[Test Data 33](#_Toc707782735)

[Post Development Testing 35](#_Toc1026672300)

# Section 2 – Design:

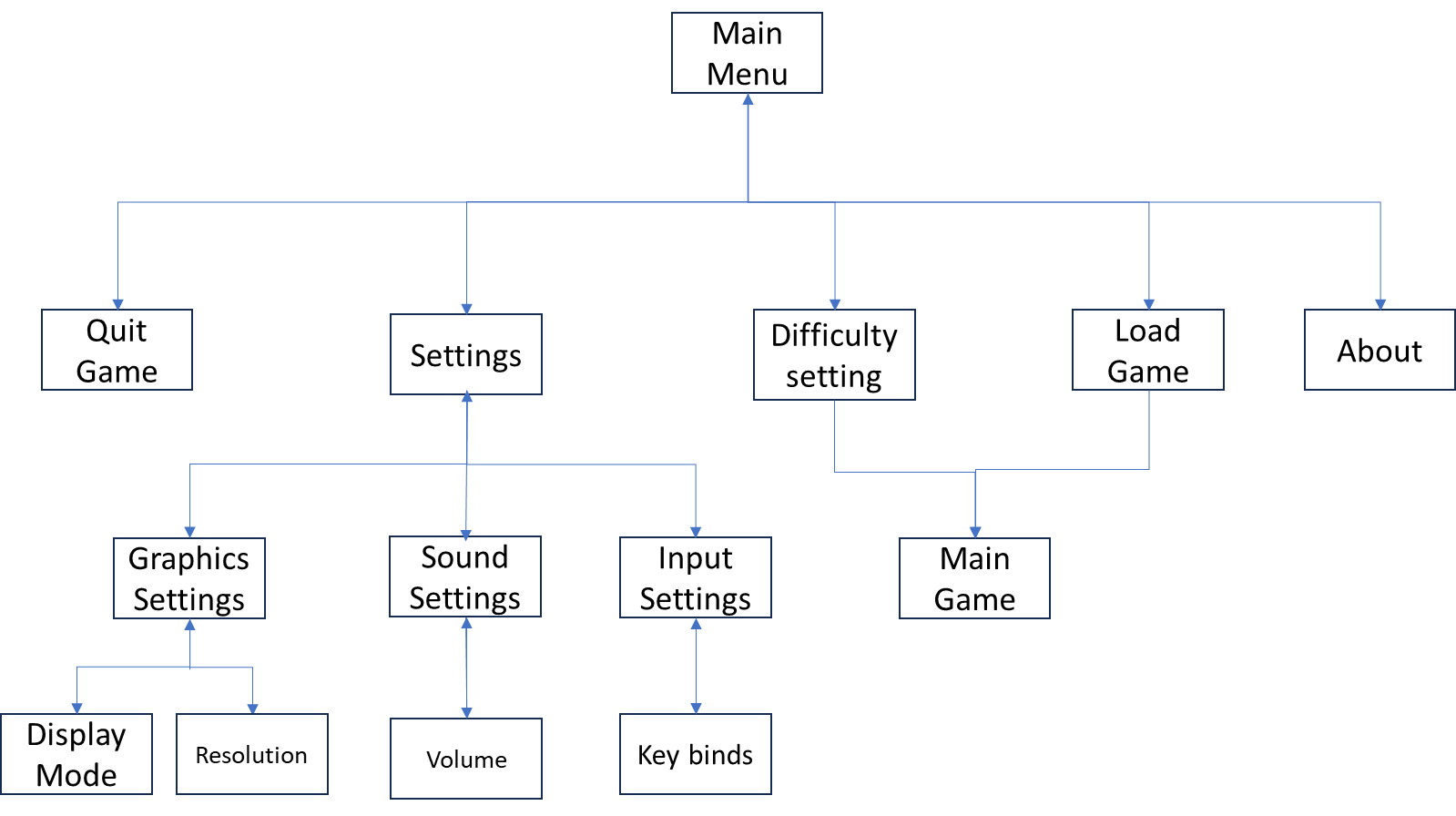
## Problem decomposition:

I will use decomposition to break down my overall project into a series of smaller and more manageable tasks. Doing this will make the development process much easier to follow as smaller problems are easier to solve than large complex problems.

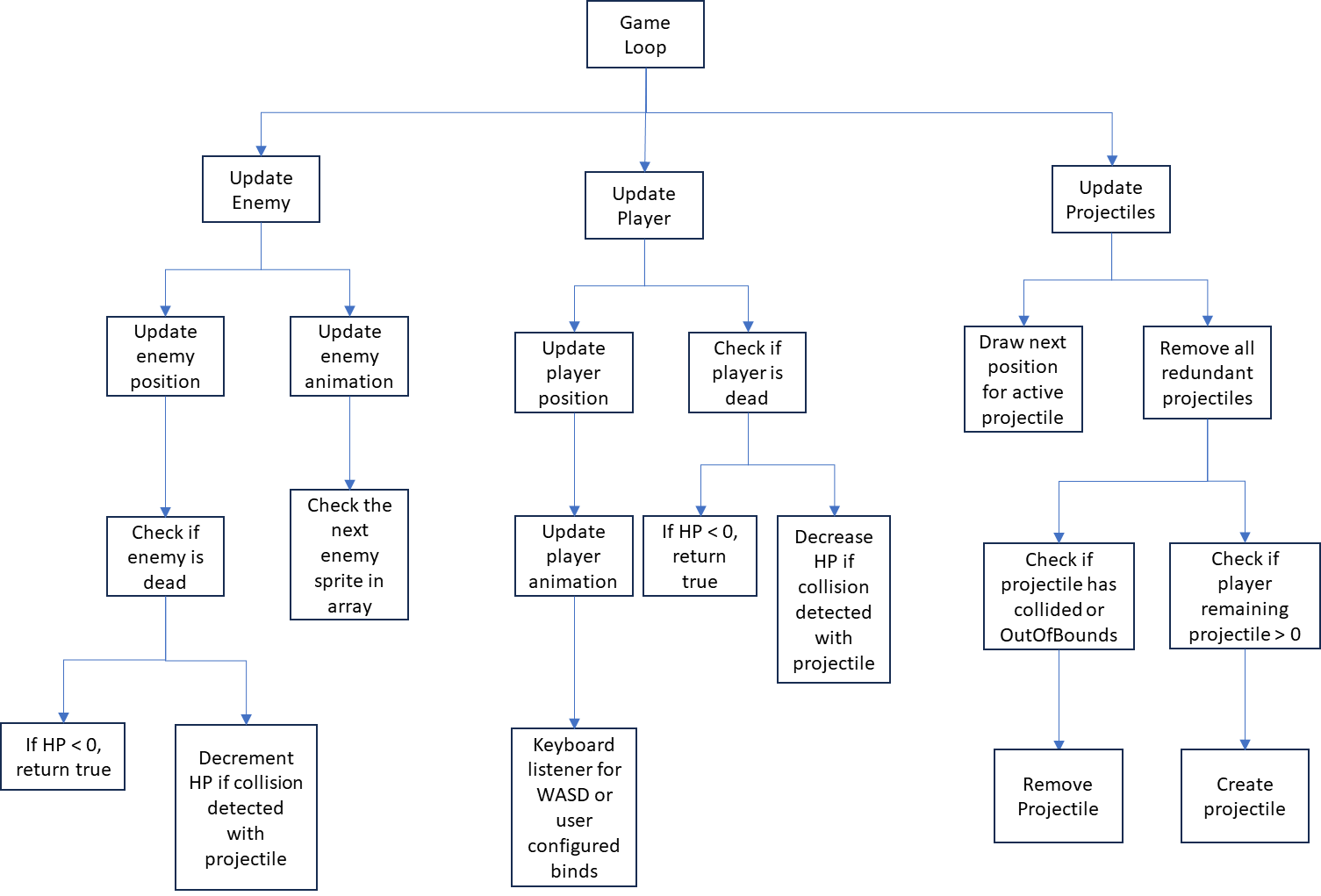
### Modular overview:

I will utilize a top-down design approach to assist me in the design process, as it will provide me with an overall view of different tasks that have to be developed.

#### Main Menu Flow Diagram:



#### Main Game Loop Flow Diagram:

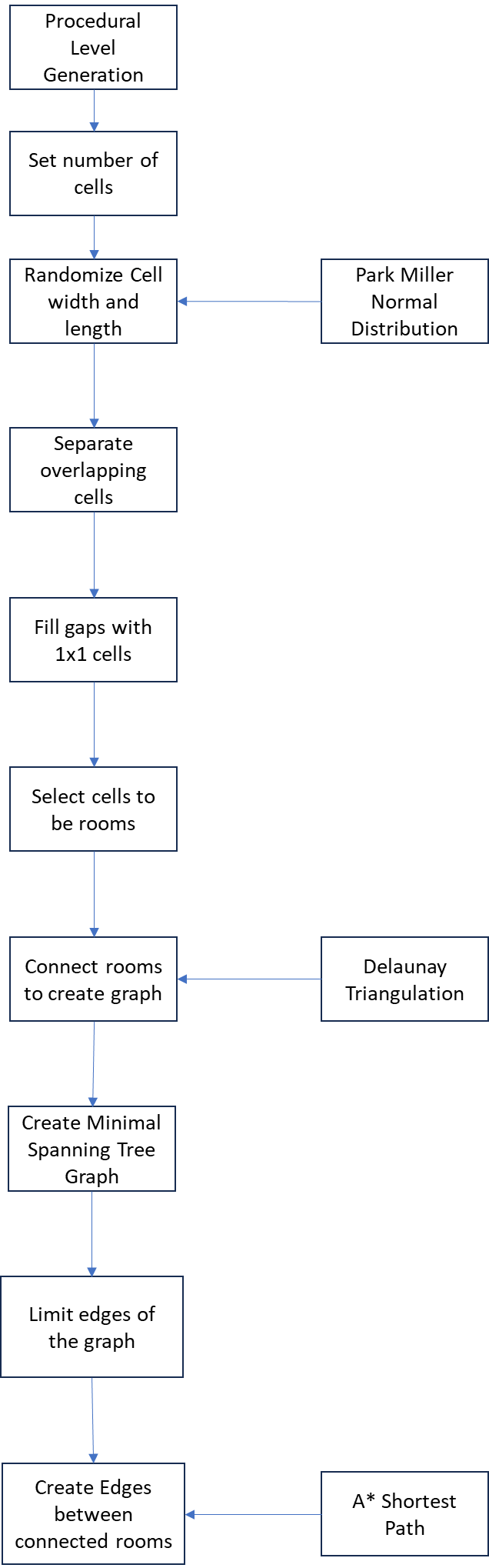


#### Game Features Flow Diagram:

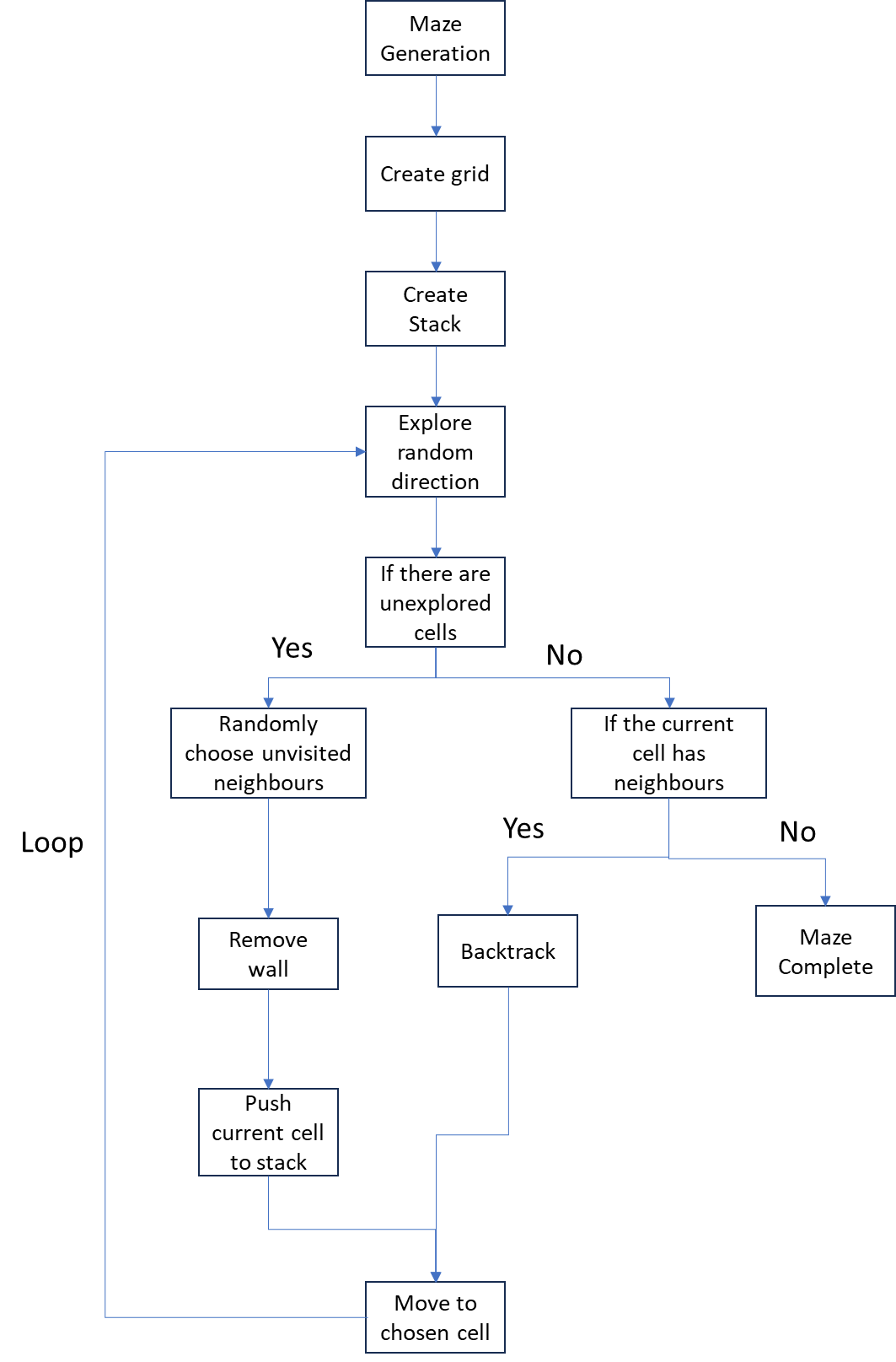
A diagram of a game

Description automatically generated

#### Procedural Generated Map Flow Diagram:



#### Randomly Generated Maze Flow Diagram:



## Description of components:

I have broken down the design of the game so that it will be much easier to design each stage of the game, such as the main menu, game features, game loop and level generation. The overall game will encapsulate the main menu, game features, game loop and level generation to deliver the overall game.

### Main Menu:

**Main Menu** – The user is displayed a GUI main menu which allows them to display 5 buttons to start the game, load previous game saves, change settings, learn more about the game and Quit game.

* **Start Game with Difficulty Selection** – The "Start Game" button with a subsequent difficulty selection scene is essential for catering to users with different skill levels and preferences.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Difficulty Level | Enemy Health | Enemy Spawns | Ammo Availability | Weapon Availability | Health Availability | Level Size | Perma-death |
| Easy | 75 HP | Very Low | Very Common | Very Common | Very Common | Small | No |
| Medium | 100 HP | Low | Common | Common | Common | Small | No |
| Hard | 200 HP | Many | Uncommon | Uncommon | Uncommon | Medium | No |
| Surviour | 400 HP | Various | Rare | Rare | Rare | Large | No |
| Grounded | 600 HP | Various | Very Rare | Very Rare | Very Rare | Large | Yes |

Providing multiple difficulty options allows users to tailor the game's challenge to their comfort level, ensuring both beginners and experienced users can enjoy the game at their own pace. This feature contributes to the inclusivity of the game, making it accessible to a broader audience and providing a more satisfying and personalized gaming experience.

* **Load Previous Game Save** – The "Load Game Save" button is crucial for enhancing the user experience. This feature allows users to resume their game from a previously saved point, providing a sense of continuity and enabling them to pick up where they left off. This is particularly valuable as it allows users to avoid starting over and maintain their investment in the game.
* **Change Settings** – The "Settings" button is essential for accommodating the diverse preferences and requirements of users. This feature allows users to customize aspects such as graphics, sound, controls, and other game-related settings. Providing these options ensures that the game can be tailored to individual preferences, making it more accessible and enjoyable for a wider audience with varying hardware configurations and personal preferences.
* **About** – Including an "About" button is important for fostering user engagement and comprehension. This button can lead to a section providing information about the game's mechanics, and any other relevant details. By offering additional context and background information, users can enhance their understanding of the game world, making their overall experience more immersive and enjoyable.
* **Quit Game** – The "Quit Game" button is important for allowing the user to leave the game application. For instance, if the user is in Fullscreen mode, the user can’t press the close window button to quit the application, so they would have to change from Fullscreen mode to windowed mode. Adding a Quit Game button will allow the user to quit the game without having to change game settings or have to press Windows and close through Task Manager. The user can also press ESC to Quit the game, but this only applies to the main menu.

### Game Features:

Having game features is vital to creating an engaging and immersive gaming experience. Level generation introduces variety and unpredictability, ensuring that players encounter new challenges and environments, enhancing replayability. Game modes provide diverse ways for players to interact with the game, this enhances variety and unpredictability, thereby broadening the appeal of the gaming experience. An inventory system adds depth by allowing the player to manage resources, enhancing strategic decision-making and progression. Lastly, enemies contribute to the dynamic nature of the game, introducing obstacles that the player must overcome, encouraging a sense of achievement and excitement. In combination, these features form the foundation of a well-rounded and captivating gaming experience, capturing and retaining the interest of players.

**Level Generation:**  
Level generation is an essential aspect of game design, offering many benefits that significantly enhance the gaming experience. Firstly, it introduces an element of unpredictability and variety, ensuring that each gameplay session is unique. This variability not only keeps the gameplay fresh and engaging but also increases replayability, as the player is presented with new challenges and scenarios. Moreover, level generation allows for the creation of expansive and dynamic game worlds, providing players with a sense of exploration and discover, adopting a more immersive and captivating gaming experience. Additionally, procedurally generated levels can adapt to different player skill levels, offering a scalable difficulty curve that caters to a broader audience. In essence, level generation serves as a cornerstone for creating rich, diverse, and endlessly entertaining game worlds.

**Game Modes:**  
Incorporating multiple game modes within a gaming experience adds layers of complexity and engagement, catering to a diverse range of player preferences and styles. The "Find the keycard " mode introduces an element of puzzle-solving and exploration, requiring players to carefully navigate the level. The level can be procedural, or Maze generated, enhancing their problem-solving skills. The "Find the exit" mode emphasizes strategic thinking and spatial awareness, challenging players to uncover the optimal path to reach their goal. The level can be procedural, or Maze generated. This enhances the strategical thinking by making it challenging for the user to traverse the level to find the exit. On the other hand, the "Kill all enemies" mode intensifies the action, promoting combat proficiency and resource management. The game mode can only be played on a procedural level. This is because the game can get frustrating for the user to traverse a maze to find small pockets of enemies, this makes the gameplay loop slower. By offering these distinct game modes, ensures a well-rounded and satisfying gaming experience that appeals to a broader audience. The variety in game modes not only adds depth to the gameplay but also enhances replayability as players can revisit the game with different objectives, keeping the experience fresh and engaging.

**Inventory System:**

The inclusion of an inventory system in a game serves as a fundamental aspect that enhances both immersion and strategic gameplay. An inventory system allows players to collect, manage, and utilize various items, adding a layer of depth to the game. Players can gather weapons, ammunition, health items and keycards throughout the game, fostering a sense of progression and accomplishment. This system promotes strategic thinking, as players must prioritize and manage their inventory space, choosing the most effective items for different situations. Moreover, an inventory system contributes to the overall narrative by connecting players with the in-game world, encouraging exploration and discovery. The inventory system not only adds a practical dimension to gameplay but also serves as a key element in shaping the player's journey and interactions within the game.

**Enemies:**

Enemies play a fundamental role in game design, contributing to the game by introducing challenges and a sense of accomplishment. Including enemies in a game adds an element of risk and reward, requiring players to strategize, adapt, and test their skills. Enemies provide a dynamic and interactive component, creating tension and engagement that is crucial for maintaining player interest. Their presence often drives the game forward, giving purpose to the player's actions. Additionally, defeating enemies offers a sense of achievement, reinforcing the player's progression and mastery of the game mechanics. Varied and well-designed enemies contribute to the game's replayability, ensuring that each encounter remains unique and poses distinct challenges. In essence, enemies are integral to creating a balanced, immersive, and enjoyable gameplay experience that captivates and motivates players throughout their gaming journey.

### Game Loop:

The Game loop is a central component in game development, responsible for orchestrating the game's real-time interactions and delivering a seamless and responsive gaming experience to users. It acts as the conductor of the game's components, ensuring that all game elements, including the user, enemies, and projectiles, come together within each frame of the game's execution. These frequent updates at the frame level enable users to witness the ongoing evolution of the game world and maintain a real-time visualization of the current game state.

Here's how the game loop is used to update these elements:

1. **Initialization:** Before entering the game loop, the game is initialized. This involves setting up the game world, loading assets, creating user and enemy objects, and defining the game's rules and mechanics.
2. **Entering the Game Loop:** Once the game is initialized, it enters the main game loop, which typically consists of three main stages: input processing, game logic update, and rendering.
3. **Input Processing:** The game loop starts by processing user input. This includes actions such as moving the user character, firing projectiles, and interacting with the game world. Input events are captured and translated into game actions.
4. **Game Logic Update:**
   * **User Update:** The game loop updates the user's character based on the actions received from the input processing stage. This includes moving the user's position, handling collisions with the environment, and responding to user commands. For example, if the user presses the "A" key (which can be configured in the settings), the user character's position is adjusted accordingly so that they would move left.
   * **Enemy Update:** Enemies are updated within the game logic stage. This involves AI routines, pathfinding, and behaviour updates. For instance, enemies may follow the user, react to changes in the environment, and make decisions about their actions, such as attacking or retreating.
   * **Projectile Update:** Projectiles, such as bullets, are updated. This includes tracking their positions, checking for collisions with other game objects (e.g., enemies or obstacles), and applying damage or effects upon impact.
5. **Collision Detection:** During the game logic update, collision detection is performed to identify interactions between game elements. For instance, the game loop checks if the user character collides with enemies if projectiles hit targets, or if objects in the game world collide with each other.
6. **Rendering:** After updating the game logic and detecting collisions, the game loop proceeds to the rendering stage. This is where the game's visuals are updated to reflect the current state of the game. The user character, enemies, projectiles, and other elements are drawn on the screen, creating the visual representation of the game world as it exists after the most recent game logic update.
7. **Repeating the Loop:** The game loop continuously repeats these stages at a high frame rate, typically 30 to 60 frames per second or more. This rapid iteration creates the illusion of smooth and continuous motion in the game. The user, enemies, and projectiles are continuously updated, and the game world responds to user input in real time.
8. **Game Over and Win Conditions:** The game loop also checks for game-over conditions, such as the user losing all their lives, as well as win conditions, defeating all enemies, or escaping to the next level. When these conditions are met, the game loop may exit or transition to a game-over screen or victory screen.

To enhance the efficiency and performance of the game loop, it's crucial to leverage multi-threading and integration with the OpenGL API, enabling distinct threads to handle specific tasks and offloading some of the computational workload to the GPU. This integration helps optimize both CPU and GPU usage, allowing for the concurrent execution of various game-related processes.

The decision to employ separate CPU threads and GPU cores for specific tasks is a strategic move aimed at achieving smooth gameplay. It protects against the risk of individual calculations or processes taking longer than anticipated, a scenario that can introduce latency in frame rendering. By distributing tasks across multiple threads and utilizing GPU resources effectively, this approach minimizes frame time latency, potentially leading to a boost in the overall framerate. The ultimate goal is to provide users with an uninterrupted and fluid gaming experience, ensuring that the game responds promptly to their actions and maintains a high level of visual fidelity.

### Procedural Generation Level:

To craft immersive game levels that captivate users with both complexity and aesthetic appeal, the procedural generation level algorithm follows a structured series of steps. These steps combine elements of randomness, geometry, and connectivity to ensure the creation of diverse and engaging game environments. By embracing these key principles, the algorithm paves the way for the generation of game levels that offer scalability, intricate layouts, and navigability, all while enhancing the overall gameplay experience.

Here is how I would develop the Procedural Level Generation algorithm:

1. **Set the Number of Cells:**
   * Determine the total number of cells to create.

Setting the number of cells to create is essential because it allows for scalability and the adjustment of map size, which is crucial for accommodating various gameplay scenarios and map complexities.

1. **Randomly Size Cells:**
   * Create cells with random width and length within a certain radius.
   * Apply Park-Miller Normal Distribution for size distortion.
   * Ensure reasonable width-to-length ratios.

Randomly sizing cells by generating random width and length within a specified radius is important to introduce diversity and complexity to the map layout. Applying the Park-Miller Normal Distribution for size distortion enhances the visual appeal and creates an irregular environment. Ensuring reasonable width-to-length ratios prevents overly stretched or squished cells, contributing to a more balanced map.

1. **Separate Overlapping Cells:**
   * Implement separation steering behaviour to eliminate cell overlaps.
   * Ensure cells are densely packed but not intersecting.

Implementing separation steering behaviour to eliminate cell overlaps is necessary for creating a visually appealing and traversable map. Overlapping cells can lead to confusion and hinder gameplay, making the separation step crucial.

1. **Fill Gaps with 1x1 Cells:**
   * Fill in any gaps left in the grid with 1x1 cells.
   * Result in a grid of variously sized, non-overlapping cells.

Filling gaps in the grid with 1x1 cells helps maintain the grid structure while optimizing space usage. This ensures that all available space is efficiently utilized, avoiding unnecessary gaps and dead areas in the map layout.

1. **Identify Rooms:**
   * Select cells with width and height above a specific threshold as rooms.
   * Typically, rooms are fewer in number compared to cells.

Identifying rooms by selecting cells with width and height above a specific threshold is vital for gameplay. Rooms serve as essential areas for users to explore, complete objectives, and engage with the game world, making them a valuable component of the map.

1. **Connect Rooms with a Graph:**
   * Create a graph of rooms' centre points using Delaunay Triangulation.
   * Link all rooms together without intersecting lines.

Creating a graph of rooms' centre points using Delaunay Triangulation and linking all rooms without intersecting lines is essential to establish connectivity between rooms. This step ensures that users can move between rooms, enhancing the overall navigability of the map.

1. **Build Minimal Spanning Tree:**
   * Construct a Minimal Spanning Tree from the graph of rooms.
   * Ensures all rooms are connected with the shortest possible corridors.

Constructing a Minimal Spanning Tree from the graph of rooms is crucial as it guarantees that all rooms are connected with the shortest possible corridors. This minimizes redundancy, streamlines navigation, and optimizes the layout for gameplay purposes.

1. **Introduce Loops for Variety:**
   * Reincorporate a limited number of edges from the Delaunay Triangulation.
   * Create loops in the layout for added complexity.

Reincorporating a limited number of edges from the Delaunay Triangulation and creating loops in the layout adds interest and non-linearity to the map layout. This step enhances gameplay variety and makes the map more engaging and dynamic.

1. **Convert Graph to Corridors:**
   * For each edge in the graph, create lines between connected rooms by using A\* Shortest Path.
   * Cells not used for rooms become corridor tiles where lines intersect.

Converting the graph to corridors by creating lines between connected rooms using the A\* Shortest Path algorithm is vital for gameplay. It transforms the abstract graph into tangible corridors that users can navigate, providing a structured environment for their in-game actions. The use of cells not designated as rooms for corridor tiles where lines intersect ensures that the map's design remains consistent and functional.

### Random Generation Maze:

To generate the maze, the recursive backtracker algorithm will be utilized. The algorithm leverages randomization, backtracking, and a stack data structure to generate mazes with natural branching patterns while exploring the entire grid.

Here is how I would develop the Maze Generation algorithm:

1. **Initialize the Maze:**
   * Start with an empty grid to represent the maze.

Starting with an empty grid to represent the maze is essential because it provides a clean slate for the maze generation process. This ensures that the algorithm begins with no pre-existing paths or obstacles, allowing it to construct the maze from scratch.

1. **Create a Stack:**
   * Set up a stack data structure to track the path being explored.

Setting up a stack data structure is a crucial part of the algorithm as it serves to track the path being explored. The stack is justifiable because it maintains a history of the cells visited, enabling the algorithm to backtrack when necessary. This history is pivotal for creating branching paths and preventing the maze from becoming a single linear route.

1. **Explore in Random Directions:**

While unexplored cells remain:

* + 1. **Randomly Choose an Unvisited Neighbour:**
       - Pick a neighbouring cell that hasn't been visited and isn't part of the maze.
    2. **Remove the Wall:**
       - Eliminate the wall between the current cell and the chosen cell.
    3. **Push Current Cell to Stack:**
       - Store the current cell on the stack.
    4. **Move to Chosen Cell:**
       - Make the chosen cell the new current cell.

While unexplored cells remain, the algorithm randomly chooses an unvisited neighbouring cell and removes the wall between the current cell and the chosen cell. This randomness is justified as it introduces the element of unpredictability and ensures that the maze features natural branching patterns, making it more challenging and interesting for users.

1. **Backtrack Until a Viable Path is Found:**

If the current cell has no unvisited neighbours:

**Backtrack:**

* + - * Pop a cell from the stack to backtrack.
      * Make it the current cell.

If the current cell has no unvisited neighbours, the algorithm backtracks by popping a cell from the stack and making it the current cell. This backtracking mechanism is justified because it prevents the algorithm from getting stuck in dead ends and facilitates the discovery of unexplored paths, ultimately contributing to the maze's completeness.

1. **Repeat Until the Maze is Complete:**
   * Repeat steps 3 and 4 until all cells in the grid have been visited.

Repeating steps 3 and 4 until all cells in the grid have been visited is essential to ensure a comprehensive exploration of the entire grid. This repetition guarantees that the algorithm exhaustively constructs the maze without leaving any areas unvisited or paths uncreated.

1. **Finish the Maze Generation:**
   * When all cells have been visited, the maze generation is complete.

When all cells have been visited, the maze generation process is considered complete. This final step is justified because it marks the endpoint of the algorithm's operation and signifies that the maze has been successfully generated with all cells connected and no unexplored areas remaining.

## Algorithms:

This section will highlight some of the algorithms that will be involved in developing the game.

### Collision Detection Algorithm:

The collision detection algorithm is a crucial component that ensures the proper interaction between different entities and objects within the game's 2D environment. Its fundamental role is to prevent objects from passing through each other. The algorithm would be called under a single function that would be recurringly called by the main game loop. For instance, when a user or an enemy fires a projectile, the algorithm continuously monitors the projectile's movement, halting its motion upon collision with either the user, enemy, walls, or other surrounding objects, thereby enforcing the game's rules and mechanics.

1. function RectangleCollide(rect1, rect2){

2. if (rect1.x + rect1.width > rect2.x && rect1.x < rect2.x + rect2.width) {

3. if (rect1.y + rect1.height > rect2.y && rect1.y < rect2.y + rect2.height) {

4. Return True

5. }

6. }

7. Return False

8. }

**Collision Decision:**

Lines 2-5:

* If both horizontal and vertical overlaps are true, the algorithm concludes that there is a collision.
* Returns True to indicate a collision.

Lines 6-7:

* If either horizontal or vertical overlap is false, the algorithm concludes that there is no collision.
* Returns False to indicate no collision.

The algorithm is designed to work with rectangles, but it can be adapted for other shapes such as squares or circles.

The collision detection algorithm operates with a time complexity of O(1), constant time. This efficiency is achieved because the algorithm's execution time remains constant, regardless of the number of the rectangles or the game environment. The algorithm's straightforward structure, involving basic arithmetic operations and conditional checks, ensures that the time required to determine the presence or absence of a collision is consistently low, making it well-suited for real-time applications such as games.

### Movement Algorithm:

The movement algorithm would be responsible for moving the user's sprite. The movement key binds can be edited by the user within settings and are stored under a configuration file. The algorithm would be called under a single function that would be recurringly called by the main game loop.

Below is a Configuration file which will store the key binds with an associated action. This will interact with the movement algorithm.

1. {

2. "moveUp": "W",

3. "moveDown": "S",

4. "moveLeft": "A",

5. "moveRight": "D",

6. "weapon": {

7. "aim": "MouseLeft",

8. "fire": "MouseRight"

9. }

10. }

**Key Bindings Configuration file (Lines 1-6):**

Lines 1-6:

* A JSON configuration file (config.json) contains key bindings associated with specific actions.
* Examples include "W" for moving up, "S" for moving down, "A" for moving left, and "D" for moving right.

1. function loadKeyBindings(){

2. configFile = open("config.json", "read")

3. keyBindings = parseJson(configFile)

4. }

5.

6. function movement(){

7. isMovingUp = isKeyPressed(keyBindings["moveUp"])

8. isMovingDown = isKeyPressed(keyBindings["moveDown"])

9. isMovingLeft = isKeyPressed(keyBindings["moveLeft"])

10. isMovingRight = isKeyPressed(keyBindings["moveRight"])

11.

12. userspeed = 2.0

13. if isMovingUp {

14. userY += userspeed

15. }

16. if isMovingDown {

17. userY -= userspeed

18. }

19. if isMovingLeft {

20. userX -= userspeed

21. }

22. if isMovingRight {

23. userX += userspeed

24. }

25. }

**Loading Key Bindings (Lines 1-4):**

Lines 1-4:

* The loadKeyBindings function reads key bindings from the configuration file.
* It attempts to open the file in read mode and parses its content into a variable called keyBindings.

**Movement Function (Lines 6-25):**

Lines 6-24:

* The movement function is called within the main game loop to update the game's state based on user input.
* It checks whether specific keys corresponding to user movement are currently pressed (W, A, S, D).

Lines 7-10:

* The isMovingUp, isMovingDown, isMovingLeft, and isMovingRight variables are set based on whether the respective keys are pressed.

Lines 12-23:

* The user's movement speed is defined as 2.0 units (userspeed).
* The user's position (userX and userY) is updated based on the values of the movement-related variables.
* If isMovingUp is true, the user's Y-coordinate is increased.
* If isMovingDown is true, the user's Y-coordinate is decreased.
* If isMovingLeft is true, the user's X-coordinate is decreased.
* If isMovingRight is true, the user's X-coordinate is increased.

The movement algorithm shows a constant time complexity, O(1), as it consistently performs key input processing and user position updates in a manner that is not dependent on the size of the game environment or the number of key bindings.

### Shooting Mechanic:

The shooting and aiming algorithm are a pivotal mechanism within the game, composing the interaction between the player's weapon and the game environment. Central to this algorithm is the coordination of aiming direction and projectile creation, triggered by user inputs. When the player initiates the aiming process, the algorithm dynamically computes the direction towards the mouse cursor to a designated target. Subsequently, upon the player's command to shoot, the algorithm instantiates projectiles along the computed trajectory, simulating the firing action.

This algorithm plays a continuous role in the game loop, actively responding to user input and ensuring the synchronization of aiming and shooting actions. Its iterative execution within the main game loop maintains real-time responsiveness, allowing players to precisely aim and fire projectiles. Much like the movement algorithm and the collision detection algorithm, the shooting and aiming mechanism is integral to enforcing the game's rules and mechanics, fostering an immersive and dynamic gaming experience.

1. public Weapon() {

2. loadKeyBindings();

3. }

4.

5. private void loadKeyBindings() {

6. JSONParser parser = new JSONParser();

7. FileReader configFile = new FileReader("config.json");

8. JSONObject jsonBindings = (JSONObject) parser.parse(configFile);

9.

10. JSONObject weaponBindings = (JSONObject) jsonBindings.get("weapon");

11.

12. keyBindings = new KeyBindings(weaponBindings);

13. }

14.

15. public void aim() {

16. if (isKeyPressed(keyBindings.get("aim"))) {

17. Point mousePosition = getMousePosition();

18. aimDirection = Math.atan2(mousePosition.getY() - player.getPosition().getY(),

19. mousePosition.getX() - player.getPosition().getX());

20. }

21. }

22.

23. public void shoot() {

24. if (isKeyPressed(keyBindings.get("fire")) && cooldown <= 0) {

25. Projectile projectile = new Projectile(aimDirection, player.getPosition());

26. GameWorld.addProjectile(projectile);

27. cooldown = 500;

28. }

29. }

30.

31. private boolean isKeyPressed(int keyCode) {

32.

33. }

34.

35. private Point getMousePosition() {

36.

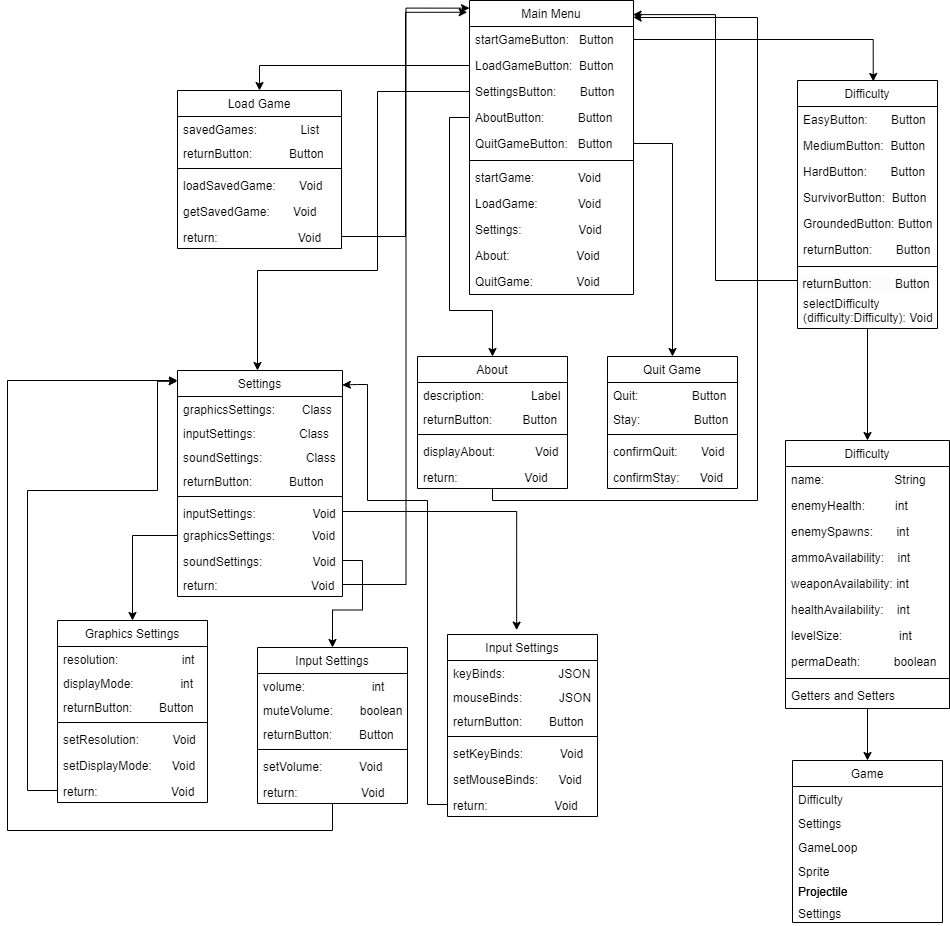
37. }

* **Lines 1-2:** The **Weapon** class constructor initializes a new instance of the **Weapon** class and immediately calls the **loadKeyBindings** method.
* **Lines 4-11:** The **loadKeyBindings** method is responsible for loading key bindings from a JSON configuration file ("config.json"). It uses the **JSONParser** to parse the JSON file, extracts the key bindings under the "[weapon](#_Movement_Algorithm:)" key, and creates a new **KeyBindings** object to store the loaded bindings.
* **Lines 13-18:** The **aim** method checks if the key corresponding to the "aim" action is pressed using the **isKeyPressed** method. If true, it calculates the **aimDirection** based on the mouse position relative to the player's position.
* **Lines 20-25:** The **shoot** method checks if the key corresponding to the "shoot" action is pressed and if the cooldown is not active. If true, it creates a new **Projectile** with the calculated **aimDirection** and the player's position, adds it to the **GameWorld**, and sets the cooldown.
* **Lines 27-28:** The **isKeyPressed** method checks if a specific key is pressed.
* **Lines 30-31:** The **getMousePosition** method will get the current mouse position.

The calculation of the aiming direction using trigonometric functions, which generally has a constant time complexity, denoted as O(1). Similarly, the instantiation of projectiles involves memory allocation and data structure manipulation, typically achieved with O(1) complexity, assuming efficient data structures. Processing user input, which includes checking for key presses or mouse clicks, is also commonly executed in constant time. As a result, the overall time complexity of the shooting and aiming algorithm is often considered constant, O(1). This is well-suited for real-time applications, so there wouldn’t be a performance impact on this algorithm.

## Interface Design

The interface for the project will allow the user to navigate easily. Thus, a minimalistic User Interface is ideal. The UI should have bold contrasting colours and large lettering to cater to users with poor vision. I have already created an initial Flowchart of the Main Menu, which I have expanded upon by using a UML class diagram. The class diagram shows the key methods and attributes of the main classes and how they link to each other. The diagram in the [Modular Overview](#_Main_Menu_Flow) Section can be integrated into the class diagram, as scenes are represented by separate classes. Each class includes attributes such as buttons, Labels, and checkboxes with their required methods in code. Using the UML Diagram, a section of the class diagram can be made to show scenes and which scenes link to each other.



In the diagram, the Main Menu is shown to the user on startup. The user can 8 different scenes: Start Game, Select Difficulty, Load Game, Settings, About and Quit. The user will first be presented with 5 buttons at the start: Start Game, Load Game, Settings, About and Quit.

While in the main menu, the user can navigate across the menu by using either keyboard inputs by pressing the Arrow Keys or WASD, ESC or they can move their mouse cursor to the respective buttons. This dual input functionality has been integrated to enhance the game's accessibility and user-friendliness. By offering both methods of navigation, the game caters to a diverse range of user preferences and accommodates various hardware configurations. This approach ensures that users can choose the input method that best suits their needs, contributing to a more inclusive gaming experience. It also aligns with the principle of user familiarity, as many gamers are accustomed to using both keyboard and mouse inputs in different games.

**Start Game** – When the user presses the button, the scene will change to the Difficulty Scene through the "**selectDifficulty**" method. The Difficulty Scene shows 5 different difficulties ([Easy, Medium, Hard, Survivor,](#_Main_Menu:) Grounded) which the user can select. When the user hovers over a difficulty option, a small description will be provided to help the user select what difficulty they want to play. A difficulty selection is necessary so that users can enjoy the game by catering to their different skill levels.

**Load Game** – When the user presses the button, the scene will change and the user straight into the game. This is useful because the user would likely become frustrated if they couldn’t save their game and would have lost all their progress on a previous playthrough if they didn’t die and would remove the replayability factor of the game. Implementing a load game feature allows the user to continue from where they left off, making the game more user-friendly and accommodating longer play sessions.

**Settings** – When the user presses the button, the scene will change to another scene which will display 2 buttons: Graphics Settings, Sound Settings and Input Settings.

**Graphics Settings:**

When the user presses the Graphics Settings button, the scene will change again via the "**GraphicsSettings**" method. The user has the option to change their Screen Resolution from 640x480 to 1920x1080 and Display Mode which can be Windowed, Fullscreen or Borderless Fullscreen. This ensures usability and accessibility as the screen size must be large enough to display the scene at a correct ratio and may need to be resizable or full screen to allow compatibility for all displays.

The user can press ESC or click the bottom left button to go back to the settings scene.

**Sound Settings:**

The user can access Sound settings by navigating to the “Sound Button”, the scene will change again via the "**SoundSettings**" method. The user has the option to change the game volume by either adjusting the slider or typing in a value between 0-100. This allows for usability and accessibility for the user as they have the freedom of adjusting how loud they want to have their game audio. Due to the game being in a school environment, the audio of the game is likely to be muted, so there is a dedicated mute button, so that other users are not distracted while you are playing the game.

The user can press ESC or click the bottom left button to go back to the settings scene.

**Keybind Settings:**

The user can access Keybind settings by navigating to the “**Keybind**” Button, the scene will change via the “**KeybindSettings**” method. Key binds will be listed with their associated in-game action. This allows them to customize their gaming experience by adjusting key bindings, making the game more user-friendly and less confusing for users who haven’t played the game yet and have immediately gone straight to settings.

The user can press ESC or click the bottom left button to go back to the settings scene.

Using 4 scenes for accessing graphics, sound and input settings makes it easier for users to navigate menus without getting lost, making it quicker for users to get into the game.

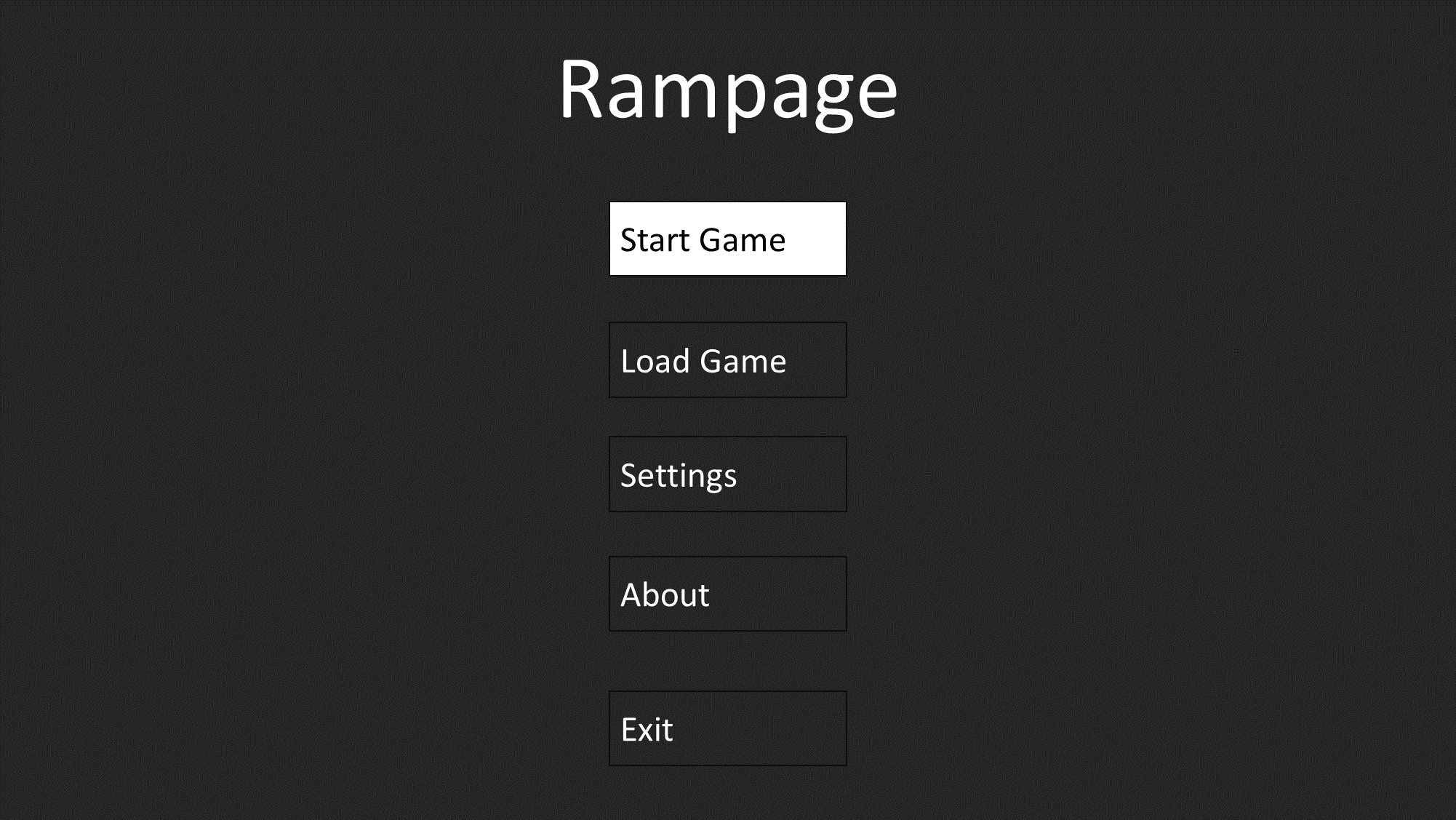
**About** – When the user presses the button, the scene will change to another scene which will display information about how to play the game by providing the user with straightforward game instructions. This feature enhances the user experience by offering clarity on game mechanics, objectives, and any specific rules or controls. It ensures users quickly understand the game, reducing frustration and improving enjoyment for both new and returning users.

**Quit** – When the user presses the button or presses ESC while in the Main Menu, the scene will change to another scene which will display 2 buttons if they are sure they want to quit the game or stay in the game. A "Quit Game" button with confirmation is essential to prevent accidental game exits and to ensure a smooth user experience. Users might accidentally press the quit button, leading to frustration and potential loss of progress. The confirmation dialogue provides a safety mechanism, requiring users to confirm their intent to exit the game. This feature minimizes the risk of accidental quits and offers a user-friendly way to handle game closure, enhancing overall satisfaction and preventing user frustration.

### Screen Designs

The following are the ideal designs for the Main Menu. These may be changed throughout the development stage as I will be in contact with the client to talk about the progress of the project and their needs might have changed from the initial start of the project to the next time they are informed about the project. The CPU alone can handle the main menu screens as the main menu is not resource intensive unlike the main game which will definitely need to use multi-threading and GPU support. Therefore, JavaFX can be used to design the main menu screens as the CPU is only having to handle buttons, labels, colours, and keyboard inputs.

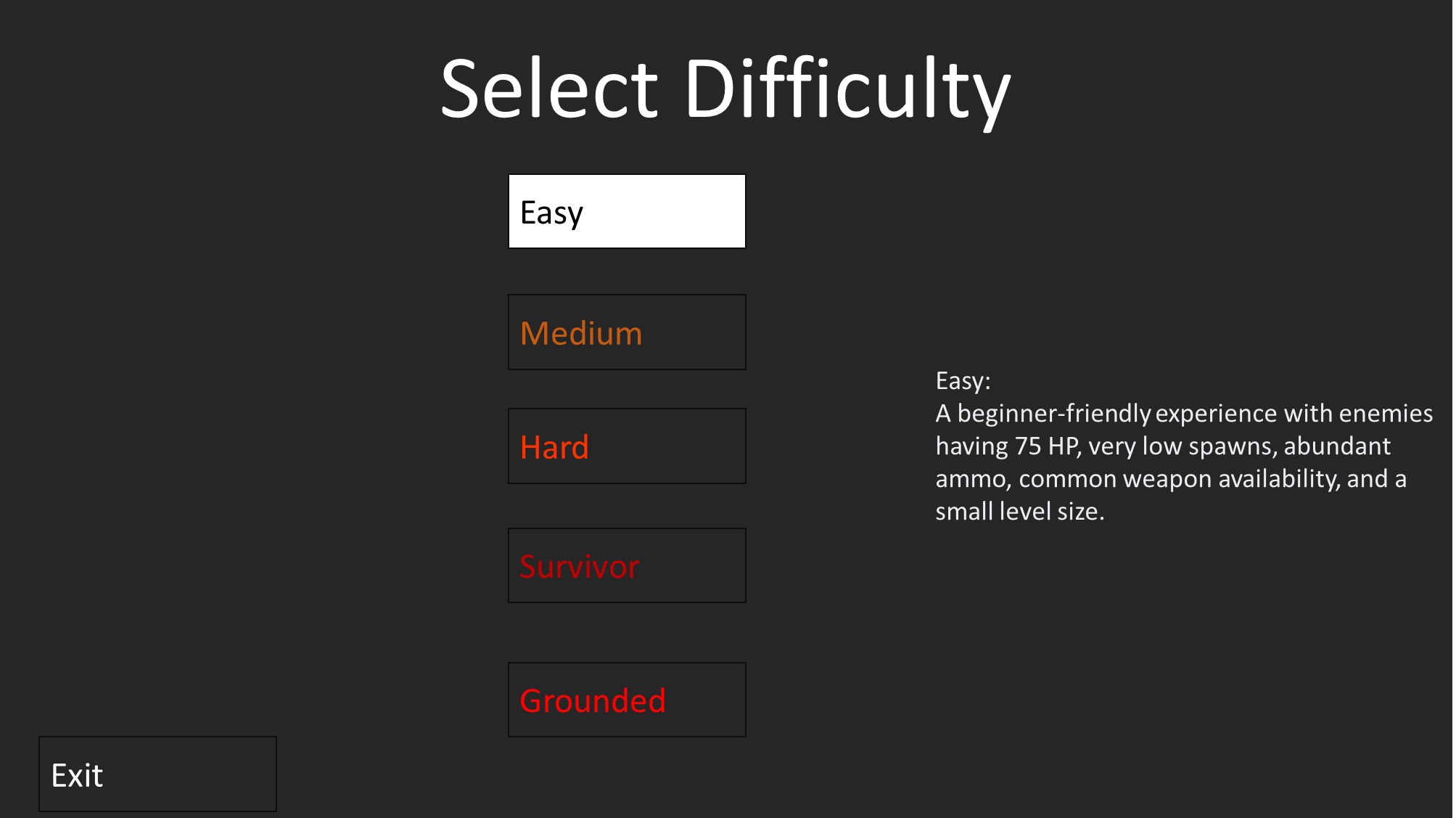
### Main Menu



Usability:

The main menu serves as the central hub for users to access essential sub-menus required to play the game. Its user-friendliness is ensured by the presence of clear and distinct buttons, each representing a different sub-menu. This setup minimizes user confusion and streamlines navigation, saving users valuable time. Users have multiple means of interaction, allowing them to use the mouse to hover over buttons or utilize the keyboard (WASD or Arrow keys) for efficient navigation. The dynamic button colour change upon hovering provides clear feedback to users, signalling both the button's interactivity and the specific button they are interacting with. This feature enhances accessibility and ensures that users can easily discern which buttons are clickable and which scenes they can access, contributing to a more intuitive and enjoyable user experience.

### Difficulty Select:



Usability:

Upon pressing the "Start Game" button, users are seamlessly transitioned to the Difficulty scene. Similar to the main menu, this scene is designed with user-friendliness in mind. Users can employ mouse interaction or keyboard inputs (WASD or Arrow keys) to navigate across the scene. The intuitive button colour change feature remains consistent, offering visual cues to users. A small description will be shown when you hover over a difficulty. This is so that the user can decide what difficulty that they want to use. These user-centric design choices guarantee that users can quickly understand and execute actions within the game, enhancing overall accessibility. When the user clicks on their desired difficulty, there will be a confirmation scene to make sure that this is the difficulty that the user wants to play on. This reduced the likelihood of a user accidentally pressing the wrong difficulty.

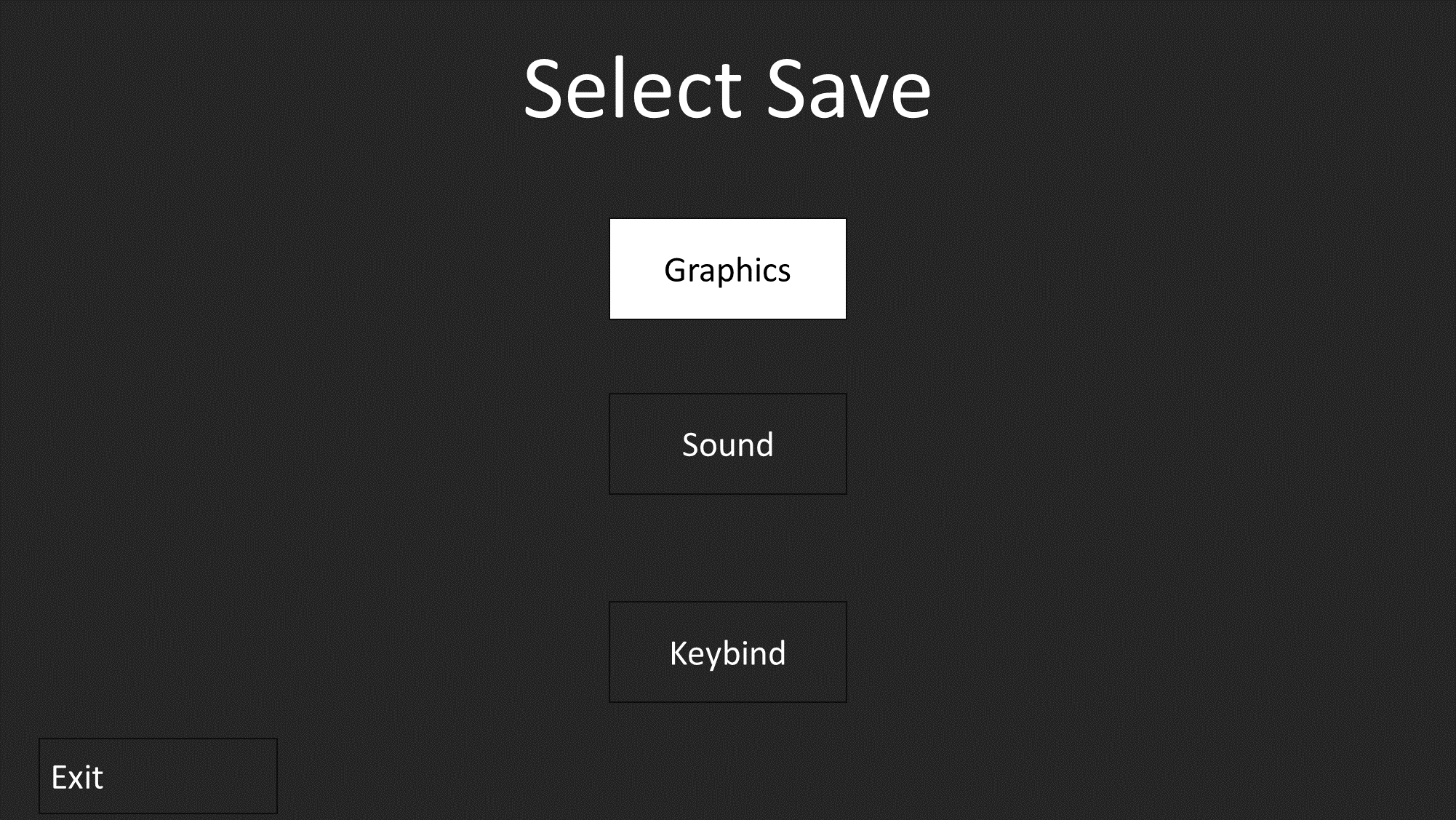
### Load Game:



Usability:

The "Load Game" feature facilitates ease of access to continue previous gameplay sessions. Its user-friendliness is paramount, ensuring that users can effortlessly pick up where they left off as they can pick what save they want to play by checking the difficulty, level, and date of save with the included time stamp. The interface maintains consistency with the rest of the game, allowing users to employ both mouse and keyboard interactions for navigation. This consistency, coupled with the dynamic button colour changes, offers users a familiar and accessible experience, which is particularly important for preserving user engagement.

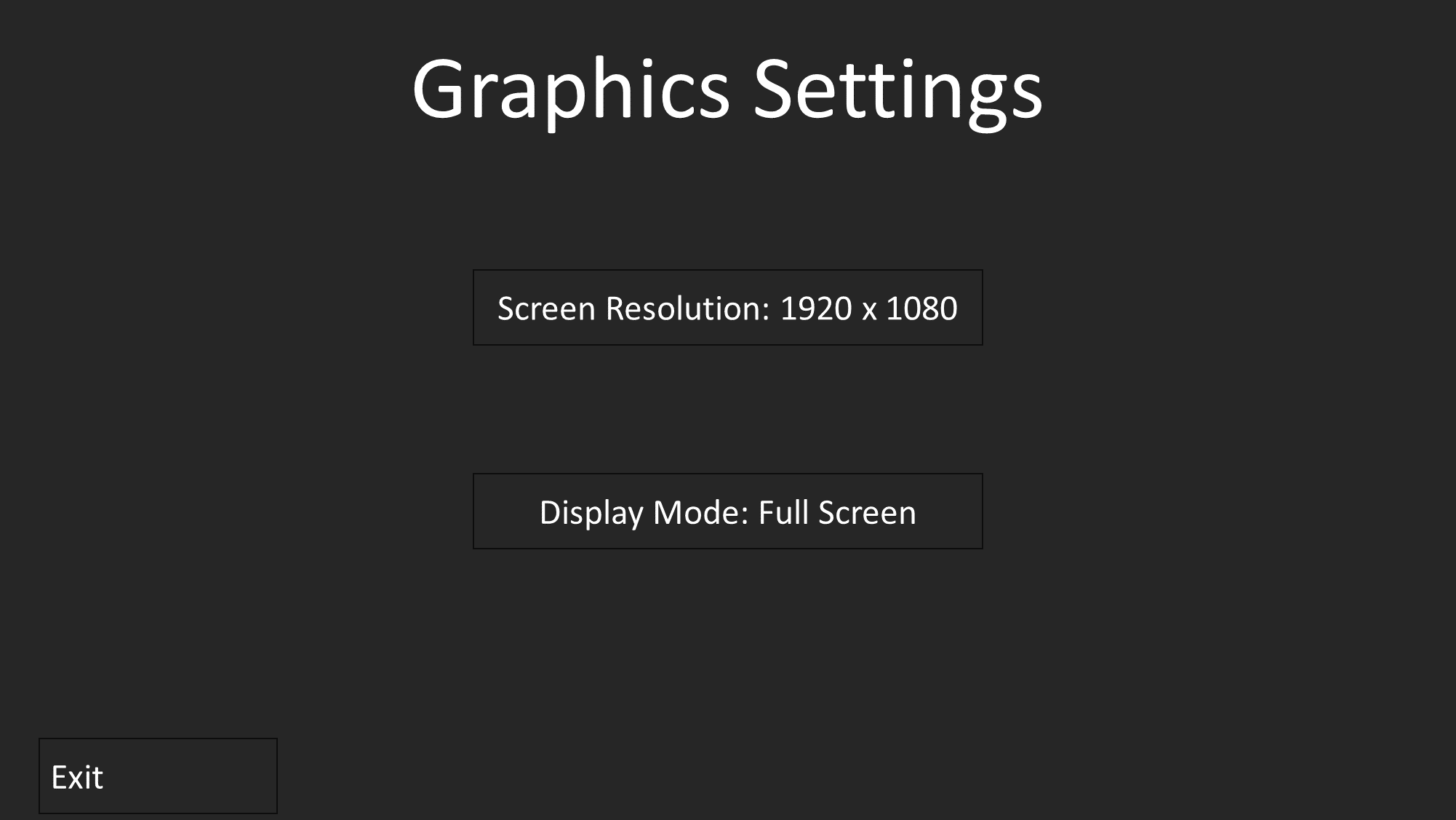
### Settings:



Usability:

The "Settings" menu is crucial for users to customize their gaming experience. Its design prioritizes user-friendliness, making it easy for users to configure their preferences and controls. As with other scenes, users can navigate using both mouse and keyboard inputs (WASD or Arrow keys). The dynamic button colour changes serve as visual guides, highlighting clickable elements. This consistency in design across scenes ensures that users can efficiently and intuitively manage their game settings, promoting accessibility and satisfaction.

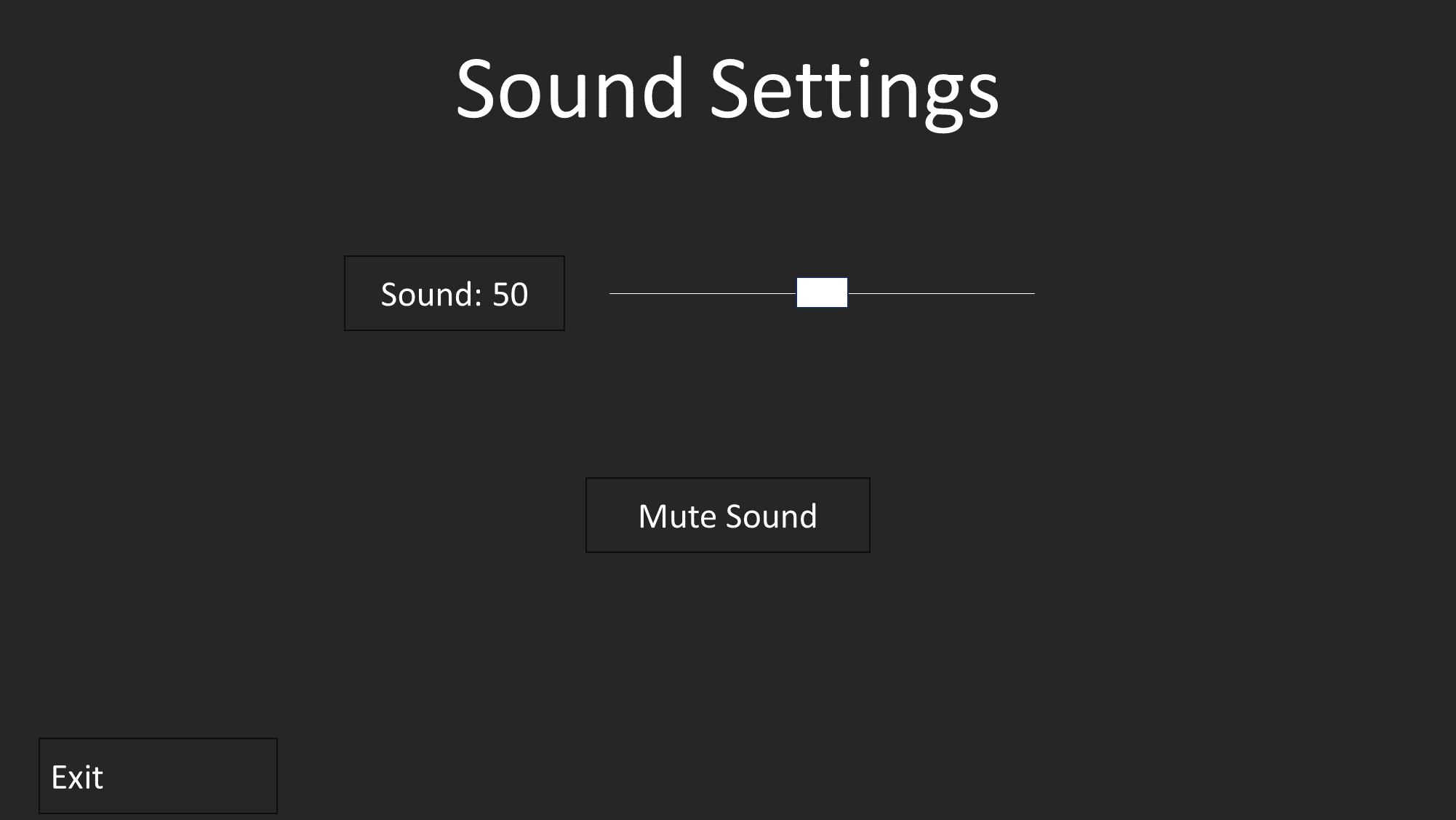
### Graphics:



Usability:

The “Graphics Settings” menu allows players to adjust screen resolution and display mode, which is essential for enhancing the usability and accessibility of a game. These features recognize the diverse hardware configurations among users and allows them to tailor the visual experience based on their preferences and system capabilities. Screen resolution adjustments accommodate a range of monitors and display setups, ensuring optimal clarity and visual fidelity. Meanwhile, the ability to toggle between display modes, such as Fullscreen or windowed, caters to players' preferences and varying multitasking needs. This flexibility not only provides a personalized gaming experience but also prevents potential discomfort or inconvenience for users with specific display requirements.

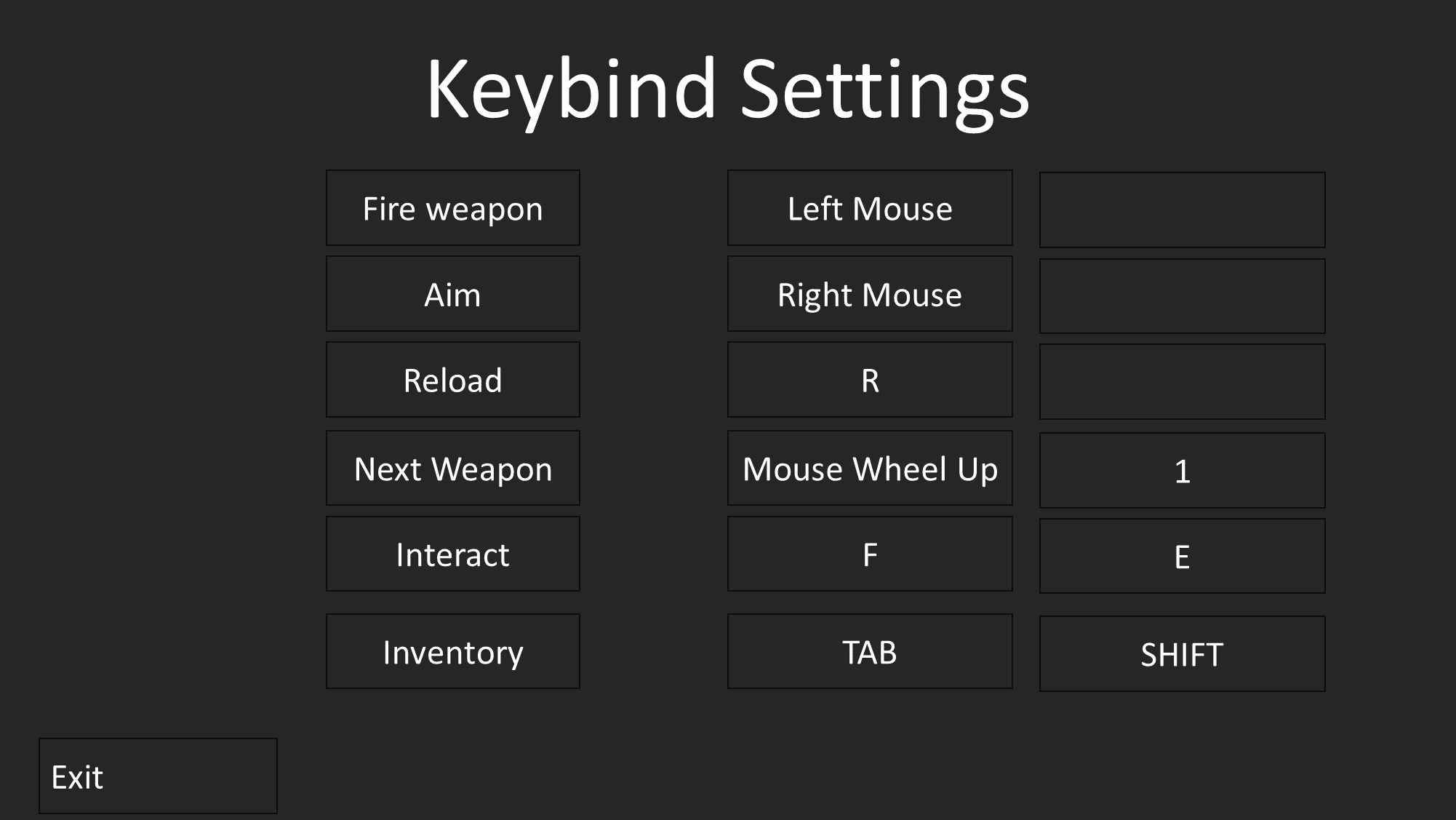
### Sound:



Usability:

The “Sound Setting” menu allows players to adjust the volume with a volume slider or they have the option to input a numeric value between 0 to 100, along with a mute button, optimizing the user experience in a game. These features cater to the diverse preferences and auditory sensitivities of players, allowing them to fine-tune the audio output to a level that suits their comfort and environment. The volume slider provides an intuitive and interactive way for users to set the audio precisely, while the numerical input option adds a quantitative dimension for those who prefer exact values. The inclusion of a mute button offers quick accessibility for situations where players need to silence the audio promptly. By integrating these sound settings, the game not only acknowledges the importance of audio customization but also demonstrates a commitment to inclusivity and user-friendly design, ensuring that players can enjoy the game with a personalized and immersive auditory experience.

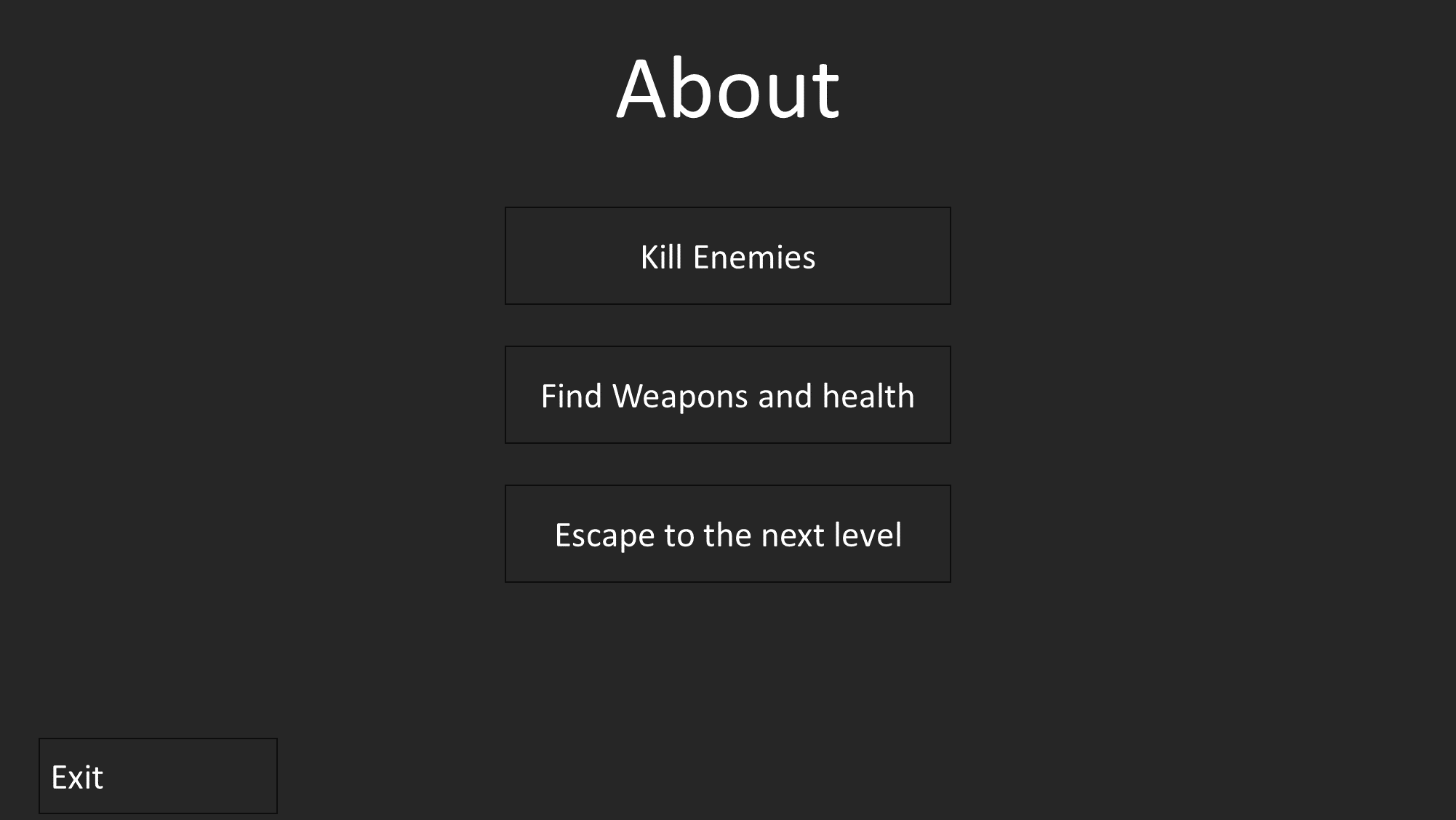
### Keybind:



Usability:

The “Keybind Setting” Scene allows users to customize inputs for various in-game actions, contributes to the accessibility and individualization of the gaming experience. Offering the flexibility to assign two different key binds for each action caters to a wide spectrum of player preferences, ensuring that users can choose key combinations that align with their comfort and familiarity. This feature is particularly beneficial for accommodating diverse input devices, allowing players to tailor the controls to their preferred keyboard layout or peripherals. Additionally, the ability to assign custom key binds addresses accessibility concerns, providing an inclusive gaming environment for users with specific physical or ergonomic requirements.

### About:



Usability:

The "About" scene provides users with vital information on how to play the game, enhancing their understanding of the game mechanics, objectives. Clarity is key to usability here, and the user interface is designed to ensure straightforward access to this information. Users can navigate the "About" scene using familiar mouse or keyboard inputs. The dynamic button colour changes persist as a user-friendly feature, helping users identify interactive elements. In this way, the scene promotes accessibility, ensuring that both new and returning players can quickly grasp the game's essential aspects, ultimately reducing frustration and enhancing enjoyment.

## Data Requirements

Data requirements are the essential elements of information and input necessary to support various aspects of the project. They play a critical role in ensuring the proper functioning and enhancing the user experience. User profiles are integral for storing individual progress, high scores and more, offering a personalized and competitive gameplay environment. Difficulty setting data is required to adjust the game's challenge, accommodating users of varying skill levels and preferences. Information about resolution and display modes optimizes graphics settings, ensuring the game can be played on a wide range of hardware configurations. Data concerning keyboard and mouse inputs enables users to customize controls to their liking, making the game more user-friendly and adaptable to individual playstyles. The JSON configuration file serves as a data repository for saving and loading user settings, maintaining consistency and personalization across gaming sessions. Therefore, data requirements are essential to tailor the game's functionality, visuals, and controls, enhancing the overall user experience by providing accessibility, personalization, and satisfaction for users.

Here’s a data dictionary defining key variables that will be used in the project:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Data Type** | **Storage Size** | **Validation Check** | **Justification** |
| menuScene | Scene | - | Display the Game title, Start Game, Load Game, Settings, About and Quit | The main menu scene is essential for organizing the user experience and providing a starting point for game access. Users value a structured main menu. |
| startGameButton | Button | - | Transition to the difficulty Scene. | These buttons are the primary means of user interaction, ensuring intuitive navigation and access to various game features. Users expect simple buttons. |
| loadGameButton | Button | - | Transition to the main game Scene. | Easy access to loading the game is crucial for user convenience and enjoyment. Users rely on this feature for resuming gameplay. |
| settingsButton | Button | - | Transition to the settings Scene. | Access to settings allows users to tailor their gaming experience. Users expect user-friendly configuration options. |
| aboutButton | Button | - | Transition to the about Scene. | Providing information about the game ensures user clarity and understanding. Users value clear instructions. |
| quitButton | Button | - | Transition to the quit Scene. | A quit button offers users a convenient way to exit the game. Users expect such functionality for a seamless experience. |
| difficultyScene | Scene | - | Display 6 buttons:  Easy, Medium, Hard, Survivor, Grounded, return | Access to different game difficulty levels ensures users can tailor the game to their skill level and preferences. Users appreciate challenge options. |
| settingsScene | Scene | - | Display 3 buttons:  Graphics, Input, and return | Access to settings, including graphics and input options, allows users to customize their gaming experience. User-friendly configurations are expected. |
| graphicsScene | Scene | - | Display 2 lists and button:  Change resolutions, Change display mode, return | Access to graphics settings enables users to adjust their display preferences for compatibility. Users expect resizable and full screen options. |
| KeybindsScene | Scene | - | Display 1 list, 1 button:  Key binds, return | Access to input settings allows users to configure their key bindings for a personalized gaming experience. User-friendly key binding options are essential. |
| aboutScene | Scene | - | Display 1 label, 1 button:  How to play, return | Providing an About scene with game instructions ensures user clarity on gameplay mechanics, objectives, and controls. Users appreciate straightforward guidance. |
| returnButton | Button | - | Returns to previous scene | The "return" button enhances the user experience by enabling easy navigation within settings or menus, offering flexibility and convenience. |
| moveUp | JSON Variable | 1 Byte | Player moves up when key is pressed | Player moves up when the corresponding key is pressed, facilitating essential character movement in the game. |
| moveDown | JSON Variable | 1 Byte | Player moves down when key is pressed | Player moves down when the corresponding key is pressed, providing complete control over character navigation. |
| moveLeft | JSON Variable | 1 Byte | Player moves left when key is pressed | Player moves left when the corresponding key is pressed, allowing for fluid horizontal movement within the game environment. |
| moveRight | JSON Variable | 1 Byte | Player moves right when key is pressed | Player moves right when the corresponding key is pressed, enabling seamless traversal to the right in the game world. |
| aim | JSON Variable | 1 Byte | Player aim weapon when key is pressed | Player aims the weapon when the corresponding key is pressed, a fundamental action for targeting and engaging with in-game elements. |
| fire | JSON Variable | 1 Byte | Weapon fires projectile when key is pressed. Projectile counter decrements. Inventory ammo decrements | Weapon fires a projectile when the corresponding key is pressed. The projectile counter decrements, providing a means of combat and interaction. |
| reload | JSON Variable | 1 Byte | Projectile counter resets when key is pressed. Inventory ammo resets | Projectile counter resets when the corresponding key is pressed, enabling the reloading of the weapon for continued engagement. |
| nextWeapon | JSON Variable | 1 Byte | Player switches to equipped weapon in inventory when key is pressed | Player switches to the equipped weapon in the inventory when the corresponding key is pressed, allowing for dynamic adaptation to different combat scenarios. |
| inventory | JSON Variable | 1 Byte | Inventory menu opens when key is pressed | Inventory menu opens when the corresponding key is pressed, offering access to stored items, and enhancing player control and decision-making. |
| interact | JSON Variable | 1 Byte | Item added to inventory or inventory menu opens (to insert keycard in door) when key is pressed | An item is added to the inventory, or the inventory menu opens (e.g., to insert a keycard in a door) when the corresponding key is pressed, enabling key interactions in the game. |
| return | JSON Variable | 1 Byte | Game pauses or transitions to previous scene when key is pressed | The game pauses or transitions to the previous scene when the corresponding key is pressed, providing a mechanism for user control and navigation within the game interface. |
| player | Sprite or Player | - | Displays the Player Sprite | Game characters are pivotal to the gameplay, directly impacting user satisfaction. Users have an interest in characters that engage and satisfy users. |
| enemy | Sprite or Enemy | - | Displays the Enemy Sprite | Enemies are essential for creating challenges and excitement in the game. Users rely on well-designed enemies for engaging gameplay. |
| isDead | Boolean | 1 byte | When user/enemy health = 0, return true | Determining when a player or enemy is defeated directly influences game mechanics. Users expect a consistent and fair system for character defeat. |
| rooms | int | 1 byte | Check how many rooms are in the level | These variables are essential for generating and maintaining game elements, shaping the dynamic gameplay experience. Users rely on well-constructed game elements for an engaging experience. |
| rectArray | rect[] | rooms | Are there the same number of rooms | Procedurally generated rooms contribute to diverse gameplay experiences. Users expect varied room layouts for increased replayability. |
| projectiles | projectile[] | - | User shouldn’t be able to shoot if array is empty | Projectiles are fundamental for controlling user ammunition and attacks, influencing game mechanics and satisfying users. Users rely on accurate and responsive projectile behaviour. |
| isCollided | Boolean | 1 byte | - | Detecting collisions accurately is critical for consistent game physics. This variable aligns with user expectations and the need for the game to respond correctly to interactions. |
| moveAngle | double | 2 bytes | Must be within [0, 360] or [0, 2π]  Degrees or Radians | Variables controlling character movement are crucial for creating dynamic gameplay. Users desire precise control over these variables to enjoy responsive and realistic movement. |
| moveSpeed | double | 2 bytes | Must be non-negative | Ensures that character movement remains realistic and predictable within the game. Allowing negative values would result in physically implausible and erratic movement, which can detract from the user's immersion and gaming experience. |
| isUserMoving | Boolean | 1 byte | - | Understanding user motion is vital for gameplay dynamics. Users expect the game to accurately reflect their input, ensuring a satisfying gaming experience. |
| score | int | 4 bytes | - | These variables allow users to track their performance and scores throughout the game. Users value feedback on their progress and achievements in the game. |
| scoreString | String | 24 bytes | - | Score is cast from int to String to be displayed within a Label, so it's viewable to the user. Users appreciate clear and accessible score presentation.  Precise character positioning is essential for balanced and engaging gameplay. Users rely on accurate character placement for a fair gaming experience. |
| userX | int | 4 bytes | - |
| userY | int | 4 bytes | - |
| enemyX | int | 4 bytes | - |
| enemyY | int | 4 bytes | - |
| randomX | int | 4 bytes | - | Used for procedural generation and enemy spawns, to create diverse rooms with varying enemy locations, increasing the replayability factor. Users expect diverse and challenging environments. |
| randomY | int | 4 bytes | - |

## Test Strategy

A Test Strategy is a guiding framework for the testing process of a software application or system. This strategic plan outlines the scope, objectives, and approach to testing, defining the overall testing strategy. It clarifies whether testing will be manual or automated, identifies testing levels and types, and details the methodologies and techniques to be used. The test strategy covers essential aspects of testing, including the test environment, test data, entry and exit criteria for different testing levels, and the delivery of key features.

### Test Strategy

Testing is a necessary phase used to ensure that the project is being developed correctly in a systematic order, utilizing the success criteria.

The early development stage will implement white box testing to quickly identify any syntax, logic, or runtime errors in each of the prototypes. Since white box testing assesses all possible outcomes of an algorithm, this allows for the elimination of any syntax or logic errors and helps optimize code from main functions as all algorithms can be debugged to understand each output. This will ensure a fluid development stage as subroutines make code much easier to write, read, and understand with fewer errors in a more efficient manner.

Additionally, I will use black box testing. This will mainly be used towards the final stages of development. Black box testing is used to assess the game from the perspective of the user, not from the game developer. This is because users play each level in their own way, some may try to see if they can walk through walls, another might stay behind cover. These different scenarios need to be tested so that the game works as intended, and so the user doesn’t get frustrated with any bugs that they might encounter. Black box testing gives you the actual output of the algorithm based on given knowledge about the algorithms. This means that no technical knowledge is required about the code, so any user can easily understand this testing method. This will be used at the end of every prototype to ensure that to mitigate any bugs in code.

Alpha Testing will also be utilized. Alpha testing can only be used when you have a complete prototype that functions as intended. So, this could be used once the core features of the game have been implemented and works as intended. This testing strategy is primarily utilized by developers to gain feedback about the any bugs that may arise or add any improvements to algorithms like the level generation algorithm might need adjusting so that rooms are more spaced out to improve the user’s gameplay experience.

Beta testing will be used after improvements have been made to the alpha version of the project. Once the improvements have been implemented, the project can be tested in the wild by a small selected group of users. This allows for more feedback that will be given to me, so that the final product is fit for the general user base.

### Test Plan

The Test Plan is a pivotal for the development process, ensuring that the application functions as intended and meets the predefined success criteria. It encompasses a series of test cases and scenarios, ranging from pre-development testing during the application design phase to post-development testing to validate the final product. This plan outlines the critical tests, test items, and expected outcomes to guarantee a seamless and user-friendly experience, addressing aspects like user accessibility, controls, in-game mechanics, and UI design. The plan not only serves as a roadmap for testing but also as a quality assurance tool, aligning the application with the expectations and requirements of stakeholders and end-users.

Usability testing is when the developer samples users to test out the game and observes their playstyle to determine the effectiveness of in game interfaces. This is useful for the developer as they get direct feedback about how to improve in-game interfaces to make interfaces easy to understand and quick to use, so that the user can get straight back into the game.

Success criteria acceptance demonstrates the product to the user once all of the success criteria features have been implemented into the game. This is useful for the developer as the user can give feedback so that certain features of the success criteria can be added, removed, or modified to improve the user experience.

Destructive testing is used in conjunction with black box testing to test invalid input data to see if they have don’t create any issues with the game. This is quite a vital test since Keybind inputs can be configured by the user in the settings. The new input Keybind should work as the previous Keybind and the previous Keybind shouldn’t work.

#### Test Data

Testing is necessary to ensure that the project runs as intended.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Item** | **Test Type** | **Test Data** | **Expected Outcome** | **Justification** |
| On Application Start up | Normal | None | Fullscreen application opens and loads the main menu. | Ensure the application starts as expected, delivering a seamless experience for users. |
| Start Game button pressed | Normal | Button click | Transition to the difficulty select scene. | Verify that pressing the "Start Game" button leads to the difficulty selection scene. |
| Easy button pressed | Normal | Button click | Confirmation message displayed. | Ensure that selecting "Easy" triggers a confirmation message for user choice confirmation. |
| Medium button pressed | Normal | Button click | Confirmation message displayed. | Verify that choosing "Medium" results in a confirmation message to confirm user selection. |
| Hard button pressed | Normal | Button click | Confirmation message displayed. | Ensure that selecting "Hard" prompts a confirmation message to confirm the user's choice. |
| Survivor button pressed | Normal | Button click | Confirmation message displayed. | Confirm that pressing "Survivor" leads to a confirmation message confirming the user's selection. |
| Grounded button pressed | Normal | Button click | Confirmation message displayed. | Ensure that selecting "Grounded" triggers a confirmation message to confirm the user's choice. |
| Go Back pressed | Normal | Button click | Transition to the Main Menu from Difficulty Select. | Verify that using the "Go Back" action in the Difficulty Select scene returns users to the Main Menu. |
| Load Game button pressed | Normal | Button click | Transition to the main game. | Confirm that pressing "Load Game" takes the user directly to the main game scene, enabling them to resume their progress seamlessly. |
| Settings button pressed | Normal | Button click | Transition to the settings scene. | Verify that pressing the "Settings" button transitions the application to the settings scene, enabling users to customize their preferences and controls. |
| Graphics Settings button pressed | Normal | Button click | Transition to the Graphics settings scene. | Confirm that selecting "Graphics Settings" navigates users to the graphics settings scene, where they can adjust screen resolution and display mode for optimal compatibility. |
| Go Back pressed | Normal | Button click | Transition from Input Settings to Settings scene. | Ensure that using the "Go Back" action in the Input Settings scene returns users to the main Settings scene, streamlining the user experience. |
| Input Settings button pressed | Normal | Button click | Transition to the Input settings scene. | Verify that pressing "Input Settings" leads users to the input settings scene, where they can customize key bindings and controls, enhancing user-friendliness. |
| Go Back pressed | Normal | Button click | Transition from Input Settings to Settings scene. | Confirm that selecting "Go Back" from the Input Settings scene navigates users back to the main Settings scene, offering a convenient return path. |
| Go Back pressed | Normal | Button click | Transition from Settings to the Main Menu. | Verify that pressing "Go Back" in the Settings scene returns users to the Main Menu, providing an efficient way to navigate between the main menu and settings. |
| About button pressed | Normal | Button click | Transition from Main Menu to the About scene. | Confirm that selecting the "About" button from the Main Menu transitions users to the About scene, offering easy access to information about the game. |
| Quit button pressed | Normal | Button click | Prompt with "Yes" and "No" buttons for quitting. | Ensure that pressing "Quit" triggers a confirmation prompt with "Yes" and "No" options, allowing users to decide whether to exit the application. |
| Yes button pressed | Normal | Button click | Application closes. | Verify that choosing "Yes" after the quit prompt terminates the application, allowing users to exit the game as expected. |
| No button pressed | Normal | Button click | Return to the main menu scene. | Confirm that selecting "No" after the quit prompt returns users to the main menu scene, providing an alternative to continue playing or explore other game features. |

#### Post Development Testing

|  |  |  |
| --- | --- | --- |
| **Test Data** | **Test Area** | **Justification** |
| Procedural Generation | Success Criteria | Verify that the procedural level generation creates diverse and solvable levels. This ensures that users have a challenging and engaging experience, meeting the success criteria for level design and player progression. |
| Maze Generation | Success Criteria | Confirm that maze generation adheres to design specifications, guaranteeing the creation of solvable mazes. This validation is crucial to meeting success criteria related to maze complexity and ensuring a fair and enjoyable gameplay experience. |
| Movement Input | Success Criteria | Ensure that player movement mapped to WASD functions smoothly, aligning with user expectations. This test guarantees that the controls meet success criteria and provide an intuitive gaming experience, enhancing overall gameplay satisfaction. |
| Enemy Overlap | Success Criteria | Validate that enemies are correctly destroyed upon player overlap without causing errors. This test ensures smooth gameplay and enhances competitiveness by preventing unexpected issues during player-enemy interactions. |
| Player Score | Success Criteria | This test needs to be done to ensure that the score is calculated correctly by multiplying points by the stage reached. Correctly calculating the score avoids unfair scenarios such as wrong score as well as improving the replayability of the game. |
| Player Location (x, y) | Success Criteria | Check that the player's location is appropriately clamped within the maze boundaries. This prevents unintended player movements outside the playable area, ensuring a consistent gaming experience and preventing players from getting stuck or having to restart. |
| Player Health | Success Criteria | Confirm that player health does not go below zero, ensuring that the UI (health bar) accurately reflects the correct value. This test prevents confusion and maintains a clear representation of the player's health status during the game. |
| Enemy Count | Success Criteria/ Destructive | Test maximum enemy count to observe effects on the game and verify that it doesn't cause errors in the enemy spawning loop. This ensures the game can handle the specified number of enemies, meeting success criteria for enemy behaviour and spawning mechanisms. |
| Projectile Count | Success Criteria/ Destructive | Test maximum projectile count to observe effects on the user's system and ensure the game doesn't crash. This validation guarantees that the game remains stable under heavy projectile usage, meeting success criteria for projectile management and system performance. |
| Menu Buttons | Usability | Test menu buttons to ensure they navigate to the correct scenes and perform the intended functions. This usability test satisfies user expectations, allowing for efficient navigation and enhancing the overall user experience in terms of menu interaction. |