Mega Man X

Sprint 4
December 1, 2024
Client: Dr. Galloway

Name	**Email Address**
Ryan Rutledge	ryan.rutledge325@topper.wku.edu
Kaden Groves	kaden.groves757@topper.wku.edu
Eli Adkisson	elijah.adkisson534@topper.wku.edu
A. R. Chessor	anthony.chessor038@topper.wku.edu

CS 360
Fall 2024
Project Technical Documentation

Contents

1 Introduction **1**
1.1 Project Overview . . . 1
1.2 Project Scope . . . 1
1.3 Technical Requirements . . . 2
1.3.1 Functional Requirements . . . 2
1.3.2 Non-Functional Requirements . . . 3
1.4 Target Hardware Details . . . 4
1.5 Software Product Development . . . 5

2 Modeling and Design **6**
2.1 System Boundaries . . . 6
2.1.1 Physical . . . 6
2.1.2 Logical . . . 6
2.2 Wireframes and Storyboard . . . 6
2.3 UML . . . 6
2.3.1 Class Diagrams . . . 6
2.3.2 Use Case Diagrams . . . 9
2.3.3 Use Case Scenarios Developed from Use Case Diagrams (Primary, Secondary) . . . 9
2.3.4 Sequence Diagrams . . . 10
2.3.5 State Diagrams . . . 12
2.3.6 Component Diagrams . . . 13
2.3.7 Deployment Diagrams . . . 14
2.4 Requirements Traceability Table . . . 15
2.5 Version Control . . . 16
2.6 Data Dictionary . . . 17
2.7 User Experience . . . 18
2.7.1 Gameplay Diagram . . . 18
2.7.2 Gameplay Objectives . . . 19
2.7.3 User Skillset . . . 19
2.7.4 Gameplay mechanics . . . 20
2.7.5 Gameplay Items . . . 20
2.7.6 Gameplay Challenges . . . 24
2.7.7 Gameplay Menu Screens . . . 25
2.7.8 Gameplay Heads-Up Display . . . 25
2.7.9 Gameplay Art Style . . . 26
2.7.10 Gameplay Audio . . . 26

3 Non-Functional Product Details **26**
3.1 Product Security . . . 26
3.1.1 Approach to Security in all Process Steps . . . 26
3.1.2 Security Threat Model . . . 26
3.1.3 Security Levels . . . 27
3.2 Product Performance . . . 27
3.2.1 Product Performance Requirements . . . 27
3.2.2 Measurable Performance Objectives . . . 28
3.2.3 Application Workload . . . 28
3.2.4 Hardware and Software Bottlenecks . . . 28
3.2.5 Synthetic Performance Benchmarks . . . 29
3.2.6 Performance Tests . . . 30

4 Software Testing **31**
4.1 Software Testing Plan Template . . . 31
4.2 Unit Testing . . . 32
4.2.1 Source Code Coverage Tests . . . 34
4.2.2 Unit Tests and Results . . . 34
4.3 Integration Testing . . . 34
4.3.1 Integration Tests and Results . . . 36
4.4 System Testing . . . 36
4.4.1 System Tests and Results . . . 36
4.5 Acceptance Testing . . . 36
4.6.1 Acceptance Tests and Results . . . 36

5 Conclusion **36**

6 Appendix **36**
6.1 Software Product Build Instructions . . . 36
6.2 Software Product User Guide . . . 37
6.3 Source Code with Comments . . . 37
4.6 Software Testing Plan Template . . . 63

List of Figures

1 Diagram that shows the physical boundaries of our system. 7
2 Diagram that shows the logical boundaries of our system. 8
3 Mega Man X Storyboard 9
4 Diagram of A singleton design pattern that implements in game music.(GameMusic.cs on page 46) 10
5 Diagram of an observer design pattern that detects bullet collision with enemies.(Enemy.cs on pages 44-45) 11
6 Diagram of a strategy design pattern that shows the different bullet types.(PlayerBullet.cs on pages 49-50) 12
7 Diagram of a facade design pattern that is used for the score management system and its connection to database.(See pages 39-44 for DatabaseController.cs, pages 54-55 for ScoreSaver.cs, and page 55 for TimeAndScoreController.cs) 13
8 Use case diagram of the over arching design of Mega Man X gameplay 14
9 This shows the use case scenario for enemy player movement. 15
10 This Image represents the use case Scenario for the user login system. 16
11 This Image represents the use case Scenario for the player movement. 17
12 This Image represents the use case Scenario for player shooting scenario. 18
13 This Image represents the use case Scenario for the store scoring scenario. 19
14 This Image represents the use case Scenario for the take damage scenario. 20
15 Sequence diagram of player attack sequence 21
16 State diagram representing the user interface states. 21
17 Component diagram that shows the connection between the scripts and database in the game. . . 22
18 Connection between the game and the local database all on the client's system. 23
19 Gameplay Diagram. 24
20 Menu Screens Wireframe 25
21 Security Threat Model 27
22 Application Workload Testing 28
23 File I/O Throughput 29
24 CPU Total Events 30
25 Execution Time 30
26 Line Graph Visual of Memory Usage in Game Scene 31
27 Detailed Memory Usage in Game Scene 31
28 CPU Usage in Game Scene 31
29 Acceptance Testing One Template1 63
30 Acceptance Testing One Template2 64
31 Acceptance testing results page 1 65
32 Acceptance testing results page 2 66

1 Introduction

1.1 Project Overview

The Mega Man X Replica project aims to create a faithful recreation of the classic action-platformer game Mega Man X. The project will focus on replicating core gameplay mechanics, graphics, and overall experience to bring the iconic 1993 game to a new audience while preserving its nostalgic charm.

The Mega Man X Replica project will recreate the classic starting level called Central Highway, however due to time contraints and the nature of the boss fight in the original game we will not be able to implement the boss fight with Vilel. We hope to add more later on, but with the limited time we have we'll make a demo to show what a fresh, new look could do for the classic SNES game. We want to provide an enjoyable experience for newcomers to the series and hardcore fans as well, and won't change much from how the game played back when it was released in 1993.

Boss found at the end of the level Central Highway named Vile

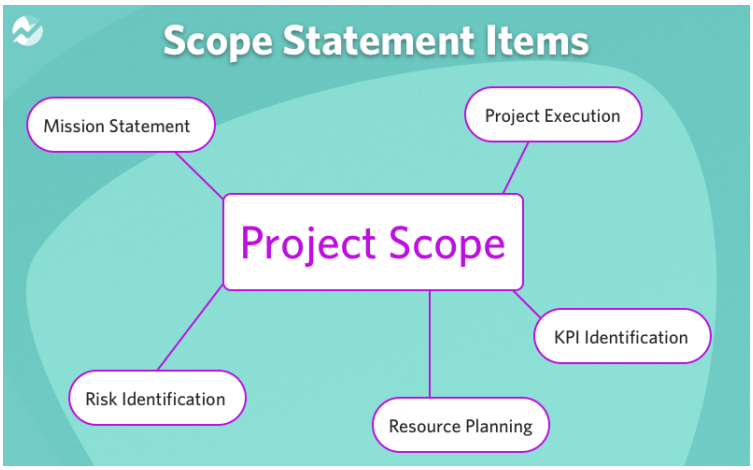
While we're making a replica of the game that feels and plays close to the original Mega Man X, we'll also go ahead and address bugs and remove some possible oversights that we're left in the original, such as spots on the stages that players shouldn't be able to reach. However, we also understand that the game is loved for the way it is currently, and will be diligent about not reinventing the wheel.

1.2 Project Scope

In order to finish the project, it will take us about fourteen weeks. As far as what work will be done in those fourteen weeks, we'll need to write thirty-two pages of documentation in our technical document, which includes overviews, requirements, diagrams, testing, wireframes, storyboard and more. We've assigned two to three people to help compile all that is need and to type out all the information for this document. We'll also need to make an eight pages of an organizational document that will explain our schedule, distribution of work, risk management, software process model and more. We will also need to make our own replica of Mega Man X using the Unity game engine, and will be making scripts and remaking the classic levels from the SNES game. We will also need presentations every three to four weeks to show our progress over the 14 weeks we have to finish all the deliverables. The benefits of this project is experience for us students on how to properly communicate and coordinate a team of people to efficiently finish a project while different people work on different parts of the project. Getting familiar with working as a team, which is common place in the work field, will benefit us greatly.

Our project will need to be finished by November 30, 2024 that has our completed level, eight page organizational document, thirty-two page technical document, and presentation to show off the completed product. Before

the final deadline, our team will be tasked with making presentations to show our progress that are due on the dates of September 2, 2024, September 30, 2024, October 28, 2024, and on November 25, 2024. We will also submit our documents with the presentations, which will still be a work in progress until November 30, 2024.



Project Scope example

1.3 Technical Requirements

1.3.1 Functional Requirements

Mandatory Functional Requirements
1. Should allow the main character to be controlled by the keyboard or gamepad.
2a. Should allow multiple players to create separate accounts.
2b. Should implement a login system for users and administrators.
3. Should save the state/progress of all players and allow players to continue once they've logged into their account.
4. Should display a splash screen when launched which also shows top scores and the login prompt.
Extended Functional Requirements
5. Should implement basic enemy logic scripts.
6a. Should allow the Fire function to be held indefinitely.
6b. Should allow different functionality after the Fire function has been held for a certain amount of time.
7. Should allow player to take damage when hit by enemy

The aforementioned functional requirements for this project are included here and enforced in order to make sure that the game functions well and users enjoy their experience. The reasons for these requirements are:

1. Central to the project is the requirement for the main character to be controlled via keyboard or gamepad, providing flexibility in gameplay.

2. The project also focuses on user management by allowing multiple players to create individual accounts, which supports personalized gaming experiences by keeping player progress, and creates a secure login system for both users and administrators that is crucial to managing access and protecting user data.

3. To ensure continuity, the project includes a feature to save players' progress, allowing them to resume their journey upon logging in.

4. Additionally, the project aims to create an engaging entry point with a splash screen that displays top scores and a login prompt, setting the tone for the game.

5. The project also includes enemy logic such as movement and attack to create engaging action for the player.

6. The project includes a feature that allows the player to hold their attack to charge it, providing a more powerful attack.

Image showing Mega Man X using his charge shot

These features collectively aim to deliver a seamless and secure gaming environment for all users just like the original Mega Man X.

1.3.2 Non-Functional Requirements

The non-functional requirements outlined for this project are crucial to ensuring the quality and accuracy of the original game. The reasons for these requirements are:

1. Utilizing the Unity game engine and development environment is essential, as it provides a reliable platform for creating robust and creative games such as Mega Man X.

2. Creating a complete replica of the game ensures that the project meets the expectations set by the original, preserving the gameplay experience.

3. Allowing the use of pre-developed media assets speeds up the development, enabling the team to focus on core programming tasks.

4. Requiring all source code to be developed by the CS 360 team fosters innovation, learning, and originality.

5. The inclusion of a database is vital for managing player data, user accounts, and game progress securely and efficiently.

6. Gathering and optimizing performance, security, and user interface metrics is necessary to deliver a secure, and intuitive gaming experience, ensuring the game runs efficiently and meets modern standards for usability and data protection.

7. Given our time allotted to our team to make a working replica of Mega Man X, we decided it would be best to make the game run as smoothly as possible on just one platform. Given that we all use Windows, it would make the most sense to develop the game for this operating system.

Mandatory Non-Functional Requirements
1. Should be implemented using the Unity game engine and development environment.
2. Should create a complete replica of the game Mega Man X.
3. Can use pre-developed images, artwork, audio, and other related media.
4. All source code must be developed by the CS360 project team
5. Must use a database
6a. Performance metrics should be gathered and optimized
6b. Security metrics should be gathered and optimized.
6c. User interface metrics should be gathered and optimized.
Extended Non-Functional Requirements
7. Must be executable on the Windows OS.
8. Should retain stable frame rate and process user actions swiftly
9. The game should be able to handle multiple enemies on the screen at a time
10. HUD should actively update with current changes in the game.
11.Input validation must ensure that only appropriate data is processed by the application.
12. The system should implement effective session management to handle user sessions securely and efficiently.
13. Access control mechanisms should restrict unauthorized users from accessing sensitive areas of the application.
14. Error handling and logging must be included to capture and manage unexpected events or failures.
15. Data protection measures should safeguard user data, including encryption of sensitive information.
16. Database security protocols must prevent unauthorized access and ensure data integrity in storage and retrieval.
17. General coding practices should be followed to improve code maintainability, readability, and performance.
18. Password management and authentication protocols must secure user accounts and protect against unauthorized access.

Database Example

All of these non-functional requirements take into consideration the time given to use and the demands of the client or target audience, and to ensure both of their satisfaction upon the release of the game.

1.4 Target Hardware Details

Example of the Windows PC the game would run on

- CPU: The game will be a demo, and a small one at that. There shouldn't be any hardware intensive rendering, so an average CPU should be able to run the game at less than 20-30 percent utilization.
- Ram: Less than 4 GB
- Persistent Storage Space: Any type of storage should be fine, but HDD or SSD is recommended. Size should be no more than 1-2 GB

- Network Connection: Wi-Fi or Ethernet should be just fine
- Network Bandwidth: Any bandwidth should be acceptable, just enough to log in to the database and pull small amounts of data. 10 KB upload and download should do the trick
- Output devices: Monitor required to play the game, speakers/headphones are recommended in order to hear game audio and music
- Input devices: Keyboard is required to play the game, while a mouse will be required to navigate the log in menu and leaderboard

The game will be tested to ensure that all of these hardware details are true before the full release of the game to make certain that these statements mentioned are true.

1.5 Software Product Development

Unity logo

- IDEs: The project will be developed utilizing the Unity game engine. Our team will utilize the Unity IDE so that we may develop the code, test, and optimize it in a reliable environment. In addition to Unity, we will also be using Visual Studio to write our scripts for the project. This is due to how Visual Studio has a lot of support for C Sharp, and since some members of the team are already acquainted with Visual Studio.
- Software Languages: The language used by the software developers for the code in the project will be C Sharp. Our team's reason for using C Sharp is that it shares the most compatibility with unity engine and

is readily accessible when using Visual studio code, as well as having team members that are familiar with the high-level programming language.

- Version Control: The project will be Version Controlled using GitHub, and the team has been instructed on how to successfully use Git and GitHub for file sharing. This platform enables our team to manage and synchronize code efficiently, as well as leave notes on what exactly happened between changes in certain files.
- Tools for Collaboration: Google Drive will be used when necessary for different tasks. So far, we have used Google Docs for working on documentation and Google Slides for working on our presentations. In the future, Google Spreadsheet may or may not be used by the team as well.
- Sprite Editor: The Unity Sprite Editor will be used by the software engineers for Asset Generation, however due to the nature of the project it is likely that most sprites will be found online.

2 Modeling and Design

2.1 System Boundaries

2.1.1 Physical

The physical system boundaries are extremely important as hardware provides the components for the system to operate. At the center of this is the CPU, this component is extremely useful for executing instructions. This amount of ram is important as it provides the system with temporary storage. The input components are important to give users functionality so that they can operate with the system. The output components such as the monitor is extremely important so that they can see how the program is functioning visually. The memory is important as we need to be able to download the game to a non volatile memory on the computer.

2.1.2 Logical

The logical system boundaries outlines the components of our system that set boundaries. One logical boundary is the unity event handling , this is important as it determines the interactions that we perform within unity. The windows OS is another logical boundary which is important as the software needs to operate on OS. C Sharp is another logical boundary as it is crucial with writing scipts that work well with unity. Another logical boundary is SQLite Database, this is important as the database needs to fluently work and be able to perform crud operations. These boundaries collectively form the backbone of system functionality.

2.2 Wireframes and Storyboard

At the start of the game, the player's character spawns with the goal of reaching the end of the level before time runs out. As they progress, they encounter various foes, beginning with "Spikey," one of the game's challenging enemies. Throughout the level, players find health capsules, which restore their health and provide a valuable resource as they face tougher encounters. Another adversary, "Ball De Voux," requires strategic attacks to defeat, while flying enemies aggressively dash at the player, adding urgency to every step forward. Players must also navigate obstacles like pitfalls, testing their agility and timing to avoid dangerous falls. Ultimately, the player reaches the final challenge: a boss battle against "Vile." To complete the level, they must conquer Vile in a fierce showdown, marking the victory as they master the level's obstacles, enemies, and time constraints.

2.3 UML

2.3.1 Class Diagrams

Singleton - This design is used to ensure that their is only one instance of the class exists at a time. The problem is that through a game the music may change whether its from the menu screen to the in game screen or level to level, and without proper audio management its possible to have overlapping game music. The solution to this is using the singleton design patter to ensure that only one instance of music is being played at a time. This is implemented by the get instance method checking if any instance of music exists if it doesn't it will create the

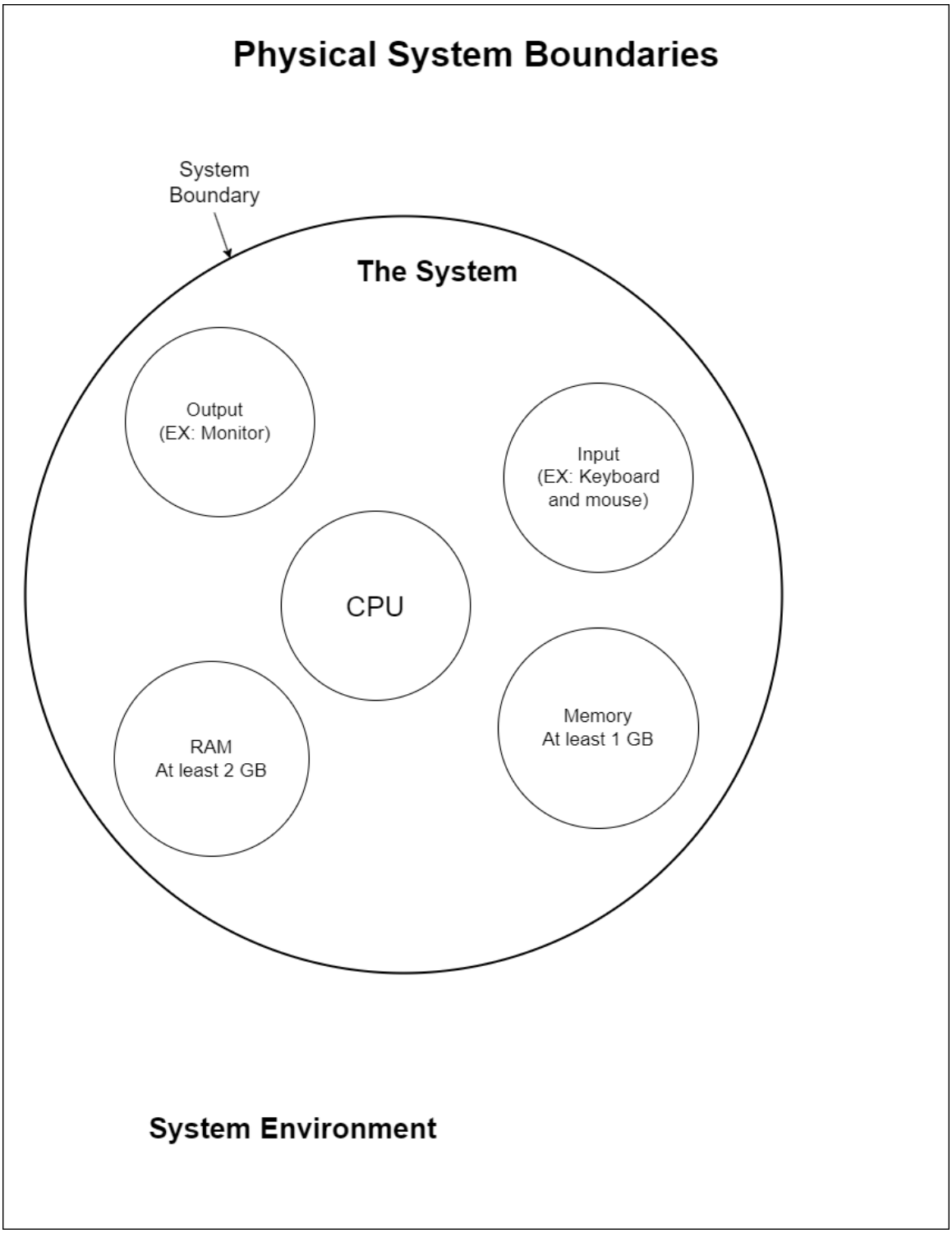


Figure 1: Diagram that shows the physical boundaries of our system.

object, then the game music has a system of single instance music which ensures only one instance is being played. This design pattern could work with the observer pattern to know when the game state changes from levels or screens. This design pattern is typically used of login systems that use database connections. Our team is pretty confident in our decision that the singleton design pattern will be incredibly helpful with this task.

Observer - One of the behavioral design patterns that our team is implementing is the observer. An observer operates by having listener objects paying attention to a subject object when a certain event occurs the listeners will be notified and update. The problem at hand is that detecting collisions with bullet and enemy can be quite complex to keep account of. We plan on using the observer pattern as we will have the health bar observing the

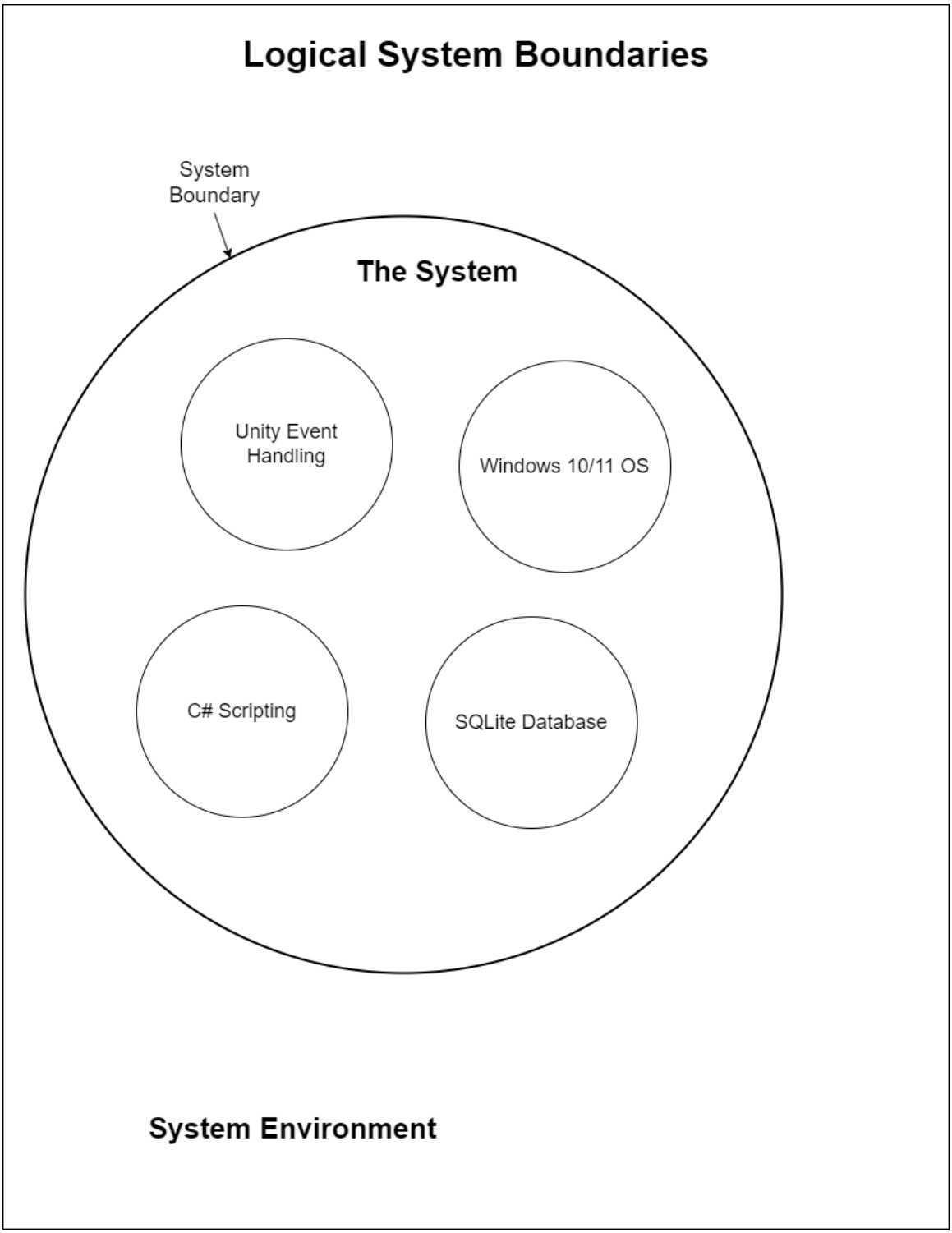


Figure 2: Diagram that shows the logical boundaries of our system.

collision of the bullet object on the enemy object, this should be simple to implement as unity already has the foundation of collision detection. If the collision occur the health bar that is the listener will auto update. Another design pattern that will help with implementation is a state behavioral pattern or a strategy design pattern to know how it will interact. The observer pattern is often used to record the state f an object. Our team decided that implementing an observer for our enemy damage system would be beneficial as it simplifies implementation.

Facade - The structural design pattern that our team decided on was the facade design pattern. The facade pattern is typically used to make simple interface on top of a complex system. The player needs to be able to save their scores to the database the score is managed separately by a scoreboard and saved in database. We think

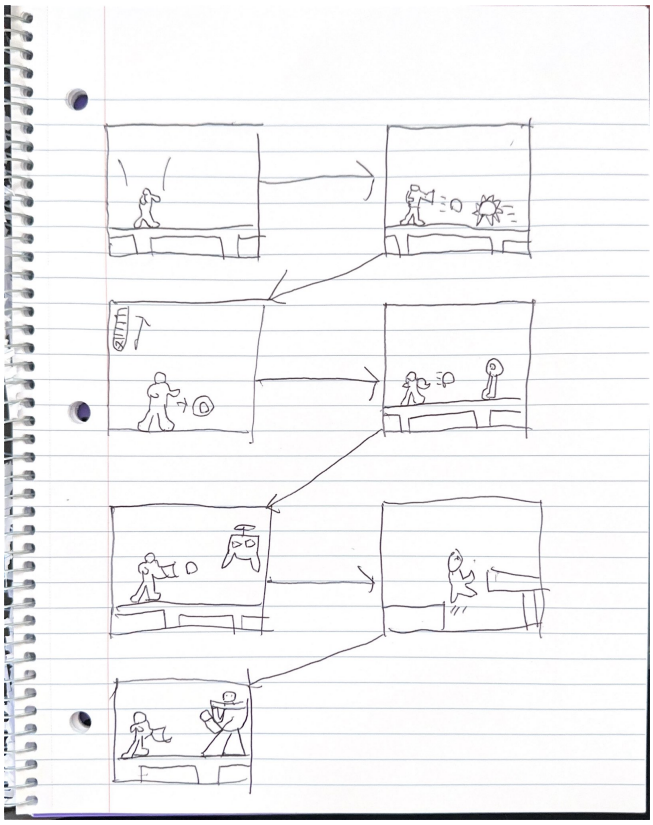


Figure 3: Mega Man X Storyboard

using the facade pattern will help simplify our saving scores system. The implementation consists of scoresaver calling get score from the manager and then saves it to the sql database. No other design patterns are required for this pattern. The facade pattern is commonly used when needing to save player progress. Our team thinks using this pattern will be crucial for simplifying our db saving.

2.3.2 Use Case Diagrams

The diagram represents the use case of the overall gameplay of the Mega Man X project. This diagram gives a visual representation of the relationship between the actors and use cases. The User interacts with various in-game actions such as PlayerShoot, PlayerMove, Pause Game, Take Damage, and Login. The CPU Enemy performs actions like EnemyShoot, EnemyMove, and also causes the player to Take Damage. Additionally, a Database actor is responsible for Storing Scores during gameplay. The diagram visually demonstrates how the player's actions, enemy behavior, and scorekeeping system are interconnected throughout the game.

2.3.3 Use Case Scenarios Developed from Use Case Diagrams (Primary, Secondary)

EnemyMovement: Use case scenario diagram that goes over how the enemy will move. Enemy movement can change depending on the enemy itself, and can also be affected by the player and its surroundings.

Login: Use case scenario diagram that explains how the user login system works for the user as well as how the game will handle the login credentials with the database.

PlayerMove: Use case scenario diagram that goes over how the player will move Mega Man in the game state. Mega Man's movement will also be affected by the terrain and enemies.

PlayerShoot: Use case scenario that explains the special gun firing system in Mega Man X. It goes over the different powers of the bullet types, and how you need to "charge" your weapon to make the stronger bullets.

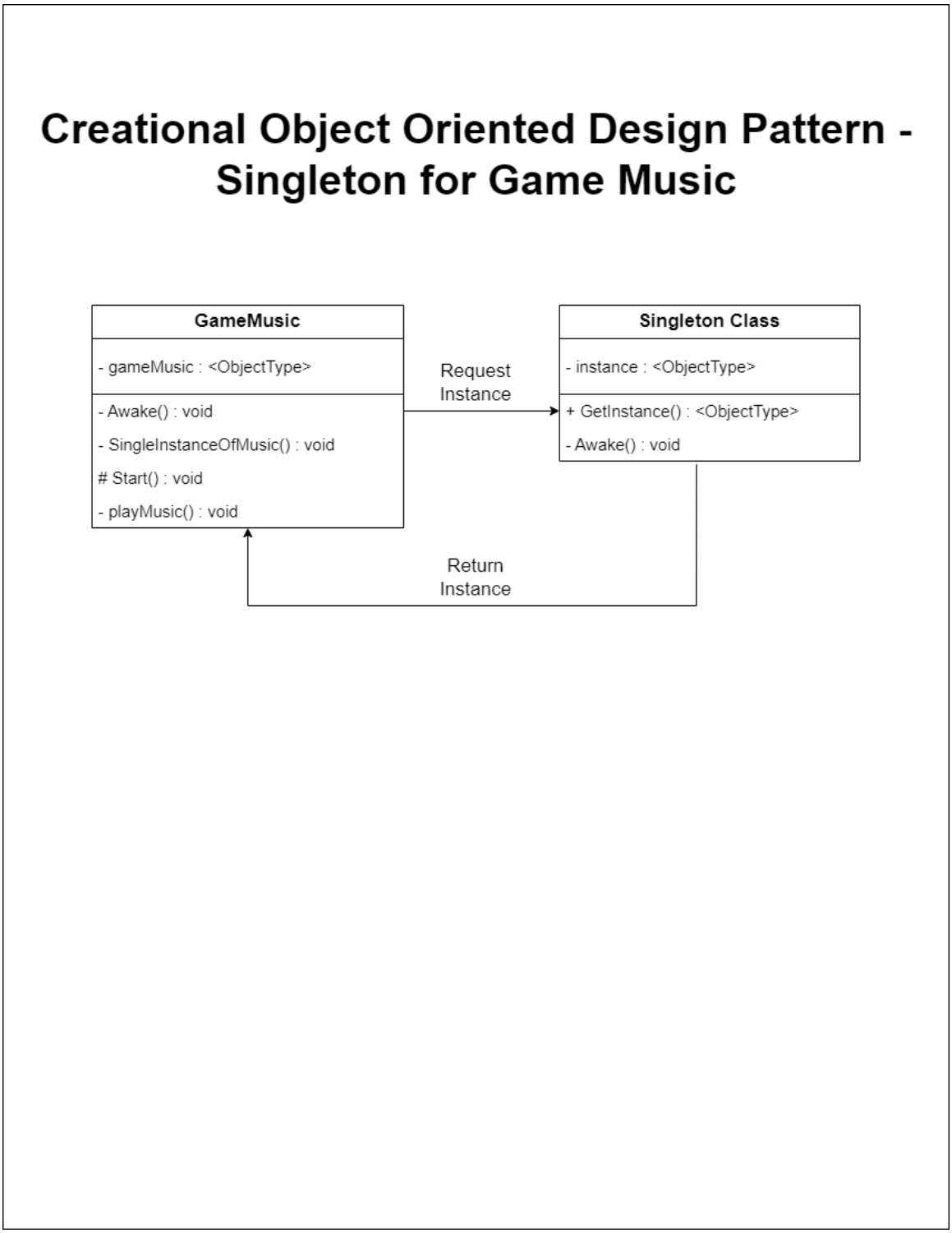


Figure 4: Diagram of A singleton design pattern that implements in game music.(GameMusic.cs on page 46)

StoreScores: Use case scenario that goes over how the game will store the scores into a local database for the user to look back on. Will display the highest scores on the game and who scored them.

TakeDamage: Use case scenario that elaborates on the damage system inside of our game, and how players will take damage, as well as what goes on in the game scripts while that happens.

2.3.4 Sequence Diagrams

The diagram illustrates a Player Attack Sequence for a shot-charging mechanic, where the player's attack power depends on how long a button is held. Initially, when the user presses the button, the player performs a

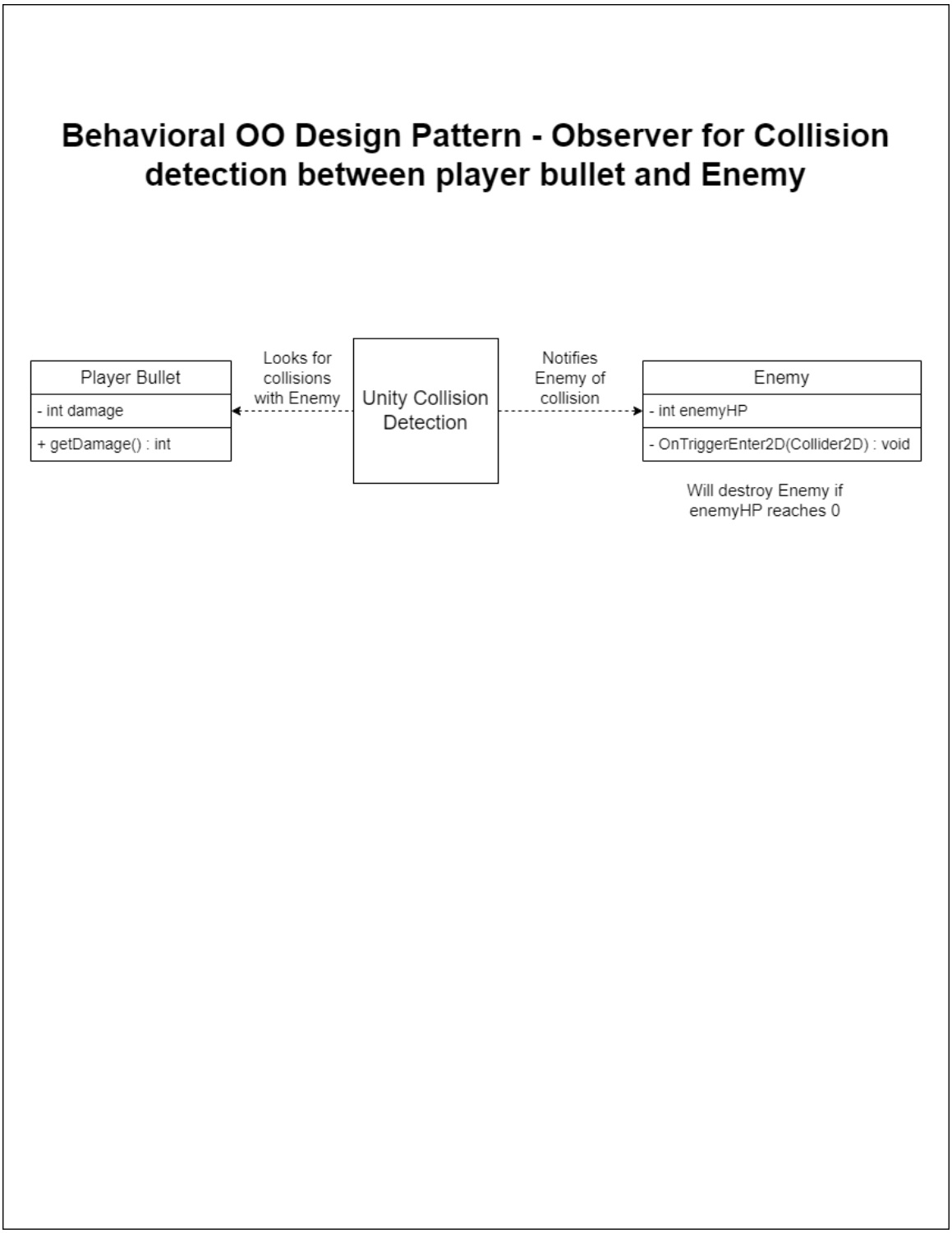


Figure 5: Diagram of an observer design pattern that detects bullet collision with enemies.(Enemy.cs on pages 44-45)

weak shot. If the button is held down, a timer starts, initiating the charge process. Upon releasing the button after a brief hold, the player executes a stronger shot, known as a strong shot. If the button is held even longer, the shot power increases further, leading to a fully charged shot. This sequence shows how holding and releasing the button affects the power level of the attack.

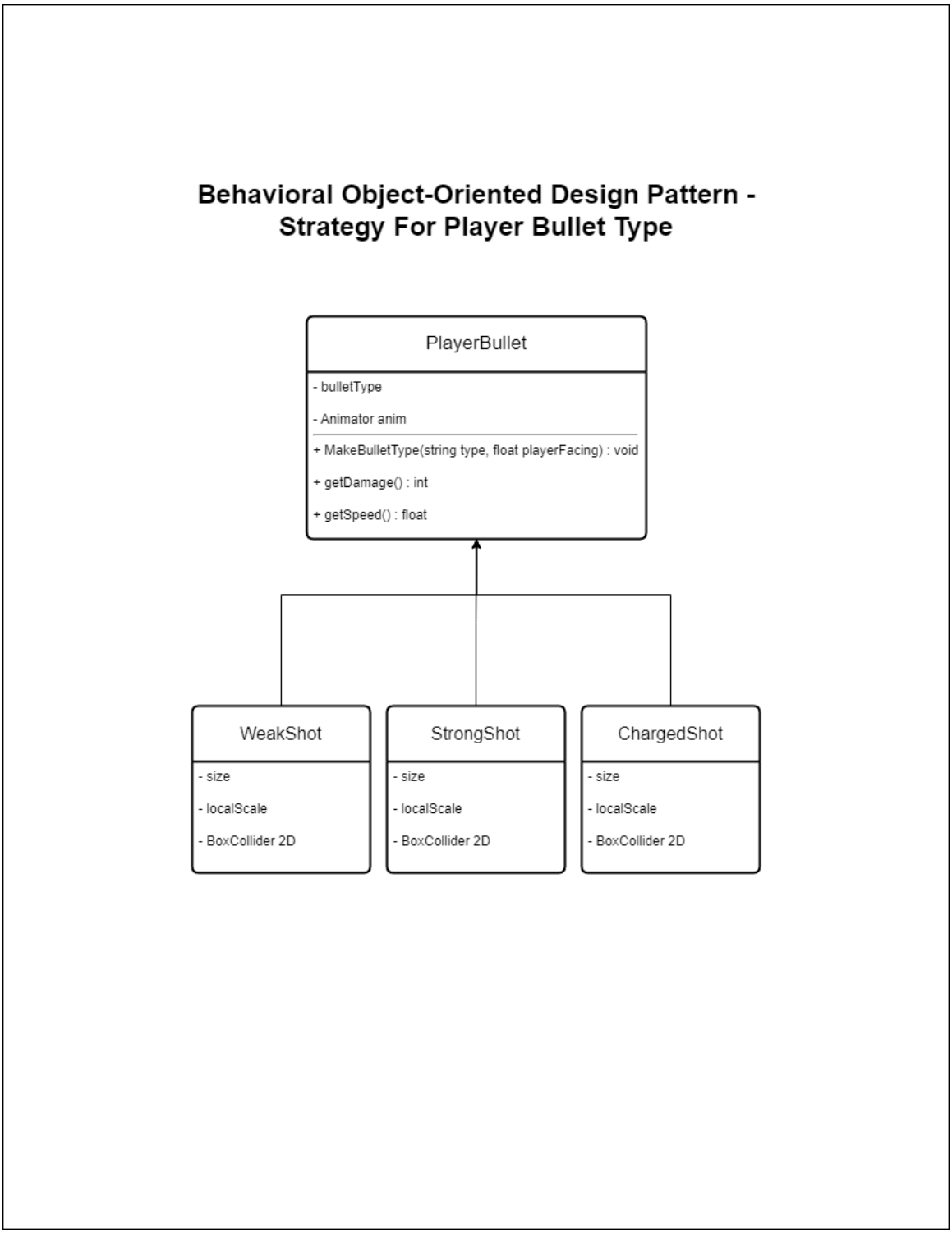


Figure 6: Diagram of a strategy design pattern that shows the different bullet types.(PlayerBullet.cs on pages 49-50)

2.3.5 State Diagrams

The diagram outlines the flow of a user interface for our Mega Man X replica project. The user begins at the Login screen, which leads to the Menu. From the Menu, the user can navigate to various options, including starting the game by selecting Mega Man X, adjusting settings in Options, or exiting the game via Quit. During gameplay, the user can press Pause, which allows access to both the Options menu and the Menu. Navigation between screens is facilitated by Back buttons, enabling the user to return to previous screens. The Quit button fully exits the system. Each interaction is triggered by buttons, guiding the user through the interface

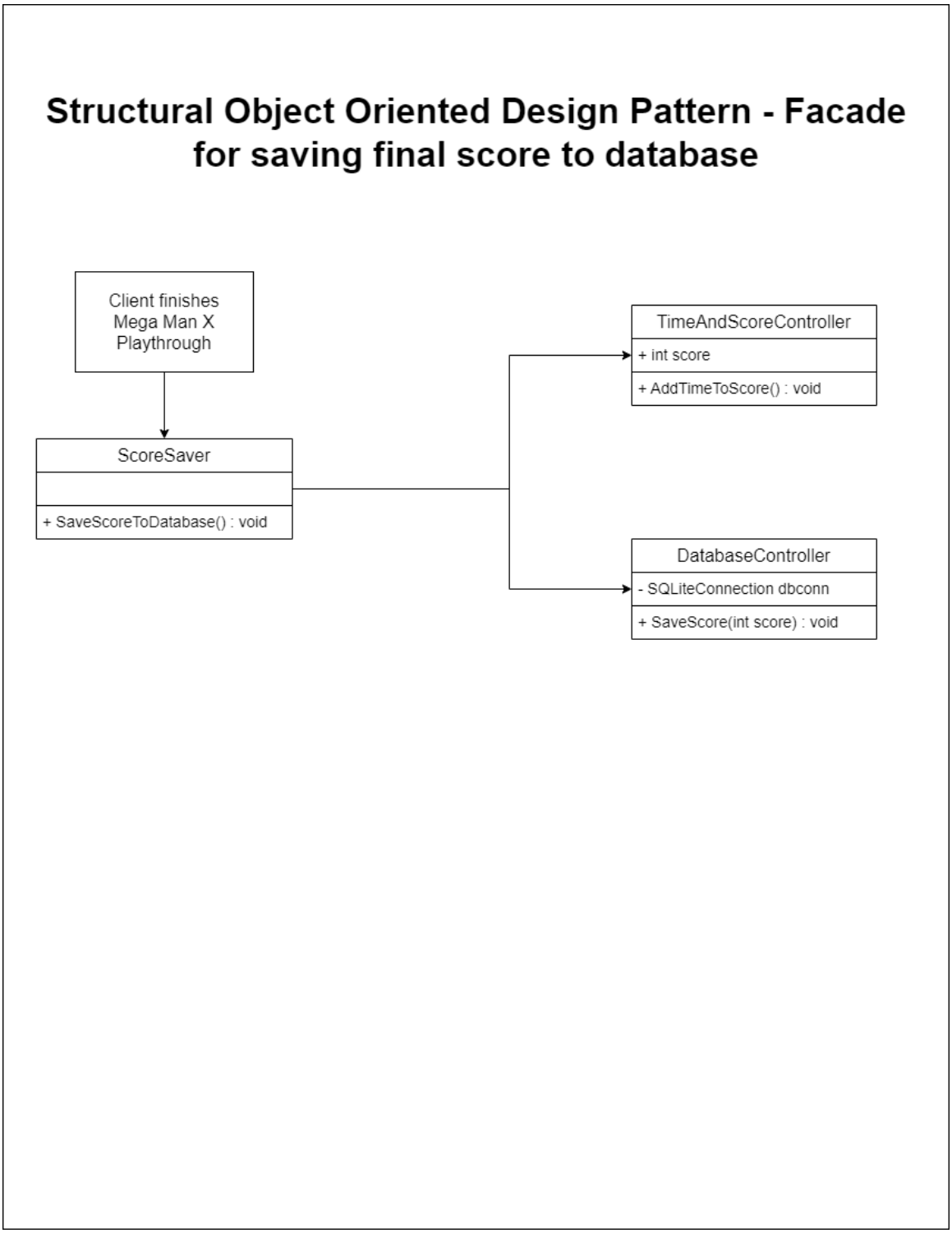


Figure 7: Diagram of a facade design pattern that is used for the score management system and its connection to database.(See pages 39-44 for DatabaseController.cs, pages 54-55 for ScoreSaver.cs, and page 55 for TimeAndScoreController.cs)

2.3.6 Component Diagrams

The Component Diagram for the Scoreboard and Score Saving System focuses on the internal software architecture and the interactions between the components of the system. The SaveManager coordinates operations by communicating with the ScoreboardManager, responsible for updating and displaying scores, and the DatabaseManager, which handles database operations. Communication between these components is through internal function calls or REST APIs if they are deployed on separate networks. The DatabaseManager interacts with the

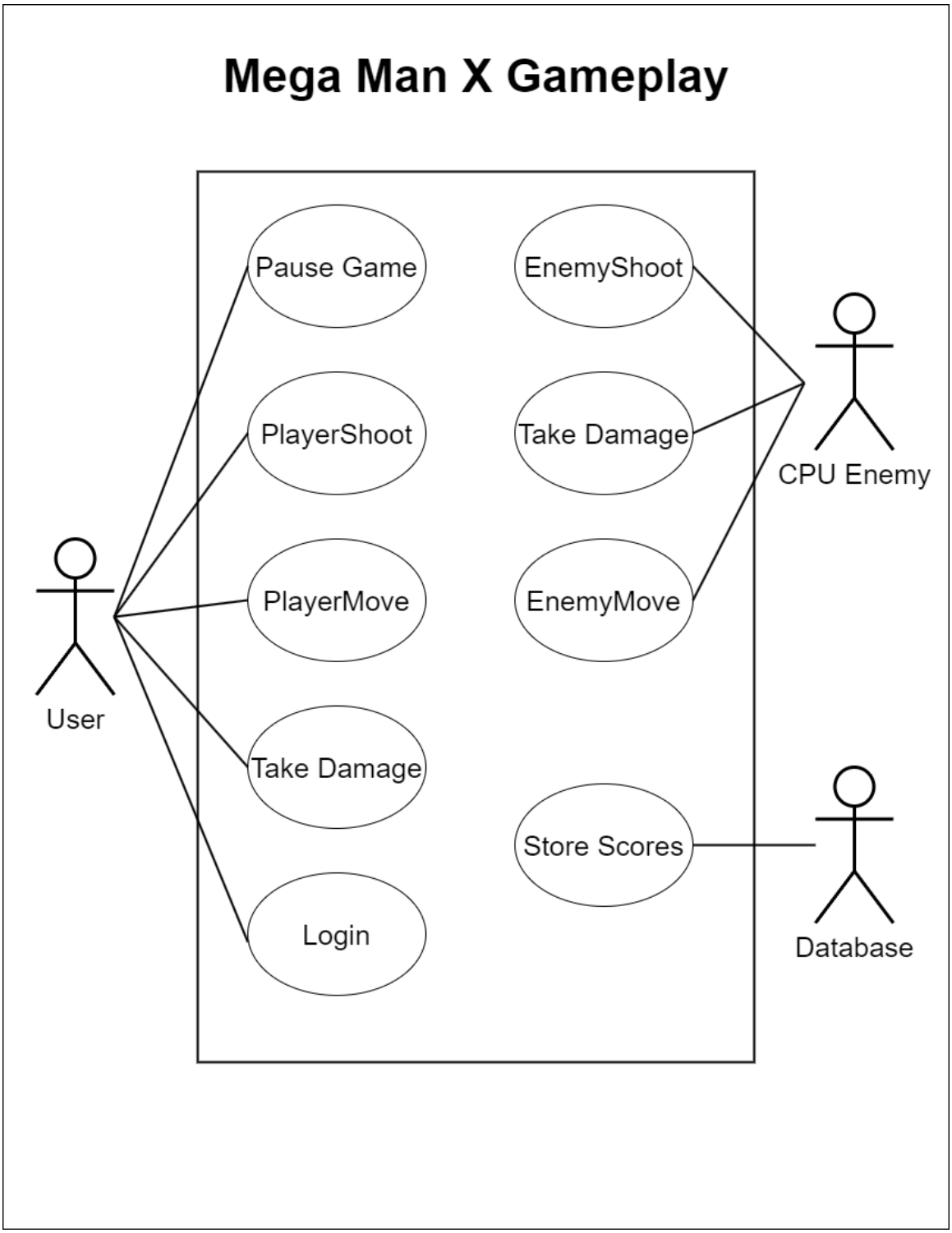


Figure 8: Use case diagram of the over arching design of Mega Man X gameplay

SQLite Database using SQL queries via standard protocols. This diagram also highlights the source code and dependencies, mapping the key components like save_manager.py, scoreboard_manager.py, and database_manager.py to their roles in the system. The focus here is on how the software components interact logically, ensuring that each part of the system communicates effectively and securely.

2.3.7 Deployment Diagrams

The Deployment Diagram for the Scoreboard and Score Saving System shows the physical or virtual locations where key components are deployed and how they communicate. The SaveManager is at the core, coordinating

Use Case: Enemy Movement
ID:UC006
Actor: Enemy
Preconditions: Game is in play state Enemy has spawned in
Primary scenario: 1. Enemy will be spawned in once player has moved close enough(will be done outside of the player's vision). 2. Enemy will follow basic movement function provide inside their script and will move in a passive manner. 3. Once the player has moved close enough to the enemy, the enemy will "lock on" to the player and change it's movement pattern. 4.The enemy may move to the player to attack, may run away, or may not change it's behavior at all. This depends on the enemy's movement script. 5. If the player moves outside the range, the enemy will change it's behavior back to being passive if needed. 6. If the player moves even further away, the enemy will be despawned.
Secondary scenarios: InvalidMovementScript StageCollision

Figure 9: This shows the use case scenario for enemy player movement.

interactions between the ScoreboardManager, which updates and displays scores, and the DatabaseManager, which handles score data storage and retrieval. The DatabaseManager communicates with an external SQLite Database, which stores scores persistently. Communication between these components is shown in terms of physical deployment nodes or devices, and protocols between them, such as network communication to an external database. This diagram shows how data flows between systems, illustrating where each component runs and how they interact over the network. All score-related actions are controlled centrally through the SaveManager, with communication across different physical or virtual systems.

2.4 Requirements Traceability Table

Our requirements tracability table goes over all of our requirements as well as all of our use case scenario diagrams to ensure that we not only have enough requirements to successfully convey what kind of game we are making, but also to ensure that we don't have any unnecessary requirements that don't match the scope of our game. Every "X" inside of the table shows that a particular use case scenario covers a requirement defined in our functional requirements, and we have at least one "X" for each requirement, showing that we don't have too little or too many requirements.

Use Case: Login
ID:UC001
Actor: User
Preconditions: Has the game on their machine.
Primary scenario: 1.user launches the game on their device. 2. The game system will prompt the user with a login screen. 3. Users will input their credentials into the allotted areas. 4.The user will then click the login button. 5.The game system will check the Database and validate the credentials 6. The user will retrieve their game data and settings. 7. Game loads player data and last save state. 8. User is then prompted with the menu screen to play.
Secondary scenarios: InvalidUserCredentials

Figure 10: This Image represents the use case Scenario for the user login system.

2.5 Version Control

Our team follows a collaborative and organized approach to version control for the Mega Man X project. For documentation, we use GitHub, ensuring everyone is on the same page by coordinating through Discord. To prevent version conflicts, we join a dedicated voice channel before making any changes, avoiding simultaneous edits that could lead to corrupt files or multiple versions. When committing updates to GitHub, we always include detailed notes to keep team members informed of changes. Our team follows a collaborative and organized approach to version control for the Mega Man X project. For documentation, we use GitHub, ensuring everyone is on the same page by coordinating through Discord. To prevent version conflicts, we join a dedicated voice channel before making any changes, avoiding simultaneous edits that could lead to corrupt files or multiple versions. When committing updates to GitHub, we always include detailed notes to keep team members informed of changes.For the development side, we rely on Unity's built-in version control, which allows quick and easy integration. This system lets multiple team members work on the project simultaneously without significant issues, helping us maintain smooth progress. Unity's version control also makes it easier to manage different versions, minimizing conflicts and facilitating easier collaboration. Our approach to version control ensures that documentation and development tasks are aligned and handled efficiently, with clear communication and minimized risks of errors. The combination of GitHub and Unity helps streamline our workflow while keeping everything well-coordinated.For

Use Case: PlayerMove
ID:UC002
Actor: User
Preconditions: Logged into the system.
Primary Scenario: 1.User presses the movement function key. 2.The game system checks the assigned action for the key. 3.The system determines the corresponding action (e.g., move left, move right, jump). 4.The game system executes the action, causing Mega Man to perform the designated movement. 5.The system plays the appropriate animation (e.g., walking or jumping) and sound effects associated with the movement. 6.The game updates Mega Man's position on the screen based on the movement direction and distance. 7.The game checks for any obstacles or boundaries in the movement path. 8.If an obstacle is detected, the system prevents the movement and plays a sound or visual cue indicating the obstruction. 9.If the move is valid, Mega Man completes the action, and the system updates the game state accordingly. 10.The user is provided with feedback through visual effects or UI updates, confirming the successful movement (e.g., health bar change, score increment, etc.).
Secondary scenarios: InvalidMovementKey MovementBlockedByObstacle

Figure 11: This Image represents the use case Scenario for the player movement.

the development side, we rely on Unity's built-in version control, which allows seamless integration. This system lets multiple team members work on the project simultaneously without significant issues, helping us maintain smooth progress. Unity's version control also makes it easier to manage different versions, minimizing conflicts and facilitating easier collaboration. Our approach to version control ensures that documentation and development tasks are aligned and handled efficiently, with clear communication and minimized risks of errors. The combination of GitHub and Unity helps streamline our workflow while keeping everything well-coordinated.

2.6 Data Dictionary

The data dictionary defines and organizes key data structures for the Mega Man X replica game by establishing consistent variables for gameplay elements and interactions. Core elemnts of the game, such as the player (player, maxHealth, currentHealth), enemies (ballDeVouxHealth), and power-ups (smallCapsuleHealing), are structured as objects with specified data types. This approach standardizes variable naming conventions, making the code more readable and maintainable. For instance, using maxHealth and currentHealth as integer variables allows for straightforward health management and reduces potential errors. The dictionary also defines major UI elements like score and timer, ensuring these are displayed and updated accurately. By defining collision effects and enemy behaviors, the dictionary enables complex gameplay mechanics. Elements like bossHealth and smallCapsuleHealing

Use Case: PlayerShoot
ID:UC004
Actor: User
Preconditions: Logged into the game Has game in the play state
Primary scenario: 1. User presses the fire button 2. If the user hold the fire button for less than one second, a yellow pellet that deals low damage will be shot. 3. If the user holds the button for one second but less than three seconds, the pellet will be upgraded to a strong shot, which will deal moderate damage. 4.If the user hold the button for at least three seconds, the strong shot will then upgrade to a charged shot, which will deal a large amount of damage. 5. When the fire button is released, the bullet fired will depend on the length of time the fire button was held, which could be held forever. 6. After one of the three types of bullets was fired, it will either collide with an enemy or be destroyed after going too far away from the character(will be handled off screen).
Secondary scenarios: InvalidFireButton BulletDestroyedByObstacle

Figure 12: This Image represents the use case Scenario for player shooting scenario.

simplify interactions by defining health impacts. This structure helps organize variables essential for the game's core mechanics, player experience, and debugging, ultimately assist development and enables organized, consistent updates across the codebase.

2.7 User Experience

2.7.1 Gameplay Diagram

This flowchart outlines the gameplay flow for a Mega Man X replica level. The player begins by advancing through the level, encountering enemies they must defeat to proceed. Failure to defeat an enemy results in a game over. After each enemy encounter, the player may have the opportunity to collect a health capsule, which restores health. If no capsule is available, they continue toward the boss with their current health. Upon reaching the boss, the player must defeat it to win the level. If the boss is not defeated, the game ends. This design captures the linear, action-focused gameplay of Mega Man X, with health management and boss battles central to progression.

Use Case: Store Scores
ID:UC005
Actor: Database
Preconditions: Player has completed level
Primary scenario: 1. After fully completing the level, the game will take the score from destroying enemies in the level. 2. The game will then convert the time taken to beat the level and multiply it by 10 to turn it into a score to subtract from the score from enemies destroyed to give the final score. 3. The final score will be added to the SQLite3 database, along with the time taken to complete the level and the username of the person who made the score and time. 4. The score and time will be displayed for the user to see, with options to retry or to return to the main menu.
Secondary scenarios: InvalidUsername InvalidScoreFormat InvalidTimeFormat

Figure 13: This Image represents the use case Scenario for the store scoring scenario.

2.7.2 Gameplay Objectives

Our game brings back the nostalgic and challenging essence of the original Mega Man X, offering players an exciting action-platformer experience. With fast-paced gameplay, players must navigate through dynamic levels, face unique enemies, and overcome various obstacles. The goal of the game is to defeat all enemies in each level while managing player health and resources, leading to a final boss battle, this boss is vile which is a very important character throught the mega man franchise. We also included a unique scoring system in which we factor in the enemies defeated, the damage taken, and the amount of time it takes you to complete to make a competitive and challenging experience driving users to try to out do others scores.

2.7.3 User Skillset

Our game offers a nostalgic experience that doesn't require an intense skillset but still provides a rewarding challenge. While it's not the most competitive or difficult game, players will need basic gaming skills and a strategic mindset to succeed. They must plan how to approach each level, tactfully defeat enemies, and keep their character safe. Quick decision-making is key, as time plays an important role in progressing through the level to maximize user score. This remake blends classic elements with modern gameplay, making it enjoyable for both seasoned players and those revisiting the genre for a fun, retro-inspired experience.

Use Case: Take Damage
ID:UC003
Actor: User
Preconditions: Logged into the game Has game in the play state
Primary scenario: 1. User runs into enemy attack or into the collision box of an enemy. 2. Game will check for the amount of damage dealt by the enemy. 3. Game will then take away that amount of damage from the player's health. 4. Game will update visually to show the amount of health lost 5. If player has no health left, game will play an animation and sound effect showing the player has lost all their health.
Secondary scenarios:

Figure 14: This Image represents the use case Scenario for the take damage scenario.

2.7.4 Gameplay mechanics

Our game follows a time-based action-platformer style. Players control a single character and must navigate levels, defeat enemies, and avoid obstacles in real-time. There are no turn-based mechanics or random chances; success depends on the player's skill and strategy. The goal is to eliminate enemies while managing resources like health and ammo. Quick decision-making is critical as players are racing against time to obtain the highest score possible. There are no team-based elements; the game focuses on individual performance. Victory comes through efficient elimination of enemies and careful navigation of each level's challenges within the time limit.

2.7.5 Gameplay Items

Our team's Mega Man X project will feature select original in-game items, focusing solely on the first level. While we won't include the power-ups that appear after each boss, we will incorporate energy capsules, which are vital for the healing mechanic. These capsules will come in two varieties: regular and large. Regular energy capsules will restore a smaller amount of health, while large energy capsules will provide a more significant healing boost. This design choice aims to maintain the essence of the original game while offering players a balanced approach to health management in this initial level. We're excited to bring this classic gameplay element to life.

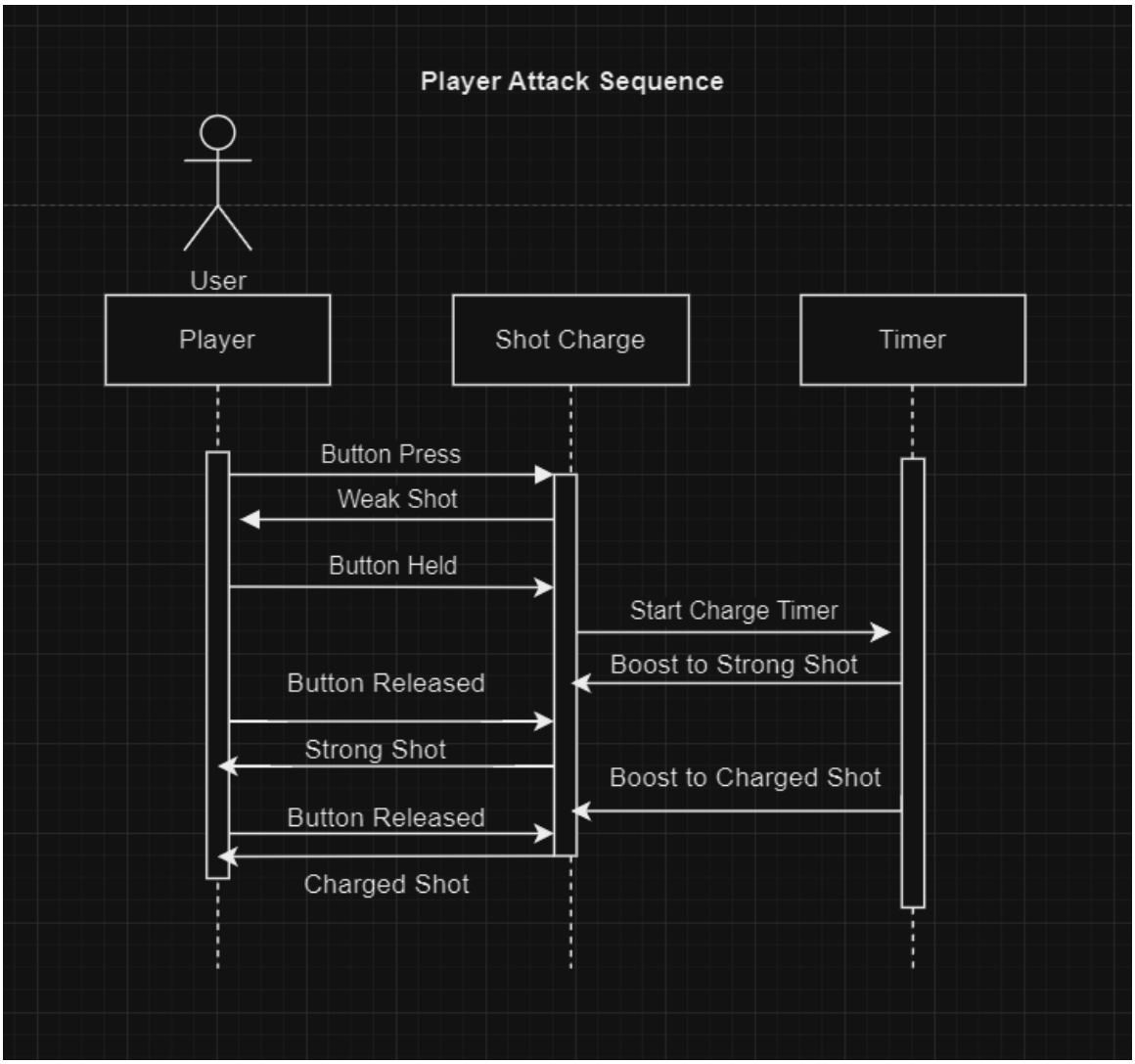


Figure 15: Sequence diagram of player attack sequence

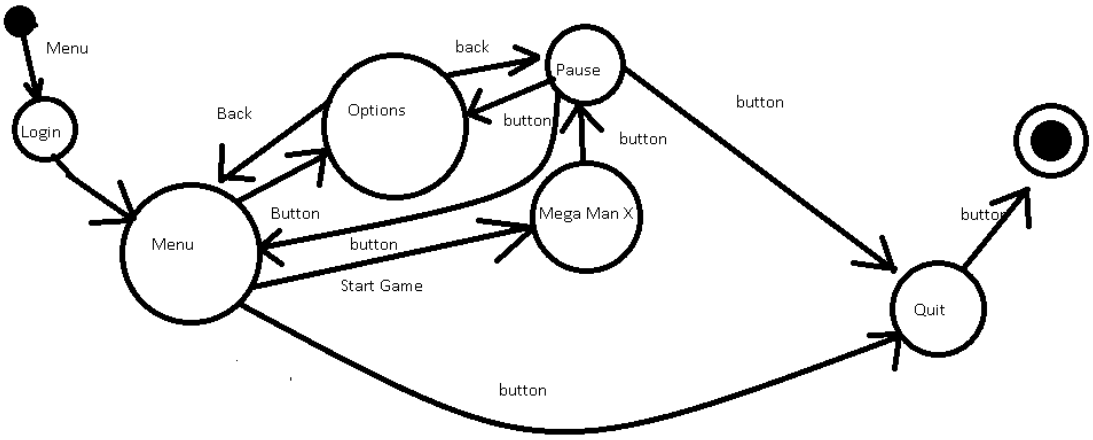


Figure 16: State diagram representing the user interface states.

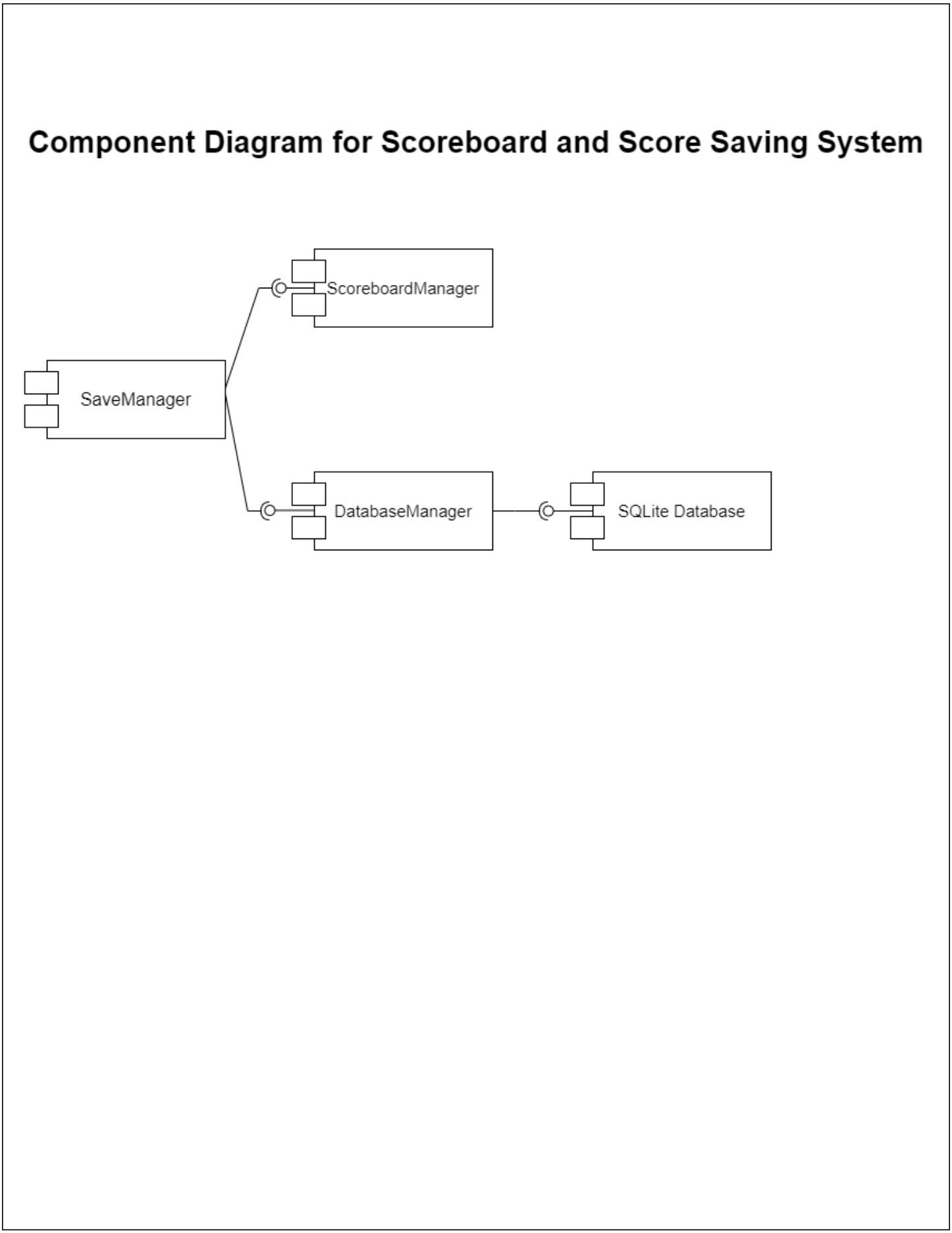


Figure 17: Component diagram that shows the connection between the scripts and database in the game.

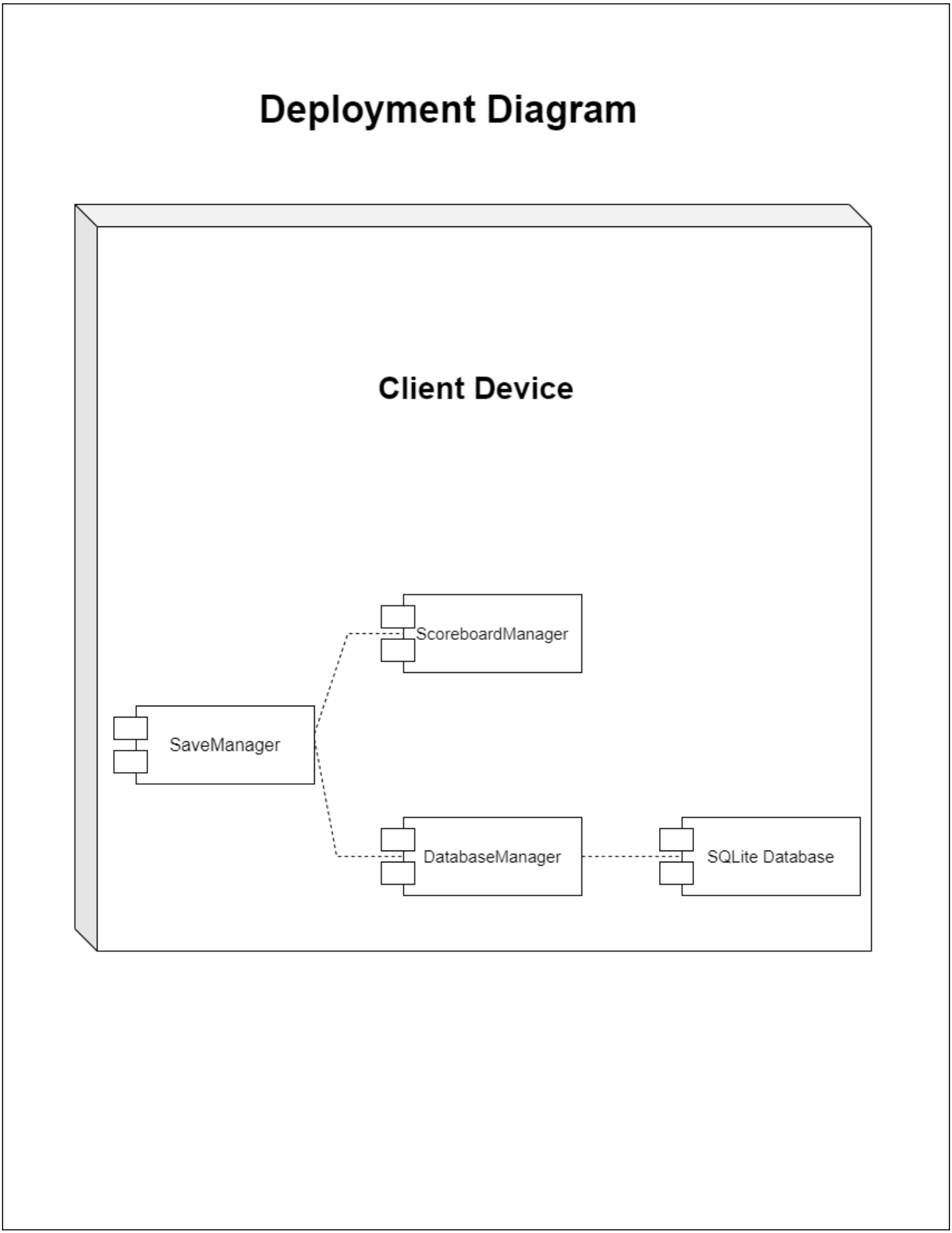


Figure 18: Connection between the game and the local database all on the client's system.

	UC001	UC002	UC003	UC004	UC005	UC006
Function Req. 1		X				
Function Req. 2	X					
Function Req. 3					X	
Function Req. 4	X					
Function Req. 5						X
Function Req. 6				X		
Function Req. 7			X			

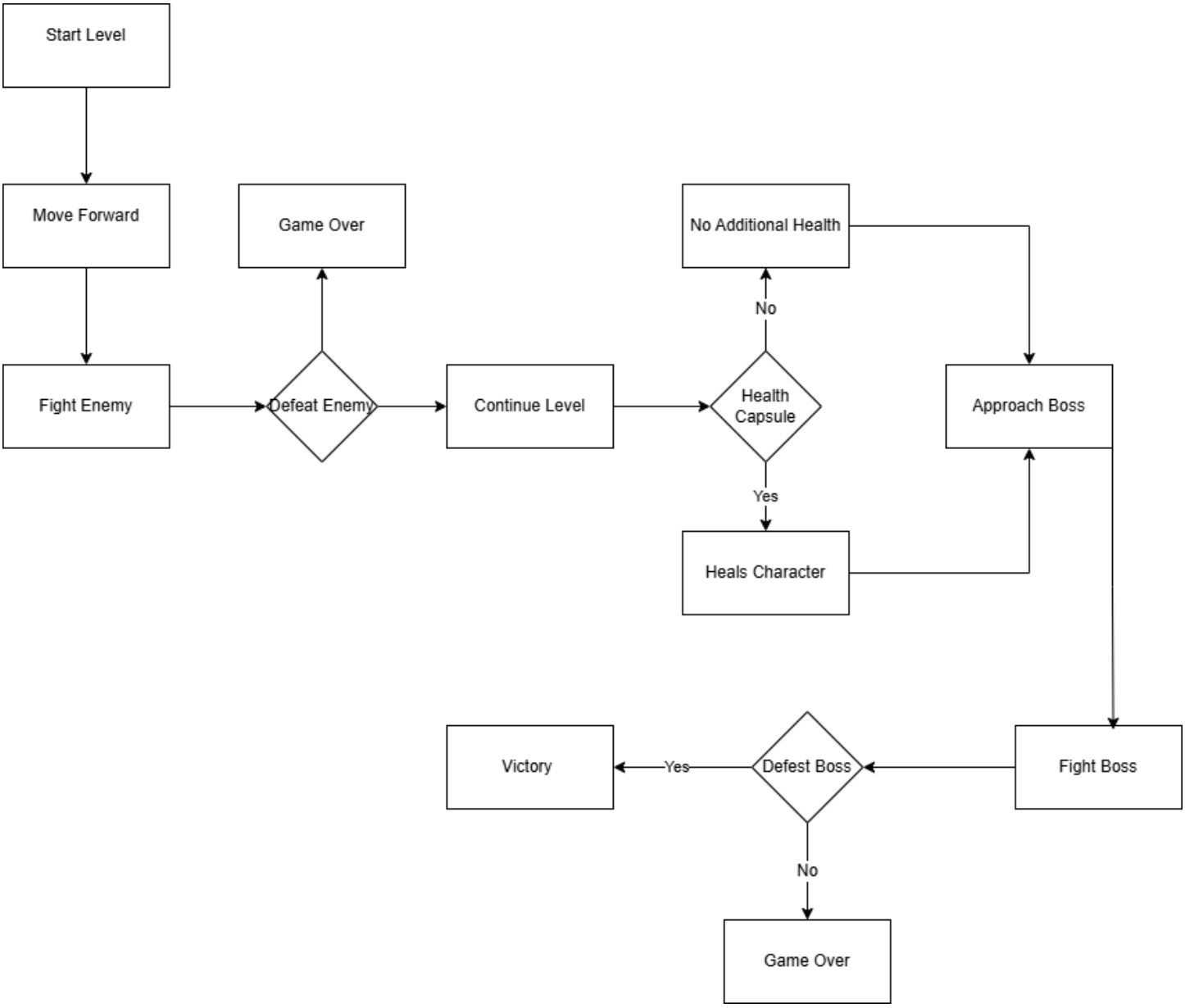


Figure 19: Gameplay Diagram.

2.7.6 Gameplay Challenges

As players progress through the game, they encounter increasingly difficult enemies, complex level designs, and challenging obstacles that test their skills. To handle these difficulties, players must rely on quick reflexes, strategic use of their available weapons, and precise platforming. Essential tools, such as health-restoring energy capsules and weapon power-ups, are scattered throughout the level to aid players in overcoming tough enemies and bosses. Additionally, players may find hidden areas or bonus items that provide extra advantages, encouraging

exploration. The game ensures that players have access to these essential tools to maintain balance, keeping the gameplay challenging but fair.

2.7.7 Gameplay Menu Screens

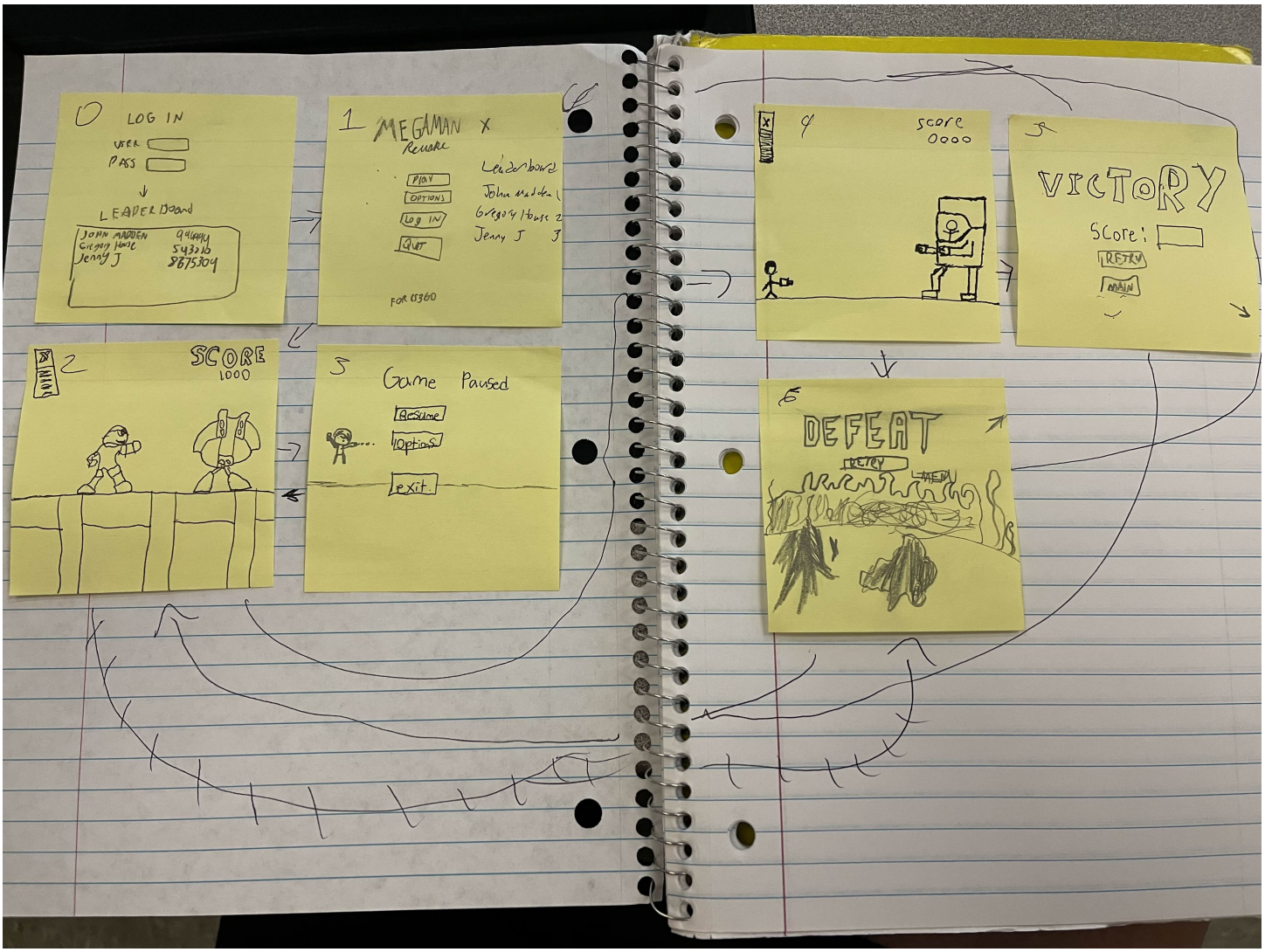


Figure 20: Menu Screens Wireframe

The wireframe for the "Mega Man X" project outlines several key game screens. It starts with a Login Screen where users input their username and password, and view the leaderboard featuring top players and scores. The Main Menu offers options to Play, view the Leaderboard, access Options, Log In, or Quit the game. During gameplay, the Gameplay Screen displays the player battling an enemy, with a score counter at the top. Players can pause the game using the Pause Menu, which has options to Resume, view Options, or Exit. After completing the level, a Victory Screen shows the player's score with options to Retry or return to the Main Menu. If defeated, a Defeat Screen displays Retry and Main Menu options.

2.7.8 Gameplay Heads-Up Display

Our team's implementation of Mega Man X includes a simple yet effective HUD that highlights key game features. One of the most critical elements is the health bar, as it determines whether the player can continue progressing. The HUD also displays a timer, reminding players they have only 10 minutes to complete the level. Additionally, the HUD emphasizes player score, introducing a competitive element by tracking and showcasing each player's progress. Our unique approach to scoring allows players to compete for leaderboard rankings, fostering

an engaging challenge to reach the top spot. This HUD design keeps gameplay informative, competitive, and immersive.

2.7.9 Gameplay Art Style

Our project, a replica of the SNES Mega Man X game, aims to closely match the original visual style by using similar two-dimensional sprites. As a 2D platformer, the game will feature pixelated, retro-style characters and objects, staying true to the nostalgic aesthetic of the SNES era. The characters, enemies, and in-game elements will maintain the cartoonish, vibrant appearance seen in the original, with bold colors and exaggerated designs. This approach helps preserve the classic, iconic look of Mega Man X while delivering an authentic experience to players familiar with the original game.

2.7.10 Gameplay Audio

Our project features MP4 files that handle all in-game character sounds and the audio for actions performed. We also included original music for the game, specifically a remix of the iconic Mega Man X soundtrack. This remix was created by one of our team members, Chessor, who collaborated with a talented guitarist to produce a fresh, exciting version of the music. The new track brings a modern twist to the classic tunes while staying true to the original spirit of the game. This custom audio adds a unique feel to the gameplay and enhances the overall experience for players.

3 Non-Functional Product Details

3.1 Product Security

3.1.1 Approach to Security in all Process Steps

The approach that our team took for security involved revising the technical document to embed thorough security measures across Requirements, Modeling and Design, and Implementation stages. In Requirements, we implemented input validation to ensure only safe, correctly formatted inputs, reducing injection attack risks and safeguarding data integrity. We also established secure session management practices with unique session tokens to guard against unauthorized access. In Modeling and Design, we reinforced access control with role-based data access, reducing unauthorized data exposure, and enhanced error handling and logging by ensuring incidents are logged securely without disclosing system details, while structured error messages remain brief and non-revealing. For data protection, we applied encryption and secure database access protocols, safeguarding sensitive information in storage and transit. In the Implementation phase, we fortified database security with least privilege access and prepared statements to prevent SQL injection. General coding practices focused on consistent secure code quality, adding another layer of defense. For authentication, we ensured secure password management with encrypted storage and regular updates to protect user credentials. This layered approach created a robust security framework, protecting the application comprehensively at every stage of development.

3.1.2 Security Threat Model

The Security Threat Model for the Mega Man X replica project identifies key trust boundaries and associated security risks to help ensure safe interactions within the system. The User/Game Boundary poses a risk of unauthorized access, where users may attempt to exploit in-game data or modify game files. To mitigate this, the system requires authentication and authorization, with salted and hashed passwords protecting user credentials. At the Web Server/Database Boundary, there is a risk of SQL Injection attacks from unvalidated input in requests from both users and admins. This can be mitigated by using parameterized SQL queries and input validation, along with firewalls and activity monitoring to prevent malicious SQL commands from compromising the database. For Admin Access, there is a risk of privilege escalation attacks, where unauthorized users may try to gain admin privileges. Role-Based Access Control (RBAC) and multi-factor authentication (MFA) are implemented to secure admin accounts and sensitive database functions. Additionally, to protect the Data Flow between the game and the database, encryption protocols like HTTPS are used to secure SQL calls, preventing data interception or

tampering. These targeted security practices enhance resilience against common attack vectors, ensuring safe interactions for both users and admins within the system.

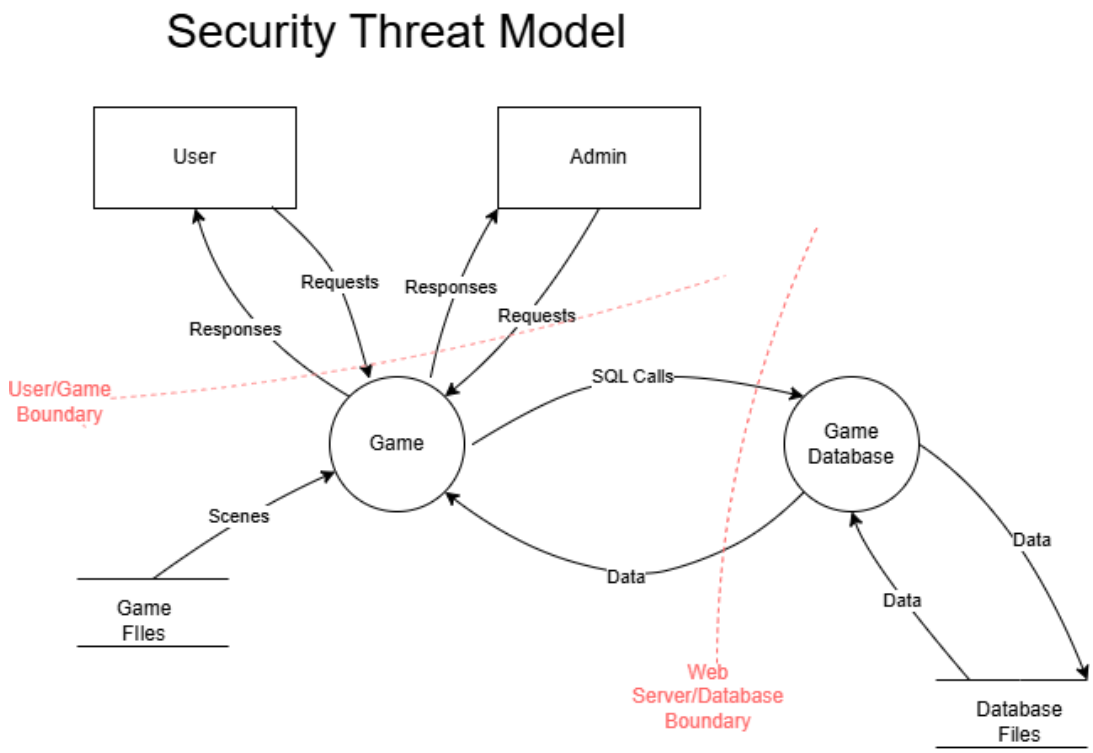


Figure 21: Security Threat Model

3.1.3 Security Levels

In our Mega Man X replica project, we plan to implement role-based access control (RBAC) to define distinct security levels for general users and administrators. General users will have access to gameplay features, allowing them to engage fully with the game while keeping sensitive functions off-limits. In contrast, administrators will have elevated permissions, granting access to critical functions and screens protected by an admin password. Only users authenticated as administrators will be able to perform sensitive operations, ensuring data integrity and security. To further enhance account security, our database will salt and hash passwords, making it significantly more challenging for accounts to be compromised. Additionally, we will incorporate parameterization in our SQL queries to prevent SQL injection attacks, a common vulnerability. To further strengthen account security, we will salt and hash all passwords in our database, making it considerably more difficult for attackers to compromise user accounts. This layered security approach will help safeguard user data while maintaining a seamless gaming experience.

3.2 Product Performance

3.2.1 Product Performance Requirements

The Mega Man X replica project's performance requirements are optimized to ensure smooth and responsive gameplay on a system with Windows 11 Home, running on an Intel Core i5-10500H CPU at 2.50 GHz with 6 cores. To align with these specifications, the game is designed to maximize resource usage within system limits, focusing specifically on CPU efficiency, memory management, and file I/O operations. Given the 6-core, 2.50 GHz CPU, the game sustains smooth processing by reducing resource contention, ensuring actions are handled swiftly. Additionally, file I/O operations, such as loading level data and assets, are optimized to minimize disk access time, preventing lag. Testing has validated these configurations, demonstrating they operate within the system's performance thresholds for optimal gameplay.

3.2.2 Measurable Performance Objectives

The performance objectives for the Mega Man X replica game emphasize efficient resource management and high responsiveness in gameplay. Specifically, CPU utilization should remain below 80 percent during peak gameplay to ensure that the CPU has adequate resources for core game functions, minimizing lag in player interactions. The file I/O throughput targets a minimum of 95 MiB/s for reads and 60 MiB/s for writes, which will help maintain high throughput for loading assets and reduce wait times across levels. Additionally, the average file I/O latency should be kept to 5 milliseconds per operation, supporting seamless in-game resource access. Database query response times are also critical, with a target of less than 50 milliseconds per query, ensuring that vital data, such as player stats and leaderboards, is retrieved quickly for a smooth gameplay experience. To enhance visual performance, the game should maintain a consistent frame rate of at least 30 FPS during active gameplay, verifying that fluid motion and responsiveness are preserved during player actions. Lastly, level load times should not exceed 2 seconds to ensure quick transitions between levels, keeping players engaged without extended waiting periods. Each of these objectives is designed to balance resource usage with responsiveness, ensuring the system can handle high demands without compromising gameplay quality.

3.2.3 Application Workload

Our team took a straightforward approach to collect this data. We developed a C sharp script that leverages the Unity SceneManager to monitor the active scene and records the time spent in each scene in seconds. This script is attached to an empty object on the login screen with a "Do Not Destroy" setting, ensuring it persists across scenes and tracks time accurately. Additional programming then calculates these times as percentages. For simplicity, I calculated individual time allocations using a calculator to determine the proportion of time spent in each game scene. This approach allows us to visualize where users spend their time, enabling data-driven decisions to improve the application's usability and design. We have an insufficient amount of people to test and get a real number on this value, but we had a small group of people play and the numbers we collected are show in this data visualization. This visualization shows that for the average of these 5 players there the time was allocated to being in the LoginScene:7.75%, MainMenu:7.75%, MainScene: 81.75%, Victory: 4.19%, Defeat: 2.13%.

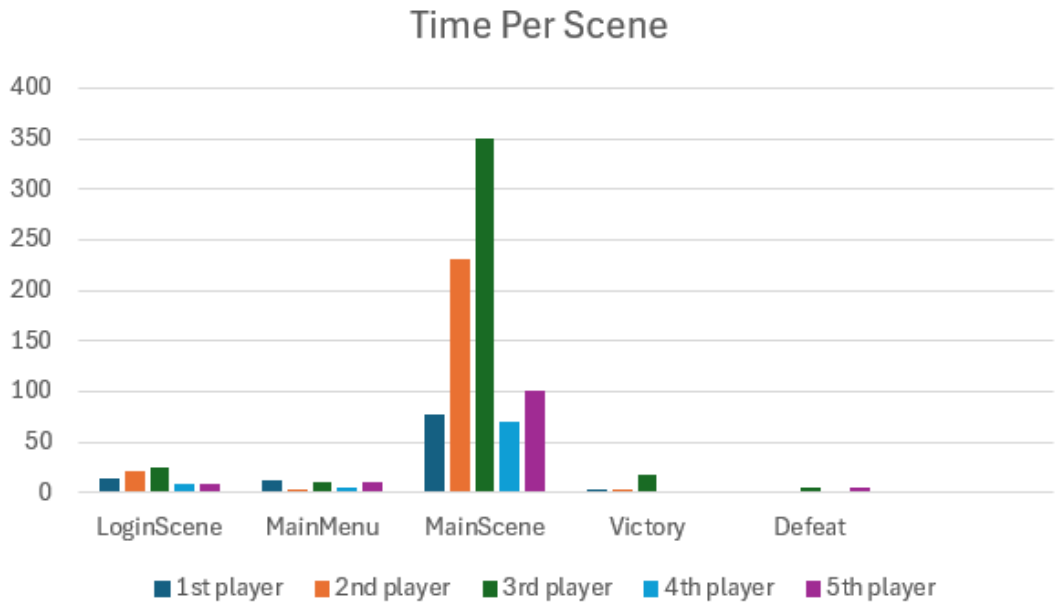


Figure 22: Application Workload Testing

3.2.4 Hardware and Software Bottlenecks

Based on the Sysbench CPU test results, several hardware and software bottlenecks can be identified. First, the test indicates CPU saturation, as evidenced by the diminishing returns in performance with increasing thread

counts. For instance, while the performance reached approximately 3,693,594.02 events per second at 16 threads, it only marginally increased to 3,703,025.26 events per second when moving to 64 threads. This lack of significant improvement suggests that the CPU may be reaching its operational capacity, unable to effectively utilize additional threads, leading to wasted processing potential.On the software side, thread management and scheduling may also contribute to the observed bottlenecks. The increasing standard deviation in execution times, particularly at higher thread counts, suggests that the software's handling of threads may not optimally utilize available CPU cores, potentially leading to some threads being starved of processing time while others are over-scheduled.

3.2.5 Synthetic Performance Benchmarks

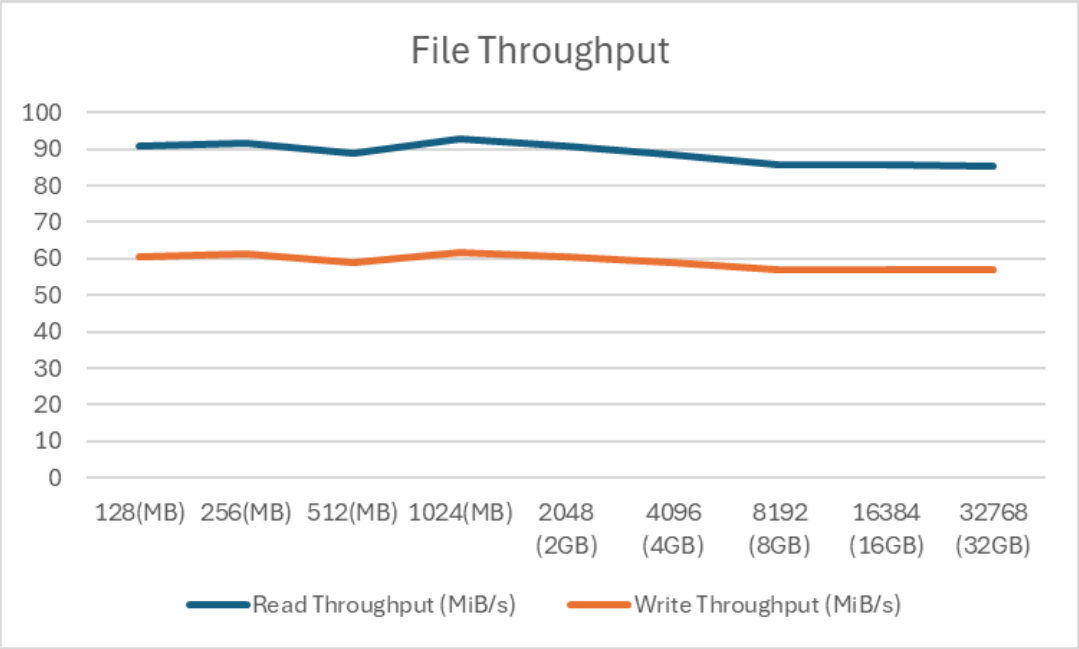


Figure 23: File I/O Throughput

The file I/O benchmark results support the statement regarding the Mega Man X replica project's optimization for smooth gameplay. The 6-core Intel Core i5-10500H CPU effectively manages processing needs, particularly at lower thread counts, confirming reduced resource contention. Read and write throughput remains acceptable (around 90 MiB/s and 60 MiB/s, respectively) for files up to 1 GB, validating optimized file I/O operations. While throughput slightly declines for larger files, it does not significantly hinder gameplay. Overall, testing demonstrates that the game's configurations operate well within the system's performance limits, ensuring responsiveness and low latency during typical load conditions. Testing the Mega Man X replica project's performance across different thread counts revealed optimal gameplay at 4–8 threads. Initial scaling (1, 2, 4 threads) shows improved resource utilization, with diminishing returns appearing beyond 8 threads due to the system's 6-core CPU. Higher thread counts (16–64) add context-switching overhead, leading to a performance plateau. Contributing factors include CPU core limitations, cache contention, and potential disk I/O bottlenecks, especially when parallel threads access files. To maintain responsiveness and efficiency, focusing on memory management and minimizing I/O contention can help optimize gameplay within the system's capabilities. The execution times for the select statements (ranging from 0.07 to 0.12 seconds) demonstrate that the database operations are efficient and align with the Mega Man X replica project's performance requirements. The consistency in execution times indicates that the database is optimized for quick data retrieval, crucial for maintaining smooth gameplay. All times fall within acceptable limits for real-time applications, ensuring that database operations will not introduce significant latency. Additionally, the similar performance across various queries suggests scalability, confirming that the database can efficiently handle increased complexity and data volume as the game evolves.

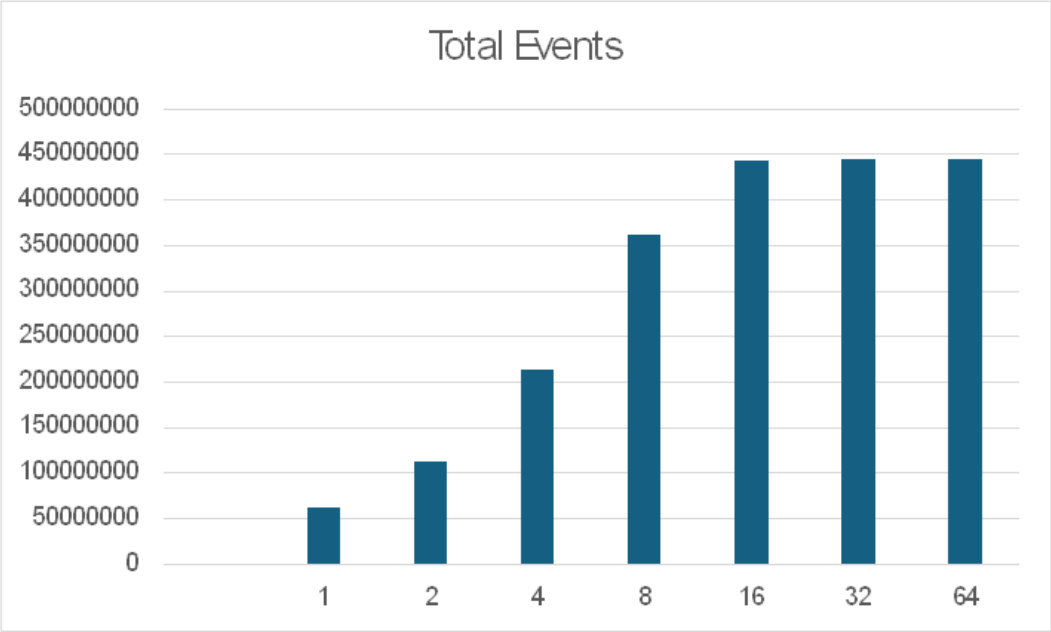


Figure 24: CPU Total Events

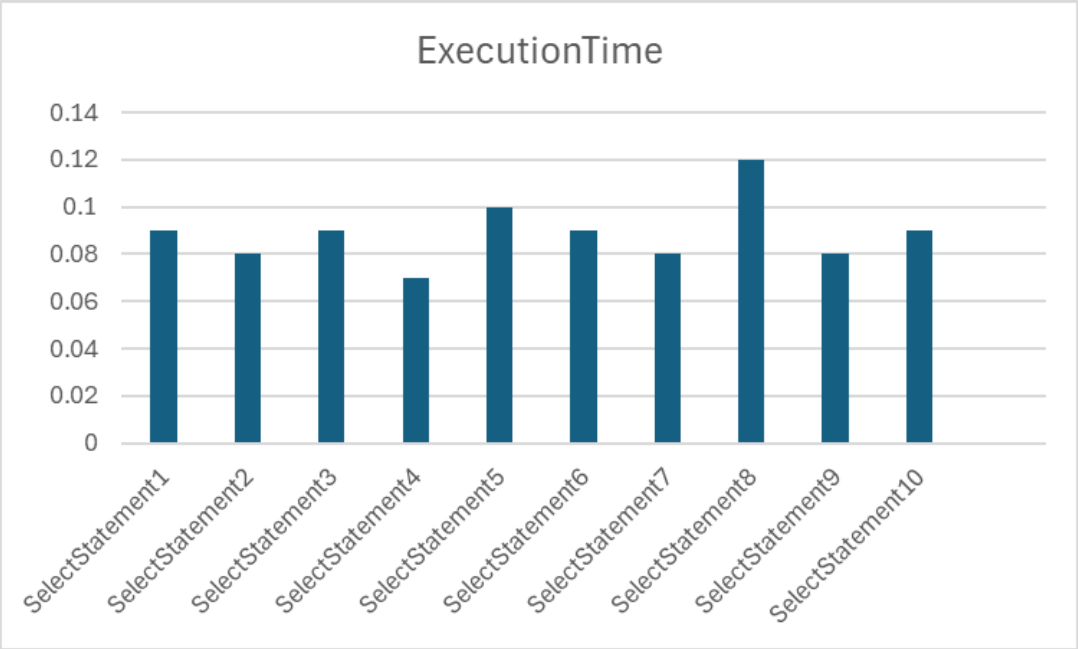


Figure 25: Execution Time

3.2.6 Performance Tests

Using the Unity Profiler that was built into the Unity Game Engine, we ran tests for the performance of the game scene, which would be the most resource intensive of the scenes. Looking at Figure 28 for the CPU Usage line graph, you can see that the game was consistently at or above 60 FPS and most of our CPU was used by scripts and the "Others" category, which consists of a lot of Asynconous loops and coroutines that we're either made by us or provided by the Unity engine itself. You can see that our game uses 1.87 GB of storage, with a lot of it being in use while inside of the game scene itself. This was to be expected, since most of our time spent working on the game is inside of the game scene, with the other scenes there to either let the user know they won or lost or to navigate the game. Our performance looks good so far, however there are some spikes in performance of the CPU that we will likely look into what caused them, and appear to have something to do with collisions with enemy objects either from the player or the player bullets. More time will be needed to fix the issue, but for

now the game is in a good spot in terms of performance.

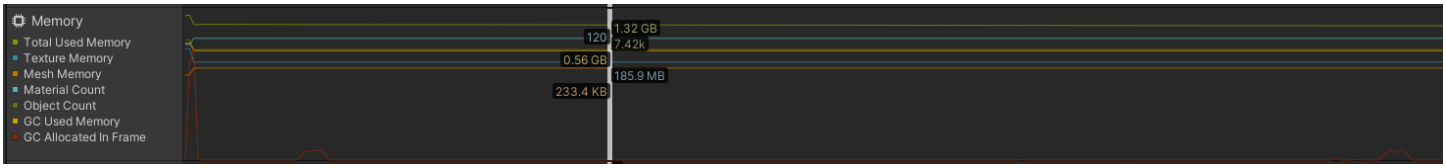


Figure 26: Line Graph Visual of Memory Usage in Game Scene

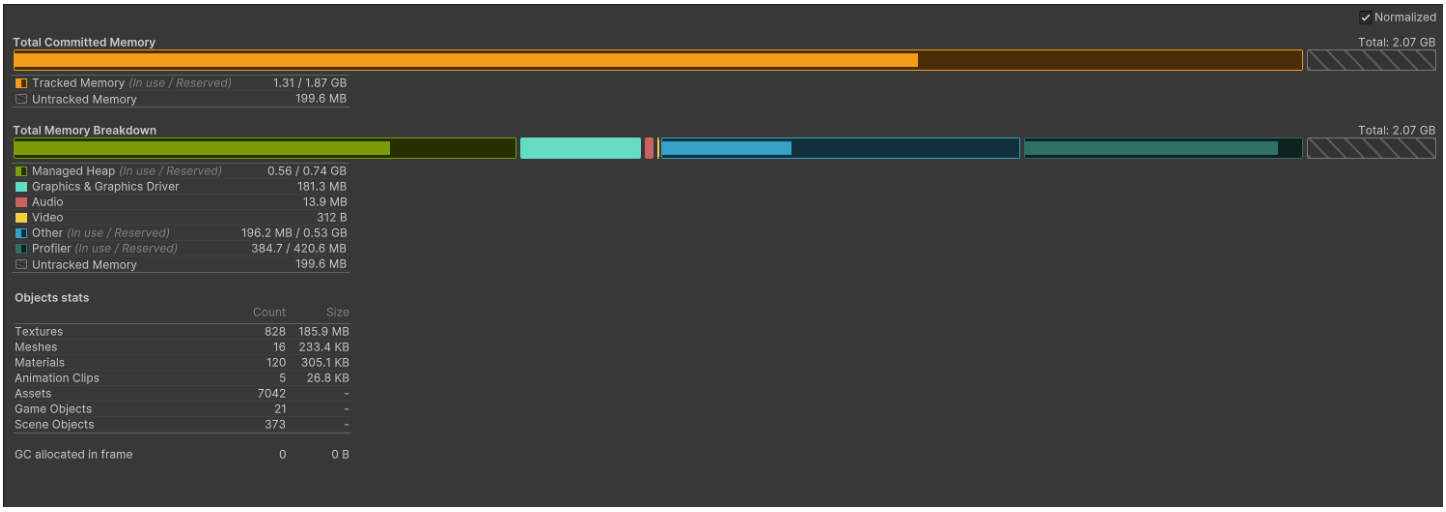


Figure 27: Detailed Memory Usage in Game Scene

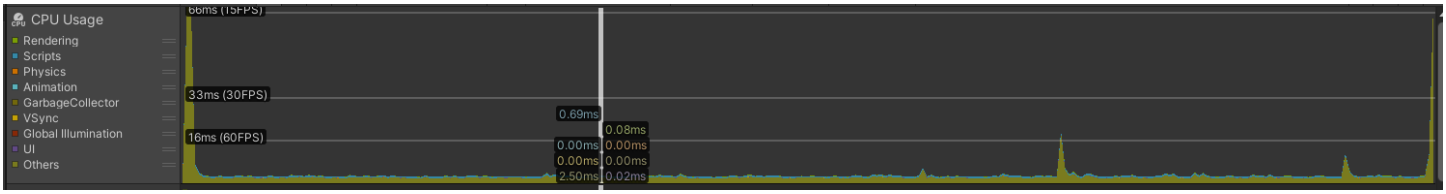


Figure 28: CPU Usage in Game Scene

4 Software Testing

4.1 Software Testing Plan Template

Test Plan Identifier: UnitTest(001), IntegrationTest(002), System(003), AcceptanceTest(004) **Introduction:** Unit Test: We conducted two seperate unit tests within our project. The first unit test is the time and score controller the other was a unit test for the database controller. These tests where to ensure that each of the functions within these classes or units work properly. IntegrationTest: our integration testing served the purpose of ensuring that the database controller and the time and score controller worked in tandem. System Testing: We conducted two diffrent system tests on our project one of which is documentation testing and boot testing. AcceptanceTesting: acceptance testing is utilized to ensure that the project meets the expectations of the client.

Test item: UnitTest:The TimeAndScoreController class manages game timer and score functionalities in the Mega Man X project. It ensures accurate tracking and display of the remaining game time and the player's score. It interacts with Unity's UI components (TextMeshProUGUI) and relies on mock objects for isolated testing. The class operates within the gameplay scene and supports core player interactions. The DatabaseController class manages database interaction ensuring that things our pushed and pulled from the database correctly. IntegrationTest: The TimeAndScoreController and DatabaseController are the primary components under test.

The TimeAndScoreController handles in-game score and time updates, while the DatabaseController manages data storage and retrieval. The integration test validates the synchronization between the two components, ensuring that game states (time and score) are accurately stored in and retrieved from the database, maintaining consistency across sessions. System Testing: With this testing we are generally testing the project as a whole. Acceptance Testing: This testing is done manually by the client after the team creates a template.

Features to test/not to test: UnitTest: Some features that our unit tests our handling is whether or not the score updates properly as the player destroys enemies. As for the Database Controller it ensures that scores our being properly pushed to the database. Integration Test: This test tests whether the database management controller and the time and score controller work properly with one another. System Testing: In this tests we are ensuring that they can build properly and that it follows the documentation. During this test our team created a template of the requirements that where designated to the project where the client manually checked.

Approach: UnitTesting: We performed unit testing by using unit's built in testing framework. We were able to use Nunit which allowed us to develop test plan cases. We construct to build mock tests and is ran in playmode which means when it is ran the test occurs. IntegrationTesting: We utilized uniy's built in testing framework in order to run the test cases. This testing is automated so that the user does not have to manually test. System Testing: During this test we are manually comparing the documentation to the code, but we are also making sure that the product builds. AcceptanceTest: This testing is manual and must be performed by the client, the client must go line by line.

Test deliverables: The deliverables for all of our tests is whether the test is able to pass or fail criteria. This is important in each of the tests as this is the overall goal of all tests.

Item pass/fail criteria: UnitTest: For the unit tests we are checking whether the score updates, whether the time is correct, user creation, login functionality, scores save, checks if user is admin. Integration Test: The integration tests make sure that the database controller and time and score controller work together. Add time bonus and save test as well as the save score integration test. System Test: The pass fail criteria is that it builds properly and that the code adheres to documentation. AcceptanceTest: In this testing the pass or fail criteria is based on whether the different functional and non functional requirements are met.

Environmental needs: UnitTest and IntegrationTest: The infrastructure that is needed in order to execute these test cases is to ensure that unity is installed and that you have the testing package. From their you need to make sure their implemented correctly and run. System Tests: The only infastrucure needed in order to test is a complete project and documentation. Acceptance Testing: The only infastructure needed is a complete project and the acceptance test template in order to complete testing.

Responsibilities: UnitTest: Unit testing is responsible to designated certain units of the code and to determine whether that unit works as it is intended. This is done using mock setups and assertions. IntegrationTest: This is responsible to ensure that these two files work properly with one another ensuring it all works properly. SystemTest: the responsibility of this test is to ensure that the project as a whole performs as intended. AcceptanceTesting: The main responsibility of acceptance testing is to ensure that the project meets the given requirements.

Staffing and training needs: Team members had to watch youtube videos along with using internet resources in order to gather a strong understanding to implement each of the test cases into the project.

Schedule: Implementing each of the test cases took quite a large amount of time. This was mainly due to trying to understand how creating test cases inside of unity actually works as well as developing test cases that properly test.

Risks and Mitigation: The main risks that our team ran into when it came to implementing testing was a lack of technical knowledge on how testing can be conducted in unity. We were able to make a risk mitigation plan in order to ensure that this issue did not affect us super heavily by allotting our members designated amount of time in which they can research and come to understand how this testing can work. For example watching youtube tutorials on how to set up a unit test inside of unity.

Approvals: Ryan Rutledge, Kaden Groves, A.R. Chessor, Eli Atkisson 11/28/2024

4.2 Unit Testing

DatabaseControllerTests.cs

```
using System.Collections;
```

```
using System.Collections.Generic;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;
using UnityEngine.SceneManagement;
using TMPro;
using System.IO;
using System.Threading;
using UnityEditor;

[TestFixture]
public class DatabaseControllerTests
{
    private DatabaseController databaseController;
    private string testDatabasePath;

    [SetUp]
    public void SetUp()
    {
        //Generate a unique database file for each test
        string uniqueFilename = $"mytestdatabase_{System.Guid.NewGuid()}.s3db";
        testDatabasePath = Path.Combine(Application.dataPath, "Database", uniqueFilename);

        //Ensure the folder exists
        Directory.CreateDirectory(Path.GetDirectoryName(testDatabasePath));

        //Initialize the DatabaseController with the unique path
        var gameControllerObject = new GameObject();
        databaseController = gameControllerObject.AddComponent<DatabaseController>();
        databaseController.SetDatabasePath(testDatabasePath);
        databaseController.DatabaseTestsSetup();

        //Create test users and setup data
        databaseController.CreateUser("LoginTest", "Password1234");
        databaseController.CreateAdminUser("Admin", "Password1234");
    }

    [TearDown]
    public void TearDown()
    {
        databaseController.CloseDatabaseConnection();

        System.GC.Collect();
        System.GC.WaitForPendingFinalizers();

        bool isDeleted = false;
        int attempts = 5;
        for (int i = 0; i < attempts; i++)
        {
            try
            {
                if (File.Exists(testDatabasePath))
                {
                    FileUtil.DeleteFileOrDirectory(testDatabasePath);
                    Debug.Log($"Successfully deleted test database: {testDatabasePath}");
                    isDeleted = true;
                    break;
                }
            }
            catch (IOException ex)
            {
                Debug.LogWarning($"Attempt {i + 1} to delete database failed: {ex.Message}");
                Thread.Sleep(2000);
            }
        }

        if (!isDeleted)
        {
            Debug.LogError($"Failed to delete test database after {attempts} attempts.");
        }
    }


    [Test]
    public void UserCreationTest(){
        //Should create new user
        Assert.AreEqual(true, databaseController.CreateUser("TestUser", "Test1234"));
        //Should not create new user due to a user called TestUser already existing inside SQLite database
        Assert.AreEqual(false, databaseController.CreateUser("TestUser", "Test1234"));
    }

    [Test]
    public void UserLoginTest(){
        //Correct login information, should be a successful login
        Assert.AreEqual(true, databaseController.LoginUser("LoginTest", "Password1234"));
        //Username does not exist in the database, login should fail
        Assert.AreEqual(false, databaseController.LoginUser("LoginTestFail", "Password1234"));
        //Username does exist in the database, but password is incorrect, login should fail
        Assert.AreEqual(false, databaseController.LoginUser("LoginTest", "WrongPassword"));
    }

    [Test]
    public void UserAdminCheckTest(){
        //Is admin, should return true
        databaseController.LoginUser("Admin", "Password1234");
        Assert.AreEqual(true, databaseController.IsAdmin());
        //Is not admin, should return false
        databaseController.LoginUser("LoginTest", "Password1234");
        Assert.AreEqual(false, databaseController.IsAdmin());
    }

    [Test]
```

```
    public void SaveScoreTest(){
        databaseController.LoginUser("LoginTest", "Password1234");
        databaseController.SaveScore(6780);

        Assert.AreEqual(true, databaseController.CheckForScoreInDatabase(6780));
    }
}
```

TimeAndScoreControllerTests.cs

```
using System.Collections;
using System.Collections.Generic;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;
using UnityEngine.SceneManagement;
using TMPro;
public class TimeAndScoreControllerTests
{
    private TimeAndScoreController timeAndScoreController;
    private TextMeshProUGUI mockTimerText;
    private TextMeshProUGUI mockScoreText;


    [SetUp]
    public void SetUp()
    {
        // Set up the GameObject and attach the script
        var gameControllerObject = new GameObject();
        timeAndScoreController = gameControllerObject.AddComponent<TimeAndScoreController>();


        // Set up mock UI components
        var timerTextObject = new GameObject();
        mockTimerText = timerTextObject.AddComponent<TextMeshProUGUI>();
        var scoreTextObject = new GameObject();
        mockScoreText = scoreTextObject.AddComponent<TextMeshProUGUI>();


        // Assign mock UI components to the controller
        timeAndScoreController.timerText = mockTimerText;
        timeAndScoreController.scoreText = mockScoreText;


        // Initialize values
        timeAndScoreController.timeRemaining = 600f; // 10 minutes
        timeAndScoreController.score = 0;
    }


    [Test]
    public void Score_UpdatesCorrectly()
    {
        // Test score updates independently
        timeAndScoreController.UpdateScore(100);
        Assert.AreEqual(100, timeAndScoreController.score, "Score should update correctly.");
        Assert.AreEqual("0100", mockScoreText.text, "Score text should display in 4-digit format.");
    }


    [Test]
    public void DisplayTime_CorrectlyFormatsTime()
    {
        // Simulate a specific time and manually call DisplayTime
        timeAndScoreController.timeRemaining = 125f; // 2 minutes and 5 seconds
        typeof(TimeAndScoreController).GetMethod("DisplayTime", System.Reflection.BindingFlags.NonPublic | System.Reflection.Bindin
            ?.Invoke(timeAndScoreController, new object[] { 125f });
        Assert.AreEqual("02:05", mockTimerText.text, "Timer should display correct time format.");
    }

}
```

4.2.1 Source Code Coverage Tests

Text goes here.

4.3 Software Testing Plan Template

Test Plan Identifier: UnitTest(001),

Introduction: Unit Test: We conducted two seperate unit tests within our project. The first unit test is the time and score controller the other was a unit test for the database controller. These tests where to ensure that each of the functions within these classes or units work properly.

Test item: UnitTest:The TimeAndScoreController class manages game timer and score functionalities in the Mega Man X project. It ensures accurate tracking and display of the remaining game time and the player's score. It interacts with Unity's UI components (TextMeshProUGUI) and relies on mock objects for isolated testing. The class operates within the gameplay scene and supports core player interactions. The DatabaseController class manages database interaction ensuring that things our pushed and pulled from the database correctly.

Features to test/not to test: UnitTest: Some features that our unit tests our handling is whether or not the score updates properly as the player destroys enemies. As for the Database Controller it ensures that scores our being properly pushed to the database.

Approach: UnitTesting: We performed unit testing by using unit's built in testing framework. We were able to use Nunit which allowed us to develop test plan cases. We construct to build mock tests and is ran in playmode which means when it is ran the test occurs.

Test deliverables: The deliverables for all of our tests is whether the test is able to pass or fail criteria. This is important in each of the tests as this is the overall goal of all tests.

Item pass/fail criteria: UnitTest: For the unit tests we are checking whether the score updates, whether the time is correct, user creation, login functionality, scores save, checks if user is admin. Integration Test:

Environmental needs: UnitTestt: The infrastructure that is needed in order to execute these test cases is to ensure that unity is installed and that you have the testing package. From their you need to make sure their implemented correctly and run.

Responsibilities: UnitTest: Unit testing is responsible to designated certain units of the code and to determine whether that unit works as it is intended. This is done using mock setups and assertions. **Staffing and training needs:** Team members had to watch youtube videos along with using internet resources in order to gather a strong understanding to implement each of the test cases into the project.

Schedule: Implementing each of the test cases took quite a large amount of time. This was mainly due to trying to understand how creating test cases inside of unity actually works as well as developing test cases that properly test.

Risks and Mitigation: The main risks that our team ran into when it came to implementing testing was a lack of technical knowledge on how testing can be conducted in unity. We were able to make a risk mitigation plan in order to ensure that this issue did not affect us super heavily by allotting our members designated amount of time in which they can research and come to understand how this testing can work. For example watching youtube tutorials on how to set up a unit test inside of unity.

Approvals: Ryan Rutledge, Kaden Groves, A.R. Chessor, Eli Atkisson 11/28/2024

4.3.1 Unit Tests and Results

The first unit test shown is used to test the database controller. This test includes multiple cases of user creation, user login, checking admin login, and score saving. The second unit test shown is used to test score and time updates. This test includes updating and formatting the score and manually updating and displaying the time.

4.4 Integration Testing

IntegrationTests.cs

```
using System.Collections;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;
using TMPro;
using System.IO;
using UnityEditor;
using System.Threading;

public class IntegrationTests
{
    private GameObject timerAndScoreObject;
    private TimeAndScoreController timeAndScoreController;

    private GameObject databaseObject;
    private DatabaseController databaseController;

    private string testDatabasePath;

    [SetUp]
    public void Setup()
    {
        // Setup test database
        string uniqueFilename = $"mytestdatabase_{System.Guid.NewGuid()}.s3db";
        testDatabasePath = Path.Combine(Application.dataPath, "Database", uniqueFilename);

        Directory.CreateDirectory(Path.GetDirectoryName(testDatabasePath));

        // Setup DatabaseController
        databaseObject = new GameObject("DatabaseController");
        databaseController = databaseObject.AddComponent<DatabaseController>();
```

```
        databaseController.SetDatabasePath(testDatabasePath);
        databaseController.DatabaseTestsSetup();
        databaseController.CreateUser("TestUser", "Password1234");

        // Setup TimeAndScoreController
        timerAndScoreObject = new GameObject("TimeAndScoreController");
        timeAndScoreController = timerAndScoreObject.AddComponent<TimeAndScoreController>();
        timeAndScoreController.scoreText = new GameObject("ScoreText").AddComponent<TextMeshProUGUI>();
        timeAndScoreController.timerText = new GameObject("TimerText").AddComponent<TextMeshProUGUI>();

        // Mock login
        databaseController.LoginUser("TestUser", "Password1234");
    }

    [Test]
    public void SaveScoreIntegrationTest()
    {
        // Update score and save to database
        timeAndScoreController.UpdateScore(150);
        databaseController.SaveScore(timeAndScoreController.score);

        // Verify that the score is saved
        Assert.IsTrue(databaseController.CheckForScoreInDatabase(150), "Score not saved in database.");
    }

    [Test]
    public void AddTimeBonusAndSaveTest()
    {
        // Simulate game timer and add time bonus to score
        timeAndScoreController.timeRemaining = 120; // 2 minutes
        timeAndScoreController.AddTimeToScore();
        databaseController.SaveScore(timeAndScoreController.score);

        // Verify that the bonus is added and saved
        int expectedScore = (int)(120 * 10); // Time bonus
        Assert.AreEqual(expectedScore, timeAndScoreController.score, "Score mismatch after adding time bonus.");
        Assert.IsTrue(databaseController.CheckForScoreInDatabase(expectedScore), "Time bonus score not saved.");
    }

    [TearDown]
    public void TearDown()
    {
        Object.Destroy(timerAndScoreObject);
        Object.Destroy(databaseObject);
        databaseController.CloseDatabaseConnection();

        System.GC.Collect();
        System.GC.WaitForPendingFinalizers();

        bool isDeleted = false;
        int attempts = 5;
        for (int i = 0; i < attempts; i++)
        {
            try
            {
                if (File.Exists(testDatabasePath))
                {
                    FileUtil.DeleteFileOrDirectory(testDatabasePath);
                    Debug.Log($"Successfully deleted test database: {testDatabasePath}");
                    isDeleted = true;
                    break;
                }
            }
            catch (IOException ex)
            {
                Debug.LogWarning($"Attempt {i + 1} to delete database failed: {ex.Message}");
                Thread.Sleep(2000);
            }
        }

        if (!isDeleted)
        {
            Debug.LogError($"Failed to delete test database after {attempts} attempts.");
        }
    }
}
```

4.4.1 Integration Tests and Results

The integration test is used to test user login, score saving, and time bonus consecutively. First, it creates a mock user. Next, the user's score is updated and saved to the database. Last, time bonus is calculated and incorporated into the saved score.

4.5 Software Testing Plan Template

Test Plan Identifier: IntegrationTest(002) **Introduction:** IntegrationTest: our integration testing served the purpose of ensuring that the database controller and the time and score controller worked in tandem.

Test item: IntegrationTest: The TimeAndScoreController and DatabaseController are the primary components under test. The TimeAndScoreController handles in-game score and time updates, while the DatabaseController manages data storage and retrieval. The integration test validates the synchronization between the two

components, ensuring that game states (time and score) are accurately stored in and retrieved from the database, maintaining consistency across sessions.

Features to test/not to test: Integration Test: This test tests whether the database management controller and the time and score controller work properly with one another.

Approach: IntegrationTesting: We utilized uniy's built in testing framework in order to run the test cases. This testing is automated so that the user does not have to manually test.

Test deliverables: The deliverables for all of our tests is whether the test is able to pass or fail criteria. This is important in each of the tests as this is the overall goal of all tests.

Item pass/fail criteria: Integration Test: The integration tests make sure that the database controller and time and score controller work together. Add time bonus and save test as well as the save score integration test.

Environmental needs: IntegrationTest: The infrastructure that is needed in order to execute these test cases is to ensure that unity is installed and that you have the testing package. From their you need to make sure their implemented correctly and run.

Responsibilities: IntegrationTest: This is responsible to ensure that these two files work properly with one another ensuring it all works properly.

Staffing and training needs: Team members had to watch youtube videos along with using internet resources in order to gather a strong understanding to implement each of the test cases into the project.

Schedule: Implementing each of the test cases took quite a large amount of time. This was mainly due to trying to understand how creating test cases inside of unity actually works as well as developing test cases that properly test.

Risks and Mitigation: The main risks that our team ran into when it came to implementing testing was a lack of technical knowledge on how testing can be conducted in unity. We were able to make a risk mitigation plan in order to ensure that this issue did not affect us super heavily by allotting our members designated amount of time in which they can research and come to understand how this testing can work. For example watching youtube tutorials on how to set up a unit test inside of unity.

Approvals: Ryan Rutledge, Kaden Groves, A.R. Chessor, Eli Atkisson 11/28/2024

4.6 System Testing

4.6.1 System Tests and Results

These System tests that we performed were done manually. We performed a team testing session in which we all went through each section of the documentation and the project itself and analyzed whether the two matched.

4.7 Software Testing Plan Template

Test Plan Identifier: SystemTest(003) **Introduction:** System Testing: We conducted two diffrent system tests on our project one of which is documentation testing and boot testing.

Test item: System Testing: With this testing we are generally testing the project as a whole via testing the project with the documentation in mind. Also testing whether the project is built correctly. . **Features to test/not to test:** System Testing: In this tests we are ensuring that they can build properly and that it follows the documentation. During this test our team created a template of the requirements that where designated to the project where the client manually checked.

Approach: System Testing: During this test we are manually comparing the documentation to the code, but we are also making sure that the product builds. **Test deliverables:** The deliverables for all of our tests is whether the test is able to pass or fail criteria. This is important in each of the tests as this is the overall goal of all tests.

Item pass/fail criteria: System Test: The pass fail criteria is that it builds properly and that the code adheres to documentation. **Environmental needs:** System Tests: The only infastrucure needed in order to test is a complete project and documentation. **Responsibilities:** SystemTest: the responsibility of this test is to ensure that the project as a whole performs as intended.

Staffing and training needs: Team members had to watch youtube videos along with using internet resources in order to gather a strong understanding to implement each of the test cases into the project.

Schedule: Implementing each of the test cases took quite a large amount of time. This was mainly due to trying to understand how creating test cases inside of unity actually works as well as developing test cases that properly test.

Risks and Mitigation: The main risks that our team ran into when it came to implementing testing was a lack of technical knowledge on how testing can be conducted in unity. We were able to make a risk mitigation plan in order to ensure that this issue did not affect us super heavily by allotting our members designated amount of time in which they can research and come to understand how this testing can work. For example watching youtube tutorials on how to set up a unit test inside of unity.

Approvals: Ryan Rutledge, Kaden Groves, A.R. Chessor, Eli Atkisson 11/28/2024

4.8 Acceptance Testing

4.9.1 Acceptance Tests and Results

According to the acceptance test plan that we created we passed all of the tests that our client was able to test during our meeting. As for the other items that where tested that were let blank this was do to our testing template being larger then what the client actually needed to perform tests on. This gives us a 100% pass rate when it came to our acceptance tests that the client tested.

5 Conclusion

In conclusion, this project has been an incredible learning experience, allowing our team to gain valuable insights into software engineering and the dynamics of working collaboratively in a professional setting. This technical document served as a crucial guide throughout our development cycle, helping us plan, organize, and execute our tasks effectively. One major challenge we encountered was time management. Despite our initial enthusiasm and ambitious goals, we realized in the third sprint that our scope was overly ambitious given the timeline. By narrowing our focus, we were able to set more realistic objectives and successfully deliver a functional product. While we couldn't implement every feature we originally envisioned, this decision enabled us to prioritize core functionalities and ensure a polished final output. This project has provided us with a solid foundation in software development practices, including planning, teamwork, and adaptive problem-solving. The experience gained will undoubtedly prove valuable in future endeavors, equipping us with the tools and confidence to tackle more complex projects. Moving forward, there is significant potential to expand upon this project. Enhancements could include adding the remaining features initially planned, refining the existing functionality, and incorporating user feedback to create an even more engaging and robust application. This project has set the stage for continuous learning and improvement.

6 Appendix

6.1 Software Product Build Instructions

To transfer the current state of the Mega Man X game project to a new computer, you'll need to download both the project's documentation and the Unity files. Start by cloning our documentation repository, which we manage using Git and GitHub. Simply run git clone 'projecturlongithub' to copy all technical and organizational documentation to your device. For the Unity project files, you have two options: a team member could send you the files directly, allowing you to load them into Unity, or we could add you as a seat in Unity's built-in version control system using your Unity account. This would grant you direct access to the game's scripts, scenes, and other assets, enabling you to keep files up-to-date within Unity. Within the unity control to make sure you are up to date just go to the incoming changes tab and apply them to your work space. Following these steps will ensure you have the full and current setup of the Mega Man X project on your computer.

6.2 Software Product User Guide

The Software Product User Guide provides an accessible overview for both general users and administrative users of our Mega Man X game. For general users, the guide covers essential functionalities such as registration, login, and gameplay instructions. Once registered and logged in, players can engage with the game, score points, and have their achievements recorded in the system. A leaderboard is available, showcasing the top 10 high scores, fostering a competitive yet engaging experience. On the administrative side, the guide details additional functionalities unique to admins. Administrators have access to an exclusive scene where they can view a comprehensive list of all players who have ever logged into the game, allowing for effective tracking and record maintenance. This section equips both types of users with the necessary information to navigate and utilize the game's features seamlessly.

6.3 Source Code with Comments

BallGuyMovement.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BallGuyMovement : MonoBehaviour
{
    [SerializeField]public float speed = 0.25f;
    [SerializeField]private bool isActivated = false;
    public Transform player;

    void Update()
    {
        if (isActivated)
        {

            if (IsPlayerInCameraView())
            {
                Vector3 direction = (player.position - transform.position).normalized;
                transform.position += direction * speed * Time.deltaTime;
            }
        }
    }

    void OnBecameVisible()
    {
        isActivated = true;
    }

    private bool IsPlayerInCameraView()
    {
        Vector3 viewportPoint = Camera.main.WorldToViewportPoint(player.position);
        return viewportPoint.x >= 0 && viewportPoint.x <= 1 && viewportPoint.y >= 0 && viewportPoint.y <= 1;
    }
}
```

CameraController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraController : MonoBehaviour
{
    [SerializeField] private Transform player;
    [SerializeField] private float aheadDistance;
    [SerializeField] private float cameraSpeed;
    [SerializeField] private float minX;
    [SerializeField] private float maxX;
    private float lookAhead;

    // Update is called once per frame
    void Update()
    {
        float targetX = player.position.x + lookAhead;
        float clampedX = Mathf.Clamp(targetX, minX, maxX);
        transform.position = new Vector3(clampedX, transform.position.y, transform.position.z);
        lookAhead = Mathf.Lerp(lookAhead, aheadDistance * player.localScale.x, Time.deltaTime * cameraSpeed);
    }
}
```

CameraViewportHandler.cs

```
using UnityEngine;

[ExecuteInEditMode]
[RequireComponent(typeof(Camera))]
public class CameraViewportHandler : MonoBehaviour
{
    public enum Constraint { Landscape, Portrait }

    #region FIELDS
```

```
public Color wireColor = Color.white;
public float UnitsSize = 1; // size of your scene in unity units
public Constraint constraint = Constraint.Portrait;
public static CameraViewportHandler Instance;
public new Camera camera;

public bool executeInUpdate;

private float _width;
private float _height;
//*** bottom screen
private Vector3 _bl;
private Vector3 _bc;
private Vector3 _br;
//*** middle screen
private Vector3 _ml;
private Vector3 _mc;
private Vector3 _mr;
//*** top screen
private Vector3 _tl;
private Vector3 _tc;
private Vector3 _tr;
#endregion

#region PROPERTIES
public float Width
{
    get
    {
        return _width;
    }
}
public float Height
{
    get
    {
        return _height;
    }
}

// helper points:
public Vector3 BottomLeft
{
    get
    {
        return _bl;
    }
}
public Vector3 BottomCenter
{
    get
    {
        return _bc;
    }
}
public Vector3 BottomRight
{
    get
    {
        return _br;
    }
}
public Vector3 MiddleLeft
{
    get
    {
        return _ml;
    }
}
public Vector3 MiddleCenter
{
    get
    {
        return _mc;
    }
}
public Vector3 MiddleRight
{
    get
    {
        return _mr;
    }
}
public Vector3 TopLeft
{
    get
    {
        return _tl;
    }
}
public Vector3 TopCenter
{
    get
    {
        return _tc;
    }
}
public Vector3 TopRight
{
    get
    {
        return _tr;
```

```
        }
    }
    #endregion

    #region METHODS
    private void Awake()
    {
        camera = GetComponent<Camera>();
        Instance = this;
        ComputeResolution();
    }

    private void ComputeResolution()
    {
        float leftX, rightX, topY, bottomY;

        if (constraint == Constraint.Landscape)
        {
            camera.orthographicSize = 1f / camera.aspect * UnitsSize / 2f;
        }
        else
        {
            camera.orthographicSize = UnitsSize / 2f;
        }

        _height = 2f * camera.orthographicSize;
        _width = _height * camera.aspect;

        float cameraX, cameraY;
        cameraX = camera.transform.position.x;
        cameraY = camera.transform.position.y;

        leftX = cameraX - _width / 2;
        rightX = cameraX + _width / 2;
        topY = cameraY + _height / 2;
        bottomY = cameraY - _height / 2;

        //*** bottom
        _bl = new Vector3(leftX, bottomY, 0);
        _bc = new Vector3(cameraX, bottomY, 0);
        _br = new Vector3(rightX, bottomY, 0);
        //*** middle
        _ml = new Vector3(leftX, cameraY, 0);
        _mc = new Vector3(cameraX, cameraY, 0);
        _mr = new Vector3(rightX, cameraY, 0);
        //*** top
        _tl = new Vector3(leftX, topY, 0);
        _tc = new Vector3(cameraX, topY, 0);
        _tr = new Vector3(rightX, topY, 0);
    }

    private void Update()
    {
#if UNITY_EDITOR

        if (executeInUpdate)
            ComputeResolution();

#endif
    }

    void OnDrawGizmos()
    {
        Gizmos.color = wireColor;

        Matrix4x4 temp = Gizmos.matrix;
        Gizmos.matrix = Matrix4x4.TRS(transform.position, transform.rotation, Vector3.one);
        if (camera.orthographic)
        {
            float spread = camera.farClipPlane - camera.nearClipPlane;
            float center = (camera.farClipPlane + camera.nearClipPlane) * 0.5f;
            Gizmos.DrawWireCube(new Vector3(0, 0, center), new Vector3(camera.orthographicSize * 2 * camera.aspect, camera.orthogra
        }
        else
        {
            Gizmos.DrawFrustum(Vector3.zero, camera.fieldOfView, camera.farClipPlane, camera.nearClipPlane, camera.aspect);
        }
        Gizmos.matrix = temp;
    }
    #endregion

}
```

CopterMovement.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CopterMovement : MonoBehaviour
{
    [SerializeField] public float speed = 0.25f;
    [SerializeField] private bool isActivated = false;
    public Transform player;

    [SerializeField] private float heightOffset = 2f;
    [SerializeField] private float attackRange = 1.5f;
    private bool movingToLeft = true;

    private void Start()
    {
```

```
    }

    void Update()
    {
        if (isActivated)
        {
            if (IsPlayerInCameraView())
            {
                // Calculate the direction to the player
                Vector3 playerDirection = (player.position - transform.position).normalized;

                // Check if the copter is within attack range
                if (Vector3.Distance(transform.position, player.position) < attackRange)
                {
                    // Stay in position if colliding with the player (Static behavior)
                    transform.position = new Vector3(transform.position.x, player.position.y + heightOffset, transform.position.z);
                }
                else
                {
                    // Determine the target position for the V maneuver
                    Vector3 targetPoint;
                    if (movingToLeft)
                    {
                        targetPoint = new Vector3(player.position.x - 2, player.position.y + heightOffset, player.position.z); // L
                    }
                    else
                    {
                        targetPoint = new Vector3(player.position.x + 2, player.position.y + heightOffset, player.position.z); // R
                    }

                    // Move towards the target point
                    Vector3 swoopDirection = (targetPoint - transform.position).normalized;
                    transform.position += swoopDirection * speed * Time.deltaTime;

                    // Check if the copter reached the target point
                    if (Vector3.Distance(transform.position, targetPoint) < 0.1f)
                    {
                        // Toggle the direction for V movement
                        movingToLeft = !movingToLeft;
                    }
                }
            }
        }
    }

    void OnBecameVisible()
    {
        isActivated = true; // Activate the copter when it becomes visible
    }

    private bool IsPlayerInCameraView()
    {
        Vector3 viewportPoint = Camera.main.WorldToViewportPoint(player.position);
        return viewportPoint.x >= 0 && viewportPoint.x <= 1 && viewportPoint.y >= 0 && viewportPoint.y <= 1;
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.transform == player)
        {
            // Handle collision with player here (optional)
            // You could add logic to decrease health or trigger an event if needed.
        }
    }
}
```

CreditsController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class CreditsController : MonoBehaviour
{
    // Start is called before the first frame update
    public void Start()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }

    // Update is called once per frame
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Escape)){
            SceneManager.LoadScene("MainMenu");
            Debug.Log("MAIN MENU LOADED");
        }
    }
}
```

DatabaseController.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using System.Data.Common;
```

```
using Mono.Data.Sqlite;
using TMPro;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Security.Cryptography;
using System.Text;
using System.IO;

public class DatabaseController : MonoBehaviour
{
    string conn;
    private SqliteConnection dbconn;
    string sqlQuery;
    List<HighScore> highScores;
    public GameObject scoreboardPanel;
    private int mostRecentScore;

    string DATABASE_NAME = "/Database/mydatabase.s3db";

    private int loggedInUserId;

    void Start()
    {
        DontDestroyOnLoad(gameObject);
        highScores = new List<HighScore>();
        Debug.Log("Running start on database");

        string filepath;
#if UNITY_EDITOR
    filepath = Application.dataPath + DATABASE_NAME;
#else
    filepath = Application.persistentDataPath + DATABASE_NAME;
#endif

        Debug.Log($"Database location: {filepath}");

        string directoryPath = Path.GetDirectoryName(filepath);
        if (!Directory.Exists(directoryPath))
        {
            Directory.CreateDirectory(directoryPath);
            Debug.Log("Created directory: " + directoryPath);
        }

        conn = "URI=file:" + filepath;

        try{
            CreateTable();
            if(SceneManager.GetActiveScene().name.Equals("LoginScene")){
                GetScores();
                ShowScores();
            }
        }
        catch (Exception e){
            Debug.LogError("Error initializing database: " + e.Message);
        }
    }

    public void DatabaseTestsSetup()
    {
        DontDestroyOnLoad(gameObject);
        highScores = new List<HighScore>();
        using(dbconn = new SqliteConnection(conn)){
            CreateTable();
            if(SceneManager.GetActiveScene().name.Equals("LoginScene")){
                GetScores();
                ShowScores();
            }
        }
    }

    private void CreateTable()
    {
        using (dbconn = new SqliteConnection(conn))
        {
            dbconn.Open();
            using (var dbcmd = dbconn.CreateCommand())
            {
                sqlQuery = "CREATE TABLE IF NOT EXISTS PlayerData (" +
                    "id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT," +
                    "username VARCHAR(255) NOT NULL UNIQUE," +
                    "password VARCHAR(255) NOT NULL," +
                    "isAdmin INTEGER NOT NULL DEFAULT 0);";
                dbcmd.CommandText = sqlQuery;
                dbcmd.ExecuteNonQuery();

                sqlQuery = "CREATE TABLE IF NOT EXISTS PlayerScores (" +
                        "id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, " +
                        "scoreById INTEGER NOT NULL, " +
                        "score INTEGER NOT NULL, " +
                        "FOREIGN KEY(scoreById) REFERENCES PlayerData(id)); ";
                dbcmd.CommandText = sqlQuery;
                dbcmd.ExecuteNonQuery();
            }
        }
    }

    private void GetScores()
    {
        Debug.Log("Running Get Scores Method");
        using (dbconn = new SqliteConnection(conn))
        {
```

```
        Debug.Log("Opening Database");
        dbconn.Open();
        using(var dbcmd = dbconn.CreateCommand()){
            Debug.Log("Database open");

            string sqlQuery = "SELECT PlayerData.username, PlayerScores.id, PlayerScores.score " +
                              "FROM PlayerScores INNER JOIN PlayerData ON PlayerScores.scoreById = PlayerData.id " +
                              "ORDER BY PlayerScores.score DESC LIMIT 10;";
            dbcmd.CommandText = sqlQuery;

            Debug.Log("Added Sqlite statement");

            using (IDataReader reader = dbcmd.ExecuteReader())
            {
                Debug.Log("Created reader to read database");
                while (reader.Read())
                {
                    string username = reader.GetString(0);
                    int score = reader.GetInt32(2);
                    highScores.Add(new HighScore(score, username));
                    Debug.Log("High Score added: Score of " + score + " by " + username);
                }
                Debug.Log("Closed Reader");
            }
        }

        Debug.Log("Close connection to database");
    }
}

private void ShowScores()
{
    //Start index after title and headers
    int scoreStartIndex = 2;

    for (int i = 0; i < highScores.Count; i++)
    {
        // Access each score panel by index
        Transform scorePanel = scoreboardPanel.transform.GetChild(scoreStartIndex + i);

        if (scorePanel.childCount >= 3)
        {
            //Get the Name and the Score children
            TextMeshProUGUI nameText = scorePanel.GetChild(1).GetComponent<TextMeshProUGUI>();
            TextMeshProUGUI scoreText = scorePanel.GetChild(2).GetComponent<TextMeshProUGUI>();

            // Update the Name and Score fields
            if (nameText != null && scoreText != null)
            {
                nameText.text = highScores[i].username;
                scoreText.text = highScores[i].score.ToString();
                Debug.Log($"Updated UI: Username: {highScores[i].username}, Score: {highScores[i].score}");
            }
            else
            {
                Debug.LogWarning("TextMeshProUGUI components not found in expected child objects.");
            }
        }
        else
        {
            Debug.LogWarning($"Score panel {i} does not contain the expected number of children.");
        }
    }
}

public bool CreateUser(string username, string password)
{
    Debug.Log("Attempting User Creation");
    try{
        using (dbconn = new SqliteConnection(conn))
        {
            dbconn.Open();
            using(var dbcmd = dbconn.CreateCommand()){
                string sqlQuery = "INSERT INTO PlayerData (username, password) VALUES (@username, @password);";
                dbcmd.CommandText = sqlQuery;

                var usernameParam = dbcmd.CreateParameter();
                usernameParam.ParameterName = "@username";
                usernameParam.Value = username;
                dbcmd.Parameters.Add(usernameParam);

                var passwordParam = dbcmd.CreateParameter();
                passwordParam.ParameterName = "@password";
                passwordParam.Value = HashPassword(password);
                dbcmd.Parameters.Add(passwordParam);

                dbcmd.ExecuteNonQuery();

                dbconn.Close();
            }
        }
    }
    catch(SqliteException ex){
        if (ex.ErrorCode == SQLiteErrorCode.Constraint)
        {
            Debug.Log("Username already exists in the database.");
            return false;
        }
    }
    catch(Exception e){
        Debug.Log(e);
        return false;
```

```
        }

        return true;
    }

    public bool CreateAdminUser(string username, string password)
    {
        Debug.Log("Attempting User Creation");
        try{
            using (dbconn = new SqliteConnection(conn))
            {
                dbconn.Open();
                using(var dbcmd = dbconn.CreateCommand()){
                    string sqlQuery = "INSERT INTO PlayerData (username, password, isAdmin) VALUES (@username, @password, 1);";
                    dbcmd.CommandText = sqlQuery;

                    var usernameParam = dbcmd.CreateParameter();
                    usernameParam.ParameterName = "@username";
                    usernameParam.Value = username;
                    dbcmd.Parameters.Add(usernameParam);

                    var passwordParam = dbcmd.CreateParameter();
                    passwordParam.ParameterName = "@password";
                    passwordParam.Value = HashPassword(password);
                    dbcmd.Parameters.Add(passwordParam);

                    dbcmd.ExecuteNonQuery();
                }
                dbconn.Close();
            }
        }
        catch(SqliteException ex){
            if (ex.ErrorCode == SQLiteErrorCode.Constraint)
            {
                Debug.Log("Username already exists in the database.");
                return false;
            }
        }
        catch(Exception e){
            Debug.Log(e);
            return false;
        }

        return true;
    }

    public bool LoginUser(string username, string password)
    {
        Debug.Log("Attempting User Login");
        try
        {
            using (dbconn = new SqliteConnection(conn))
            {
                dbconn.Open();
                using(var dbcmd = dbconn.CreateCommand()){
                    string sqlQuery = "SELECT id FROM PlayerData WHERE username = @username AND password = @password;";
                    dbcmd.CommandText = sqlQuery;

                    var usernameParam = dbcmd.CreateParameter();
                    usernameParam.ParameterName = "@username";
                    usernameParam.Value = username;
                    dbcmd.Parameters.Add(usernameParam);

                    var passwordParam = dbcmd.CreateParameter();
                    passwordParam.ParameterName = "@password";
                    passwordParam.Value = HashPassword(password);
                    dbcmd.Parameters.Add(passwordParam);

                    using (IDataReader reader = dbcmd.ExecuteReader())
                    {
                        if (reader.Read())  // Check if there is at least one result
                        {
                            loggedInUserId = reader.GetInt32(0);
                            Debug.Log($"Login successful. User ID: {loggedInUserId}");
                            return true;
                        }
                        else
                        {
                            Debug.Log("Username and Password not found in the database");
                            return false;
                        }
                    }
                }
            }
        }
        catch (SqliteException ex)
        {
            Debug.Log("Failure when interacting with SQLite Database: " + ex);
            return false;
        }
        catch (Exception e)
        {
            Debug.Log(e);
            return false;
        }
    }

    public string HashPassword(string password)
    {
        using (SHA256 sha256 = SHA256.Create())
        {
            byte[] bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(password));
```

```
        StringBuilder builder = new StringBuilder();
        foreach (byte b in bytes)
        {
            builder.Append(b.ToString("x2")); // Convert byte to hex
        }
        return builder.ToString();
    }
}

public bool IsAdmin()
{
    using (dbconn = new SqliteConnection(conn))
    {
        dbconn.Open();
        using(var dbcmd = dbconn.CreateCommand()){
            string sqlQuery = "SELECT isAdmin FROM PlayerData WHERE id = @userId;";
            dbcmd.CommandText = sqlQuery;

            var idParam = dbcmd.CreateParameter();
            idParam.ParameterName = "@userId";
            idParam.Value = loggedInUserId;
            dbcmd.Parameters.Add(idParam);

            using (IDataReader reader = dbcmd.ExecuteReader())
            {
                if (reader.Read())
                {
                    int isAdmin = reader.GetInt32(0);
                    return isAdmin == 1;
                }
            }
        }
    }
    return false;
}

public void SaveScore(int score){
    using (dbconn = new SqliteConnection(conn)){
        dbconn.Open();
        using(var dbcmd = dbconn.CreateCommand()){
            string sqlQuery = "INSERT INTO PlayerScores (scoreById, score) VALUES (@scoreById, @score);";
            dbcmd.CommandText = sqlQuery;

            var scoreByIdParam = dbcmd.CreateParameter();
            scoreByIdParam.ParameterName = "@scoreById";
            scoreByIdParam.Value = loggedInUserId;
            dbcmd.Parameters.Add(scoreByIdParam);

            var scoreParam = dbcmd.CreateParameter();
            scoreParam.ParameterName = "@score";
            scoreParam.Value = score;
            dbcmd.Parameters.Add(scoreParam);

            dbcmd.ExecuteNonQuery();
        }
    }

    mostRecentScore = score;
}

//FOR TESTING PURPOSES ONLY
public bool CheckForScoreInDatabase(int score){
    using (dbconn = new SqliteConnection(conn))
    {
        dbconn.Open();
        using (var dbcmd = dbconn.CreateCommand())
        {
            string sqlQuery = "SELECT score FROM PlayerScores WHERE scoreById = @userId AND score = @score LIMIT 1;";
            dbcmd.CommandText = sqlQuery;

            var idParam = dbcmd.CreateParameter();
            idParam.ParameterName = "@userId";
            idParam.Value = loggedInUserId;
            dbcmd.Parameters.Add(idParam);

            var scoreParam = dbcmd.CreateParameter();
            scoreParam.ParameterName = "@score";
            scoreParam.Value = score;
            dbcmd.Parameters.Add(scoreParam);

            using (IDataReader reader = dbcmd.ExecuteReader())
            {
                if (reader.Read())
                {
                    int scoreInDb = reader.GetInt32(0);
                    Debug.Log($"Score found in DB: {scoreInDb}");
                    return scoreInDb == score;
                }
            }
        }

        return false;
    }
}

public void CloseDatabaseConnection()
{
    if (dbconn != null)
    {
        if (dbconn.State == ConnectionState.Open)
        {
            dbconn.Close();
```

```
                Debug.Log("Database connection closed.");
            }
            dbconn.Dispose();
            dbconn = null;
        }
    }

    //FOR TESTING PURPOSES ONLY
    public void SetDatabasePath(string customPath)
    {
        conn = "URI=file:" + customPath;
    }

    public int GetMostRecentScore(){
        return mostRecentScore;
    }

    public void logout(){
        loggedInUserId = 0;
        mostRecentScore = 0;
    }

    public int GetCurrentUserId(){
        return loggedInUserId;
    }
}

class HighScore{
    public int score = 0;
    public string username;

    public HighScore(int score, string username)
    {
        this.score = score;
        this.username = username;
    }
}
```

DeathFloor.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
public class DeathZone : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            HandleDefeat();
        }
    }

    void HandleDefeat()
    {
        Debug.Log("Player fell off the map! Game Over.");
        SceneManager.LoadScene("Defeat");
    }
}
```

DefeatMenuController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class DefeatSceneController : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }

    public void Retry()
    {
        SceneManager.LoadScene("MainScene");
        Debug.Log("MAIN SCENE LOADED.");
    }

    public void GoToMainMenu(){
        SceneManager.LoadScene("MainMenu");
        Debug.Log("MAIN MENU LOADED");
    }
}
```

Enemy.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Enemy : MonoBehaviour
{
    public int maxHealth = 3;
```

```
private int currentHealth;
public int damage = 2;
public int pointValue = 100;
private bool isDead = false;

public AudioSource audioSource;
public AudioClip hitSound;

public GameObject smallHealthCapsulePrefab;
public GameObject largeHealthCapsulePrefab;

void Start()
{
    hitSound = Resources.Load<AudioClip>("Audio/enemy_hit");
    audioSource = gameObject.AddComponent<AudioSource>();
    audioSource.playOnAwake = false;
    audioSource.volume = 1f;
    currentHealth = maxHealth;
}

public void TakeDamage(int damage)
{
    audioSource.PlayOneShot(hitSound);
    currentHealth -= damage;

    if (currentHealth <= 0)
    {
        currentHealth = 0;

        Die();
    }
}

private void Die()
{

    if (isDead) return;
    isDead = true;


    GameObject.Find("TimeAndScoreController").GetComponent<TimeAndScoreController>().UpdateScore(pointValue);
    Debug.Log("Enemy defeated! Worth " + pointValue + " points!");


    SpawnHealthCapsule();

    //Destroys the visuals of the enemy
    Destroy(gameObject.GetComponent<SpriteRenderer>());
    Destroy(gameObject.GetComponent<CircleCollider2D>());

    StartCoroutine(PlayDeathSoundAndDestroy());
}

private void SpawnHealthCapsule()
{

    int randomNumber = Random.Range(1, 10);


    if (randomNumber >= 4 && randomNumber <= 6)
    {

        Instantiate(smallHealthCapsulePrefab, transform.position, Quaternion.identity);
        Debug.Log("Small health capsule spawned.");
    }
    else if (randomNumber >= 7 && randomNumber <= 9)
    {

        Instantiate(largeHealthCapsulePrefab, transform.position, Quaternion.identity);
        Debug.Log("Large health capsule spawned.");
    }
    else
    {
        Debug.Log("No health capsule spawned.");
    }
}

void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("PlayerBullet"))
    {
        Debug.Log("Bullet collided with Enemy");

        PlayerBullet bullet = other.GetComponent<PlayerBullet>();
        if (bullet != null)
        {
            int bulletDamage = bullet.GetDamage();
            Debug.Log("Bullet damage: " + bulletDamage);
            TakeDamage(bulletDamage);
            bullet.Explode();
            Debug.Log("Waiting for explosion animation");
            StartCoroutine(DestroyAfterExplosion(other.gameObject, 0.25f));
        }
    }
}

private IEnumerator DestroyAfterExplosion(GameObject bullet, float delay)
{
    yield return new WaitForSeconds(delay);
    Debug.Log("Explosion animation is over");
    Destroy(bullet);
}
```

```
    private IEnumerator PlayDeathSoundAndDestroy()
    {
        audioSource.PlayOneShot(hitSound);

        //Wait for the audio to finish playing
        yield return new WaitForSeconds(hitSound.length);

        //Fully destroy the GameObject
        Destroy(gameObject);
    }
}
```

EnergyCapsule.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class EnergyCapsule : MonoBehaviour
{
    public string capsuleType; // "Small" or "Large"

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            var playerHealth = other.GetComponent<PlayerHealth>();
            if (playerHealth != null)
            {
                playerHealth.HealingCapsule(capsuleType);
                Destroy(gameObject); // Destroy the capsule after healing
            }
        }
    }
}
```

GameMusic.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameMusic : MonoBehaviour
{
    [SerializeField] GameObject musicLoop;
    [SerializeField] [Range(0,1)] float musicVolume = 0.8f;

    private void Awake()
    {
        SingleInstanceOfMusic();
    }

    private void SingleInstanceOfMusic()
    {
        if(FindObjectsOfType(GetType()).Length > 1)
        {
            Destroy(gameObject);
        }
    }

    void Start()
    {
        GameObject music = Instantiate(musicLoop, Camera.main.transform.position, Quaternion.identity);
    }
}
```

GameSaver.cs

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;

public class GameSaver : MonoBehaviour
{
    private int currentUserId;
    private string filePath, directoryPath;
    TimeAndScoreController timeAndScoreController;
    PlayerHealth playerHealth;
    [SerializeField] GameObject[] enemies;
    [SerializeField] GameObject checkpoint;
    void Start(){
        currentUserId = GameObject.Find("DatabaseController").GetComponent<DatabaseController>().GetCurrentUserId();
        timeAndScoreController = GameObject.Find("TimeAndScoreController").GetComponent<TimeAndScoreController>();
        playerHealth = GameObject.Find("Mega Man X").GetComponent<PlayerHealth>();

        directoryPath = Path.Combine(Application.dataPath, "PlayerSaves");
        filePath = Path.Combine(directoryPath, $"playersaveid{currentUserId}.txt");

        //Ensure the directory exists, if not it creates the directory
        if (!Directory.Exists(directoryPath))
        {
            Directory.CreateDirectory(directoryPath);
        }

        //Check to see if there is a file save for the user id
        if(File.Exists(filePath)){
```

```
            LoadGame();
        }
    }
    public void SaveGame()
    {
        checkpoint.GetComponent<BoxCollider2D>().enabled = false;

        // Retrieve necessary components and data
        float currentTimeLeft = timeAndScoreController.timeRemaining;
        int currentScore = timeAndScoreController.score;
        int currentHealth = playerHealth.GetCurrentHealth();

        Debug.Log($"Directory path is: {directoryPath}");
        Debug.Log($"File path is: {filePath}");

        // Save data to the file
        using (StreamWriter writer = new StreamWriter(filePath))
        {
            writer.WriteLine(currentTimeLeft);
            writer.WriteLine(currentScore);
            writer.WriteLine(currentHealth);
        }

        Debug.Log("Game saved to " + filePath);
    }

    public void LoadGame(){
        checkpoint.GetComponent<BoxCollider2D>().enabled = false;

        float currentTimeRemaining = 0f;
        int currentScore = 0;
        int currentHealth = 0;

        using (StreamReader reader = new StreamReader(filePath))
        {
            if (float.TryParse(reader.ReadLine(), out float timeRemaining)){
                currentTimeRemaining = timeRemaining;
            }
            if (int.TryParse(reader.ReadLine(), out int score)){
                currentScore = score;
            }
            if (int.TryParse(reader.ReadLine(), out int health)){
                currentHealth = health;
            }
        }

        timeAndScoreController.timeRemaining = currentTimeRemaining;
        timeAndScoreController.score = currentScore;
        playerHealth.SetCurrentHealth(currentHealth);

        //Destroy the first 16 enemies in the game that come before the checkpoint when loading the game
        for(int i = 0; i<=15; i++){
            Destroy(enemies[i]);
        }

        //Move Player and Camera to the checkpoint
        Transform playerTransform = GameObject.Find("Mega Man X").GetComponent<Transform>();
        Transform cameraTransform = GameObject.Find("Main Camera").GetComponent<Transform>();

        playerTransform.position = new Vector3(170.6f, 0.76f, -5f);
        cameraTransform.position = new Vector3(170.6f, 0f, -10f);
    }

    public void DeleteCheckpointSave(){
        if(File.Exists(filePath)){
            File.Delete(filePath);
        }
    }
}
```

gunvolt.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Gunvolt : MonoBehaviour
{
    public GameObject projectilePrefab; // Reference to the projectile prefab
    public Transform player; // Reference to the player's transform
    public float attackRange = 10f; // Range at which Gunvolt will attack
    public float fireRate = 1f; // Rate of fire for projectiles

    private float nextFireTime = 5f;

    private void Update()
    {
        // Check distance to player
        if (Vector2.Distance(transform.position, player.position) < attackRange)
        {
            Attack();
        }
    }

    private void Attack()
    {
        // Check if Gunvolt can fire a projectile based on the fire rate
        if (Time.time >= nextFireTime)
        {
            // Create a projectile
            GameObject projectile = Instantiate(projectilePrefab, transform.position, Quaternion.identity);
```

```
            // Set direction of the projectile towards the player
            Vector2 direction = (player.position - transform.position).normalized;
            projectile.GetComponent<MissileProjectile>().SetDirection(direction);

            // Update the next fire time
            nextFireTime = Time.time + fireRate;
        }
    }
}
```

LoginMenuController.cs

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class LoginMenu : MonoBehaviour
{
    public GameObject loginPanel, signUpPanel, scoreboardPanel, notificationPanel, DatabaseController;
    public TMP_InputField loginUsername, loginPassword, signupUsername, signupPassword, signupConfPassword;
    public TextMeshProUGUI notifTitleText, notifMessageText;

    public void Start()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }
    public void OpenLoginPanel(){
        loginPanel.SetActive(true);
        scoreboardPanel.SetActive(true);
        signUpPanel.SetActive(false);
    }

    public void OpenSignUpPanel(){
        loginPanel.SetActive(false);
        scoreboardPanel.SetActive(false);
        signUpPanel.SetActive(true);
    }

    public void LoginUser(){
        if(string.IsNullOrWhiteSpace(loginUsername.text) || string.IsNullOrWhiteSpace(loginPassword.text)){
            ShowErrorNotificationMessage("ERROR", "BLANK PASSWORD OR USERNAME");
            return;
        }

        else if(loginUsername.text.Length < 4  || loginUsername.text.Length > 11)
        {
            ShowErrorNotificationMessage("ERROR", "USERNAME TOO LONG OR SHORT, NEEDS TO BE 4-11 CHARACTERS LONG");
            return;
        }
        else if(loginPassword.text.Length < 8 || loginPassword.text.Length >= 20)
        {
            ShowErrorNotificationMessage("ERROR", "PASSWORD TOO LONG OR SHORT, NEEDS TO BE 8-20 CHARACTERS LONG");
            return;
        }

        bool loginSuccess = DatabaseController.GetComponent<DatabaseController>().LoginUser(loginUsername.text, loginPassword.text)

        if(loginSuccess){
            SceneManager.LoadScene("MainMenu");
        }
        else{
            ShowErrorNotificationMessage("ERROR", "USERNAME/PASSWORD INVALID, USER NOT FOUND, CHECK SPELLING AND MAKE SURE YOUR ACCO
        }
    }

    public void SignUpUser(){
        if(string.IsNullOrWhiteSpace(signupUsername.text) || string.IsNullOrWhiteSpace(signupPassword.text) || string.IsNullOrWhiteS
            ShowErrorNotificationMessage("ERROR", "BLANK PASSWORD OR USERNAME");
            return;
        }
        else if(signupUsername.text.Length < 4  || signupUsername.text.Length > 11)
        {
            ShowErrorNotificationMessage("ERROR", "USERNAME TOO LONG OR SHORT, NEEDS TO BE 4-11 CHARACTERS LONG");
            return;
        }
        else if(signupPassword.text.Length < 8 || signupPassword.text.Length >= 20)
        {
            ShowErrorNotificationMessage("ERROR", "PASSWORD TOO LONG OR SHORT, NEEDS TO BE 8-20 CHARACTERS LONG");
            return;
        }
        else if(!signupConfPassword.text.Equals(signupPassword.text)){
            ShowErrorNotificationMessage("ERROR", "PASSWORD DOES NOT MATCH CONFIRMATION PASSWORD, BE SURE TO DOUBLE CHECK SPELLING"
            return;
        }

        bool userCreated = DatabaseController.GetComponent<DatabaseController>().CreateUser(signupUsername.text, signupPassword.text

        if(userCreated){
            OpenLoginPanel();
        }
        else{
            ShowErrorNotificationMessage("ERROR", "USER FAILED TO BE CREATED, USERNAME IS ALREADY IN USE OR INVALID FORMAT FOR USERN
        }
    }

    public void ShowErrorNotificationMessage(string title, string message){
```

```
        notifTitleText.text = title;
        notifMessageText.text = message;

        notificationPanel.SetActive(true);
    }

    public void CloseErrorNotificationMessage(){
        notifTitleText.text = "";
        notifMessageText.text = "";

        notificationPanel.SetActive(false);
    }
}
```

MainMenu.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour
{
    public GameObject editDatabaseButton;

    public void Start()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;

        if(GameObject.Find("DatabaseController").GetComponent<DatabaseController>().IsAdmin()){
            editDatabaseButton.SetActive(true);
        }
        else{
            editDatabaseButton.SetActive(false);
        }
    }
    public void StartGame()
    {

        SceneManager.LoadScene("MainScene");
        Debug.Log("MAINSCENE LOADED.");
    }
    public void OpenSettings()
    {
        SceneManager.LoadScene("SettingsScene");
    }

    public void QuitGame()
    {
        Application.Quit();

        //If Unity Editor, stop play mode
#if UNITY_EDITOR
        UnityEditor.EditorApplication.isPlaying = false;
#endif
    }
    public void Logout()
    {
        GameObject.Find("DatabaseController").GetComponent<DatabaseController>().logout();
        SceneManager.LoadScene("LoginScene");
    }

    public void LoadDatabaseEditor()
    {
        SceneManager.LoadScene("DatabaseEditor");
    }

    public void ShowCredits(){
        SceneManager.LoadScene("Credits");
    }
}
```

MissileProjectile.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class MissileProjectile : MonoBehaviour
{
    public float speed = 12f;
    public float lifetime = 3f;

    private Vector2 direction;

    private void Start()
    {
        Destroy(gameObject, lifetime);
    }

    private void Update()
    {
        transform.Translate(direction * speed * Time.deltaTime);
    }


    public void SetDirection(Vector2 newDirection)
    {
```

```
        direction = newDirection.normalized;
    }

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            collision.GetComponent<PlayerHealth>().TakeDamage(4);
            Destroy(gameObject);
        }
    }
}
```

PlayerBullet.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using UnityEngine;

public class PlayerBullet : MonoBehaviour
{
    [SerializeField] private GameObject weakBullet;
    [SerializeField] private GameObject strongBullet;
    [SerializeField] private GameObject chargedBullet;
    private string bulletType;
    private Animator anim;

    //CHESSOR 11.19.24
    public AudioSource audioSource;
    public AudioClip fireSoundWeak;
    public AudioClip fireSoundCharged;
    public AudioClip fireSoundStrong;

    private void Awake(){
        fireSoundWeak = Resources.Load<AudioClip>("Audio/buster");
        fireSoundCharged = Resources.Load<AudioClip>("Audio/strongbuster");
        fireSoundStrong = Resources.Load<AudioClip>("Audio/mediumbuster");
        audioSource = gameObject.AddComponent<AudioSource>();
        anim = transform.GetComponent<Animator>();
    }

    public void MakeBulletType(string type, float playerFacing){
        if(type.Equals("Weak")){
            bulletType = "Weak";
            transform.localScale = weakBullet.GetComponent<Transform>().localScale;
            transform.GetComponent<BoxCollider2D>().size = weakBullet.GetComponent<BoxCollider2D>().size;
            transform.localScale = new Vector3(playerFacing*Mathf.Abs(transform.localScale.x), transform.localScale.y, transform.loc
            audioSource.PlayOneShot(fireSoundWeak);
            anim.SetTrigger("WeakBullet");
        }
        else if(type.Equals("Strong")){
            bulletType = "Strong";
            transform.localScale = strongBullet.GetComponent<Transform>().localScale;
            transform.GetComponent<BoxCollider2D>().size = strongBullet.GetComponent<BoxCollider2D>().size;
            transform.localScale = new Vector3(playerFacing*Mathf.Abs(transform.localScale.x), transform.localScale.y, transform.loc
            audioSource.PlayOneShot(fireSoundStrong);
            anim.SetTrigger("StrongBullet");
        }
        else if(type.Equals("Charged")){
            bulletType = "Charged";
            transform.localScale = chargedBullet.GetComponent<Transform>().localScale;
            transform.GetComponent<BoxCollider2D>().size = chargedBullet.GetComponent<BoxCollider2D>().size;
            transform.localScale = new Vector3(playerFacing*Mathf.Abs(transform.localScale.x), transform.localScale.y, transform.loc
            audioSource.PlayOneShot(fireSoundCharged);
            anim.SetTrigger("ChargedBullet");
        }
    }
    public int GetDamage(){
        int damage = 0;

        if(bulletType.Equals("Weak")){
            damage = 1;
        }
        else if(bulletType.Equals("Strong")){
            damage = 4;
        }
        else if(bulletType.Equals("Charged")){
            damage = 10;
        }

        return damage;
    }
    public float GetSpeed(){
        float speed = 0f;

        if(bulletType.Equals("Weak")){
            speed = 8f;
        }
        else if(bulletType.Equals("Strong")){
            speed = 10f;
        }
        else if(bulletType.Equals("Charged")){
            speed = 12f;
        }

        return speed;
    }

    public void Explode(){
```

```
        transform.GetComponent<Rigidbody2D>().velocity = Vector2.zero;
        anim.SetTrigger("Explode");
    }
}
```

PlayerHealth.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class PlayerHealth : MonoBehaviour
{
    [SerializeField] public int maxHealth = 20;
    [SerializeField] private int currentHealth = 20;
    [SerializeField] public Image healthBarImage; // Reference to the health bar image
    [SerializeField] public Sprite[] healthBarSprites; // Array to hold different health bar sprites
    [SerializeField] public int smallHealAmount = 2;
    [SerializeField] public int largeHealAmount = 8;
    public AudioSource audioSource;
    public AudioClip healthSound;
    public AudioClip deathSound;

    void Awake(){
        currentHealth = maxHealth;
    }

    void Start()
    {
        healthSound = Resources.Load<AudioClip>("Audio/refill");
        deathSound = Resources.Load<AudioClip>("Audio/death");
        UpdateHealthBar();
    }

    public void TakeDamage(int damage)
    {
        currentHealth -= damage;

        if (currentHealth < 0)
        {
            currentHealth = 0;

        }

        UpdateHealthBar();

        if (currentHealth == 0)
        {
            // for some reason it breaks  being able to die. audioSource.PlayOneShot(deathSound); -Chessor
            Defeat();
        }
    }

    public void Heal(int amount)
    {
        currentHealth += amount;
        // for some reason it breaks picking up the healing orb audioSource.PlayOneShot(healthSound); - Chessor
        if (currentHealth > maxHealth)
        {
            currentHealth = maxHealth;
        }

        UpdateHealthBar();
    }

    void Defeat()
    {
        Debug.Log("Defeat.");
        SceneManager.LoadScene("Defeat");
    }

    public void HealingCapsule(string capsuleType)
    {
        switch (capsuleType)
        {
            case "Small":
                Heal(smallHealAmount);
                break;
            case "Large":
                Heal(largeHealAmount);
                break;
            default:
                Debug.LogWarning("Unknown capsule type!");
                break;
        }
    }

    private void UpdateHealthBar()
    {
        if (healthBarSprites.Length > 0)
        {
            int spriteIndex = Mathf.Clamp((currentHealth * healthBarSprites.Length) / maxHealth, 0, healthBarSprites.Length - 1);
            healthBarImage.sprite = healthBarSprites[spriteIndex];
        }
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        Debug.Log("Collision Occurred");
```

```
        if (other.CompareTag("Enemy"))
        {
            Debug.Log("Collision with Enemy");
            var enemy = other.GetComponentInParent<Enemy>();
            if (enemy != null)
            {
                Debug.Log("Enemy is not null");
                TakeDamage(enemy.damage);
            }
            else
            {
                Debug.Log("Enemy is null, what're you DOING");
            }
        }
    }

    public int GetCurrentHealth(){
        return currentHealth;
    }

    public void SetCurrentHealth(int newHealthValue){
        currentHealth = newHealthValue;
        UpdateHealthBar();
    }
}
```

PlayerMovement.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.SceneManagement;

public class PlayerMovement : MonoBehaviour
{
    private Rigidbody2D m_Rigidbody;
    private Animator anim;
    private BoxCollider2D boxCollider;
    [SerializeField] private LayerMask groundLayer;
    [SerializeField] private LayerMask wallLayer;
    public float m_speed = 5f;
    public float jumpforce = 10f;
    private float wallJumpCooldown;
    private float horizontalInput;

    public AudioSource audioSource;
    public AudioClip collectSound;
    public AudioClip landSound;
    //public GameObject bulletPrefab;

    //public AudioClip collectSound;

    void Awake()
    {
        landSound = Resources.Load<AudioClip>("Audio/land");
        m_Rigidbody = GetComponent<Rigidbody2D>();
        anim = GetComponent<Animator>();
        boxCollider = GetComponent<BoxCollider2D>();
        Cursor.visible = false;
        Cursor.lockState = CursorLockMode.Locked;
    }

    // Update is called once per frame
    void Update()
    {
        horizontalInput = Input.GetAxis("Horizontal");
        m_Rigidbody.velocity = new Vector2(horizontalInput * m_speed, m_Rigidbody.velocity.y);

        if (horizontalInput > 0.01f){
            transform.localScale = new Vector3(4.5f, transform.localScale.y, transform.localScale.z);
        }
        else if (horizontalInput < -0.01f){
            transform.localScale = new Vector3(-4.5f, transform.localScale.y, transform.localScale.z);
        }

        //Wall jump implementation and checks
        if(wallJumpCooldown > 0.2f){
            m_Rigidbody.velocity = new Vector2(horizontalInput * m_speed, m_Rigidbody.velocity.y);

            if(onWall() && !isGrounded()){
                m_Rigidbody.gravityScale = 0;
                m_Rigidbody.velocity = Vector2.zero;
            }
            else
                m_Rigidbody.gravityScale = 1.75f;

            if (Input.GetKeyDown(KeyCode.X))
            {
                Jump();
            }
        }
        else
            wallJumpCooldown += Time.deltaTime;

        anim.SetBool("Run", horizontalInput != 0);
        anim.SetBool("Grounded", isGrounded());

        //if(Input.GetKeyDown(KeyCode.W) || Input.GetKeyDown(KeyCode.A) || Input.GetKeyDown(KeyCode.S) || Input.GetKeyDown(KeyCode.D
        //{
```

```
            //footstepSource.Play();
            //footstepSource.loop = true;
        //}

        //if(!Input.anyKey)
        //{
            //footstepSource.loop = false;
        //}

        //if (Input.GetKeyDown(KeyCode.Mouse0))
        //{
            //Fire();
        //}
    }

    private void Fire()
    {
        //GameObject bullet = Instantiate(bulletPrefab, transform.position, transform.rotation);
    }

    private void Jump()
    {
        if(isGrounded()){
            m_Rigidbody.velocity = new Vector2(m_Rigidbody.velocity.x, jumpforce);
            anim.SetTrigger("Jump");
        }
        else if (onWall() && !isGrounded()){
            if(horizontalInput == 0){
                m_Rigidbody.velocity = new Vector2(-Mathf.Sign(transform.localScale.x) * 10, jumpforce/4);
                transform.localScale = new Vector3(-Mathf.Sign(transform.localScale.x) * 4.5f, transform.localScale.y, transform.loc
            }
            else
                m_Rigidbody.velocity = new Vector2(-Mathf.Sign(transform.localScale.x) * 20, jumpforce/2);

            wallJumpCooldown = 0f;
        }
    }

    private bool isGrounded(){
        RaycastHit2D raycastHit2D = Physics2D.BoxCast(boxCollider.bounds.center, boxCollider.bounds.size, 0, Vector2.down, 0.1f, gro
        return raycastHit2D.collider != null;
    }

    private bool onWall(){
        RaycastHit2D raycastHit2D = Physics2D.BoxCast(boxCollider.bounds.center, boxCollider.bounds.size, 0, new Vector2(transform.l
        return raycastHit2D.collider != null;
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if(other.gameObject.tag.Equals("Checkpoint")){
            GameObject.Find("GameSaver").GetComponent<GameSaver>().SaveGame();
        }
        if(other.gameObject.tag.Equals("Finish")){
            other.GetComponent<ScoreSaver>().SaveScoreToDatabase();
            SceneManager.LoadScene("Victory");
            Debug.Log("MAIN MENU LOADED");
        }
    }
}
```

PlayerShooting.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerShooting : MonoBehaviour
{
    [SerializeField] private float attackCooldown;
    private Animator anim;
    private float cooldownTimer = 10f; //Allows player to immeadiately fire when loaded into game without waiting for the cooldown c
    [SerializeField] private float chargeTime = 0f;
    private bool chargingBuster = false;
    [SerializeField] private GameObject PlayerBullet;

    public AudioSource audioSource;
    public AudioClip chargingSound;
    private void Awake()
    {
        audioSource = new AudioSource();
        chargingSound = Resources.Load<AudioClip>("Audio/charging");
        anim = GetComponent<Animator>();
    }

    private void Update()
    {
        cooldownTimer += Time.deltaTime;

        if (Input.GetKeyDown(KeyCode.C) && cooldownTimer >= attackCooldown)
        {

            chargingBuster = true;
            chargeTime = 0f;
            anim.SetBool("Charging", true);
            //audioSource.PlayOneShot(chargingSound);
        }
        if (Input.GetKey(KeyCode.C) && chargingBuster)
        {

            chargeTime += Time.deltaTime;
```

```
        }
        if (Input.GetKeyUp(KeyCode.C) && anim.GetBool("Charging"))
        {
            chargingBuster = false;
            Shoot();
            anim.SetTrigger("Fire");
            anim.SetBool("Charging", false);
        }
    }

    private void Shoot()
    {
        Vector3 bulletPositionOffset = new Vector3(Mathf.Sign(transform.localScale.x) * 0.85f, 0.06f, 0);

        GameObject bullet = Instantiate(PlayerBullet, transform.position + bulletPositionOffset, Quaternion.identity);

        float playerFacing = Mathf.Sign(transform.localScale.x);

        if (chargeTime <= 0.5f)
        {
            bullet.GetComponent<PlayerBullet>().MakeBulletType("Weak", playerFacing);
        }
        else if (chargeTime > 0.5f && chargeTime <= 1.5f)
        {
            bullet.GetComponent<PlayerBullet>().MakeBulletType("Strong", playerFacing);
        }
        else if (chargeTime > 1.5f)
        {
            bullet.GetComponent<PlayerBullet>().MakeBulletType("Charged", playerFacing);
        }

        float projSpeed = bullet.GetComponent<PlayerBullet>().GetSpeed();
        bullet.GetComponent<Rigidbody2D>().velocity = new Vector2(playerFacing*projSpeed, 0);

        cooldownTimer = 0;
    }
}
```

SceneLoggers/SceneTimeTrackers.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Collections.Generic;
using System.IO;

public class SceneTimeTracker : MonoBehaviour
{
    private float startTime;
    private Dictionary<string, float> sceneTimes = new Dictionary<string, float>();

    void Awake()
    {
        DontDestroyOnLoad(gameObject);
        SceneManager.sceneLoaded += OnSceneLoaded;
        SceneManager.sceneUnloaded += OnSceneUnloaded;
    }

    void Start()
    {
        startTime = Time.time;
    }

    void OnSceneLoaded(Scene scene, LoadSceneMode mode)
    {
        startTime = Time.time;  // Reset start time for the new scene
    }

    void OnSceneUnloaded(Scene scene)
    {
        LogSceneTime(); // Log time when leaving a scene
    }

    void LogSceneTime()
    {
        string sceneName = SceneManager.GetActiveScene().name;
        float elapsedTime = Time.time - startTime;

        if (elapsedTime < 0.1f) return;  // Skip logging if elapsed time is too short

        if (sceneTimes.ContainsKey(sceneName))
            sceneTimes[sceneName] += elapsedTime;
        else
            sceneTimes[sceneName] = elapsedTime;

        startTime = Time.time;  // Reset timer after logging
    }

    void OnApplicationQuit()
    {
        LogSceneTime();  // Final log on application quit

        // Log to Console
        Debug.Log("Scene Times:");
        foreach (var entry in sceneTimes)
        {
            Debug.Log($"Scene: {entry.Key}, Time Spent: {entry.Value:F2} seconds");
        }

        // Log to File
        string filePath = Path.Combine(Application.persistentDataPath, "SceneTimeLog.txt");
        using (StreamWriter writer = new StreamWriter(filePath))
```

```
        {
            writer.WriteLine("Scene Times:");
            foreach (var entry in sceneTimes)
            {
                writer.WriteLine($"Scene: {entry.Key}, Time Spent: {entry.Value:F2} seconds");
            }
        }
        Debug.Log($"Scene time log saved to: {filePath}");
    }
}
```

ScoreSaver.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ScoreSaver : MonoBehaviour
{
    public void SaveScoreToDatabase(){
        GameObject.Find("GameSaver").GetComponent<GameSaver>().DeleteCheckpointSave();
        TimeAndScoreController timeAndScoreController = GameObject.Find("TimeAndScoreController").GetComponent<TimeAndScoreControlle
        timeAndScoreController.AddTimeToScore();
        int finalScore = timeAndScoreController.score;
        GameObject.Find("DatabaseController").GetComponent<DatabaseController>().SaveScore(finalScore);
    }
}
```

Spikey.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class VictoryMenuController : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }

    public void Retry()
    {
        SceneManager.LoadScene("MainScene");
        Debug.Log("MAIN SCENE LOADED.");
    }

    public void GoToMainMenu(){
        SceneManager.LoadScene("MainMenu");
        Debug.Log("MAIN MENU LOADED");
    }
}
```

TimeAndScoreController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;
using TMPro;
public class TimeAndScoreController : MonoBehaviour
{
    public TextMeshProUGUI timerText, scoreText;
    public float timeRemaining = 600f;
    public int score = 0;
    private bool isGameOver = false;

    void Start(){
        Debug.Log("Score and Timer running");
        scoreText.text = string.Format("{0:0000}", score);
    }

    void Update()
    {
        if (!isGameOver)
        {
            timeRemaining -= Time.deltaTime;
            DisplayTime(timeRemaining);

            if (timeRemaining <= 0)
            {
                timeRemaining = 0;
                isGameOver = true;
                HandleDefeat();
            }
        }
    }

    void DisplayTime(float timeToDisplay)
    {
        int minutes = Mathf.FloorToInt(timeToDisplay / 60);
        int seconds = Mathf.FloorToInt(timeToDisplay % 60);
        timerText.text = $"{minutes:00}:{seconds:00}";
    }
```

```
    void HandleDefeat()
    {
        Debug.Log("Time's up! Player defeated.");
        SceneManager.LoadScene("Defeat");
    }

    public void UpdateScore(int addedPoints){
        score += addedPoints;
        scoreText.text = string.Format("{0:0000}", score);
    }

    public void AddTimeToScore(){
        score += (int)timeRemaining * 10;
    }
}
```

UIManager.cs

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;

public class UIManager : MonoBehaviour
{
    [SerializeField] private GameObject pauseScreen;

    // Update is called once per frame
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Escape)){
            if(pauseScreen.activeInHierarchy){
                PauseGame(false);
                Cursor.visible = false;
                Cursor.lockState = CursorLockMode.Locked;
            }
            else{
                PauseGame(true);
                Cursor.visible = true;
                Cursor.lockState = CursorLockMode.None;
            }
        }
    }

    private void PauseGame(bool status){
        pauseScreen.SetActive(status);

        if(status)
            Time.timeScale = 0;
        else
            Time.timeScale = 1;
    }

    public void ExitToMainMenu(){
        Time.timeScale = 1;
        SceneManager.LoadScene("MainMenu");
        Debug.Log("MAIN MENU LOADED");
    }
}
```

VictoryMenuController.cs

```
using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;

public class VictoryMenuController : MonoBehaviour
{
    [SerializeField] TextMeshProUGUI finalScoreText;

    // Start is called before the first frame update
    void Start()
    {
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;

        int finalScore = GameObject.Find("DatabaseController").GetComponent<DatabaseController>().GetMostRecentScore();
        finalScoreText.text = $"Final Score: {finalScore}";
    }

    public void Retry()
    {
        SceneManager.LoadScene("MainScene");
        Debug.Log("MAIN SCENE LOADED.");
    }

    public void GoToMainMenu(){
        SceneManager.LoadScene("MainMenu");
        Debug.Log("MAIN MENU LOADED");
    }
}
```

Variable Name	Data Type	Description	Example Value
player	Object	Represents the main player character, Mega Man.	N/A
maxHealth	Integer	Maximum health of the player.	16
currentHealth	Integer	Current health of the player.	16
playerPosition	Vector2D	Starting position of the player on the left of the map.	(0,0)
score	Integer	Player's current score, displayed in the top right.	100
timer	Float	Countdown timer displayed at the top middle.	600s
ballDeVoux	Object	Enemy with moderate health, damages player on contact.	N/A
ballDeVouxHealth	Integer	Enemy health for Ball De Voux.	8
ballDeVouxDamage	Integer	Damage inflicted on contact with player.	3
spikey	Object	Enemy with low health, damages player on contact.	N/A
spikeyHealth	Integer	Enemy health for Spikey.	3
spikeyDamage	Integer	Damage inflicted on contact with player.	2
copter	Object	Flying enemy that chases player.	N/A
copterHealth	Integer	Enemy health for Copter.	3
copterDamage	Integer	Damage inflicted on contact with player.	2
voltGun	Object	High-health enemy that shoots missiles.	N/A
voltGunHealth	Integer	Enemy health for VoltGun.	15
voltGunDamage	Integer	Missile damage inflicted by VoltGun.	4
smallCapsule	Object	Small energy capsule that restores health.	N/A
smallCapsuleHealing	Integer	Health restored by small capsule.	2
largeCapsule	Object	Large energy capsule that restores more health.	N/A
largeCapsuleHealing	Integer	Health restored by large capsule.	8
floorHole	Object	Dangerous floor hole that causes instant death.	N/A
floorHoleEffect	String	Causes player death upon fall.	"Death"
boss	Object	Final boss enemy positioned at the end of the level.	N/A
bossHealth	Integer	Health of the boss.	25
bossAttackType	String	Boss's main attack type.	"DashingPunch"
healthBar	Object	Player's health bar UI element from 0 to max health.	N/A
scoreDisplay	Object	Display for the player's score in the top right.	N/A
timerDisplay	Object	Countdown timer in the top middle.	N/A
checkpoint	Object	Player's save and exit points.	N/A
victoryCondition	Boolean	Player defeats the boss to win the level.	true, false
defeatCondition	Boolean	Player dies at any point, including falling or health = 0.	true, false
collisionBoundary	Object	Defines collision area for damage and healing.	N/A

Test Case	pass/fail
SaveScoreCheck	
UserAdminCheck	
UserCreation	
UserLogin	

Table 2: TimeAndScoreController Unit Test

Test Case	pass/fail
AddTimeBonusAndSaveTest	
SaveScoreIntegrationTest	

Table 3: DatabaseController Unit Test

Test Case	pass/fail
SaveScoreCheck	pass
UserAdminCheck	pass
UserCreation	pass
UserLogin	pass

Table 4: TimeAndScoreController Unit Test

Test Case	pass/fail
AddTimeBonusAndSaveTest	pass
SaveScoreIntegrationTest	pass

Table 5: DatabaseController Unit Test

Test Case	pass/fail
AddTimeBonusAndSaveTest	
SaveScoreIntegrationTest	

Table 6: Integration Template

Test Case	pass/fail
AddTimeBonusAndSaveTest	Pass
SaveScoreIntegrationTest	Pass

Table 7: Integration Result

Test Case	pass/fail
Game Build	

Table 8: Boot Test Template

Test Case	**Result**
Title Page	
Introduction	
Project Overview	
Project Scope	
Functional Requirements	
Non-Functional Requirements	
Target Hardware Details	
Software Product Development	
System Boundaries	
Wireframes and Storyboard	
UML	
Requirements Traceability Table	
Version Control	
Data Dictionary	
User Experience Details	
Product Security	
Product Performance	
Software Testing Plan Template	
Unit Testing	
Integration Testing	
System Testing	
Acceptance Testing	
Appendix: Software Product Build Instructions	
Appendix: Software Product User Guide	

Table 9: System Testing Template for Documentation Test

Test Case	Result
Title Page	Pass
Introduction	Pass
Project Overview	Pass
Project Scope	Pass
Functional Requirements	Pass
Non-Functional Requirements	Pass
Target Hardware Details	Pass
Software Product Development	Pass
System Boundaries	Pass
Wireframes and Storyboard	Pass
UML	Pass
Requirements Traceability Table	Pass
Version Control	Pass
Data Dictionary	Pass
User Experience Details	Pass
Product Security	Pass
Product Performance	pass
Software Testing Plan Template	Pass
Unit Testing	Pass
Integration Testing	Pass
System Testing	Pass
Acceptance Testing	Pass
Appendix: Software Product Build Instructions	Pass
Appendix: Software Product User Guide	Pass

Table 10: System Testing Results for Documentation Test

Test Case	pass/fail
Game Build	Pass

Table 11: Boot Test Results

4.9 Software Testing Plan Template

Test Plan Identifier: AcceptanceTest(004) **Introduction:** AcceptanceTesting: acceptance testing is utilized to ensure that the project meets the expectations of the client.
Test item: Acceptance Testing: This testing is done manually by the client after the team creates a template.
Features to test/not to test: AcceptanceTesting: During acceptance testing we are ensuring that our project closely adheres to the scope of the project. This consists of the functional and nonfunctional requirements being met.
Approach: AcceptanceTest: This testing is manual and must be performed by the client, the client must go line by line.
Test deliverables: The deliverables for all of our tests is whether the test is able to pass or fail criteria. This is important in each of the tests as this is the overall goal of all tests.
Item pass/fail criteria: AcceptanceTest: In this testing the pass or fail criteria is based on whether the different functional and nonfunctional requirements are met.
Environmental needs: Acceptance Testing: The only infrastructure needed is a complete project and the acceptance test template in order to complete testing.
Responsibilities: AcceptanceTesting: The main responsibility of acceptance testing is to ensure that the project meets the given requirements.
Staffing and training needs: Team members had to watch youtube videos along with using internet resources in order to gather a strong understanding to implement each of the test cases into the project.
Schedule: Implementing each of the test cases took quite a large amount of time. This was mainly due to trying to understand how creating test cases inside of unity actually works as well as developing test cases that properly test.
Risks and Mitigation: The main risks that our team ran into when it came to implementing testing was a lack of technical knowledge on how testing can be conducted in unity. We were able to make a risk mitigation plan in order to ensure that this issue did not affect us super heavily by allotting our members designated amount of time in which they can research and come to understand how this testing can work. For example watching youtube tutorials on how to set up a unit test inside of unity.
Approvals: Ryan Rutledge, Kaden Groves, A.R. Chessor, Eli Atkisson 11/28/2024

Acceptance Testing

Tests	Pass/Fail
Should allow the main character to be controlled by the keyboard.	
Should allow multiple players to create separate accounts.	
Should implement a login system for users and administrators.	
Should save the state progress of players, utilizing checkpoints to retain the save state.	
Should display a splash screen when launched which also shows top scores and the login prompt.	
Should implement basic enemy logic scripts.	
Should allow the Fire function to be held indefinitely.	
Should allow different functionality after the Fire function has been held for a certain amount of time.	

Should retain stable frame rate and process user actions swiftly	

The game should be able to handle multiple enemies on the screen at a time	
HUD should actively update with current changes in the game.	
Input validation must ensure that only appropriate data is processed by the application.	
The system should implement effective session management to handle user sessions securely and efficiently.	
Access control mechanisms should restrict unauthorized users from accessing sensitive areas of the application.	
Error handling and logging must be included to capture and manage unexpected events or failures.	
Data protection measures should safeguard user data, including encryption of sensitive information.	
Database security protocols must prevent unauthorized access and ensure data integrity in storage and retrieval.	
General coding practices should be followed to improve code maintainability, readability, and performance.	
Password management and authentication protocols must secure user accounts and protect against unauthorized access.	
Do the Charged Shots work appropriately	
Does the energy capsules use RNG to spawn on enemy death	
Does falling off the map send the player to defeat screen.	

Figure 30: Acceptance Testing One Template2

Acceptance Testing

Tests	Pass/Fail
Should allow the main character to be controlled by the keyboard.	Pass
Should allow multiple players to create separate accounts.	Pass
Should implement a login system for users and administrators.	Pass
Should save the state progress of players, utilizing checkpoints to retain the save state.	Pass
Should display a splash screen when launched which also shows top scores and the login prompt.	Pass
Should implement basic enemy logic scripts.	Pass
Should allow the Fire function to be held indefinitely.	Pass
Should allow different functionality after the Fire function has been held for a certain amount of time.	Pass
Should allow player to take damage when hit by enemy	Pass
Should be implemented using the Unity game engine and development environment.	Pass
Should create a close replica of the game Mega Man X.	Pass
Can use pre-developed images, artwork, audio, and other related media.	Pass
Must utilize a database	Pass
Performance metrics should be gathered and optimized	
Security metrics should be gathered and optimized.	
User interface metrics should be gathered and optimized.	
Must be executable on the Windows OS.	Pass

Figure 31: Acceptance testing results page 1

Should retain stable frame rate and process user actions swiftly	Pass

The game should be able to handle multiple enemies on the screen at a time	Pass
HUD should actively update with current changes in the game.	Pass
Input validation must ensure that only appropriate data is processed by the application.	
The system should implement effective session management to handle user sessions securely and efficiently.	
Access control mechanisms should restrict unauthorized users from accessing sensitive areas of the application.	
Error handling and logging must be included to capture and manage unexpected events or failures.	
Data protection measures should safeguard user data, including encryption of sensitive information.	
Database security protocols must prevent unauthorized access and ensure data integrity in storage and retrieval.	
General coding practices should be followed to improve code maintainability, readability, and performance.	
Password management and authentication protocols must secure user accounts and protect against unauthorized access.	
Do the Charged Shots work appropriately	Pass
Does the energy capsules use RNG to spawn on enemy death	Pass
Does falling off the map send the player to defeat screen.	Pass

Figure 32: Acceptance testing results page 2