Rampage Technical Document

Sprint 4
12/1/2023

Name	**Email Address**
Andrea Florence	andrea.florence180@topper.wku.edu
Tameka Ferguson	tameka.ferguson183@topper.wku.edu
Shikha Sawant	shikha.sawant059@topper.wku.edu
Aaron Beasley	aaron.beasley908@topper.wku.edu

CS 360
Fall 2023
Dr. Michael Galloway
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 . . . 2
- 1.4 Target Hardware Details . . . 3
- 1.5 Software Product Development . . . 4

2 Modeling and Design **4**
- 2.1 System Boundaries . . . 4
  - 2.1.1 Physical . . . 4
  - 2.1.2 Logical . . . 4
- 2.2 Wireframes and Storyboard . . . 5
- 2.3 UML . . . 5
  - 2.3.1 Class Diagrams . . . 5
  - 2.3.2 Use Case Diagrams . . . 5
  - 2.3.3 Use Case Scenarios Developed from Use Case Diagrams (Primary, Secondary) . . . 6
  - 2.3.4 Sequence Diagrams . . . 6
  - 2.3.5 State Diagrams . . . 8
  - 2.3.6 Component Diagrams . . . 8
  - 2.3.7 Deployment Diagrams . . . 8
- 2.4 Traceability Table . . . 9
- 2.5 Version Control . . . 9
- 2.6 Data Dictionary . . . 12
- 2.7 User Experience . . . 13
  - 2.7.1 Gameplay Diagram . . . 13
  - 2.7.2 Gameplay Objectives . . . 13
  - 2.7.3 User Skillset . . . 13
  - 2.7.4 Gameplay mechanics . . . 14
  - 2.7.5 Gameplay Items . . . 15
  - 2.7.6 Gameplay Challenges . . . 15
  - 2.7.7 Gameplay Menu Screens . . . 15
  - 2.7.8 Gameplay Heads-Up Display . . . 16
  - 2.7.9 Gameplay Art Style . . . 17
  - 2.7.10 Gameplay Audio . . . 17

3 Non-Functional Product Details **18**
- 3.1 Product Security . . . 18
  - 3.1.1 Approach to Security in all Process Steps . . . 18
  - 3.1.2 Security Threat Model . . . 20
  - 3.1.3 Security Levels . . . 21
- 3.2 Product Performance . . . 22
  - 3.2.1 Product Performance Requirements . . . 22
  - 3.2.2 Measurable Performance Objectives . . . 22
  - 3.2.3 Application Workload . . . 22
  - 3.2.4 Hardware and Software Bottlenecks . . . 23
  - 3.2.5 Synthetic Performance Benchmarks . . . 23
  - 3.2.6 Performance Tests . . . 23

4 Software Testing **24**
4.1 Software Testing Plan Template . . . 24
4.2 Unit Testing . . . 24
4.2.1 Source Code Coverage Tests . . . 24
4.2.2 Unit Tests and Results . . . 24
4.3 Integration Testing . . . 25
4.3.1 Integration Tests and Results . . . 25
4.3.2 Integration Tests and Results . . . 26
4.4 System Testing . . . 26
4.4.1 System Tests and Results . . . 27
4.5 Acceptance Testing . . . 27
4.5.1 Acceptance Tests and Results . . . 27

5 Conclusion **27**

6 Appendix **27**
6.1 Software Product Build Instructions . . . 27
6.2 Software Product User Guide . . . 27
6.3 Source Code with Comments . . . 27
6.3.1 ChangeInputs.cs . . . 27
6.3.2 PlayFabManager.cs . . . 28
6.3.3 OpenGame.cs . . . 32
6.3.4 PlayerMovement.cs . . . 32
6.3.5 HealthBarManager.cs . . . 34
6.3.6 ScoreManager.cs . . . 35

List of Figures

1 Image of wireframe. . . . 5
2 Image of storyboard. . . . 6
3 Prototype diagram showing how most of our . . . 7
4 Observer diagram shows . . . 7
5 Façade diagram shows the relationship between the player and enemies. . . . 8
6 Use case diagram showing the player interactions with the game. . . . 9
7 Use Case Scenario 1 . . . 10
8 Use Case Scenario 2 . . . 10
9 Sequence diagram on the overall software function. . . . 11
10 State diagram on the overall game state. . . . 12
11 Component diagram showing the how aspects of our game feeds into Unity and interfaces. . . . 13
12 Deployment diagram showing how our game is deployed. . . . 14
13 Requirements Traceability Table. . . . 15
14 Data dictionary table game objects. . . . 16
15 Data dictionary table collisions. . . . 17
16 Data dictionary table game mechanics. . . . 17
17 Data dictionary table user interface. . . . 18
18 Data dictionary table audio. . . . 19
19 Diagram of Rampage Gameplay. . . . 20
20 Image of Register Screen. . . . 21
21 Image of Login Screen. . . . 21
22 Image of Pause and Main Menus. . . . 37
23 Images of the wireframes for main menu and gameplay. . . . 38
24 Model showing trust boundaries and security risks. . . . 38
25 Graph depicting the number of threads vs execution time. . . . 39
26 Graph depicting file size vs throughput. . . . 39
27 Integration testing visual. . . . 40
28 System testing table results. . . . 40

1 Introduction

1.1 Project Overview

The main goal of this project is to recreate one level of the game Rampage as close to the original as possible. We must create the game with accurate sprites which we have found some through Spriters Resource. The background would be more difficult than we thought to find because there aren't any background sprites for the game on the internet that match the art style. The mechanics of the gameplay appear easier to implement. The sound effects of the original game are also hard to find. With limited resources on the table, we are prepared to do whatever it takes to find the best substitutes to make our remake close to the original.

Our client and audience, Dr. Galloway also wants us to document whatever we have done in our project so he can be sure that everyone in our group does our part. All members are expected to use Unity because there was no other platform that we could use to create games. We are also tasked with creating a Gantt chart to stay organized and document the times that we spent working on our project. This project provides users with a game that challenges hand-eye coordination. Players must be able to think and move quickly to avoid being shot by soldiers as they destroy structures with a punching mechanic, while keeping their health up. Another thing is the character would be able to climb buildings and will react when being shot by soldiers. The game will be made in Unity and shared with other members of the team. We also intend to add a sign in/sign up system for other players who want to play our game. There will also be a character selection scene so you can play as one of the three characters from the original game. We are also making sure that the character does not have the power to cheat the game like actual console games do. Overall, there are things that we must do to keep this project in motion for all members.

1.2 Project Scope

The scope of this project includes the writing of two written reports, the technical and organizational documents, a presentation, and a CATME evaluation. The technical document includes the project overview, scope, requirements, model and design, non-functional details, and testing of software. The organizational document includes the team's organizational approach, task schedule, team's known progress in project, and managing possible risks. The presentation focuses on giving the client an overview of project, progress on project and future plans. The CATME evaluation lets client know how each team member contributes to project, how team members interact with each other, if the team members check on each other's progress, and overall interactions with the project. The benefit for this project is to acquire skills pertaining to software engineering. This entails learning how to analyze requirements, determining the needs for a desired project outcome, user interface, design, construction, testing and maintenance. Along with this, we get to do this with a team who is also learning as the project progresses. As a team, we expect that the outcome for this project would be that we are able to successfully remake a level of Rampage as close to the original game as possible.

The tasks for the project would be completing the technical and organizational documentations, the presentations, evaluations and completion of the final product. With documents, each team member will be given a specific part of the document that they would have to contribute to. The presentation will be created based on the technical and organizational documents that we have done. The evaluations will be done once preparations for the presentation have been made. The development of the final product will start in Sprint 3.

There are no costs for the project other than time cost for each team member. We will try to meet every week to discuss future plans and evaluate our current work. And the time costs are dependent on the arranging of the team members schedules. Our team consists of Aaron, Andrea, Shikha, and Tameka for this project. Unity tutorials and resources are the necessary resources for this project.

Sprint 1 documentation is due on September 7th. The first evaluation and presentation are due on September 4th. Sprint 2 documentation is due on October 6th. The presentation and CATME evaluations are due on October 4th. Sprint 3 documentation is due on November 3rd, and the CATME evaluation is due that day as well. The presentation is due on October 30th.

Mandatory Functional Requirements
1. Player Controls
a. Move left/right? Jump? Climb Buildings? Punch?
b. Accessing a menu 2. Injury
a. How much can the health lower with each hit?
3. Player Accounts
a. How will the players' data be saved? Logging In?
b. Multiple accounts?
4. NPCS/Vehicles
a. Dying from player
b. Inflicting Damage
c. Their movement and design
5. Building
a. Take damage
b. Hold NPCS
Extended Functional Requirements
1. Injury a. Programming inflictions
2. Visuals
a. Aesthetics
3. Environment
a. Environment size
b. Building size
c. What moves/doesn't move
4. Graphics
a. Color scheme

1.3 Technical Requirements

1.3.1 Functional Requirements

The mandatory functional requirements are needed to actually make a game that has all the bare minimum requirements. The requirements we have listed enable our game to have an objective that makes it unique compared to any other game. In order for the players' character to interact with the game, it needs to be able to move left and right, as well as scale buildings. We had to ask the questions "How will the player move and climb buildings, and destroy them? What controls can we assign to the computer to do these specific functions?" We might use the AWSD keys or the arrow key for the player to move, and one of the letter keys to be the punching functions. Another key can be the option for the menu, in case the player needs help with playing the game. Another function that is figuring out the player can die. The question we have to ask is "How much damage can they take before dying?" along with this, another functional requirement is managing accounts. We had to ask "How many accounts can be made for our game? Will this be a single player or multiplayer?" These are the bare minimum functions that are mandatory to successfully make a functioning game.

1.3.2 Non-Functional Requirements

The mandatory non-functional requirements are the basic requirements that we needed to consider outside of the actual game functionality itself. These pertain to the baseline characteristics on how and where the game will be built and stored upon, as well as other things. It also pertains to different performance times such as response times and loading times. The purpose of studying these requirements is to ensure the performance times are reasonable enough for the enjoyment of the player. Different storages are also types of nonfunctional requirements

Mandatory Non-Functional Requirements
1. Implementation
a. Using Unity
b. Windows software
2. Security
a. Ensure player cannot cheat
3. Metrics
a. Hardware metrics TBD
4. User Interface
a. Buttons will be the main controls
Extended Non-Functional Requirements
1. Response Time
a. Their movement and design
2. Loading Time
3. Compatibility and Accessibility
4. CPU/Memory Usage
5. Network Performance
a. Assuming this game would be accessed online
6. Storage
a. Progress storage
b. Chances of lagging?
7. Bugs
a. Debugging

we have to consider given it shouldn't be too large for our client's computer or ours. It will be developed on the software called Unity since Unity is great for developing games, and we also can specify that the platform we will be using is Windows, since that is the resources we are limited to. Much of the nonfunctional requirements are to be determined since this sprint was mainly focusing on familiarizing ourselves with the team and familiarizing ourselves with our objective, and the nonfunctional requirements are so detailed.

1.4 Target Hardware Details

Since the first sprint was mainly about familiarizing ourselves with the project and getting a basic understanding of our task at hand, we haven't really been able to look into detail about what the target hardware would be. We are going to be basing our target hardware on the assumption that this game will be accessed on Unity, so the hardware of the device that would access the game would need to be able to run Unity. Meanwhile all of our team members are using windows to access Unity, we based the details on this.

For our development team, the necessary operating system (OS) needed to run our game would be Windows 7 and newer, 64-bit versions only. The CPU would be x64 architecture with SSE2 instruction set support. The Graphics API would be DX10, DX11, and DX12 – capable GPUs. Additional requirements would be that the hardware vendor officially supported drivers. For all operating systems, the Unity Editor is supported on workstations or laptop form factors, running without emulation, container or compatibility layer.

For players the system requirements can be different depending on what software they intend to play the game on. For Android, the version would have to be 4.4(API 19)+, CPU ARMv7 with Neon Support (32-bit) or ARM64, and Graphics API would be OpenGL ES 2.0+, OpenGL ES 3.0+, Vulkan. For iOS, the version would have to be 11+, CPU A7 SoC+, and Graphics API would be Metal.

The necessary RAM needed for most users when using Unity would be 32GB and 64GB+ for those interested

in building lighting which takes more than a few hours. The persistent storage, network connection and network bandwidth are still to be determined. Output devices would be speakers for the in-game sound. Input devices would only be the keyboard to determine the movements and actions of the playable character.

1.5 Software Product Development

The IDE we are considering using for the project is Visual Studio. Since we are using Unity the default script editor is Visual Studio. All of our team members are also already somewhat familiar with Visual Studio, so it would work best for us to continue with something we know.

Another option would be to use an IDE plugin that has Unity support built in. We may consider JetBrains Rider which is a C-sharp editor. JetBrains Rider makes writing C-sharp a lot smarter. Though, none of our members have experience in C-sharp, JetBrains Rider would be very helpful in enhancing our skills and act as a good IDE plugin. The software language that we will be using is C-sharp since this is the language that Unity can understand. We have not decided on a software framework and its necessity. Our version control will be Git. Not all of our members are familiar with Git or GitHub either so this will all be a learning experience. We also have not yet discussed what asset creation tools we will be using for our project.

We use Discord to communicate and share documents for Sprint 1. Discord is a very convenient social platform that makes both connecting and conveying information between our team members more efficient. As we move onward into the project and begin using Unity, we are considering using GitHub for version control to ensure that all team members are aware of changes in our game development. We will commit changes to GitHub as we are editing the Unity file so that we all have the latest version of our software. If GitHub proves to be too difficult for us to understand or operate, we will try to share documents and track our project progress using either Google Drive or One Drive.

2 Modeling and Design

2.1 System Boundaries

2.1.1 Physical

The hardware components and resources that will be used for the software system would be the operating system, the CPU, the graphics API and disk space. For android the OS version would be 4.4 (API 19) + and for iOS 11+. For android the CPU would be ARMv7 with Neon Support (32-bit) or ARM64 and for iOS A7 Soc+. The graphics API would be OpenGL ES 2.0+, OpenGL ES 3.0+, Vulkan for android, and Metal for iOS. Then for both android and iOS, the user would need 80GB of free disk space. For the deployment environment which our software will be deployed on would be the user's PC. Since our project is small, we would like to keep everything contained to the user's device. For network/communication, if we were branching out to have different servers later on we would use latency optimization to minimize network latency and ensure smooth gameplay for users. For security, we would have authentication and authorization. With this we would create a login system that uses a database that ensures that users are authorized to play the game. We won't be focusing too much on scalability for this project as we won't have many servers for our game.

2.1.2 Logical

We had to take into consideration aspects within the scope of the logical system boundary. Some things within the logical system boundary include but are not limited to game elements, game logic, level design/terrain and state entities. The game elements we had to take into consideration were the main game elements such as the player character(which are the monsters), the buildings, and the NPCs. In terms of the game logic we had to mainly take consideration of the rules and regulations of the game, as well as the mechanisms and the algorithms. For the level design and terrain, we need to focus on how we want our game environment set up; As a group we have to think about how we want to set up the street, background, and structure of the environment as a whole. Positions and health scores are what we have to look at for the state entities of not only the player but the NPCs as well. Some things out of the logical system boundary scope are things such as user input, audio, and operating system and hardware.

2.2 Wireframes and Storyboard

From the wireframe, we have one for our start menu as well as for the main playing scene. On the start menu, you can either start the game or view the leaderboard. Instead of on-screen buttons we have implemented where you can click 'F' or 'L' on your keyboard to start game or view leaderbaord. On the main playing scene, there are outlines for the buildings we implemented as well as the HUD. Unfortunately, we were unable to implement this design for our HUD due to numerous issues with Unity and time constraints, so we ended with a much more simple HUD design. Also we settled for three buildings rather than four.

From the storyboard we created, there are 4 different scences. First, there is a login scene where a user can login with an existing account and a separate register scene where a new user can create an account. From there, they meet a start menu with two buttons, start and view scores. When teh start is clicked, the user is thrown into the playing scene where they lose if they lost all health or win if they destroy all of the buildings.

Due to unity issues, we merged our login and register functions within one singular scene. We also simplified our playing scene which had a simpler HUD function and different character and npcs functionality.

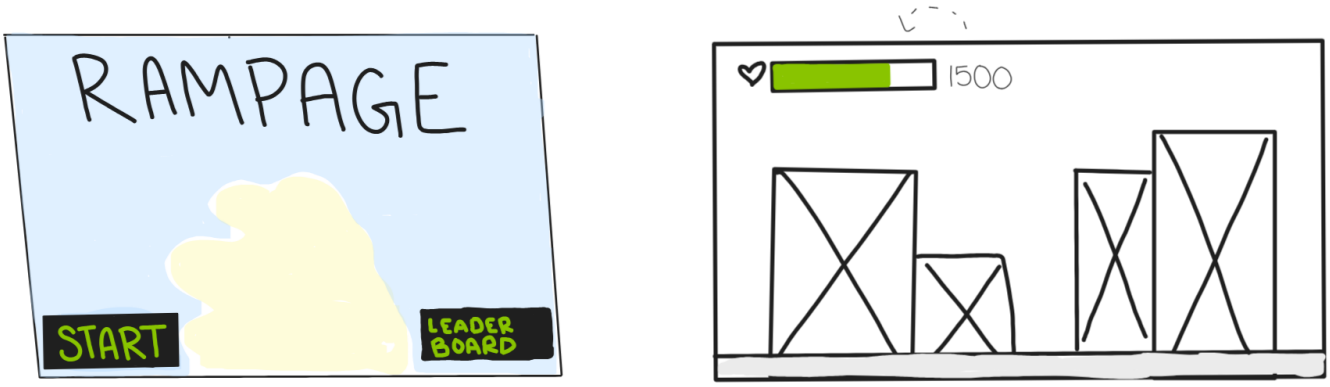


Figure 1: Image of wireframe.

2.3 UML

2.3.1 Class Diagrams

Enemy Prototype Diagram: The enemies/NPCs have multiple character types like walkers, helicopters, and tanks. The most common function walkers and helicopters have is the ability to fire at the player with all enemies having the ability to spawn.

Within our game, we have many NPCs that all share some forms of the same functionality including spawning and attacking the player. Using the prototype diagram helps us efficiently make the same methods that can go across the various types of NPCs. We have also included the civilian NPCs because although they do not necessarily harm the player's character, they still have some of the same functionalities as a soldier.

Game Objects Observer Diagram: When recreating this game, we have many objects that we need to monitor. We have many NPC objects and building objects that will take various state changes during the gameplay, and we need to implement algorithms to take these state changes into consideration. The Observer diagram is the best behavioral diagram we could use for our game since we need to observe the state changes of the other objects outside of the player themselves. The methods can also stop affecting any object that does not need the method implemented on them at any time.

Player and Enemy Façade Diagram: The more complex code is for both the players and the enemies. Scripts for the player and the enemies are made from even smaller codes behind a much bigger interface.

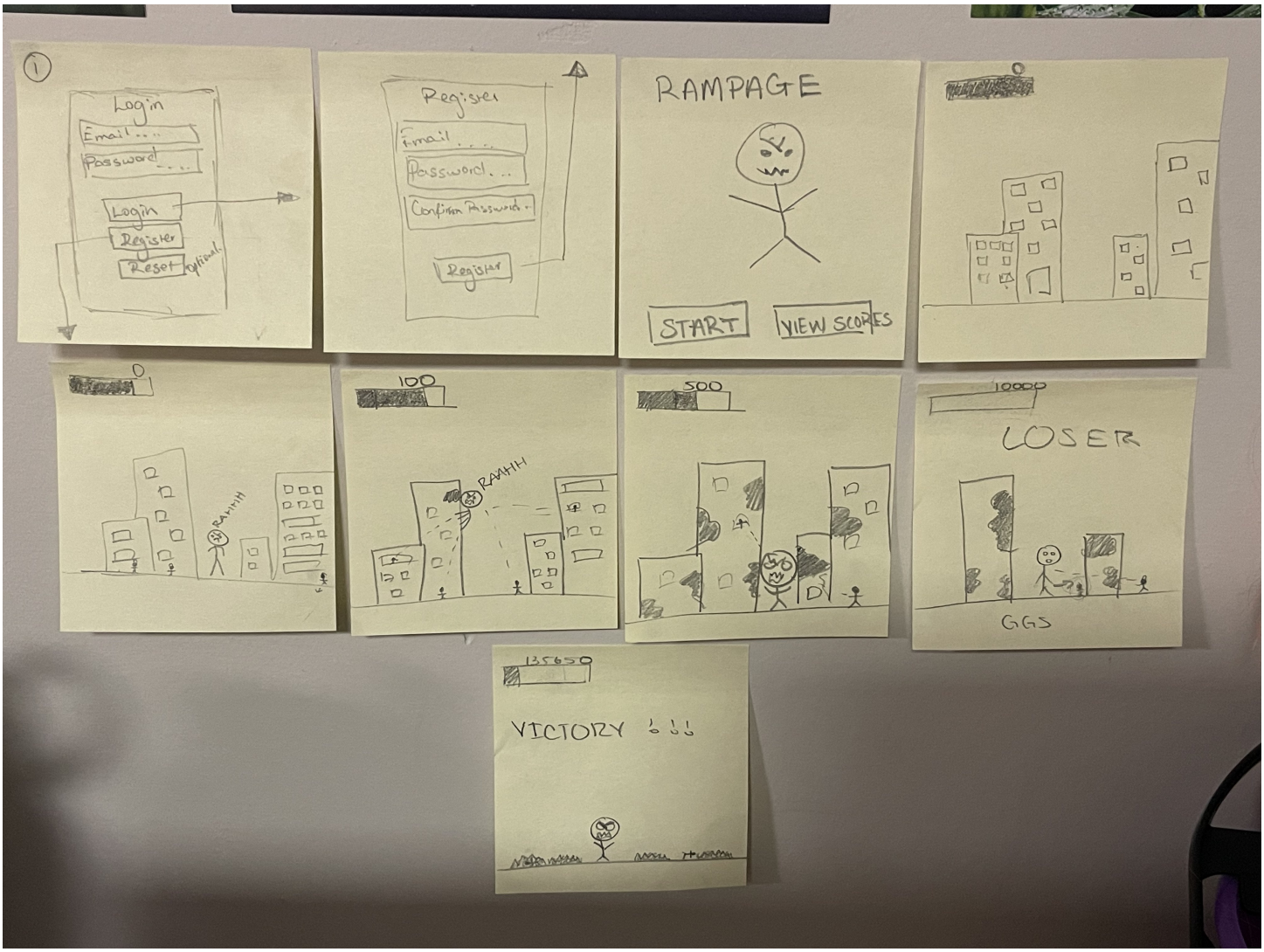


Figure 2: Image of storyboard.

We chose to sketch out the façade diagram since we are going to be recreating only one level of Rampage. Our diagram depicts our façade class holding the startGame() method, and it illustrates how the other methods and objects interact with it. Overall, our Facade diagram promotes loose coupling between the player and the system by providing a simplified interface of the gameplay. It helps manage complexity and improves maintainability in large systems.

2.3.2 Use Case Diagrams

Title: Player Activity Use Case Diagram

The user is the player and when the game is booted up, they have two choices when on startup: to play or exit the game. When the player exits, it exits the game, but when you play the game, you must create a sign in, and you can choose one of three characters to play as. Once selected, you can use the keyboard to move and punch buildings. We decided on this layout because it was easier to describe what the user can do while playing the game.

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

Primary Use Case Scenario: The user starts the game by selecting the “play game” option from the menu. The player can potentially select their character they can play as. The player then clicks advance, and the environment loads, including the buildings and NPCs. The player then gains control of the monster and they start moving around the environment using the keyboard. By doing so, they destroy buildings and gain points. The player also

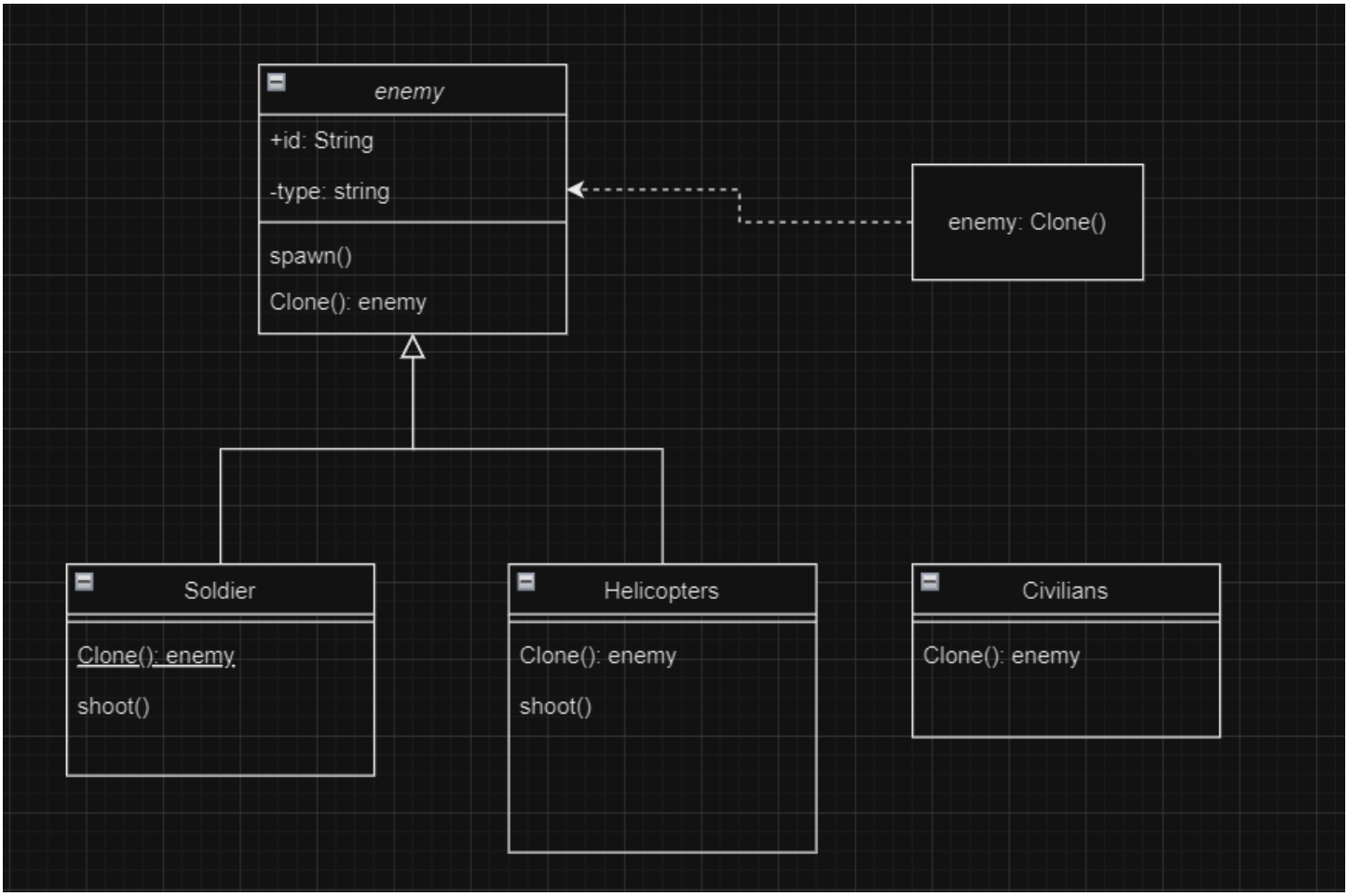


Figure 3: Prototype diagram showing how most of our

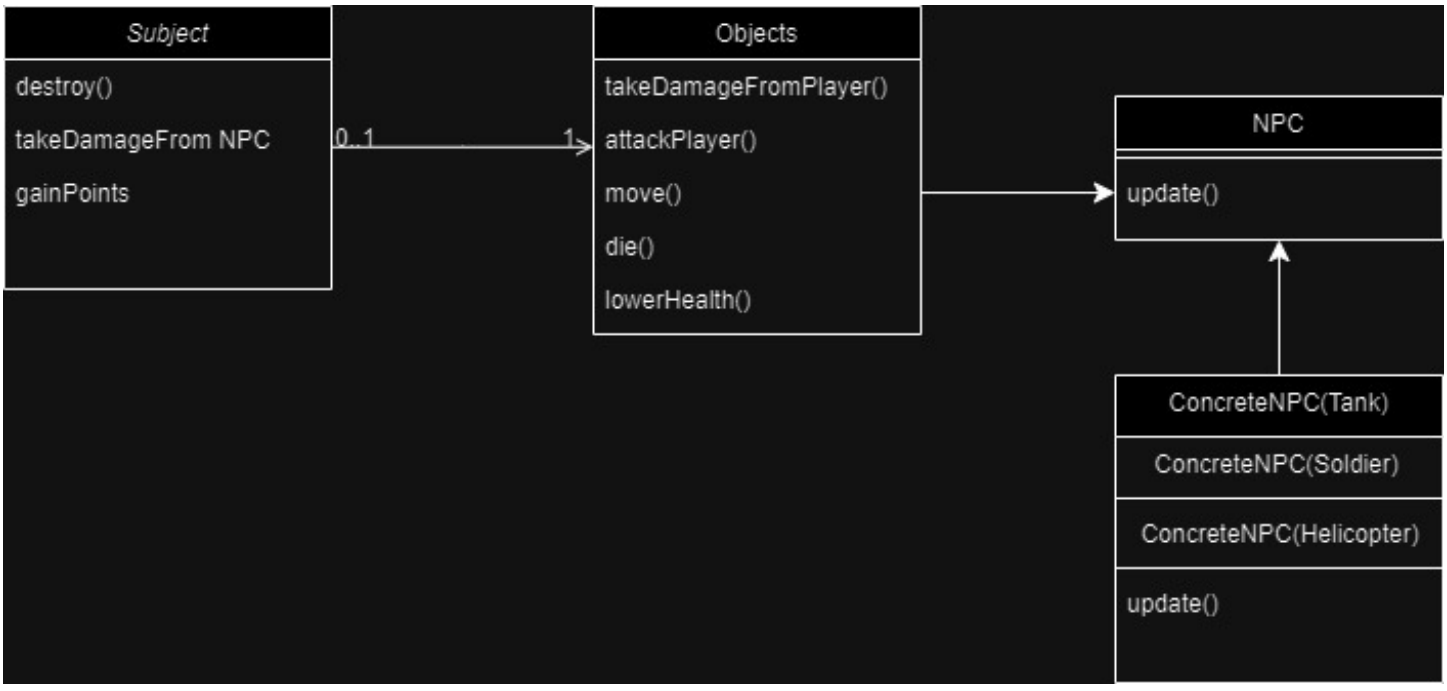


Figure 4: Observer diagram shows

much avoid the soldiers, tanks and helicopters in order to live and destroy, thus collecting points. If the player completes the objective of destroying all the buildings, they win. If they don't complete it by the time their health

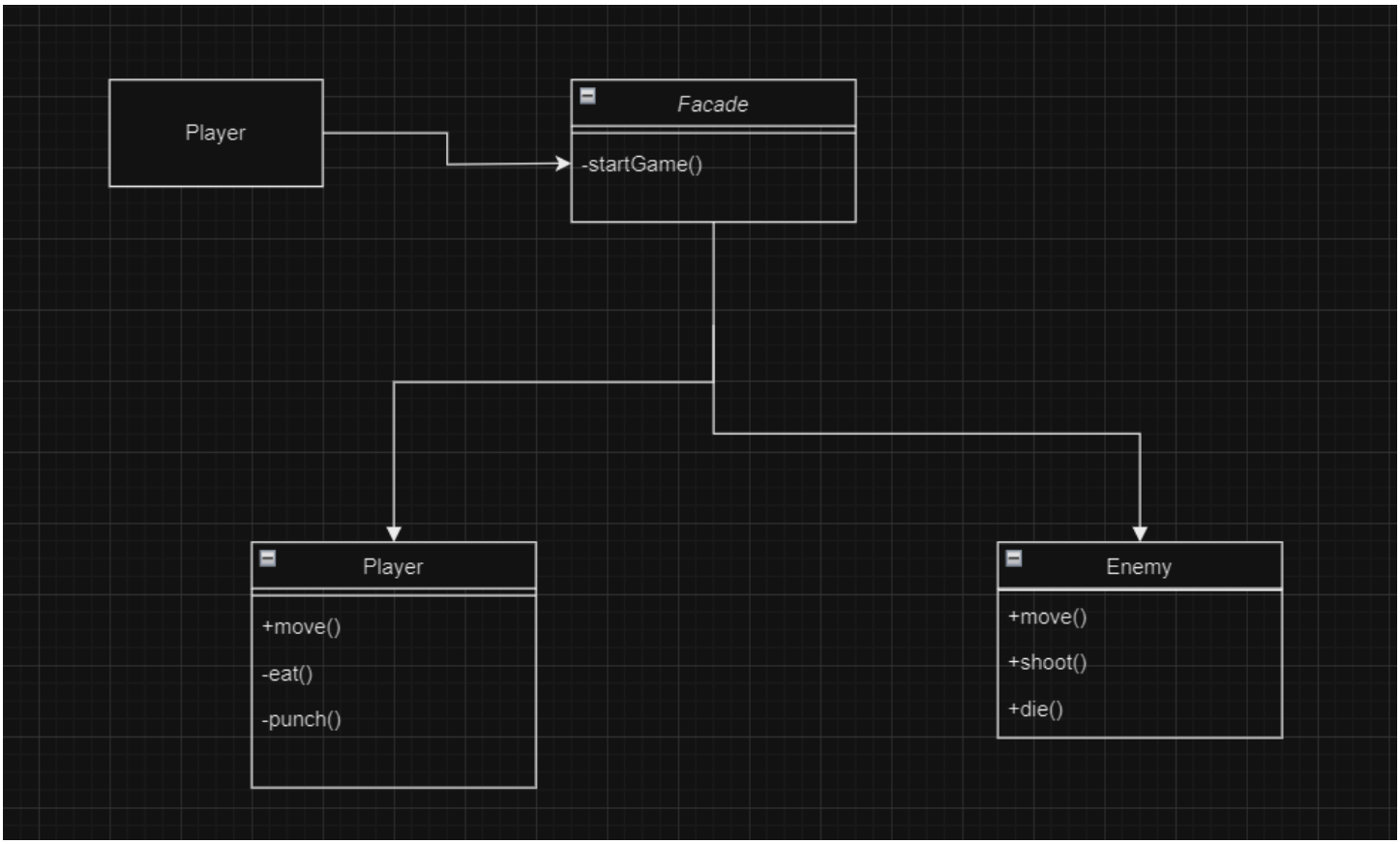


Figure 5: Façade diagram shows the relationship between the player and enemies.

runs out, they lose. After the level is complete, the score is displayed.

Secondary use case scenario: Occasionally, the player can get an alert indicating that the tanks and helicopters are approaching. The player must move the monster to avoid these new strikes. The player can also stomp on the tanks and punch the helicopters, destroying them and collecting points as well. After evading these attacks, the player resumes their task of destroying buildings.

2.3.4 Sequence Diagrams

Title: Game System Interaction Sequence Diagram

The diagram shows our overall game approach on interacting with the game system. First, once the user opens the game, the game shows the user a login prompt as well as the top scores which it will take from the database that we will use for this project. Once the user logs in, the game system will check the database to verify that the user exists within it. If the login credentials are valid, the database notifies the game system and the user is able to start playing the game. Within the gameplay, the enemy npcs will target the user and the user can kill npcs and destroy buildings. Once all buildings are destroyed, the game notifies the user that the level was completed and shows their score. The user cam then save those scores, which will be saved to the database.

2.3.5 State Diagrams

Title: Game System State Diagram

When the user starts the game, the game is in the starting state. When the user hits pause, the game goes into a paused state. When the user presses resume, the game goes back to its starting state. If the player dies or completes the level, the game goes into an end state and the player has the option of restarting the level which would put the game back into its starting state.

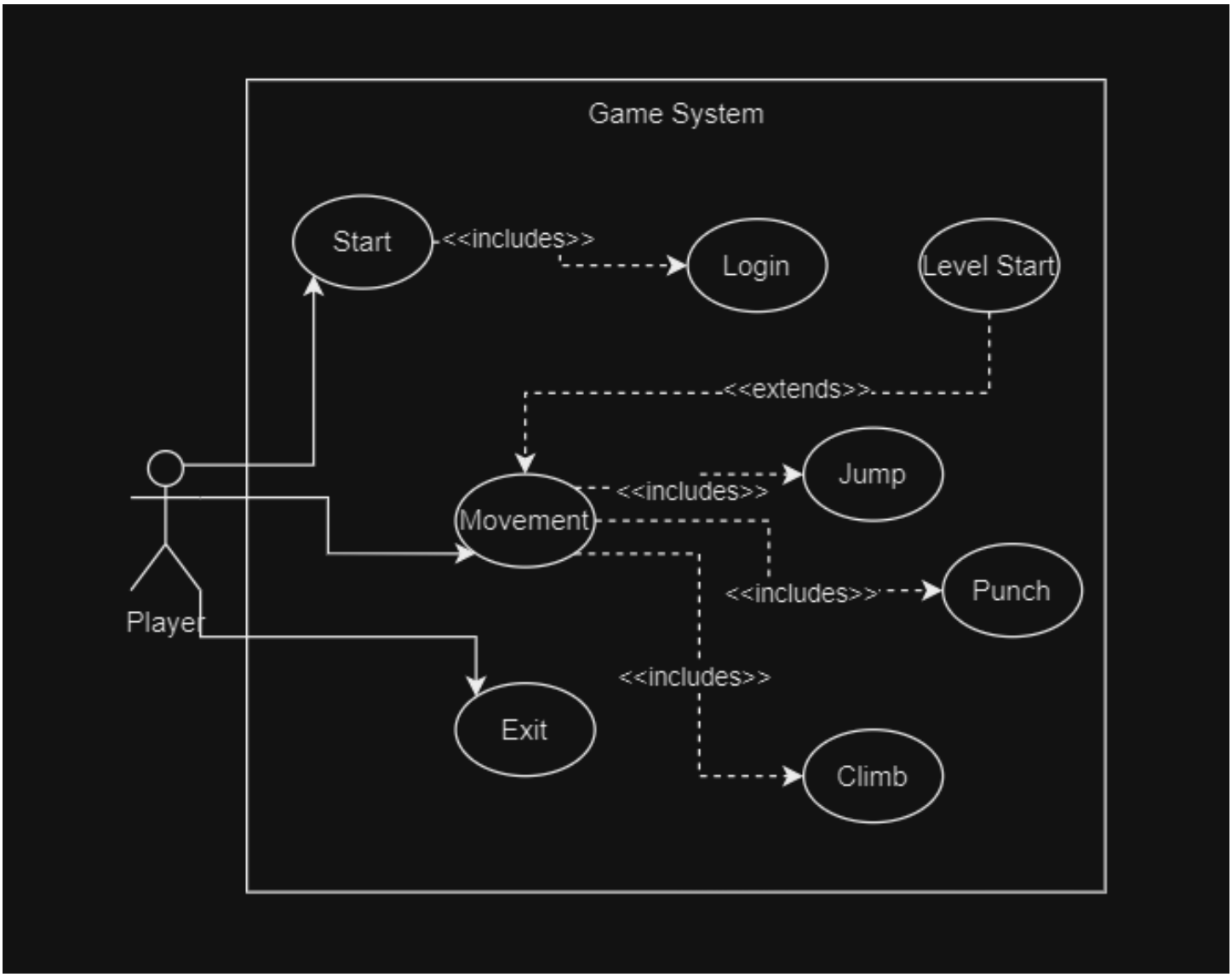


Figure 6: Use case diagram showing the player interactions with the game.

2.3.6 Component Diagrams

Title: Game Aspects Component Diagram

This diagram shows that each aspect of our game feeds into Unity and its interface(s). The game tools, which includes key controls and pause menu screens; the game contents, such as audio, graphics, settings, and the network; then the game utility, which contains the logging system and math utilities and libraries. All three of those things feed into the game engine.

2.3.7 Deployment Diagrams

Title: Game File Deployment Diagram

The deployment diagram shows that the game file containing all unity files, scripts, and assets is contained within Unity - which the user will use to play the game. All of this is occurring on the user's computer/PC. There is no outside server or connection to someone else's server or computer.

2.4 Traceability Table

The following traceability table outlines the many components of our recreation of Rampage. Each row represents a specific element that is categorized by its characteristics and status in terms of development.

	A	B	C
	Type	Primary Use Case	
	Name	Rampage	
	Summary	Recreate one level of Rampage	
	Priority	High	
	Preconditions	Client must make an account	
	Postconditions	Client is logged in/Client must know this game challenges their hand-eye coordination	
	Primary Actors	Dr. Galloway	
	Secondary Actors	NPCs	
	Main Scenario	Client starts the game by selecting "Play Game" option from the menu	
		The player can potentially select their character they can play as	
		The player then clicks advance, and the environment loads, including the buildings and NPCs	
		The player then gains control of the monster and they start moving around the environment using the keyboard	
		By doing so, they destroy buildings and gain points	
		The player also must avoid the soldiers, tanks and helicopters in order to live and destroy, thus collecting points.	
		If the player completes the objective of destroying all the buildings, they win. If they don't complete it by the time their health runs out, they lose	
		After the level is complete, the score is displayed	

Figure 7: Use Case Scenario 1

	A	B	C	D
1	Type	Secondary Use Case		
2	Name	Rampage		
3	Summary	Recreate one level of Rampage		
4	Priority	Medium		
5	Preconditions	Client must make an account		
6	Postconditions	Client is logged in/Client must know this game challenges their hand-eye coordination		
7	Primary Actors	Dr. Galloway		
8	Secondary Actors	NPCs		
9	Secondary Scenario	Occasionally, the player can get an alert indicating that the tanks and helicopters are approaching.		
10		the player must move the monster to avoid these new strikes		
11		The player can also stomp on the tanks and punch the helicopters, destroying them and collecting points as well		
12		after evading these attacks, the player resumes their task of destroying buildings.		
13				
14				
15				
16				

Figure 8: Use Case Scenario 2

This table helps us track the progress of all of these aspects of the Rampage development process. It indicates which components fall under the categories of Gameplay and Aesthetics, lists any dependencies required for each component, and provides an update on their current status. As of now, all the mentioned components are in the initial phase and have not yet been started.

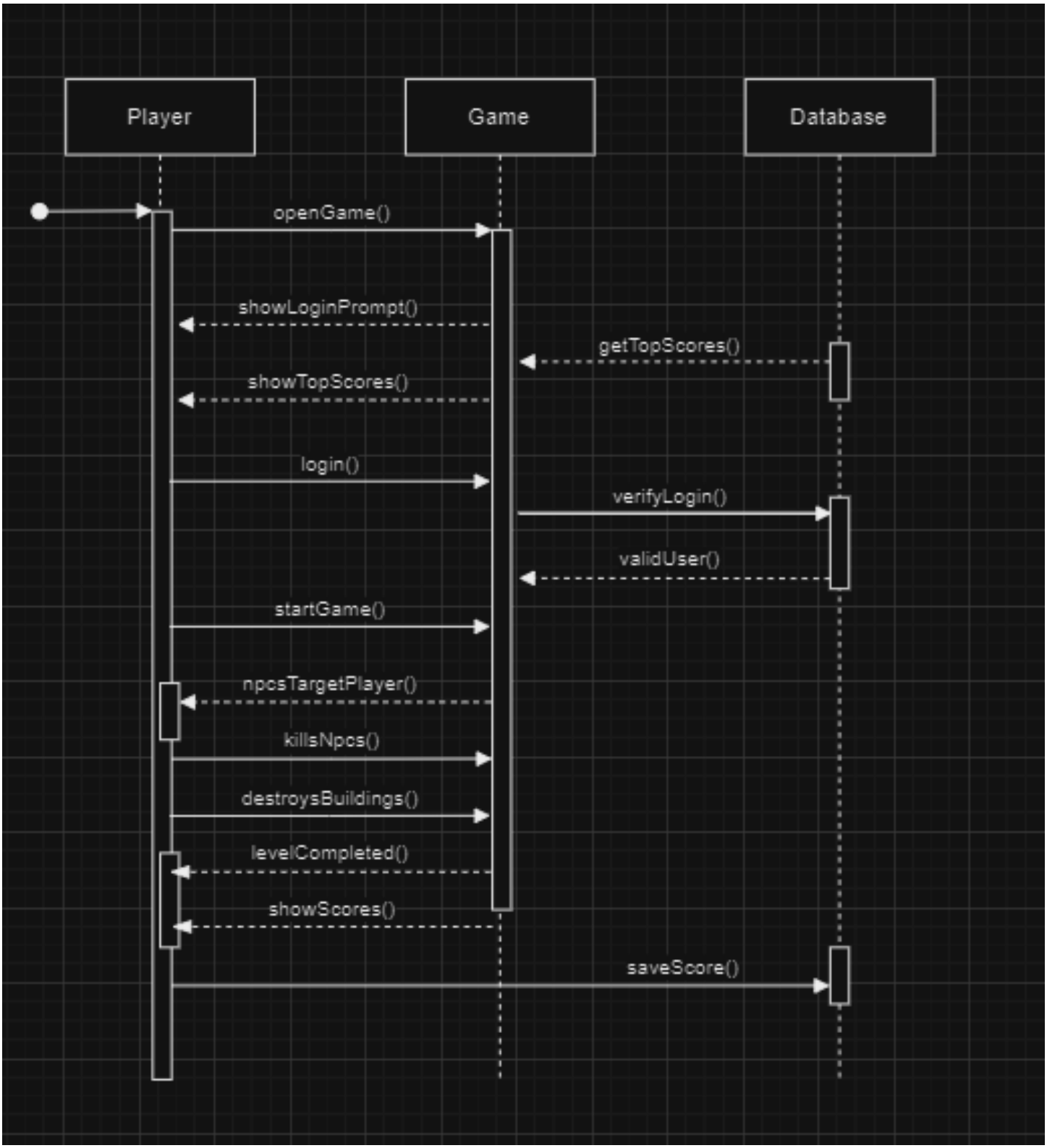


Figure 9: Sequence diagram on the overall software function.

2.5 Version Control

Version control was a big unknown when we had started this project. We were in between on using Google Drive and GitHub though we had never used GitHub before. We knew that Google Drive would have a lot of issues with syncing changes with other team members and by only supporting one member making changes at a time, so GitHub seemed like a better alternative. Though GitHub has a lot of issues of its own, we have been managing

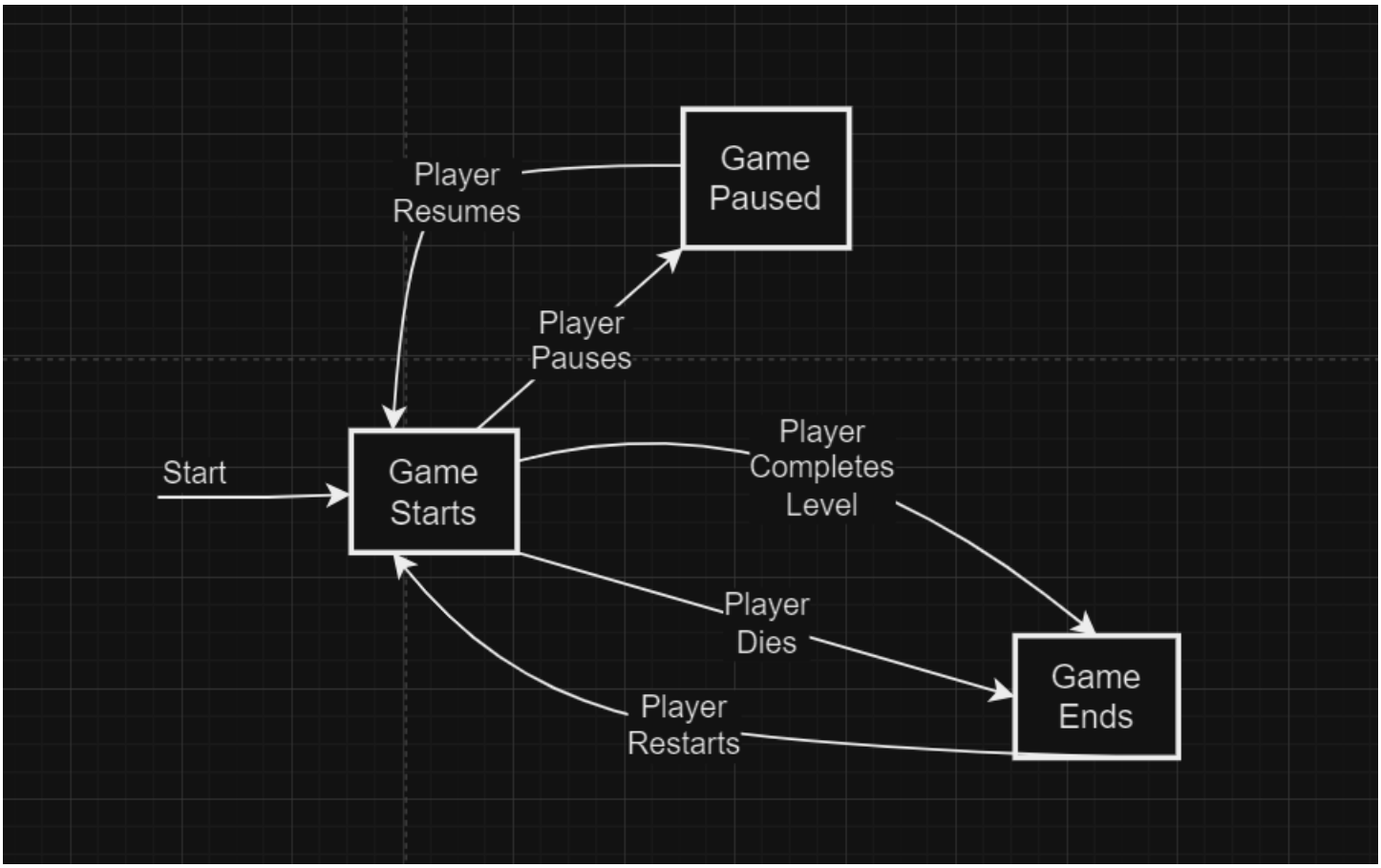


Figure 10: State diagram on the overall game state.

with it and learnt how to somewhat use it correctly. After learning the basic way of syncing GitHub with Unity from a YouTube video, the only problem we have had is that sometimes committed changes took awhile to show up on another member's laptop after pulling from origin.

We usually make commit changes to the repository after we have completed a big section of our task for the project or when we are done working on it momentarily. We name our commits based on what we are working on or have completed in those changes. We use GitHub desktop that is connected to the repository and create changes through that. At the moment, we all have been working on the same branch, the main branch, to make commits. This may have been the issue to an earlier problem that we had faced but it all sorted itself out.

2.6 Data Dictionary

The following data dictionary provides a comprehensive overview of the game elements and mechanics for the recreation of our level in the game Rampage. This table is categorized in game objects, collisions, game mechanics, user interface and audio. The monster, soldierNPC, tankNPC, planeNPC, building1, building2, building3 and background are categorized as game objects. The monster is a playable character whose role is to destroy the entire city. The soldierNPC, tankNPC and the planeNPC are the nonplayable characters whose purpose are to defend the city from the monster. building1, building2, building3 are the city buildings that the monster destroys and the city that the military are trying to protect. The background city scene's purpose is to enhance the visual experience of the game. The elements incorporated in the collision category are playerClimbBuilding, playerPunchBuilding, npcShootPlayer, and playerKillNPC. The playerClimbBuilding and playerPunchBuilding collisions define the interactions between the player and buildings using a box collider. The npcShootPlayer and the playerKillNPC collisions define the interactions between NPCs and the players using capsule colliders. The scoring rules, winning conditions and losing conditions are incorporated into the game mechanics. The scoring rules are that the players earn points for specific actions, such as eating soldiers, squishing tanks and planes, and damaging buildings. The winning conditions are to destroy all the buildings while still maintaining enough health

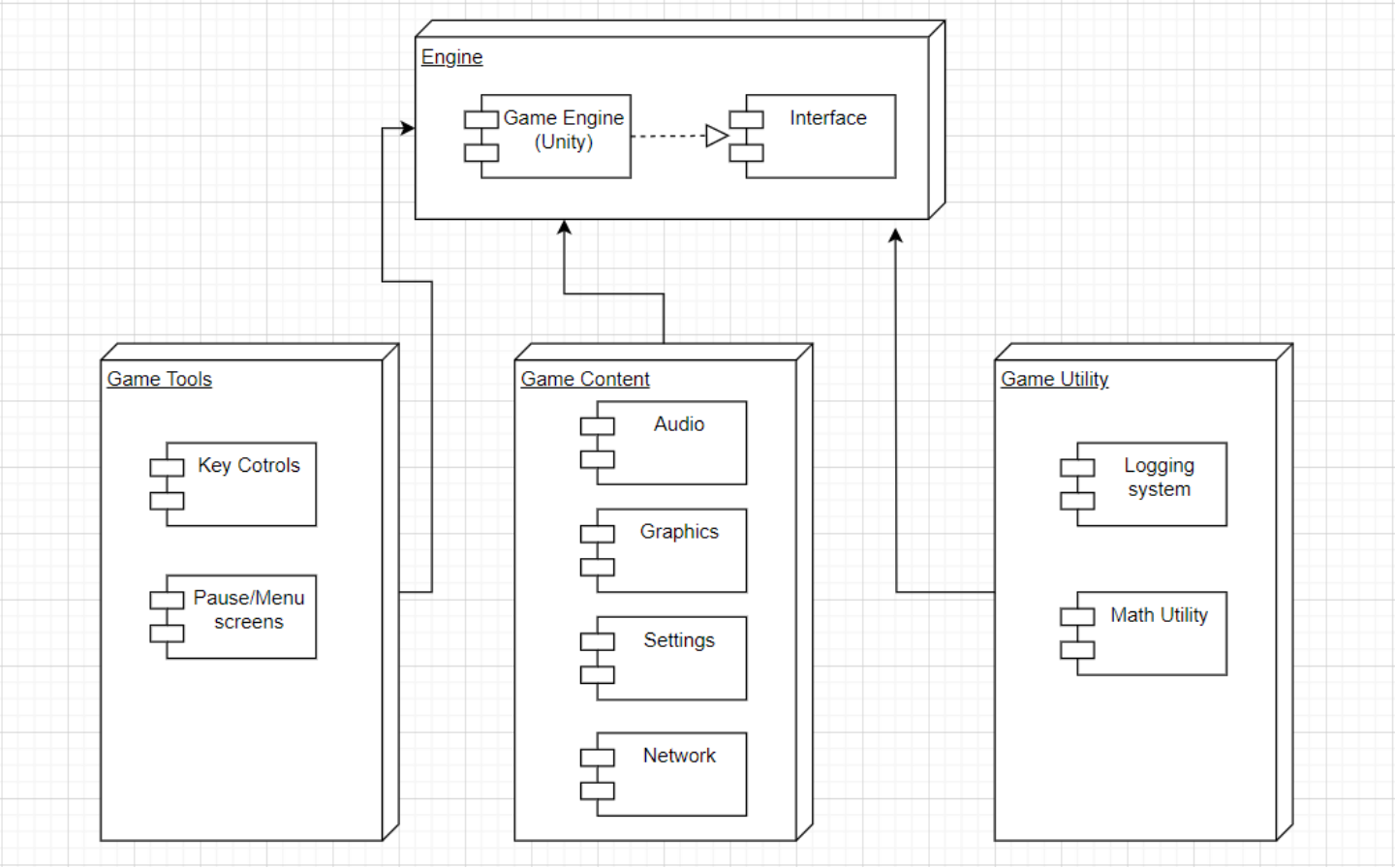


Figure 11: Component diagram showing the how aspects of our game feeds into Unity and interfaces.

to survive. The losing conditions are to die before destroying the entire city. The user interface(HUD) contains the following aspects: health bar background, health bar fill and points. The health bar background represents the initial total health that remains static at the start and as the player's health depletes. The health bar fill indicated the player's current health. It decreases as the player takes damage. The points element tracks points earned by the monster player through actions like city destruction and defeating military forces. Within the audio category, there are two elements: sound effects and the theme song. The sound effects are various sound effects like "boom" for explosions. The theme song plays in the backgrounf for entertainment purposes, independent of the game functionality.

2.7 User Experience

2.7.1 Gameplay Diagram

This diagram illustrates the flow of how our game should operate. The user will start the game up. They are then asked to login, if they do not have an account, they will create one and then login. Once they have done that, they will then have the option to start the game or look at the leaderboard. Clicking the leaderboard option will show the leaderboard and then lead the player back to the menu. The player clicks start and then the gamplay begins. They will play a character that destroys buildings, which earns them points as the buildings take damage. They will be attacked by NPCs, which will gradually deplete their health. If the player's health bar reaches zero, they will die and lose the game. However, the player can attack NPCs. When they attack, the NPCs lose health and die. If the player gets enough points and their health bar is not empty then they win and the game is over.

2.7.2 Gameplay Objectives

Our game challenges hand-eye coordination and strategy. The player must be able to make the moves they decide on quickly and efficiently using their keyboard. They also need to be able to strategize and think on the

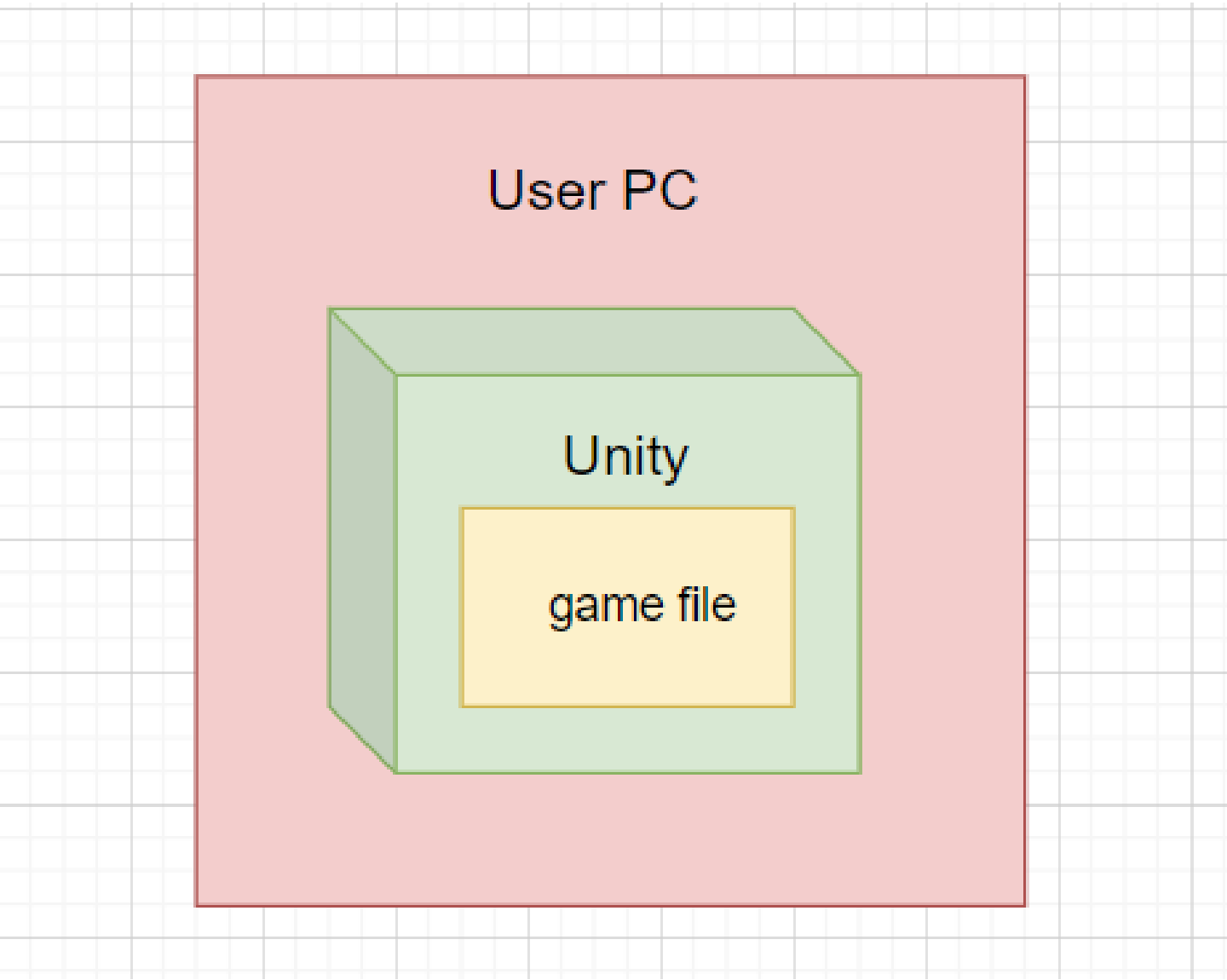


Figure 12: Deployment diagram showing how our game is deployed.

fly. They are going to be getting attacked as they try to destroy the buildings, and they don't want to die and lose the game. So they have to strategize on when to attack NPCs versus when to focus on destroying buildings and gaining the points they need to win. It's a delicate balance that requires them to both be able to strategize quickly and carry out their strategy with good hand-eye coordination. Playing this game will keep their intellect and reflexes sharp as they move throughout the game and destroy the buildings.

2.7.3 User Skillset

The user should be able to interact with the game by using their mouse or trackpad and a keyboard. There is no screen interaction capability, so the user will not be able to use a digital touchscreen keyboard – it must be physical. Even if the game supported screen interaction, a digital keyboard when get in the way of the game presentation on the screen and interfere with the gameplay. There is no memorization or puzzle solving, but they will need to develop strategies as they play and use quick reflexes to avoid the bullets being fired at them by an enemy. The user will need to have some level of muscle memory and know how to type without having to reference the keyboard. They will need to be able to control the keys without taking their eyes off of the screen so that they can still notice and react to enemy NPCs and changes in health and score while they play.

ID	Description	Attributes	Purpose	Test Cases	Dependencies	Status	Requirements
player	Playable character(Monster)	run, punch,take damage, earn points, climb buildings, jump?	Gameplay	monsterRunTest, monsterScaleBuilding, monsterPunchTest, monsterDieAnimation	N/A	In progress	health, speed, model
building_1	Building 1 that player destroys	stay static, take damage from player, crumble when enough damage is inflicted on the building?	Gameplay	building1ShowHoleTest, building1CrumbleTest	Player	In progress	size, texture
building_2	Building 2 that player destroys	stay static, take damage from player, crumble when enough damage is inflicted on the building?	Gameplay	building2ShowHoleTest, building2CrumbleTest	Player	In progress	size, texture
building_3	Building 3 that player destroys	stay static, take damage from player, crumble when enough damage is inflicted on the building?	Gameplay	building3ShowHoleTest, building3CrumbleTest	Player	In progress	size, texture
soldierNpc	Soldier	shoot, take damage from the player, deplete the player health, move in a loop	Gameplay	soldierMoveTest, soldierShootTest, soldierDieTest	Player, pathfinding	In progress	health, speed, weapon
tankNpc	Tank	shoot, take damage from the player, deplete the player health, move in a loop	Gameplay	tankMoveTest, tankShootTest, tankDieTest	Player, pathfinding	In progress	health, speed, weapon
planeNpc	Plane	shoot, take damage from the player, deplete the player health, move in a loop	Gameplay	planeMoveTest, planeShootTest, planeDieTest	Player, pathfinding	In progress	health, speed, weapon
background	Background	N/A	Aesthetics	N/A	Lighting	Completed	texture, size
audio	Audio	Play when game event occurs	Aesthetics	boomTest, crumbleTest, sfxTest,		In progress	background music, sound effects
login	Login	Create and store player info	Account	loginTest	Player, Acount	In progress	username, password
leaderbaord	Leaderboard	username and high score, date	Account	leaderboardDisplay, highScoreChange	Account	In progress	player scores
health	Health	destruction counter, total health bar outline, red actual health bar outline	Gameplay	healthCount, healthDisplay	Player, Account	In progress	N/A
points	Points	NPC death, building destruction	Gameplay	pointCount	Player, Account	In progress	scoring system

Figure 13: Requirements Traceability Table.

2.7.4 Gameplay mechanics

This game is a point-based game. The user is trying to achieve 100,000 points to win. There is no time limit, however if their health depletes to zero they lose; they do not have multiple lives. The rules are as such: the player can climb buildings and destroy them when they stop moving in certain spots. They will be getting shot at by enemy NPCs. They can jump and move to avoid bullets, but if they get hit, they will lose health. There is no way to replenish that. The user can move to the enemy and squash them to kill them – this will give them a small amount of points. There will not be multiply players so they cannot team up with someone to divide and conquer or race against. It is simply them vs. the enemies until they either get enough points to win or lose enough health to lose.

2.7.5 Gameplay Items

The user will not encounter any health tokens or powerups in our game. There are no coins, loot, or character upgrades that can be made. The user will remain as the main character, George, in his one state of being for the entirety of the game. George does not have any powers or special abilities. He can walk, jump, climb buildings, punch, and squash soldiers. That is all that the user will be able to do as they play through the game. If we continue this project in the future and add other levels, George could get 2x points for 30 seconds or something of a similar effect, however, that is out of our scope we have created for this project as we are simply recreating the first introductory level of the game. There are no special tricks at this time, especially not any that would have any effect on the gameplay.

2.7.6 Gameplay Challenges

The player has everything they need to win the second they start the game. George does not need any special abilities, he simply needs to be able to move around, destroy buildings, and squash enemies. This makes it simple enough for someone with a lower skillset to be able to enjoy playing, but something with a higher skillset could replay as many times as they want to make improvements to their score. They would know, however, that any improvements they see are a reflection of the improvement of their hand-eye coordination or their strategy; it shows personal improvement, not an easier win with assistance. They are not given a helping hand when they encounter difficulties, which forces them to be involved and strategize as they play to be able to win and potentially increase their score each time. A spot on the leaderboard would be earned through their practice work, not a gift from the game.

Game Objects			
ID	Data Type	Description	Texture
monster	Sprite Renderer	The playable character that is supposed to destroy the city	Pixels
soldierNPC	Sprite Renderer	Soldier characters that will try to save the city and try to kill or inflict as much damage as possible onto the character	Pixels
tankNPC	Sprite Renderer	Soldier characters that will try to save the city and try to kill or inflict as much damage as possible onto the character	Pixels
planeNPC	Sprite Renderer	Soldier characters that will try to save the city and try to kill or inflict as much damage as possible onto the character	Pixels
building1	Sprite Renderer	Buildings in the city that the player is destroying	Pixels
building2	Sprite Renderer	Buildings in the city that the player is destroying	Pixels
building3	Sprite Renderer	Buildings in the city that the player is destroying	Pixels
background	Image	Background city scene wow amazingggggggg :D	Pixels

Figure 14: Data dictionary table game objects.

2.7.7 Gameplay Menu Screens

The first screen the user will see when they open the game is the login screen. If they already have an account, they can just go ahead and login. If they need an account however, they can click the register button and be directed over to the registration page. Once they register, they will go back to the login page and sign in. Once they login, they will see the main menu screen. Once here, they will see three options. Option one is to see the leaderboard, which will take them to the leaderboard scene and then they will come back to the main menu. The second option is to start the game, which will take them into the game. While in game they can bring up the pause screen, which will have options to resume the game or leave the game and go back to the main menu. Once the game is over, they will see either a win screen with a congratulatory message, or they will see a lose screen that declares that they are, in fact, a loser. If they make the leaderboard, they will be announced on the win screen. After that screen displays, the credits will show and then it will move back to the main menu. The third option on the main menu is to log out of their account, which will log them out and bring them back to the main menu screen. Screenshots of screens we have completed are attached, as well as drawings of ones that are not finished yet and/or have not yet been started.

Collisions			
Collision ID	Collision Type	Description	Objects involved
playerClimbBuilding	boxCollider	The player interacts with the building by scaling it	Player, Building
playerPunchBuilding	boxCollider	The player interacts with the building by scaling it	Player, Building
npcShootPlayer	capsuleCollider	The NPC interacts with the player by shooting them	Player, NPC
playerKillNPC	capsuleCollider	The NPC interacts with the player by dying when the player interacts with them	Player,NPC

Figure 15: Data dictionary table collisions.

Game Mechanics		
Scoring Rules	Winning Conditions	Losing Conditions
50 points for each soldier eaten, 100 points for each tank squished, 150 for each plane squished, 20 points for each punch to	destroy all buildings	lose health before destroying all buildings

Figure 16: Data dictionary table game mechanics.

2.7.8 Gameplay Heads-Up Display

Our heads up display is extremely simple. They will see their health bar in the top left corner, which will change as they endure attacks from enemies. They will see their points just above the health bar, which will go up as they destroy buildings and kill their enemies. We will not have a pause button on screen as a key function will be used for that. They only have the one life and that is represented by the health bar. If it empties, they will see the lose screen. Our player does not have any ammo or maps, so we do not need to implement any kind of visual in the heads up display for that. We do not need a map because our screen does not move – they see the same display on their screen for the entire game and they are unable to move away from that. Thus, no map is needed.

2.7.9 Gameplay Art Style

Our game is a 2D cartoon pixelated style based off the visuals of games in the 1980's. The characters are basic with very little detail – we are not trying for any kind of realism here. There is no depth, as it is 2D, so depth is implied by shadowed pixels in certain areas. The bullets will simply be traveling pixels and not accurate shells, and the soldiers will be in windows, so we do not have to have full body enemies, just shoulders and up with a gun. George himself is simple ape, and doesn't rotate so much as flips and goes up and down.

User Interface		
HUD Elements	Type	Defintion
Health bar background	Image	The health bar background represents the total amount of health whenever the game starts, and whenever the player's health depletes, it will stay the same
Health bar fill	Slider	The health bar fill represents how much health the player has whenever the game starts and commences; Once the player takes damage, the health fill bar will slowly
Points	Text	When the monster player destroys the city or kill the military, the points will

Figure 17: Data dictionary table user interface.

2.7.10 Gameplay Audio

Originally, we had set out to use gameplay audio that was accurate to the original game in the eighties. However, our client has given us permission to use custom audio in our game. Our background music is two of our members humming and beatboxing together. Our character sound effects are done by one team member, while another member is handling the a cappella music to be layered over our win/lose screens and into the credits. Each member will also do various ambient sounds like buildings breaking, gun 'pew's, and helicopter whirring. These audios will attached to the appropriate sprites and scripts for each scenario, and should layer with each other well when needed. These audios will be created/recorded on various member's phones and compiled into a folder for the game to utilize. They will be mp3 files, and well labeled so that there is no confusion on which sound belongs to which aspect of the game.

3 Non-Functional Product Details

3.1 Product Security

3.1.1 Approach to Security in all Process Steps

For our game, there will be only two types of users. We have the administrators, which would be our team members, and the player, which would be those who play the game. The administrators have the ability to access the backend of the game, so they can modify code, alter game design and so on. Another thing administrators can do is view the database where they can see all of the players' login information and scores. Administrators have a separate login screen that is implemented in the Unity editor where they can log into the PlayFab account using the added PlayFab editor extension. Here we can also view data hidden from players. To see the overall

Audio	
Sound Effects	Purpose
Boom	Whenever any type of explosion happens, the boom sound effect will occur
Crumble	Whenever a building takes enough damage and the crumble animation commences, so will the crumble sound effect
Theme song	The theme song will be playing in the background with no dependency on any game function; It is solely for entertainment
Tank?	Whenever the tank moves, a sound effect will play
Plane?	Whenever the plane moves, a sound effect will play

Figure 18: Data dictionary table audio.

user information, the administrator must access the account using the web browser. There you can see all of the player's data as well as alter that data. They can also add and delete users and see logins, new users and API calls through the PlayFab account.

The player only has the ability to see what the administrators allow the player to see. That goes for the login/registration pages, the start, pause menus, the leaderboard and the actual gameplay view which includes the character. The player can only login using the in-game login system and can only view their personal information. The player can also view other players' high scores on the leaderboard but are unable to edit that information. The player has the ability to play the game and upload scores to the database without accessing the database itself.

Our IT infrastructure might be vulnerable to security breaches and performance issues due to outdated software versions, unpatched systems, and improper configurations. We are also using an internet based platform to organize our player accounts, which will open doors to many security issues. A solution to these significant issues is to implement comprehensive IT security and maintenance policies to address the following system configuration key points. To ensure servers, frameworks and system components have all the patches issues for the version, we have to try and establish a process for monitoring and applying patches for the version of software in use to address any known vulnerabilities. For disabling the directory settings, we have to configure the web server to disable directory listings to prevent potential exposure of sensitive information. In terms of restricting the web server, process and service accounts to the least privileges possible, we have to ensure that the web server, process and

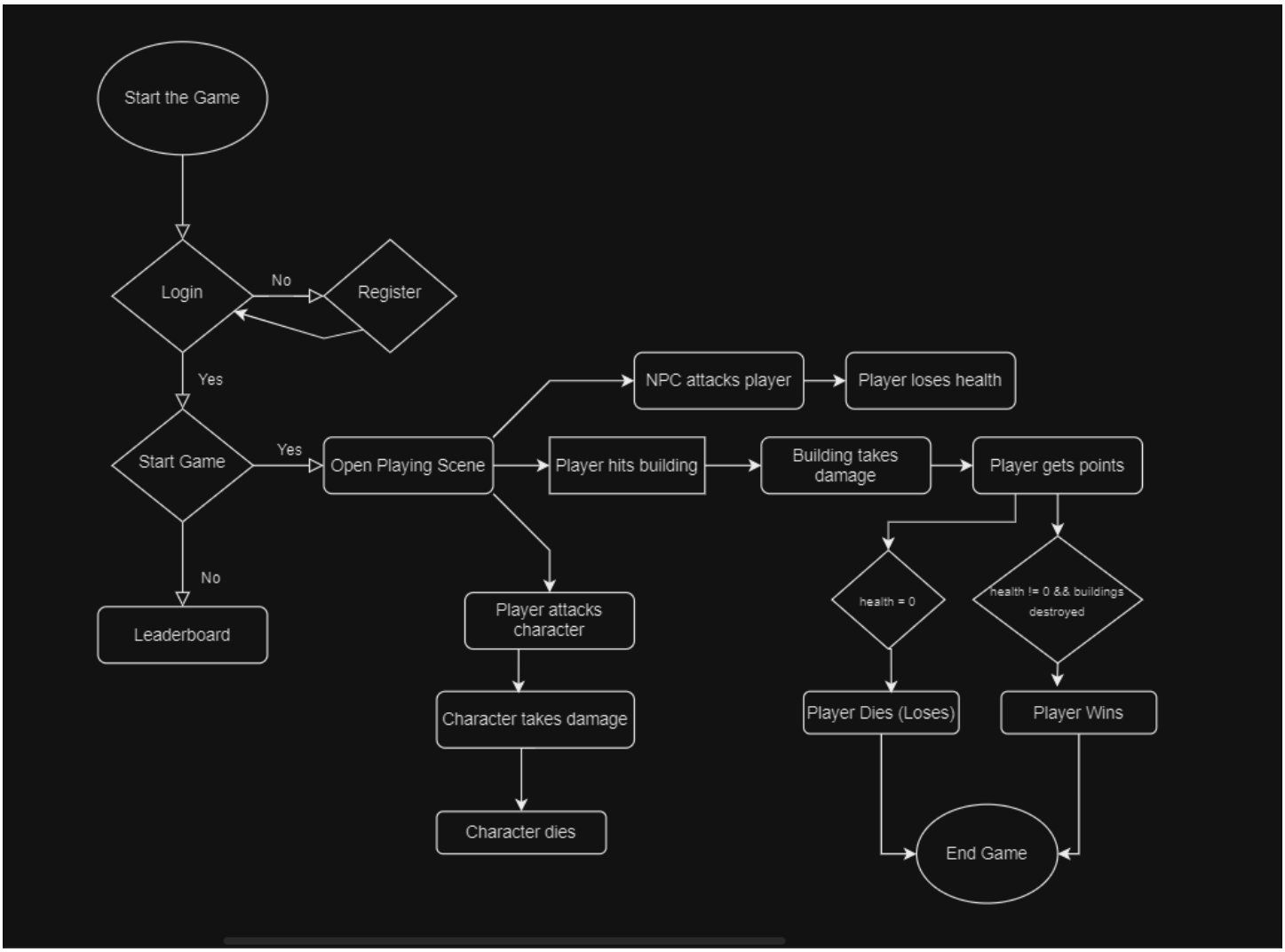


Figure 19: Diagram of Rampage Gameplay.

service accounts have the minimum privileges necessary to perform their functions, reducing the potential impact of any compromised account. To ensure that when an exception occurs that the software should fail securely, we have to implement a fail-safe mechanism that handles the exceptions securely, preventing unauthorized access or information leakage.

Within the database security, the internet based database we are using may be vulnerable to security breaches and unauthorized access due to inadequate security measures and configurations. To ensure we utilize input validation and output encoding and ensure to address meta characters and what to do when they fail, we must implement thorough input validation and output encoding to mitigate the risk of injection attacks. If validation fails, then we have to avoid executing the database command. To ensure that the variables are strongly typed, we have to ensure that the variable used in the database operations are strongly typed to prevent unintended type conversions and potential vulnerabilities. In order to use secure credentials for database access, we have to ensure strong, unique credentials for database access and ensure they are securely stored and managed. closing the database connections as soon as they are no longer needed helps free up resources and minimize the window of opportunity for potential attacks. We also must take into account that we have to eliminate any default vendor content, that may provide unnecessary information to potential attackers. Our group must also find a way to deactivate any default account that are not required to support the specific businesses requirements of the application.

3.1.2 Security Threat Model

With our game, we are using an online platform called PlayFab that is not created by Unity which can cause security risks. In the Security Threat Model, we can see that there are two trust boundaries. We believe that the

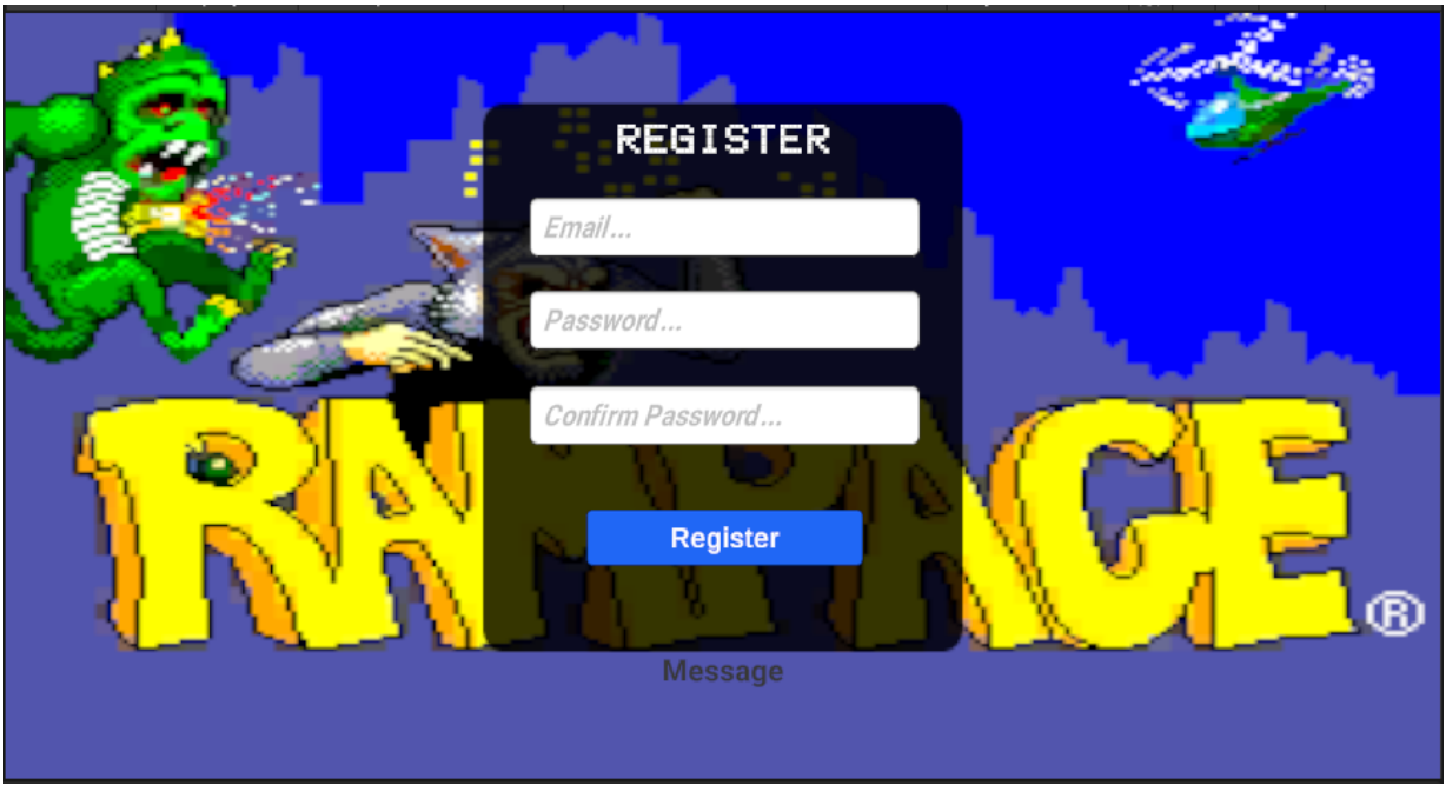


Figure 20: Image of Register Screen.

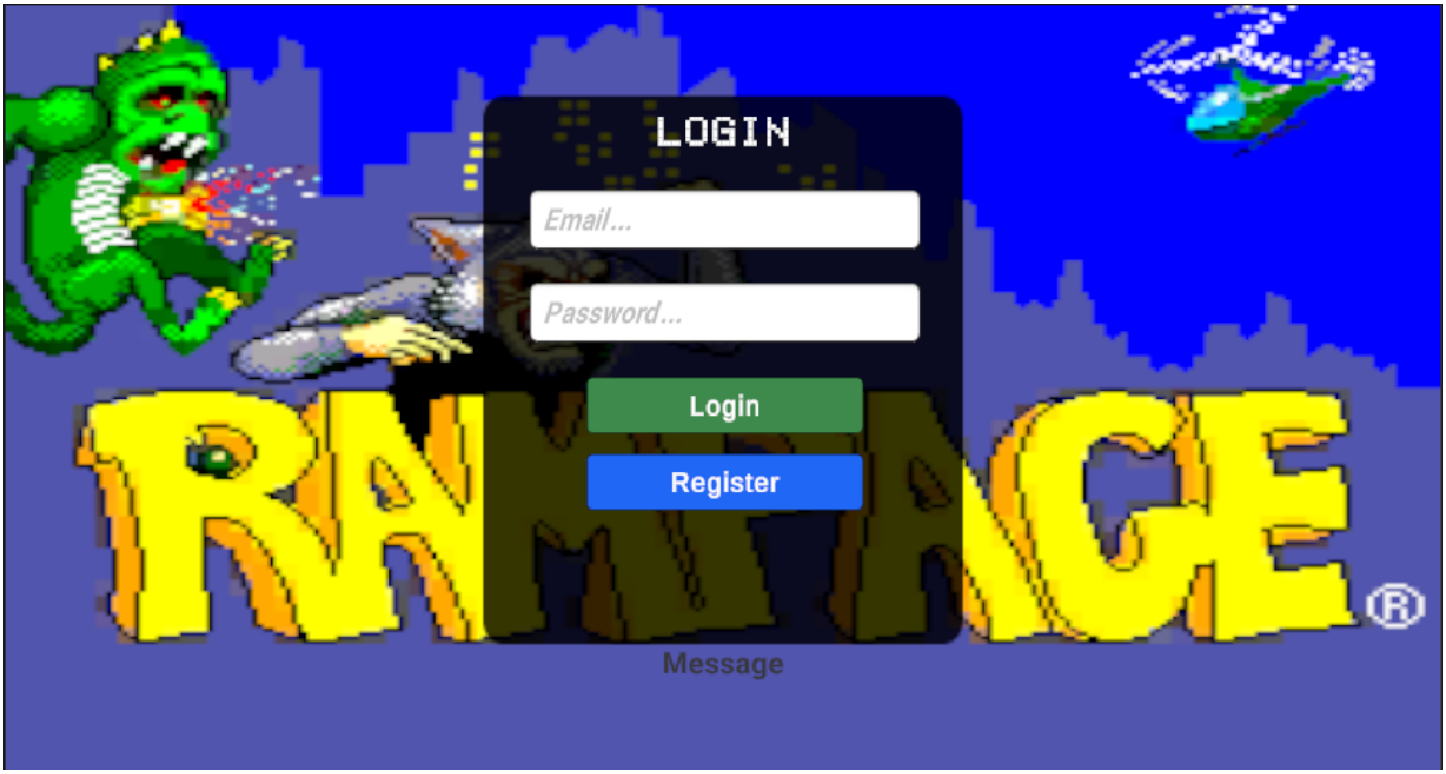


Figure 21: Image of Login Screen.

database would be the riskiest as it is an online database and if this becomes breached then data can be given to unauthorized users. Another risk with using PlayFab would be that unauthorized access to player accounts or administrative functions can lead to cheating, data manipulation, and game disruption. We have to ensure that the user cannot input certain information into, for example login, that would give them access into the game that

would allow them to cheat or steal vital user information.

We can also see in the model, that the user, whether admin or player, requests access to the game and the game calls the game database to get data from it and responds to the user after the data is received. We have to ensure that once the user requests data from the game, that the game does not display sensitive information that the user can use to cheat or gain access to certain features of the game that is not authorized for them to see.

3.1.3 Security Levels

For our game, there will be only two types of users. We have the administrators, which would be our team members, and the player, which would be those who play the game. The administrators have the ability to access the backend of the game, so they can modify code, alter game design and so on. Another thing administrators can do is view the database where they can see all of the players' login information and scores. Administrators have a separate login screen that is implemented in the Unity editor where they can log into the PlayFab account using the added PlayFab editor extension. Here we can also view data hidden from players. To see the overall user information, the administrator must access the account using the web browser. There you can see all of the player's data as well as alter that data. They can also add and delete users and see logins, new users and API calls through the PlayFab account.

The player only has the ability to see what the administrators allow the player to see. That goes for the login/registration pages, the start, pause menus, the leaderboard and the actual gameplay view which includes the character. The player can only login using the in-game login system and can only view their personal information. The player can also view other players' high scores on the leaderboard but are unable to edit that information. The player has the ability to play the game and upload scores to the database without accessing the database itself.

3.2 Product Performance

3.2.1 Product Performance Requirements

Many games have different attributes to make the performance as best as it could be. We have decided that our game should run over 60 frames per second to make sure that no matter what happens, the game will be able to run as fast as it can. The CPU should be at least 70% at best. The operating system version for the player to play the game will be Windows 7 and up for Windows, High Sierra 10.13+ for macOS, and Ubuntu 20.04 and 18.04 as well as CentOS 7 for Linux. The expected CPU needed for the game is 15 execution time but for now it sits at least 10 seconds in execution time. Analyzing and monitoring the software every few minutes will also help to ensure the game maintains high performance. For disk space, the player must have at least 80GB of space to make sure the game is functional. The application would be free for everyone to play on PC as well as Android and IOS. If needed, the user could also lower the resolution in the actual game to make things run smoother. The user must also make sure that the actual software can handle Unity games with a large disk space.

3.2.2 Measurable Performance Objectives

The goal for this project is to recreate one level of the game Rampage. We decided to split the tasks up into parts like who will be responsible for sprites and controls of the main characters, NPCs and their scripts, login, etc. For this section of the project, we went with creating basic character scripts, adding sprites, and creating separate animations. With the many sprites that are present, we are self-aware of any that could take up disk space which could slow the work process on the game. To soften the blow of doing some tasks by ourselves, we have also decided to look up YouTube videos to help give a guide on how to program the game. The main platform that the game will be available on is the PC so to make things easier, the game will be made with the control scheme of the keyboard. The next goal for the project is to go all into programming more complex parts of the code as well as doing more animations and making the NPCs functional. Sound will also be implemented at the same time as the other scripts are program and there are ideas of having actual people doing the voices instead of just relying on sound effects.

3.2.3 Application Workload

The user will generally spend about 5% of the game on the menu to at least start the game. About 75% of the user's time will be spent in the actual game. Since the gameplay takes up most of the project, the user will mostly be breaking buildings and eating civilians to win. The login will take up at least up to another 5%, it will not take long for the user to register an account which should take at least two minutes at best. As for system workload, the average time for the game to boot up will take less than 10 seconds. The application for the game will mostly be the one thing that is running on your computer for the best performance the user desires. Many applications not related to the game will more than likely cause bottlenecks in the background because they will cause either uneven CPU bottlenecks or GPU bottlenecks. They would be running on the software or on a hard drive if one possessed it. The loading time to get from one part of the game to another will also be short and it will take less than 10 seconds to load. The use of settings in the game will depend on the user, but the user will likely spend another 5% on settings. The user will have a lot of time to get used to the game, especially recreation of an old title. Our team hopes that the user has fun playing and plays with the best performance necessary.

3.2.4 Hardware and Software Bottlenecks

Bottlenecks vary depending on the computer and the space it uses. They mostly occur when either the CPU or the GPU holds back the potential of your software. Some of the issues that the user will likely encounter is high CPU when playing the game which could cause the game to slow down. To fix this problem, the users could do things like changing in-game settings by slowing down the graphics card to make sure that the rhythm matches the GPU, closing background programs by closing tasks in the task manager, overclocking RAM and CPU, etc. GPU bottlenecks can occur when the GPU card can calculate fewer images per second compared to the CPU. The way to fix a GPU bottleneck is by overclocking the component by increasing the temperature limit and increasing the clock speed, lowering the graphics in the game's setting, or even by upgrading your GPU. You can also fix GPU bottlenecks by utilizing hardware monitoring tools as these can help you search for whatever is causing your system to slow down, as well as having a high-resolution monitor which demands equal power from both the CPU and GPU. If you are playing online, network bottlenecks can also occur. They happen when there are not enough resources to ensure reliable network data delivery. They can be fixed by upgrading your network by increasing your bandwidth and improving the architecture of the network. Another way to fix a network bottleneck is to divide your service plan by separating them into different sub-networks.

3.2.5 Synthetic Performance Benchmarks

The team used Sysbench to see results of how the game should run on our target platform, the PC. According to the first graph shown, the execution time for the CPU decreases around 32 threads. This is a good thing since it shows that execution of a program will be as quick as possible without it ruining the latency of the system. This will also benefit the actual game itself since the low execution time will make the game run smoothly without trouble. The throughput, in the second graph, measures how many units of information the system can process in each amount of time. If the throughput is low, the network will not be able to transfer large files and produce high rates of latency in the system. The read time is higher than the write time since it will take more time to read everything within certain files. The common pattern of both read and write time is that the throughput increases as the file size increases. The written time's throughput increases from 3 to between 5 and 6 while read time's throughput increases from between 4 and 5 to between 8 and 9. With the throughput being less than 10, it will ensure that loading certain assets of the game to the scene will be quick and sufficient for the user. It will also ensure that with the increase of the file size, the game will still run smoothly and will read files without ruining the CPU and GPU for the player and for the team.

3.2.6 Performance Tests

Currently there are no visualization aid for most of these tests because they have yet to be implemented into the game. The tests are listed:

monsterRunTest: tests if the player can move left and right as well as jump

monsterScaleBuilding: tests if the player can interact with the building by scaling it

MonsterPunchTest: tests if the player can interact with the building by punching it

monsterDieAnimation: tests if the player has the right death animations

building1ShowHoleTest: tests if building 1 can interact with the player by showing holes after the player punches them.

building1CrumbleTest: tests if the buildings will execute the crumble animations once inflicted with enough holes

building2ShowHoleTest: tests if building 1 can interact with the player by showing holes after the player punches them.

building2CrumbleTest: tests if the buildings will execute the crumble animations once inflicted with enough holes

building3ShowHoleTest: tests if building 1 can interact with the player by showing holes after the player punches them.

building3CrumbleTest: tests if the buildings will execute the crumble animations once inflicted with enough holes

soldierMoveTest: tests if the soldiers move a certain distance then move back to their spot.

soldierShootTest: tests if soldiers can shoot ammunition by command

soldierDieTest: tests if the soldier dies when interacting with the player y being hit/squashed

tankMoveTest: tests if the tanks move a certain distance then move back to their spot.

tankShootTest: tests if tanks can shoot ammunition by command

tankDieTest: tests if the tank dies when interacting with the player y being hit/squashed

planeMoveTest: tests if the planes move a certain distance then move back to their spot.

planeShootTest: tests if planes can shoot ammunition by command

planeDieTest: tests if the plane dies when interacting with the player y being hit/squashed

boomTest: tests if the boom sound effect works when properly called with some sort of damage inflicted by the player or the NPC

crumbleTest: tests if the building crumbles when inflicted with a certain amount of damage

RegisterTest: tests if the registration is successful

loginTest: tests if the login is successful

leaderboardDisplay: tests if the proper high score displays for the accounts registered in a leaderboard fashion

highScoreChange: tests if the high score changes If new high score surpasses the old high score

healthCount: tests if the health depletes properly after damage is inflicted on the player

healthDisplay: tests if the health depleting in the health bar is displayed

pointCount: tests if the correct amount of points are added to the score

4 Software Testing

4.1 Software Testing Plan Template

Test Plan Identifier:

Introduction:

Test item:

Features to test/not to test:

Approach:

Test deliverables:

Item pass/fail criteria:

Environmental needs:

Responsibilities:

Staffing and training needs:

Schedule:

Risks and Mitigation:

Approvals:

4.2 Unit Testing

Text goes here.

4.2.1 Source Code Coverage Tests

Text goes here.

4.2.2 Unit Tests and Results

Text goes here.

4.3 Integration Testing

4.3.1 Integration Tests and Results

Test Plan Identifier: Integration Testing

Introduction: We used integration testing to combine the code and assets for our game. The main goal of using this is to combine every asset into one scene so we can fix any errors that surround the actual gameplay. The game should have a high and consistent framerate without changing settings.

Test item: The main software we used to test data is on Unity, a game engine. The games created in Unity can be played on multiple platforms like PC, Switch, X-Box, etc. The planned system it will be tested on is PC and Mac since they are the easiest to grasp and understand. We also used Sysbench to view any drops in the CPU and GPU.

Features to test/not to test: One of the main goals for the Rampage recreation is that the character should move left and right as well as jump and animations. We decided to exclude the environment as well as the environment size because they are low priority and the team wanted to focus on documentation and polishing the game more.

Approach: We tested the software by using the bottom-up method in which small parts of the program were tested before testing the main module. We had separate scenes to test the necessary parts before being put in the main scene. Then we would play it, find issues, and try to solve the code and its performance.

Test deliverables: We used Sysbench to detect any drops in framerates as well as to see if there were any bottlenecks that were present in the system. When inside the code, we would use debug log to make sure that every button-press and anything UI related would function correctly and will not hold back other aspects of the game.

Item pass/fail criteria: For entry criteria, before we start testing code, we must make sure that the test environment was set up and ready to code and the testable code was available. Exit criteria for this project is the deadline is met or most of the bug that were spotted in the code were fixed.

Environmental needs: For the game to function, it must run on a Windows 10, Mac, or any up-to-date PC device. The system should also run on at least 64GB for optimal performance. It would be best to store the game on an external hard drive with at least 64GB or a regular computer with the same number of GB.

Responsibilities: One person was responsible for main menu, the other NPCS, and the other for the main character. Everyone on our team is responsible for testing their code and placement based on what they chose to work on. We are also responsible for giving each other feedback on any element that is not right when put into the main scene.

Staffing and training needs: The staff must have some coding experience, experience with C#, or some familiarity with Unity. Anyone who doesn't have any experience with C# can get some insight from more experienced staff. The training will help with typing faster as well as being familiar with C# to make progress faster. The staff must also be able to notice any bugs and report them to the developers or detect where the problem lies and edit them out.

Schedule: Test schedule should also be noted in the Gantt Chart. Test estimation (Efforts) and high-level schedule. Schedule should be for key deliverables or important milestones. Ideally, all test deliverables included in the test plan should be scheduled. On November 21st to December 1st, we have been testing all codes for all assets. We would spend at least a few hours a day to make sure the player and menu are functioning as expected

and that audio is present. At this moment, we are 50% done with testing and we are now focusing on doing our documentation.

Risks and Mitigation: When we implemented our assets into the same scene, we assumed that the game would cause a slow frame rate, and while some of that was true, the gravity of the characters would also mess up along with it. To fix these problems, we would have to turn off background apps and edit the gravity in the Unity menu. We also had to minimize any settings like graphics and controls which sped the game up significantly. For any errors in the code, we would look over and change certain aspects of it until the code, at least, somewhat works for what is intended of it.

Approvals: As a team, we have all signed off and approved of all of our testing done on Nov. 29th with our client and Nov. 30th.

4.3.2 Integration Tests and Results

4.4 System Testing

Test Plan Identifier: SystemTesting

Introduction: System testing is done to check the behaviour of the complete software based on software requirements. In this test, we will test whether the system functionalities are behaving as expected and specified in software requirements document, that the user interfaces are user friendly and easy to operate and use, and that the system guide is correct and complete.

Test item: The software that we tested was a unity-made game that had numerous different functionalities, such as being able to login or register an account, viewing a leaderboard, and play the gameplay related to the game. The login/register and leaderboard scene use an external database to store and get its user data. We will test the functionality of these actions within the scene.

Features to test/not to test: The features that we tested were whether the user can register a new account or login with an existing account, do all of the scenes load as expected, and does the leaderboard get existing. We did not test whether there was a way for the user to cheat within the game and we did not check whether the game was compatible with other operating systems such as macOS because we ran out of time.

Approach: For the functional testing, we played the game ourselves to ensure that it was functioning as expected. To test the login/register, we would register multiple accounts as well as login with those to ensure that multiple accounts are possible. For documentation testing, we would ensure that all of our documentation is accurate to what we implemented for the game. For usability testing, we would ensure that the game is user friendly.

Test deliverables: Test cases for the system testing would be, "Is the user able to login/register an account?", "Is the user able to switch from scenes?", "Is the user able to play the game and win/lose?", "Is the user able to see the leaderboard?" All of these tests passed and fulfilled the required function.

Item pass/fail criteria: For the login/register scene to pass our needed criteria, the user would have to successfully register an account that will show up in online platform as well as be able to access that account when trying to login. For the loading of scenes, each scene would have to load successfully without errors. The leaderboard would also have to be able to access the database and obtain the leaderboard data. The player would also have to be able to win or lose within the actual gameplay. Otherwise, the tests would fail the criteria.

Environmental needs: For the software to run smoothly, the user would have to have access to Unity and a windows pc. We were unable to test whether it was playable on a Mac so we are unsure about that environment. The user would need to ensure that their system fulfilled all of the software requirements needed for Unity itself.

Responsibilities: The documented software testing was done by Tameka. The team as a whole went through the various functions of the game to ensure they fulfilled what we had all expected. The database was entirely monitored and implemented by Tameka so testing involving the database was done and seen to by her.

Staffing and training needs: The needed training would be learning more about system testing and how to go about the various types of system testing. Testing is an important part of software development so a better understanding on it would improve knowledge and skills significantly. To further increase software testing skills, we can utilise opportunities such as this course to help advance those skills. **Schedule:** For the software testing specifically, it was scheduled to be done by Nov. 30th. Based on the Gantt Chart, we scheduled and completed all of the prioritised tests during the last week of the sprint. This goes for the documentation for the testing as well as the test cases and their results.

Risks and Mitigation: A risk that could have transpired was that our database, if not implemented correctly, was not connected to our game. This would cause issues with logging in or registering as well as going further than the login/register scene. Additionally, if the database did not connect, there would be no way of viewing scores on the leaderboard either. To mitigate this, we would have to implement our database differently and not use an online platform to store user data.
Another risk would be that the system crashed due to errors within the code. This could be mitigated by code review and isolating where the issue is occurring.

Approvals: As a team, we have all signed off and approved of all of our testing done on Nov. 29th with our client and Nov. 30th.

4.4.1 System Tests and Results

4.5 Acceptance Testing

Text goes here.

4.5.1 Acceptance Tests and Results

Text goes here.

5 Conclusion

Text goes here.

6 Appendix

6.1 Software Product Build Instructions

To build our project, ensure that you have Unity installed. We are using the 2022.3.11f1 editor version so ensure that that editor us also installed on Unity. To find our shared project file, you can find it on GitHub under MikaMika2003/Rampage. From there download the zip file and open it in your unity hub. Once that opens, you should have the current state of our software product.

6.2 Software Product User Guide

As an admin user, you can log into the PlayFab database using the shared username and password to view the database. Go to PlayFab.com. Here you can view the players that have registered and the leaderboard for scores. As admin, you can also delete remove users and change user information. As a user, you can launch the game where you can either register or login to an account. After, you will be met with the start menu and you can press 'F' to start the game or 'L' to view leaderboard. Once the game is started you have to either destroy all buildings or come in contact with the npcs to be met with the game over scene. From there you can logout and login again if you wish to restart the game.

6.3 Source Code with Comments

6.3.1 ChangeInputs.cs

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

public class ChangeInputs : MonoBehaviour
{
```

```
    EventSystem system1;
    public Selectable firstInput;
    public Button submitBtn;

    // Start is called before the first frame update
    void Start()
    {
        system1 = EventSystem.current;
        firstInput.Select();
    }

    // Update is called once per frame
    // This is for when the user clicks tab to go to the next input field
    void Update()
    {
        // Checks if the tab and left shift is clicked
        if (Input.GetKeyDown(KeyCode.Tab) && Input.GetKey(KeyCode.LeftShift)) {
            Selectable previous = system1.currentSelectedGameObject.GetComponent<Selectable>().FindSe
            if(previous != null) {
                previous.Select();
            }
        }
        // Checks if the tab is clicked
        else if (Input.GetKeyDown(KeyCode.Tab)) {
            Selectable next = system1.currentSelectedGameObject.GetComponent<Selectable>().FindSelect
            if(next != null) {
                next.Select();
            }
        }
        // Checks if the enter button is clicked
        else if (Input.GetKeyDown(KeyCode.Return)) {
            submitBtn.onClick.Invoke();
            Debug.Log("Button pressed!");
        }
    }
}
```

6.3.2 PlayFabManager.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using PlayFab.ClientModels;
using PlayFab;
using System;
using UnityEngine.UI;
using Unity.VisualScripting;
using UnityEngine.SceneManagement;

// This class is used for interacting with our PlayFab database
public class PlayFabManager : MonoBehaviour
{
```

```
[Header("UI")]
public Text messageText;
public InputField emailInput;
public InputField passwordInput;
public Button registerBtn;
public Button loginBtn;
public Button Get_LB_Btn;
public Button Back_Btn;


void Start()
{
    // Assign the RegisterButton method to the button's onClick event
    if (registerBtn != null)
    {
        registerBtn.onClick.AddListener(RegisterButton);
    }

    if (loginBtn != null)
    {
        loginBtn.onClick.AddListener(LoginButton);
    }
    if (Get_LB_Btn != null)
    {
        Get_LB_Btn.onClick.AddListener(GetLeaderboard);
    }
    if (Back_Btn != null)
    {
        Back_Btn.onClick.AddListener(BackToStartScene);
    }

    AddRandomDataToLeaderboard();



}

// Scene manager that takes the user back to the start scene
public void BackToStartScene() {
    SceneManager.LoadScene("StartScene");
}

// If the register and login button is pressed, it checks to ensure that the inputs are valid
public void RegisterButton() {

    if (passwordInput.text.Length < 6) {
        messageText.text = "Password too short!";
        return;
    }

    var request = new RegisterPlayFabUserRequest {
        Email = emailInput.text,
        Password = passwordInput.text,
```

```
        RequireBothUsernameAndEmail = false
    };
    PlayFabClientAPI.RegisterPlayFabUser(request, OnRegisterSuccess, OnRegisterError);
}

void OnRegisterSuccess(RegisterPlayFabUserResult result)
{
    messageText.text = "Registered and logged in!";
    Debug.Log("PlayFab User ID: " + result.PlayFabId);
    SceneManager.LoadScene("StartScene");
}

void OnRegisterError(PlayFabError error)
{
    messageText.text = error.ErrorMessage;
    Debug.Log(error.GenerateErrorReport());
}


// If the login button is pressed then it checks user validation within the PlayFab database
public void LoginButton() {
    var request = new LoginWithEmailAddressRequest {
        Email = emailInput.text,
        Password = passwordInput.text
    };
    PlayFabClientAPI.LoginWithEmailAddress(request, OnLoginSuccess, OnLoginError);

}

void OnLoginSuccess(LoginResult result)
{
    messageText.text = "Logged in!";
    Debug.Log("Successful login/create account!");
    SceneManager.LoadScene("StartScene");
}

void OnLoginError(PlayFabError error)
{
    messageText.text = error.ErrorMessage;
    //Debug.Log("Error while logging in/creating account!");
    Debug.Log(error.GenerateErrorReport());
}


//
void OnError(PlayFabError error)
{
    //messageText.text = error.ErrorMessage;
    Debug.Log("Error while getting leaderboard");
    Debug.Log(error.GenerateErrorReport());
}

public GameObject rowPrefab;
```

```
public Transform rowsParent;

// Method to add random data to the leaderboard
public void AddRandomDataToLeaderboard()
{
    int numberOfEntries = 5; // Adjust this based on the number of entries you want to add

    for (int i = 0; i < numberOfEntries; i++)
    {
        // Generate a random score
        int randomScore = UnityEngine.Random.Range(0, 3);

        // Call the method to send the random score to the leaderboard
        SendLeaderboard(randomScore);
    }

    // After adding the random data, refresh the leaderboard

    GetLeaderboard();
}


// For leaderboard scores to send to PlayFab
public void SendLeaderboard(int score) {
    var request = new UpdatePlayerStatisticsRequest {
        Statistics = new List<StatisticUpdate> {
            new StatisticUpdate {
                StatisticName = "GainedPoints",
                Value = score
            }
        }
    };
    PlayFabClientAPI.UpdatePlayerStatistics(request, OnLeaderboardUpdate, OnError);
}

// If the score is updated it updates the leaderboard
void OnLeaderboardUpdate(UpdatePlayerStatisticsResult result)
{
    Debug.Log("Successful leaderboard sent!");

    // Check if the "B" key is pressed
    if (Input.GetKeyDown(KeyCode.B))
    {
        // Load your target scene
        SceneManager.LoadScene("StartScene");
    }
}


// Gets the leaderboard
public void GetLeaderboard() {
    Debug.Log("Attempting to get leaderboard...");
    Get_LB_Btn.interactable = false;
```

```
        var request = new GetLeaderboardRequest {
            StatisticName = "GainedPoints",
            StartPosition = 0,
            MaxResultsCount = 10
        };
        PlayFabClientAPI.GetLeaderboard(request, OnLeaderboardGet, OnError);
    }

    // Shows the leaderboard
    // Gets all the information like position, player name and score for each entry
    void OnLeaderboardGet(GetLeaderboardResult result)
    {

        Get_LB_Btn.interactable = true;

        // Remove all existing rows
        foreach (Transform item in rowsParent) {
            Destroy(item.gameObject);
        }

        foreach (var item in result.Leaderboard) {
            GameObject newGo = Instantiate(rowPrefab, rowsParent);
            Text[] texts = newGo.GetComponentsInChildren<Text>();

            // Get position on leaderboard, then player id then player score
            texts[0].text = (item.Position + 1).ToString();
            texts[1].text = item.PlayFabId;
            texts[2].text = item.StatValue.ToString();

            Debug.Log(string.Format("PLACE: {0} | ID: {1} | VALUE: {2} ",
            item.Position + 1, item.PlayFabId, item.StatValue));
        }
    }
}
```

6.3.3 OpenGame.cs

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

public class OpenGame : MonoBehaviour
{
    // Update is called once per frame
    void Update()
    {
        // Check if the "F" key is pressed
        if (Input.GetKeyDown(KeyCode.F))
        {
```

```
            // Load your target scene
            SceneManager.LoadScene("PlayingScene");
        }

        // Check if the "L" key is pressed
        if (Input.GetKeyDown(KeyCode.L))
        {
            // Load your target scene
            SceneManager.LoadScene("Leaderboard");
        }
    }
}
```

6.3.4 PlayerMovement.cs

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

// This file is used to control the main character's movements
public class PlayerMovement : MonoBehaviour
{
    private float horizontal;
    public float speed = 15f;
    public float jumpForce = 30f;
    private bool isFacingLeft = true;

    [SerializeField] private Rigidbody2D rb;
    [SerializeField] private Transform groundCheck;
    [SerializeField] private LayerMask groundLayer;
    [SerializeField] private Animator animator;


    // On start it gets the character and places it within the scene
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }

    // This used to determine and update the current  movement of the player
    void Update()
    {
        horizontal = Input.GetAxisRaw("Horizontal");
        Flip();
        animator.SetFloat("Speed", Mathf.Abs(horizontal));

        // Checks if the up key is pressed
        if(Input.GetKey("up"))
        {
            animator.SetBool("LookingUp", true);
        } else
        {
            animator.SetBool("LookingUp", false);
```

```
    }

    // Checks if the down key is pressed
    if(Input.GetKey("down"))
    {
        animator.SetBool("LookingDown", true);
    } else
    {
        animator.SetBool("LookingDown", false);
    }

    // Checks if the z key is pressed
    if(Input.GetKeyDown(KeyCode.Z))
    {
        animator.SetBool("IsPunching", true);
        StartCoroutine(WaitSecond());
        animator.SetBool("IsPunching", false);
    }

    rb.velocity = new Vector2(horizontal * speed, rb.velocity.y);

    Jumping();
}

// Introduces a wait period for character movement
IEnumerator WaitSecond(){
    yield return new WaitForSeconds(1);
}

// Checks if the space button is pressed
private void Jumping()
{
    animator.SetBool("IsJumping", !isGrounded());

    if(Input.GetButtonDown("Jump") && isGrounded())
    {
        rb.velocity= new Vector2(rb.velocity.x, jumpForce);

    }
}


// Keeps the character grounded to the added platform
private bool isGrounded()
{
    return Physics2D.OverlapCircle(groundCheck.position, 0.2f, groundLayer);
}

// Checks the direction in which the character is facing
private void Flip()
{
    if (isFacingLeft && horizontal > 0f || !isFacingLeft && horizontal < 0f)
        {
```

```
                isFacingLeft = !isFacingLeft;
                Vector3 localScale = transform.localScale;
                localScale.x *= -1f;
                transform.localScale = localScale;

        }

    }
}
```

6.3.5 HealthBarManager.cs

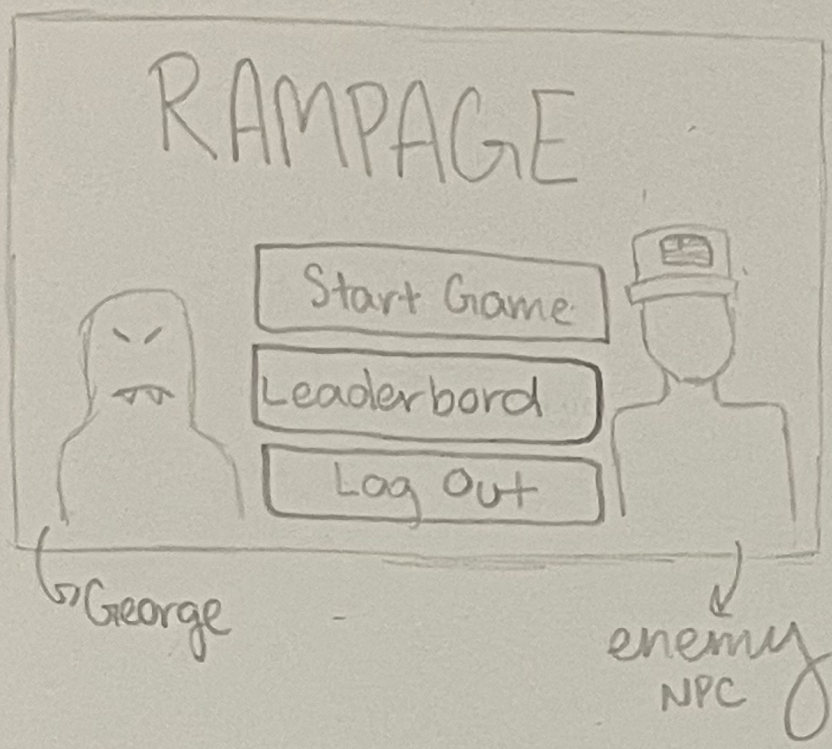
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

// This class was not used

public class HealthBarManager : MonoBehaviour
{

    // slider for the health bar
    Slider _healthSlider;

    // Gets the slider component
    private void Start()
    {
        _healthSlider = GetComponent<Slider>();
    }

    // sets the max health possible
    public void SetMaxHealth(int maxHealth)
    {
        _healthSlider.maxValue = maxHealth;
        _healthSlider.value = maxHealth;
    }

    // sets the health value
    public void SetHealth(int health)
    {
        _healthSlider.value = health;
    }


}
```

6.3.6 ScoreManager.cs

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

```
public class ScoreManager : MonoBehaviour
{
    public static ScoreManager instance;

    public Text scoreText;

    int score = 0;

    // Call the database to get the highscore and set it to highscore
    // int highscore = 0;


    // This will be called at the very start of the game before Start() is called
    // Will set an instance of the Manager that can be used in other classes
    private void Awake()
    {
        instance = this;
    }

    // Start is called before the first frame update
    void Start()
    {
        scoreText.text = score.ToString();
    }


    public void AddPoint()
    {
        //score += 1 //or however many points should be added
        scoreText.text = score.ToString();

        // To save the highscore
        //if (hignscore < score)
        //    PlayerPrefs.SetInt("hignscore", score);

    }
}
```

main menu



pause menu

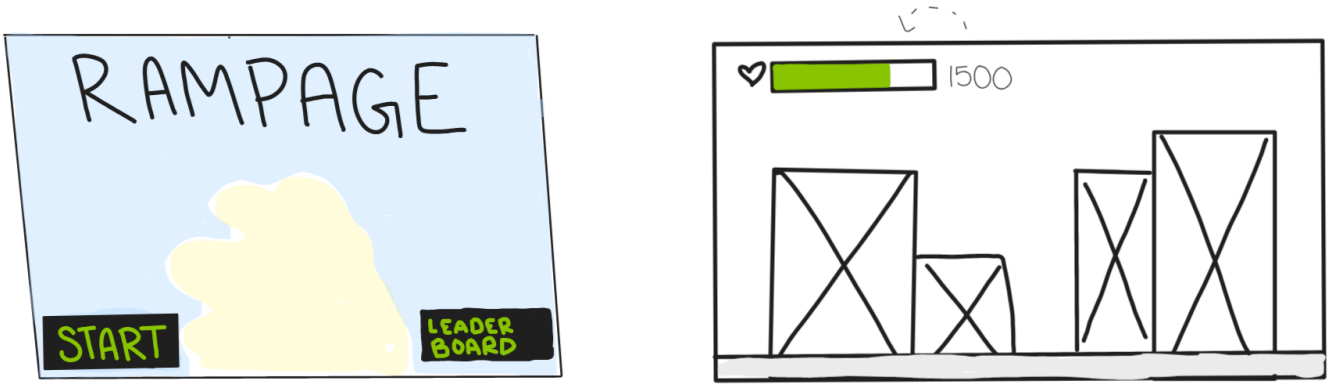


Figure 23: Images of the wireframes for main menu and gameplay.

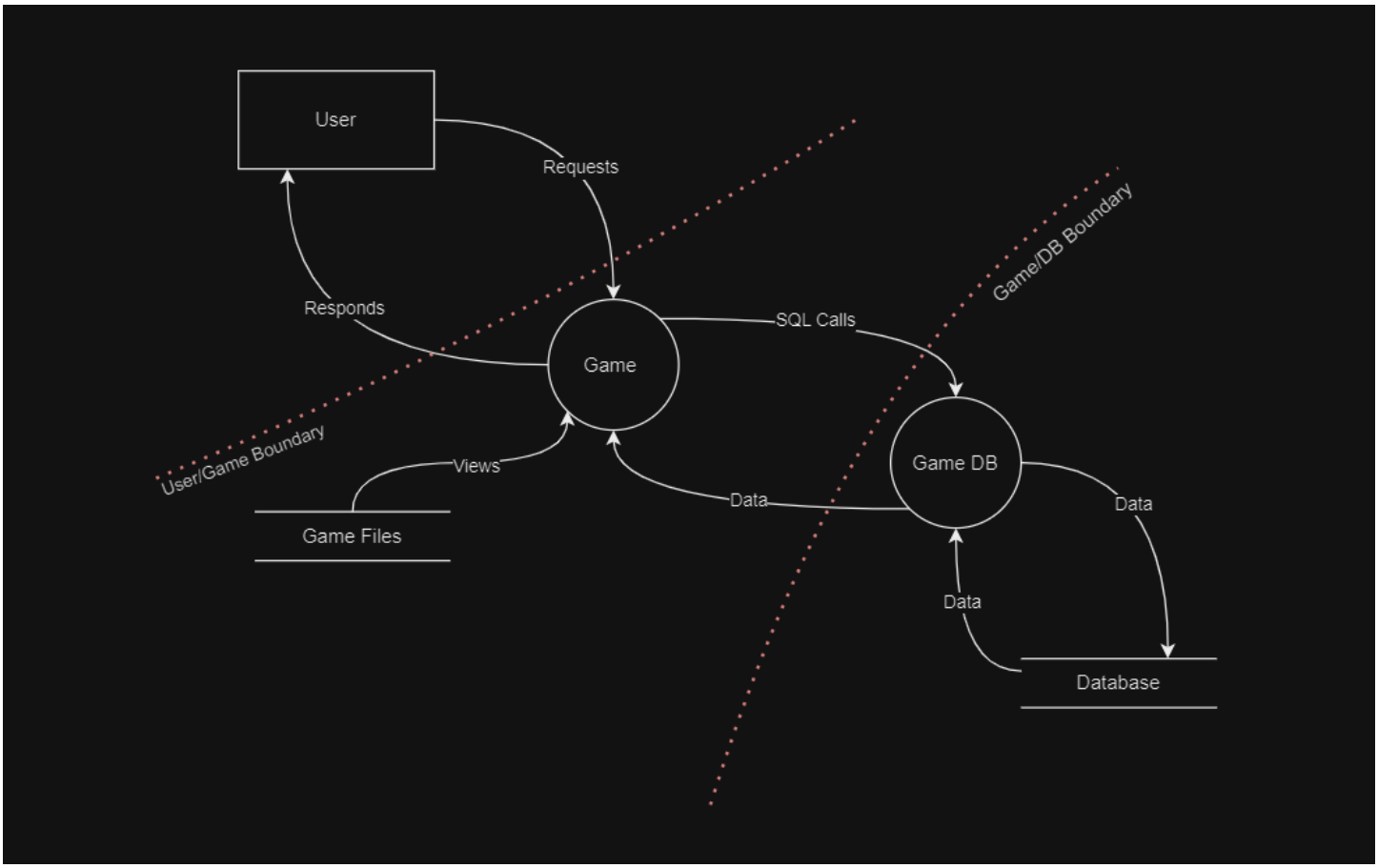


Figure 24: Model showing trust boundaries and security risks.

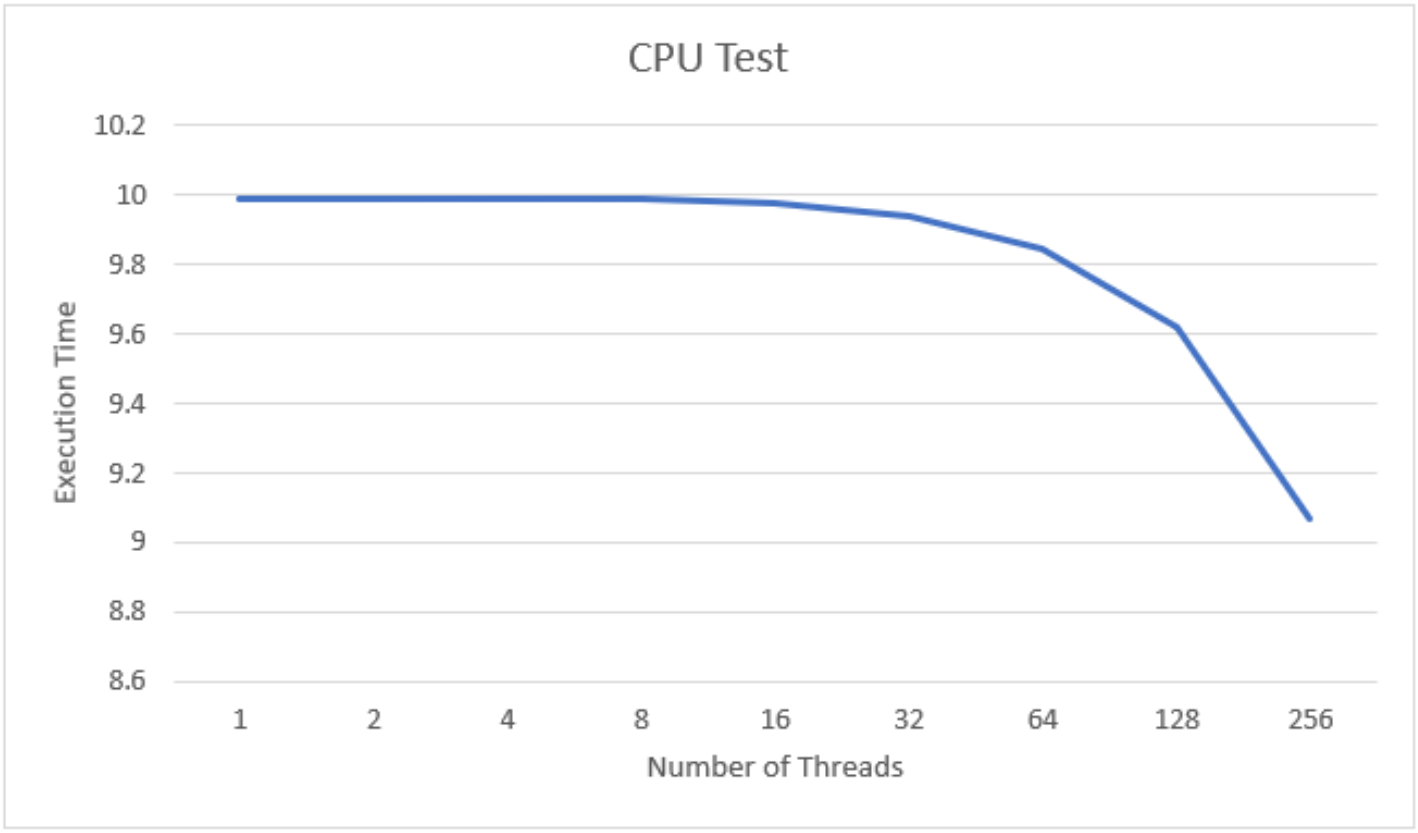


Figure 25: Graph depicting the number of threads vs execution time.

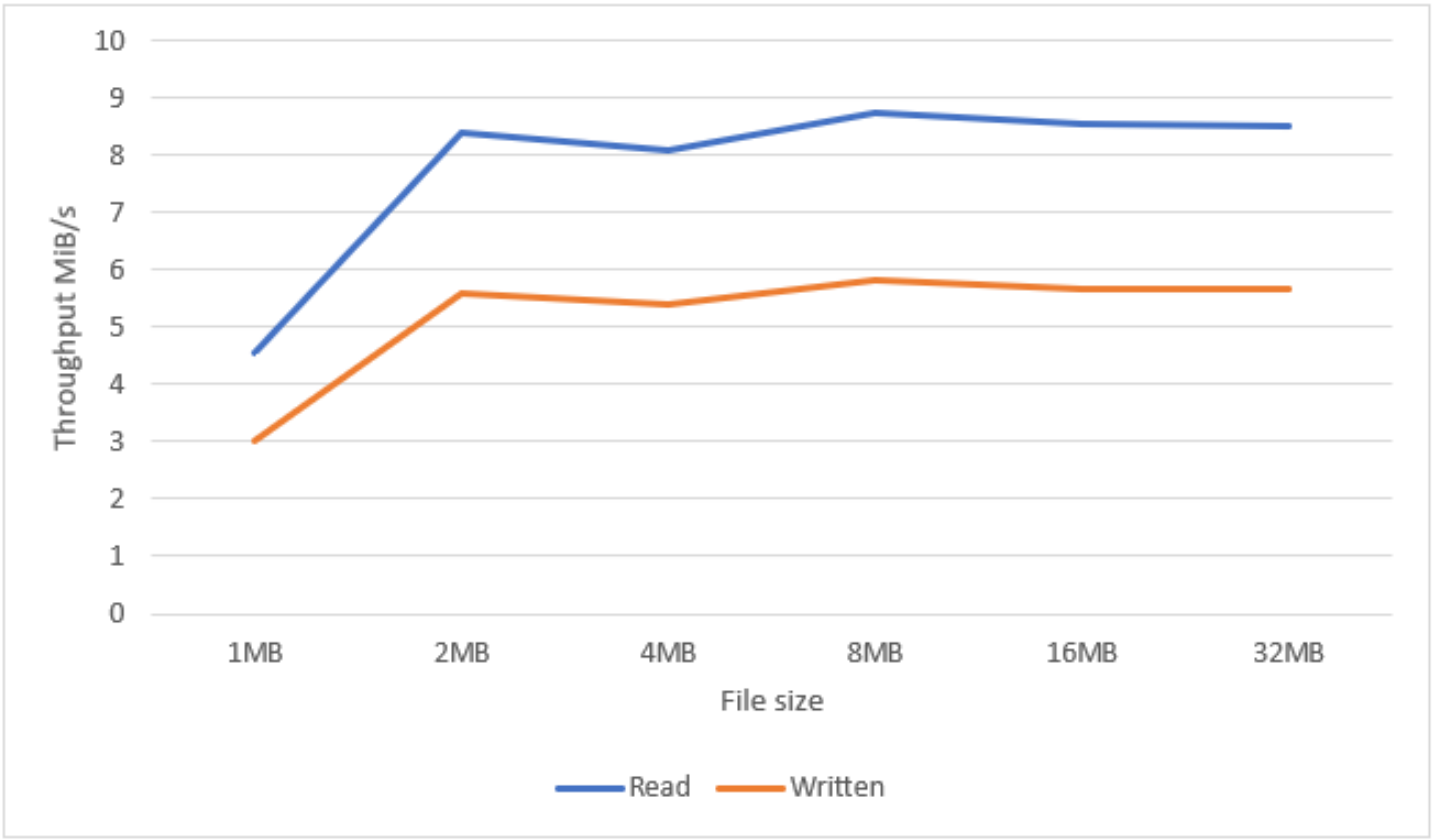


Figure 26: Graph depicting file size vs throughput.

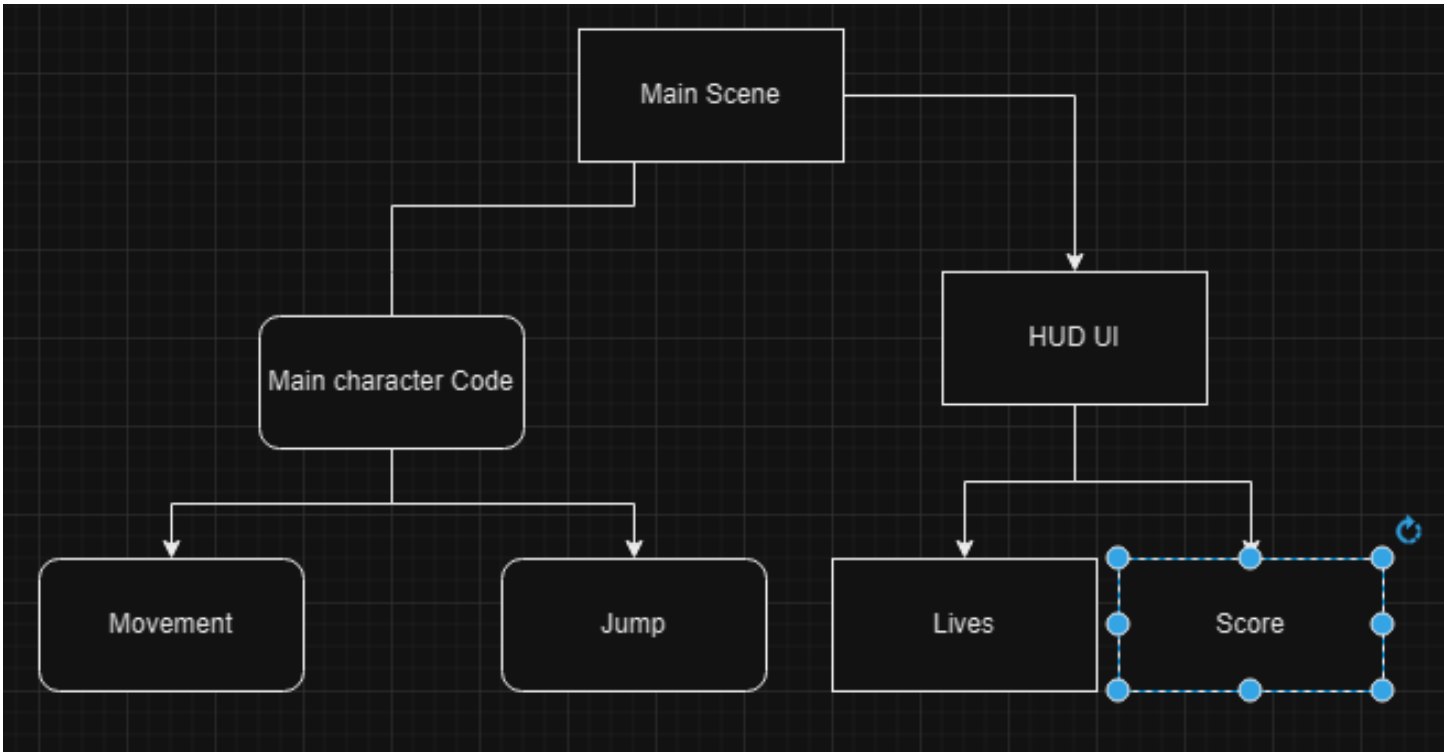


Figure 27: Integration testing visual.

STEP ID	STEP DESCRIPTION	TEST DATE	EXPECTED RESULTS	ACTUAL RESULTS	PASS / FAIL	ADDITIONAL NOTES
1.1	Register an account.	11/30/2023	You will register an account successfully.	You register an account successfully.	Pass	This will take you to the start scene.
1.2	Ensure that you cannot re-register with the same email.	11/30/2023	You cannot re-register the same email.	You cannot re-register the same email.	Pass	
1.3	Allows for multiple accounts	11/30/2023	Allows you to register more accounts.	Allowed you to register more accounts.	Pass	
2	Login with existing account.	11/30/2023	You will login successfully.	You login successfully.	Pass	This will take you to the start scene.
3	Check database for added user data.	11/30/2023	Once registering, user data should be added to the PlayFab database.	User data was added to the PlayFab database.	Pass	
4.1	View leaderboard.	11/30/2023	You can view leaderboard.	You can view leaderboard.	Pass	This generates user saved scores.
4.2	Check database for leaderboard data.	11/30/2023	You can see leaderboard data in PlayFab database.	You can see leaderboard data in PlayFab database.	Pass	
5	Check if game is user friendly.	11/30/2023	Game is user friendly.	Game is user friendly.	Pass	

Figure 28: System testing table results.