Development Project Report CTEC3451_2023_520: Isometric Rogue-like game with Cellular automata for procedural level generation.

Full Name: Chaouki Mabrouki

Institution: Computer engineering and media "CEM",

Course: Computer Games Programming - G62041_2023

P Number: P2655779

Supervisor: Dr Conor Fahy

Note: I would like to leave a not here to all my teachers who gave me the energy and resources to be able to finish this journey, it was an immense honor for me to be part of the community of DMU, and I will be forever proud for being part of this adventure, thank you for this incredible opportunity.

Table of Contents

Introduction:

This report offers an in-depth exploration of the development process behind an isometric rogue-like game created using Unity. The project encompasses a variety of complex systems, each contributing to the overall functionality and player experience of the game. These systems include procedural level generation using cellular automata, sophisticated enemy AI, mesh generation, game management, and user interface design. The game's development involved extensive iteration, testing, and problem-solving, resulting in a cohesive and challenging gameplay experience.

The primary focus of this project was the implementation of procedural level generation through cellular automata, a method chosen for its ability to create dynamic and unpredictable environments. Cellular automata offer a flexible approach to generating levels that are both diverse and structurally sound, ensuring that each playthrough presents a unique challenge to the player. The report examines the intricacies of implementing this technique, including the balance between randomness and playability, and the integration of mesh generation to create the physical structures of the game world.

In addition to procedural generation, the development of enemy AI was a critical aspect of the project. The AI was designed to navigate the procedurally generated environments, track the player, and engage in combat using various strategies. The AI system, built upon Unity's NavMeshAgent and a finite state machine (FSM), allowed for dynamic and challenging enemy behaviors. The report delves into the design and implementation of different enemy types, such as the TurtleEnemy and MageEnemy, each with its unique attack patterns and interactions with the player.

The game also features a robust projectile system, integral to both player and enemy combat mechanics. This system supports various types of projectiles, including homing projectiles that target the nearest enemy or player. The development process involved addressing challenges

related to projectile movement, collision detection, and the seamless integration of these mechanics into the broader gameplay systems.

Game management systems, including the GameOverManager and ScoreManager, play a pivotal role in the overall player experience. These systems are responsible for managing game states, tracking player performance, and providing feedback through the user interface.

The report explores the design and implementation of these systems, highlighting the challenges of ensuring their correct functionality within the game's dynamic environment. User interface (UI) design is another crucial element covered in this report. The UI components, such as the main menu, options menu, and in-game HUD, are designed to be intuitive and accessible, enhancing the player's interaction with the game.

The report discusses the considerations behind the UI design, the technical implementation, and how these elements contribute to the overall user experience.

Beyond the core systems, the report also touches on advanced mechanics, such as wave-based enemy spawning and a planned in-game store system. These features were designed to add depth and complexity to the gameplay, offering players new challenges and progression opportunities. The report concludes with a critical analysis and reflection on the project, discussing what worked well, the challenges encountered, and what could have been done differently.

By the end of this report, readers will have a comprehensive understanding of the technical complexities involved in developing this Unity-based game and the thought processes behind each design and implementation decision.

Cellular Automata and Procedural Map Generation:

At the core of the game's level design is the use of cellular automata, a computational technique known for its ability to generate complex patterns and structures from simple rules. Cellular automata are particularly well-suited for creating game environments that are both dynamic and varied, making them an ideal choice for the procedural generation of levels in this rogue-like game.

The Cellular Automata Process:

The cellular automata process begins with a grid where each cell can exist in one of several states—typically representing a wall or an open space. The state of each cell is determined by a set of rules that take into account the states of neighboring cells.

These rules are applied iteratively, allowing the initial random configuration of cells to evolve into a more structured and coherent map. In the context of this project, the cellular automata technique was employed to generate maze-like environments, with the goal of creating levels that are challenging yet navigable.

The MapGenerator class was developed to handle the implementation of cellular automata. The process begins with an initial random fill of the grid, where each cell is assigned, a state based on a predefined probability.

This step is crucial, as it sets the foundation for the subsequent iterations of the cellular automata rules. Too high a probability of walls, and the map becomes overly cluttered and difficult to navigate; too low, and the map lacks the necessary complexity to provide a challenge.

After the initial random fill, the grid undergoes several passes of the cellular automata rules. Each pass serves to smooth out irregularities, remove isolated cells, and create more coherent structures such as corridors and rooms.

The number of passes and the specific rules applied were carefully tuned to achieve a balance between randomness and structure. The result is a series of levels that are distinct from one another but share common characteristics that ensure playability.

Challenges and Solutions in Procedural Generation:

One of the primary challenges in implementing cellular automata was finding the right balance between randomness and structure. Early iterations of the algorithm produced levels that were either too chaotic, with disconnected sections and dead ends, or too predictable, offering little variation between playthroughs.

To address this, extensive testing was conducted to fine-tune the parameters controlling the initial fill probability and the subsequent cellular automata rules. The final implementation struck a balance that produced levels with a high degree of variability while maintaining coherence and playability.

In addition to the basic structure, it was essential to ensure that the generated levels were bounded by solid walls, preventing the player from exiting the play area or encountering visual glitches. This was achieved through the implementation of a bordered map technique within the MapGenerator class. The bordered map added a border of solid walls around the generated level, preserving the integrity of the game world.

Another critical aspect of procedural generation was the placement of spawn points for the player and enemies. These points needed to be strategically placed to ensure that the game started with the player in a suitable location and that enemies could be introduced into the environment in a way that provided a challenge without overwhelming the player. The MapGenerator was designed to consider these factors, placing spawn points in accessible and strategically relevant locations within the generated levels.

Mesh Generation Integration:

Once the cellular automata had generated the layout of the level, the MeshGenerator class was responsible for creating the physical geometry that represented the level's walls and floors. This process involved defining the vertices and triangles that make up the mesh, ensuring that the geometry was both accurate and optimized for performance.

One of the primary challenges in mesh generation was ensuring that the normals of the generated meshes were correctly oriented.

Normals are vectors that define the direction in which a surface faces, and they are crucial for proper lighting and collision detection within the game. Incorrectly oriented normals can lead to visual artifacts, such as surfaces appearing too dark or light, or collision detection failing to work as intended.

The MeshGenerator was designed to calculate and assign normals accurately, ensuring that the generated levels were visually consistent and functioned correctly within the game engine.

In addition to normals, the mesh generation process also had to address the issue of triangle orientation. Triangles are the basic building blocks of a mesh, and their orientation determines how they are rendered within the game. Incorrectly oriented triangles can lead to surfaces that are invisible or appear inside out.

The MeshGenerator included logic to ensure that triangles were consistently oriented, contributing to the overall stability and visual fidelity of the game.

Another challenge in mesh generation was the optimization of the generated geometry. The complexity of the level's geometry directly impacts the performance of the game, particularly in terms of rendering and collision detection. To optimize performance, the MeshGenerator was designed to minimize the number of vertices and triangles while still accurately representing the

level's structure. This was achieved by combining adjacent walls into larger segments and avoiding the creation of unnecessary geometry.

Despite these efforts, the initial implementation of the MeshGenerator encountered issues such as inverted walls and gaps in the mesh. These issues were eventually resolved through a combination of debugging, algorithm refinement, and iterative testing.

NavMesh Integration for AI Pathfinding:

The procedural generation system was also tightly integrated with Unity's NavMesh system, enabling AI agents to navigate the generated levels effectively. After each level was created by the cellular automata and mesh generation processes, a NavMesh was dynamically baked onto the terrain. This NavMesh provided the necessary data for the AI agents to perform pathfinding, allowing them to move through the environment, avoid obstacles, and reach their targets.
The dynamic generation of the NavMesh was a critical feature, as it ensured that the AI could adapt to any changes in the environment, whether they were the result of procedural generation or gameplay events. This integration was essential for enabling the complex AI behaviors discussed in the following sections of this report.

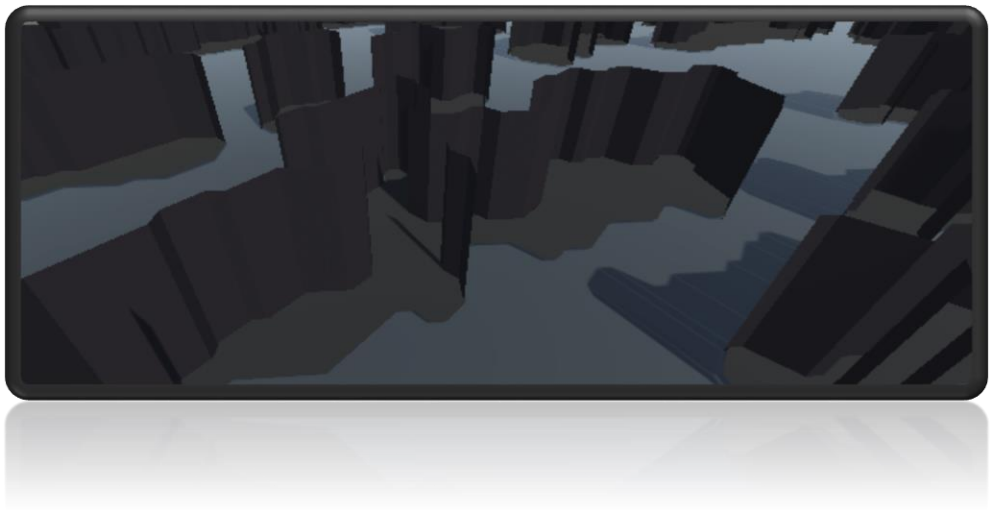

Figure 1 walls not being Generated Properly, normals inverted and triangles facing inward

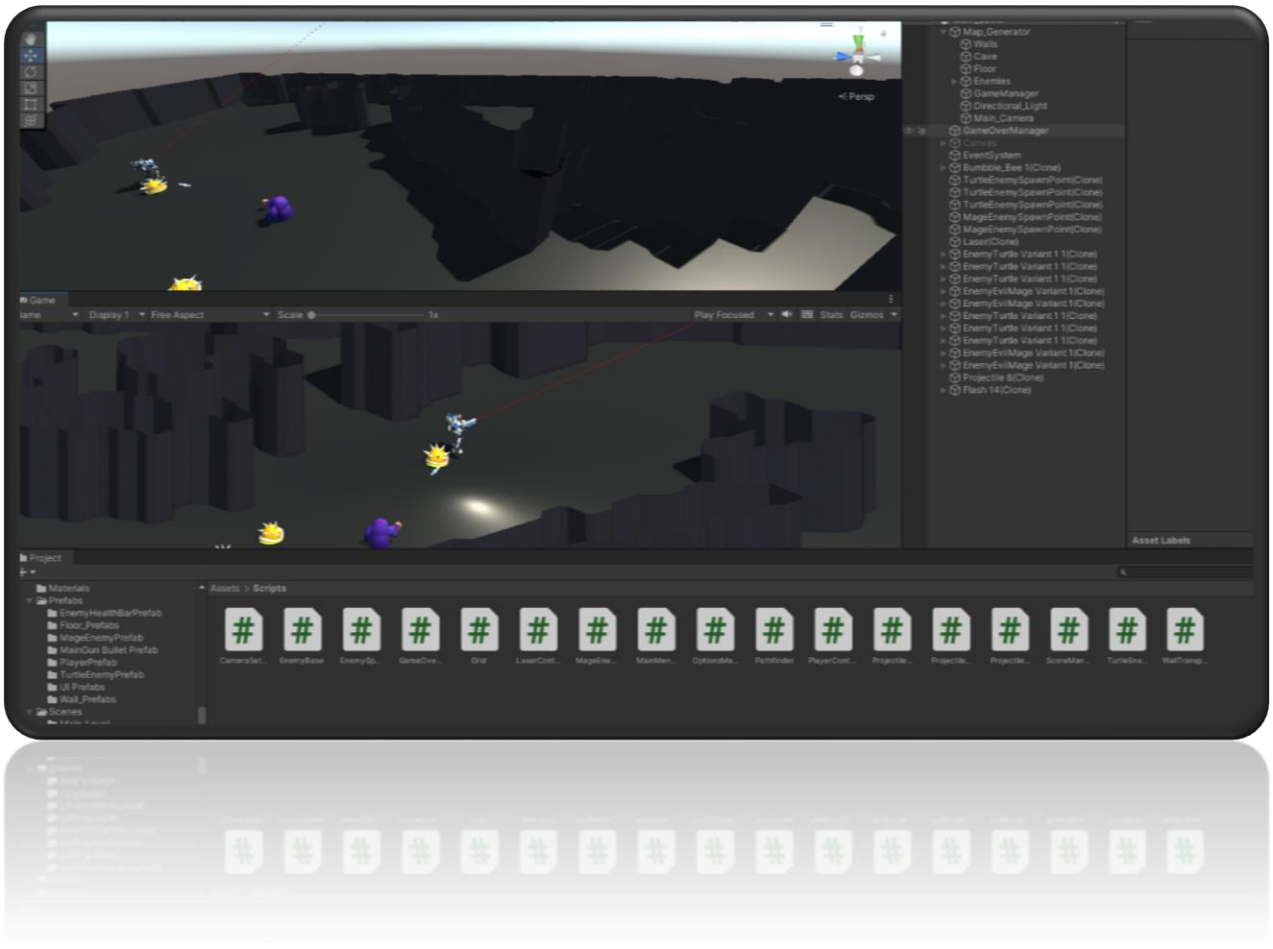

Figure 2 Walls Generated correctly and normals and triangles facing the correct angle

Enemy AI Development:

The development of enemy AI was one of the most intricate and challenging aspects of the project, requiring the integration of several systems, including pathfinding, state management, and behavior control. The AI was designed to provide a dynamic and challenging experience for the player, with enemies that could navigate the procedurally generated environments, track the player, and engage in combat using a variety of strategies.

The EnemyBase Class and Inheritance:

The EnemyBase class served as the foundation for all enemy AI in the game. This class encapsulated common behaviors such as movement, health management, and damage handling, which were shared across different enemy types.

By using inheritance, the EnemyBase class allowed for the creation of specific enemy behaviors without duplicating code, making the system both flexible and maintainable.

The use of inheritance also made it easier to extend the game with new enemy types in the future. By building upon the common functionality provided by the EnemyBase class, new enemy types could be introduced with minimal changes to the existing codebase.

This modular approach to AI development was critical for maintaining the project's scalability and ensuring that the AI system could evolve as the game expanded.

Pathfinding with NavMeshAgent:

Pathfinding was a critical component of the enemy AI, enabling enemies to navigate the procedurally generated levels and reach the player.

Unity's NavMeshAgent was used to handle pathfinding, leveraging the NavMesh that was dynamically generated based on the terrain created by the cellular automata.

The NavMeshAgent provided built-in functionality for avoiding obstacles and finding the shortest path to the player, which was essential for ensuring that the enemies could effectively track and engage the player.

The integration of the NavMeshAgent with the procedural generation system was not without its challenges. Since the NavMesh had to be generated dynamically after each level was

created, there was a risk that the AI agents might not correctly interpret the terrain, leading to pathfinding errors or inefficient movement.

To mitigate this risk, extensive testing was conducted to ensure that the NavMesh generation process produced accurate and reliable data for the AI agents to use. Additionally, the NavMeshAgent's parameters, such as speed, acceleration, and stopping distance, were carefully tuned to match the desired behavior of each enemy type.

Finite State Machine for Behavior Control:

The behavior of the AI was controlled using a finite state machine (FSM), a widely-used technique in game development for managing the different states an AI agent can be in.

The FSM allowed for smooth transitions between states based on the AI's perception of the environment and the player's actions. For example, an enemy might start in a patrolling state, where it moves through a predefined area, and then transition to a chasing state when it detects the player. If the enemy closed the distance with the player, it would then transition to an attacking state.

The FSM was implemented as a series of states within the EnemyBase class, with each state corresponding to a specific behavior.

Transitions between states were triggered by events such as the detection of the player, changes in the enemy's health, or the completion of an attack animation.

This modular approach to behavior control made it easy to add new states or modify existing ones, providing a high degree of flexibility in the design of enemy behaviors.

Specific Enemy Types: TurtleEnemy and MageEnemy:

The TurtleEnemy and MageEnemy were two specific enemy types implemented in the game, each with its unique behaviors and interactions with the player. These enemies were designed to provide distinct challenges, requiring the player to adapt their strategy based on the type of enemy they were facing.

The TurtleEnemy was designed as a melee attacker, focusing on closing the distance with the player and dealing damage in close quarters.

This enemy type had a relatively simple behavior pattern, charging towards the player and attacking when within range.

The challenge in implementing the TurtleEnemy was ensuring that it could navigate the environment efficiently and avoid getting stuck on obstacles.

The TurtleEnemy's movement was controlled by the NavMeshAgent, while its attack behavior was managed by the FSM, transitioning from chasing to attacking when within range.

The MageEnemy, on the other hand, was a ranged attacker that utilized projectile attacks. This enemy type was more complex, as it needed to maintain a certain distance from the player while attacking.

The MageEnemy used a combination of movement and attack patterns, often retreating to a safe distance before launching a projectile.

This behavior required careful coordination between the AI's movement logic and its attack routines, ensuring that the MageEnemy could both evade the player and deliver consistent damage.

The MageEnemy also made use of the FSM, transitioning between states such as evading, attacking, and retreating based on its proximity to the player and its current health.

The development of these enemy types highlighted the flexibility of the AI system, with the EnemyBase class providing a strong foundation for implementing diverse and challenging enemy behaviors.

The integration of pathfinding, state management, and behavior control allowed for the creation of enemies that were both dynamic and responsive to the player's actions, enhancing the overall challenge and engagement of the game.

Projectile Mechanics and Combat System:

Projectile mechanics played a crucial role in the combat system, both for the player and the enemies. The development of a robust projectile system was essential for creating engaging and dynamic combat encounters, requiring careful consideration of movement, collision detection, and the interaction between projectiles and other game elements.

The ProjectileMoverBase Class:

The ProjectileMoverBase class provided the core functionality for all projectiles in the game, handling essential tasks such as movement, collision detection, and damage application. This base class served as a foundation for creating specific types of projectiles, such as those used by the player and the MageEnemy.

Projectile movement was managed using Unity's physics system, with the ProjectileMoverBase class applying forces to move the projectile through the environment.

The movement logic had to be carefully tuned to ensure that projectiles traveled at the correct speed and followed the intended trajectory.

This was particularly important for the MageEnemy's projectiles, which needed to be slow enough to allow the player to evade them, but fast enough to pose a significant threat.

Collision detection was another critical aspect of the projectile system, as it determined when a projectile hit a target and needed to apply damage.

The ProjectileMoverBase class used Unity's built-in collision detection system, with projectiles being configured to detect collisions with specific layers, such as the player, enemies, or the environment. Upon collision, the projectile would trigger an event that applied damage to the target, reduced the target's health, and potentially destroyed the projectile.

Homing Projectiles:

One of the more advanced features of the projectile system was the implementation of homing projectiles, which could track and follow a target, such as the player or an enemy, within a certain range. This behavior added an extra layer of complexity to the combat, making it more difficult for the player to avoid incoming attacks and increasing the overall challenge of the game.

Homing projectiles were implemented as a specialized class that extended the ProjectileMoverBase class. This class included additional logic for detecting the nearest target within a specified radius and adjusting the projectile's trajectory to follow that target.

The homing behavior was achieved by applying a force to the projectile in the direction of the target, continuously updating the direction as the target moved.

This required careful tuning to ensure that the projectile could effectively track the target without becoming too difficult to evade.

The integration of homing projectiles into the combat system added variety to the gameplay, requiring the player to adapt their strategy based on the type of projectile they were facing.

This feature also demonstrated the flexibility of the ProjectileMoverBase class, allowing for the creation of different projectile behaviors that could be easily integrated into the broader gameplay systems.

Game Management and UI Design:

Effective game management and a well-designed user interface are essential components of any game, ensuring that players have a seamless and enjoyable experience.

This section of the report discusses the systems implemented for managing the game state, tracking scores, and providing a user-friendly interface for navigating the game's various features.

Game Over Management:

The GameOverManager was responsible for handling the game over state, a critical moment in the gameplay loop. When the player's health reached zero, the GameOverManager would trigger the game over screen, pause the game, and display the player's final score.

The manager also provided options for restarting the game or returning to the main menu, allowing the player to quickly get back into the action or exit the game.

Implementing the GameOverManager presented several challenges, particularly in ensuring that all game systems were correctly paused during the game over state.
This required careful coordination between different components of the game, such as the AI, projectiles, and UI elements, to ensure that they all responded appropriately to the game over event.

The transition to the game over state also needed to be smooth and free of bugs, providing a polished and professional experience for the player.

Score Management:

Score management was another vital aspect of the game, providing players with a tangible measure of their success and motivation to improve their performance, the ScoreManager class was responsible for tracking both time-based and kill-based scores, adding points based on the type of enemy defeated, and checking against the highest score recorded.

The ScoreManager was integrated into both the gameplay and the UI, with scores being displayed on the game over screen and the main menu, this required the ScoreManager to update the score in real-time during gameplay, ensuring that the player received immediate feedback on their performance.

The system also needed to handle the saving and loading of the highest score, using Unity's PlayerPrefs to persist this data between play sessions.

The implementation of the ScoreManager highlighted the complexity of managing dynamic systems in a real-time environment. During development, issues were encountered with accurately tracking and updating scores, particularly during intense combat scenarios.

These issues were eventually resolved through careful debugging and optimization, ensuring that the scoring system functioned reliably and provided a rewarding experience for the player.

User Interface Design:

The user interface (UI) was designed to be intuitive and easy to navigate, with a focus on providing clear and immediate feedback to the player. The main menu served as the hub for

accessing different parts of the game, including the play mode, options, and potentially the store system. The MainMenuManager class handled the navigation between these scenes, ensuring that transitions were smooth and responsive.

The main menu also displayed the highest score achieved by the player, providing an additional incentive to keep playing and improving. This was implemented by loading the highest score from PlayerPrefs when the main menu was initialized and updating the score display accordingly.

The options menu was another important part of the UI, allowing players to adjust audio settings and save their preferences using PlayerPrefs. This system ensured that player settings were preserved between play sessions, providing a consistent experience.

The implementation of the options menu involved creating a simple but effective interface for adjusting audio levels and saving these settings, which was achieved using sliders and buttons.

Level Design and Procedural Generation:

Procedural generation was at the heart of the game's level design, providing players with a unique experience each time they played. The MapGenerator class played a central role in this process, using cellular automata to generate the layout of each level. This approach allowed for the creation of diverse environments that were both challenging and unpredictable, keeping players engaged over multiple playthroughs.

Dynamic NavMesh Generation:

Once the map was generated, the NavMesh was baked onto the terrain, allowing enemy AI to navigate the environment effectively. This dynamic generation of the NavMesh was crucial for ensuring that the AI could adapt to the changing layout of the levels, providing a consistent challenge to the player.

The integration of NavMesh generation with the procedural map generation system required careful consideration of timing and resource management. NavMesh generation is a computationally intensive process, and it needed to be performed after the level was fully generated but before the AI agents began their pathfinding routines.

This was achieved by triggering the NavMesh baking process as the final step in the level generation pipeline, ensuring that the AI had accurate and up-to-date navigation data.

Enemy Spawning and Wave Progression:

Enemy spawning was another key aspect of the level design, with the EnemySpawner class responsible for placing enemies in the level and managing their progression over time.

The EnemySpawner was designed to increase the difficulty of the game by spawning more enemies and more powerful enemy types as the player progressed. This system was also tied to the scoring system, with higher scores being awarded for defeating more challenging enemies and surviving longer.

The challenge in implementing the EnemySpawner was ensuring that the difficulty curve was smooth and that the game remained challenging but fair.

This required careful tuning of the enemy spawn rates, the types of enemies spawned, and the timing of enemy waves. The EnemySpawner also needed to coordinate with the NavMesh system, ensuring that enemies were spawned in locations where they could immediately begin navigating the environment.

The progression of enemy waves was designed to increase the intensity of the game over time, providing a gradual escalation in difficulty that kept the player engaged. This was achieved by

increasing the number and strength of enemies in each successive wave, as well as introducing new enemy types with more complex behaviors.

The wave progression system added an additional layer of strategy to the game, as players needed to manage their resources and plan their movements carefully to survive the increasingly difficult encounters.

Audio Management and Advanced Mechanics:

Audio management was an essential part of creating an immersive gameplay experience. The OptionsMenu class allowed players to control the main audio volume via a slider, with settings being saved using PlayerPrefs for persistence. This system ensured that players could customize their audio experience and that these preferences were maintained across play sessions.

In-Game Audio and Effects:

In-game audio effects were implemented for player actions such as shooting and taking damage, providing immediate feedback and enhancing the overall immersion of the game. These audio cues were essential for communicating important gameplay information to the player, such as when they had successfully hit an enemy or when they were taking damage themselves. The audio effects were carefully selected and timed to match the visual and gameplay elements, ensuring a cohesive and satisfying experience.

Background music was also considered as part of the audio management system, with the potential for dynamic changes based on the game's state. For example, the music could become more intense during enemy encounters or transition to a different track when the player reached a certain milestone. Although background music was not fully implemented in the current version of the game, the structure for managing it was laid out, allowing for easy integration in future updates.

Wave-Based Enemy Spawning and Difficulty Scaling:

Wave-based enemy spawning was an advanced mechanic designed to increase the game's difficulty over time. As the player survived longer and defeated more enemies, the game would progressively spawn more challenging waves of enemies. This mechanic was closely tied to the scoring system, with higher scores being awarded for surviving longer and defeating tougher enemies.

The implementation of wave-based spawning required careful balancing to ensure that the difficulty increased at a manageable rate. If the difficulty ramped up too quickly, players might become overwhelmed and frustrated; if it increased too slowly, the game might become too easy and lose its challenge. Extensive playtesting was conducted to find the right balance, adjusting the number of enemies, their spawn rates, and their behaviors to create a challenging yet fair experience.

Planned In-Game Store System:

A planned feature that was not fully implemented due to time constraints was the in-game store system. This system would have allowed players to spend gold earned from defeating enemies to upgrade their character's stats, such as health, damage, and fire rate.

The store system would have added an additional layer of strategy to the game, as players would need to decide how to allocate their resources to maximize their chances of survival.

The design of the store system included several upgrade paths, each offering different benefits to the player. For example, upgrading health would allow the player to survive longer, while upgrading damage would increase the effectiveness of their attacks.

The store system was also intended to be integrated into the game's progression system, with more powerful upgrades becoming available as the player advanced through the game.

Although the store system was not completed in time for the final version of the game, its design and potential impact on gameplay are discussed in this report.

The addition of the store would have provided players with more choices and a greater sense of progression, enhancing the overall depth and replayability of the game.

Note that I didn't include any sound effects everything for it, but I have chosen not to ruin the experience by implementing poor quality sound effects and music, so I made all the structure necessary for audio in the game and now it's a matter of holding and dragging files and references.

Testing, Debugging, and Project Structure:

Throughout the development process, extensive testing and debugging were conducted to ensure that the game was stable and that all systems functioned as intended. This section of the report discusses the tools and techniques used for testing and debugging, as well as the overall project structure and best practices employed during development.

Logging and Debugging Tools:

Logging was a key tool used for debugging, with console outputs providing real-time feedback on the game's state. This was particularly useful for tracking the scoring system, enemy behaviors, and other dynamic elements of the game. For example, logs were used to verify that scores were being correctly calculated and updated, and that enemies were transitioning between states as expected.

Gizmos were also used in the Unity editor to visually represent important elements such as enemy detection radii, which helped in fine-tuning the AI's behavior. Gizmos allowed the development team to see how far an enemy could detect the player and adjust the detection range as needed. This visual feedback was invaluable for ensuring that the AI behaved as intended and that gameplay was balanced and fair.

Modular Project Structure:

The project was organized using a modular structure, with dedicated scripts for different gameplay and UI functionalities. This separation of concerns made the codebase easier to manage and allowed for more efficient debugging and testing. Each module, such as the AI system, projectile mechanics, and game management, was developed as a self-contained unit, with clearly defined interfaces for interacting with other parts of the game.

This modular approach also facilitated teamwork, as different members of the development team could work on separate modules without interfering with each other's work. For example, one developer could focus on improving the AI system, while another worked on the UI, with both modules being integrated into the game without conflicts. The modular structure also made it easier to add new features or modify existing ones, as changes could be made to individual modules without affecting the rest of the codebase.

Singleton Patterns for Key Managers:

Singleton patterns were employed for key managers such as the ScoreManager and PlayerController, ensuring that only one instance of these classes existed at any time. This approach helped prevent issues related to multiple instances and made the overall project more robust. For example, the ScoreManager was responsible for tracking and updating scores across different scenes and ensuring that only one instance of the ScoreManager existed at any time prevented conflicts and inconsistencies.

The use of singletons also simplified the design of the game's architecture, as it provided a clear and consistent way to access important game managers from anywhere in the codebase. This made it easier to implement features that relied on global data, such as the scoring system, and reduced the risk of bugs caused by duplicate instances or conflicting data.

Performance Optimization:

Performance considerations were a key focus during development, with efforts made to optimize the NavMesh generation and pathfinding processes, as well as UI updates and game object destruction. These optimizations were crucial for maintaining a smooth and responsive gameplay experience, particularly in more complex levels with multiple enemies and dynamic elements.

For example, the NavMesh generation process was optimized to minimize the time required to bake the NavMesh after each level was generated. This was achieved by reducing the complexity of the level geometry, combining adjacent walls into larger segments, and avoiding the creation of unnecessary triangles. The pathfinding system was also optimized by adjusting the NavMeshAgent's parameters to balance accuracy and performance.

UI updates were another area of focus for performance optimization. The UI elements, such as the score display and health bar, were updated frequently during gameplay, and these updates needed to be efficient to avoid impacting the game's frame rate. This was achieved by minimizing the number of UI updates per frame and using optimized methods for rendering text and images.

Critical Analysis and Reflection:

The development of this Unity-based game was a complex and challenging project that required careful planning, iteration, and problem-solving. Throughout the process, several aspects of the project went well, while others presented significant challenges and opportunities for improvement.

Successes in Procedural Generation and AI Development:

One of the most successful aspects of the project was the implementation of the procedural generation system. The use of cellular automata allowed for the creation of unique and varied levels that provided a consistently engaging experience for players. The integration of mesh generation and NavMesh baking with the procedural generation system was also successful, ensuring that the levels were both visually appealing and functional for AI navigation.

The AI system was another area of success, with the EnemyBase class providing a flexible foundation for implementing different enemy behaviors. The use of the NavMeshAgent for pathfinding, combined with a finite state machine for behavior management, resulted in dynamic and challenging enemy encounters. The specific behaviors of the TurtleEnemy and MageEnemy were well-implemented, providing distinct challenges for the player to overcome.

The projectile system also worked well, with the ProjectileMoverBase class providing a solid foundation for creating different types of projectiles. The homing projectiles added an extra layer of challenge to the combat, requiring the player to adapt their strategy and making the gameplay more dynamic and engaging.

Challenges in Mesh Generation and Scoring System:

However, there were also areas where the project encountered significant challenges. One of the most difficult aspects was ensuring that the mesh generation process produced correct and visually consistent results.

Early iterations of the MeshGenerator script led to issues such as inverted walls and gaps in the mesh, which required extensive debugging and refinement to resolve. These issues highlighted the complexity of procedural mesh generation and the importance of thorough testing and validation in ensuring that the generated geometry was both accurate and optimized.

Another challenge was the implementation of the scoring system, which initially encountered issues with accurately tracking and updating scores. This was eventually resolved through careful debugging, but it highlighted the complexity of managing dynamic systems in a real-time environment. The scoring system was a critical component of the game's feedback loop, and any inaccuracies could have significantly impacted the player's experience.

Reflection on Unfinished Features and Future Improvements:

The planned in-game store system was a feature that was ultimately not completed due to time constraints. This system would have added significant depth to the gameplay, providing players with more strategic options and a greater sense of progression. Given more time, this feature would have been a valuable addition to the game, offering players the opportunity to customize and upgrade their character's abilities based on their playstyle.

Looking back, there are several areas where the development process could have been improved. More time spent on planning and design, particularly for the more complex systems such as the in-game store, could have helped prevent some of the issues encountered during implementation. Additionally, a more rigorous testing process earlier in development could have identified and resolved some of the technical challenges, such as the mesh generation issues, more quickly.

In terms of future improvements, several enhancements could be made to the game based on the lessons learned during this project. For example, the AI system could be expanded to include more advanced behaviors, such as teamwork between enemies or adaptive strategies based on the player's actions. The procedural generation system could also be enhanced to create more complex and varied environments, incorporating elements such as multiple floors, traps, and environmental hazards.

Conclusion:

The development of this Unity-based game was a complex and multifaceted endeavor that required the seamless integration of various advanced systems, including procedural generation, artificial intelligence (AI), and game management. This project not only demonstrated the technical challenges inherent in creating a dynamic and engaging game environment but also underscored the importance of robust design principles, iterative testing, and continuous improvement.

One of the most significant achievements of this project was the successful implementation of the procedural generation system. By employing cellular automata, the project was able to create unique and unpredictable levels that provided players with fresh experiences on every playthrough. This approach to level design is particularly valuable in rogue-like games, where the replayability of the game is a key factor in its longevity and player engagement. The challenges faced during the development of the procedural generation system, such as balancing randomness with playability and ensuring that generated levels were both navigable and visually appealing, were substantial. However, through persistent refinement and iteration, the final implementation achieved a balance that not only met but exceeded initial expectations. This success highlights the effectiveness of cellular automata in procedural generation and sets a strong foundation for future projects that may seek to explore even more complex generative techniques.

The AI system was another cornerstone of the project, designed to deliver challenging and varied enemy behaviors that kept the gameplay dynamic and unpredictable. The use of Unity's NavMeshAgent for pathfinding, combined with a finite state machine (FSM) to manage enemy states, allowed for the creation of AI that could adapt to the changing environment and the player's actions. This adaptability was crucial in maintaining a high level of challenge throughout the game. The specific behaviors of the TurtleEnemy and MageEnemy, each with its unique attack patterns and interaction styles, provided the player with a diverse range of challenges that required different strategies to overcome. This diversity in enemy behavior not

only enriched the gameplay experience but also showcased the potential for extending the AI system to include even more complex and nuanced interactions in future developments.

Despite these successes, the project also encountered several significant challenges that provided valuable learning opportunities. One of the most persistent issues was the mesh generation process. Early iterations of the MeshGenerator script resulted in various visual and functional problems, such as inverted walls and gaps in the generated meshes. These issues not only impacted the visual integrity of the game world but also posed challenges for gameplay, as they could potentially disrupt the AI's pathfinding or the player's navigation. Resolving these issues required a deep dive into the mesh generation algorithms and an understanding of the underlying geometry processing. The experience gained from debugging and refining this aspect of the project will be invaluable for future endeavors that involve procedural geometry or similar complex systems.

Another area of difficulty was the scoring system, which initially struggled to accurately track and update the player's score in real-time. This was particularly problematic because the scoring system is a critical component of the game's feedback loop, providing players with a tangible measure of their success and progress. The issues encountered here underscored the complexity of managing dynamic data in a real-time environment, especially when multiple systems are interacting simultaneously. Through careful debugging and a methodical approach to refining the system, these challenges were ultimately overcome, but they serve as a reminder of the importance of thorough testing and validation, particularly in systems that are central to the player's experience.

One of the more ambitious aspects of the project was the planned in-game store system, which unfortunately was not fully realized due to time constraints. This feature would have added a significant strategic layer to the gameplay, allowing players to upgrade their character's abilities in response to the increasing difficulty of the game. The store system was designed to be tightly integrated with the game's progression mechanics, offering upgrades that would become

available as the player advanced. Although this feature was not completed, its design process provided valuable insights into how such systems can enhance player agency and replayability. The lessons learned from this experience highlight the importance of time management and prioritization in game development, particularly when dealing with complex features that require significant integration with existing systems.

Reflecting on the overall development process, it is clear that while the project achieved many of its goals, there were also areas where the approach could have been improved. For instance, more time dedicated to upfront planning and system design could have mitigated some of the challenges encountered later in the project, such as the mesh generation issues. Additionally, implementing a more rigorous testing protocol earlier in the development cycle might have identified these and other issues before they became more significant problems. This reflection emphasizes the need for a balanced approach to game development, where creativity and innovation are supported by solid engineering practices and a commitment to quality assurance.

The insights gained from this project will undoubtedly inform future development efforts. For example, the successful integration of procedural generation and AI systems in this project provides a strong foundation upon which to build more complex and varied game environments. Future projects could explore more sophisticated procedural generation techniques, such as multi-layered cellular automata or hybrid methods that combine rule-based generation with machine learning. Similarly, the AI system could be expanded to include more advanced behaviors, such as cooperative strategies among enemies or adaptive responses to the player's tactics.

Moreover, the challenges faced during the project, particularly those related to mesh generation and the scoring system, offer valuable lessons for improving the development process. Future projects will benefit from a more structured approach to testing and validation,

as well as a greater emphasis on planning and design, particularly for features that are critical to the game's core mechanics.

In conclusion, this project represents a significant achievement in Unity-based game development, demonstrating the potential of procedural generation, advanced AI, and dynamic game management to create engaging and replayable game experiences. While there were challenges along the way, the successes far outweighed the difficulties, and the lessons learned will serve as a strong foundation for future projects. The experience gained from this project has not only enhanced technical skills but also provided a deeper understanding of the complexities and nuances of game development, insights that will be invaluable in any future endeavors in this field.

References List:

- Watkins, R., 2016. *Procedural Content Generation for Unity Game Development*.
- Cook, M., *Procedural Generation and Information Games*. School of Electronic Engineering and Computer Science.
- Baldwin, A., Dahlskog, S., Font, J.M. and Holmberg, J., *Mixed-Initiative Procedural Generation of Dungeons using Game Design Patterns*. Faculty of Technology and Society, Malmö University.
- Capasso-Ballesteros, I.F. and De la Rosa, F., *Generating Automated Rules-Based Game Design Prototypes with MaruGen*.
- Walton, S.P., Rahat, A.A.M. and Stovold, J., *Evaluating Mixed-Initiative Procedural Level Design Tools Using a Triple-Blind Mixed-Method User Study*.
- Shaker, N., Smith, G., and Yannakakis, G.N., 2016. Evaluating content generators. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. *Procedural Content Generation in Games*. Cham: Springer, pp. 219-236.
- Liapis, A., Smith, G., and Shaker, N., 2016. Mixed-initiative content creation. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. *Procedural Content Generation in Games*. Cham: Springer, pp. 195-214.
- Shaker, N., Togelius, J., and Yannakakis, G.N., 2016. The experience-driven perspective. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. *Procedural Content Generation in Games*. Cham: Springer, pp. 171-193.
- Ashlock, D., Risi, S., and Togelius, J., 2016. Representations for search-based methods. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. Procedural Content Generation in Games. Cham: Springer, pp. 147-168.

- Ashlock, D., Risi, S., and Togelius, J., 2016. Representations for search-based methods. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. *Procedural Content Generation in Games*. Cham: Springer, pp. 147-168.
- Nelson, M.J. and Smith, A.M., 2016. ASP with applications to mazes and levels. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. Procedural Content Generation in Games. Cham: Springer, pp. 123-144.
- Cheong, Y-G., Riedl, M.O., Bae, B-C., and Nelson, M.J., 2016. Planning with applications to quests and story. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. *Procedural Content Generation in Games*. Cham: Springer, pp. 101-121.
- Nelson, M.J., Togelius, J., Browne, C., and Cook, M., 2016. Rules and mechanics. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. *Procedural Content Generation in Games*. Cham: Springer, pp. 79-99.
- Togelius, J., Shaker, N., and Dormans, J., 2016. Grammars and L-systems with applications to vegetation and levels. In: Shaker, N., Togelius, J., and Yannakakis, G.N., eds. *Procedural Content Generation in Games*. Cham: Springer, pp. 57-77.