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

Contents

List of Figures

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 created the game with accurate an player sprite and NPC sprites as close to the original as possibe. The player sprite was found through Spriters Resource, the NPCs were drawn in an online pixel art maker by Andrea. We ulitmately decided to use the night edition of the game for the background. The gameplay mechanics are largely button presses, meant to mimic the way the game was actually played on arcade machines. It is as non-reliant on screen touching as possible. The sound effects of the original game were also hard to find, leading us to take creative liberty and use our own created audio for this project. With limited resources on the table, we were prepared to do whatever it took to find the best substitutes to make our remake close to the original as we could.

Our client and audience, Dr. Galloway also informed us to document anything we completed in our project so that he could be sure that everyone in our group has done our part, as well as ensure that the project could be replicated by anyone given the documentation. All members were expected to use Unity at the client's request. We wre 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 about plans of action in order to collide with the buildings without coming into contact with any NPCs. The game was made in Unity and shared with other members of the team.There is a built-in sign in/sign up system for the player to create an account and land their score on the leaderboard. We also took security measures to ensure that the player does not have the power to cheat the game like actual console games do.

1.2 Project Scope

The scope of this project included 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 presentations focus on giving the client an overview of project, progress on project and future plans. The CATME evaluations let 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 got to do this with a team who is also learning as the project progresses. As a team, we expected 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 were to comple the technical and organizational documentations, the presentations, evaluations and the final product. With documents, each team member was given a specific part of the document that they had to make contributions to and complete. The presentations were created based on the technical and organizational documents that we had completed. The evaluations were be done once preparations for the presentation have been made. The development of the final product startes in Sprint 3. The project was completed at the end of Sprint 4.

There were no costs accumulated for the project other than time cost for each team member. We met every week to discuss future plans and evaluate our current work. The time costs were dependent on the arranging of the team members schedules and changed week by week. Our team consisted of Aaron Beasley, Andrea Florence, Shikha Sawant, and Tameka Ferguson. Unity tutorials and resources, YouTube, and class lectures were the necessary resources for this project.

Sprint 1 documentation was due on September 7th. The first evaluation and presentation were due on September 4th. Sprint 2 documentation was due on October 6th. The presentation and CATME evaluations were due on October 4th. Sprint 3 documentation was due on November 3rd, and the CATME evaluation was due that day as well. The presentation was due on October 30th. Sprint 4 documentation was due on December 1st, and the CATME evaluation is due that day as well. The presentation was due on November 27th.

1.3 Technical Requirements

1.3.1 Functional Requirements

Mandatory Functional Requirements
1. Player Controls
a. Move left/right and Jump
b. Accessing a menu 2. Player Lives
a. Player has 3 lives in gameplay.
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
5. Building
a. Disappear upon collision
b. Kill 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

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 needed to be able to move left and right. 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 used AD and the left and right arrow keys for character movement, and the spacebar to allow the character to jump. Another function that was necessary was a way to track player lives. The question we had to ask is "How much damage can they take before dying?" along with this, another functional requirement was the ability to manage accounts. We had to ask "How many accounts can be made for our game? Will this be a single player or multiplayer?" These were 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 were also types of nonfunctional requirements we had to consider given it shouldn't be too large for our client's computer or ours. It was developed

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

in the game engine called Unity since that is what the client requested. The non-functional requirements became clearer the longer we worked and developed the game. We took a bit of leeway in character movement, and in the actual design and graphics of our game and characters within it.

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 didn't begin thinking about the technical requirements until further into the second sprint. We based 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 were using windows to access Unity, so we also based the details on this fact.

For our development team, the necessary operating system (OS) needed to run our game was Windows 7 and newer, 64-bit versions only. The CPU had to be x64 architecture with SSE2 instruction set support. The Graphics API had to be DX10, DX11, and DX12 – capable GPUs. Additional requirements were that the members had 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 so we were fine from that point.

For players not running the game from the Unity engine, the system requirements are 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. 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 decided to use to program our C-sharp scripts was Visual Studio. Since we used Unity, the default script editor is Visual Studio. All of our team members were also already somewhat familiar with Visual Studio, so it worked best for us to continue with something we already had familiarization with.

Another option was to use an IDE plugin that has Unity support built in. We considered using JetBrains Rider, which is a C-sharp editor. JetBrains Rider makes writing C-sharp a lot smarter. Since, none of our members had experience in C-sharp, JetBrains Rider could have been very helpful in enhancing our skills and acting as a good IDE plugin. The software language that we used in the development of our project was C-sharp since this is the language that Unity used to build games with. Our version control will be Git. Not all of our members were familiar with Git or GitHub either so that was a learning experience for most of the team. We used an online pixel art creator to create some of the assets to use in development within our sccenes.

We used Discord to communicate and share documents for each sprint. Discord is a very convenient social platform that made both connecting and conveying information between our team members more efficient. As we moved onward into the project and began using Unity, we decided to use GitHub for version control to ensure that all team members were aware of changes in our game development and were able to build on the newest edition of the game as it was developed. We commited changes to GitHub as we were editing the Unity file so that we all have the latest version of our software.

2 Modeling and Design

2.1 System Boundaries

2.1.1 Physical

The hardware components and resources that were used for the software system were the operating system, the CPU, the graphics API and disk space. For android the OS version needs to be 4.4 (API 19) + and for iOS 11+. For android the CPU needs to be ARMv7 with Neon Support (32-bit) or ARM64 and for iOS A7 Soc+. The graphics API needs to be OpenGL ES 2.0+, OpenGL ES 3.0+, Vulkan for android, and Metal for iOS. Then for both android and iOS, the user needs need 80GB of free disk space. For the deployment environment, our software is meant to be contained on the user's PC. Since our project is small, we kept everything contained to the user's device. For network/communication, if we had branched out to have different servers later on we would have used latency optimization to minimize network latency and ensure smooth gameplay for users. However, it did not come to that at any point in the development of this project. For security, we have implemented authentication and authorization. With this we created a login system that uses a database that ensures that users are authorized to play the game before allowing them access to gameplay and the leaderboard. We did not focus too much on scalability for this project as we knew we would not be launching our product on a larger scale.

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 had to focus on how we wanted our game environment set up; As a group we had to think about how we wanted to set up the street, background, and structure of the environment as a whole. Positions and health scores are what we had 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 wireframes, we created one for our start menu as well as one 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. We also 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 the start is clicked, the user is thrown into the playing scene where they lose if they lost all lives 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.
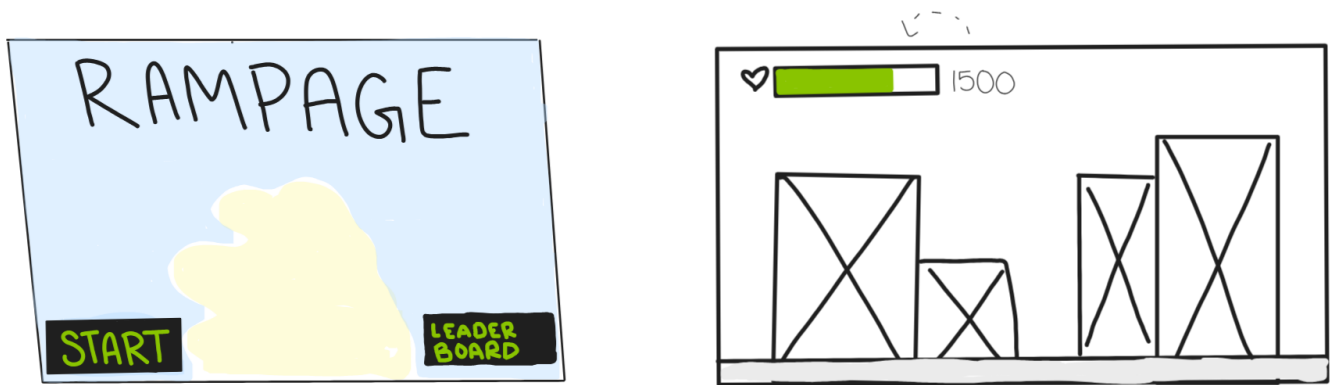
Figure 1: Image of wireframe.

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 had many objects that we needed to monitor. We have many NPC objects and building objects that 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.
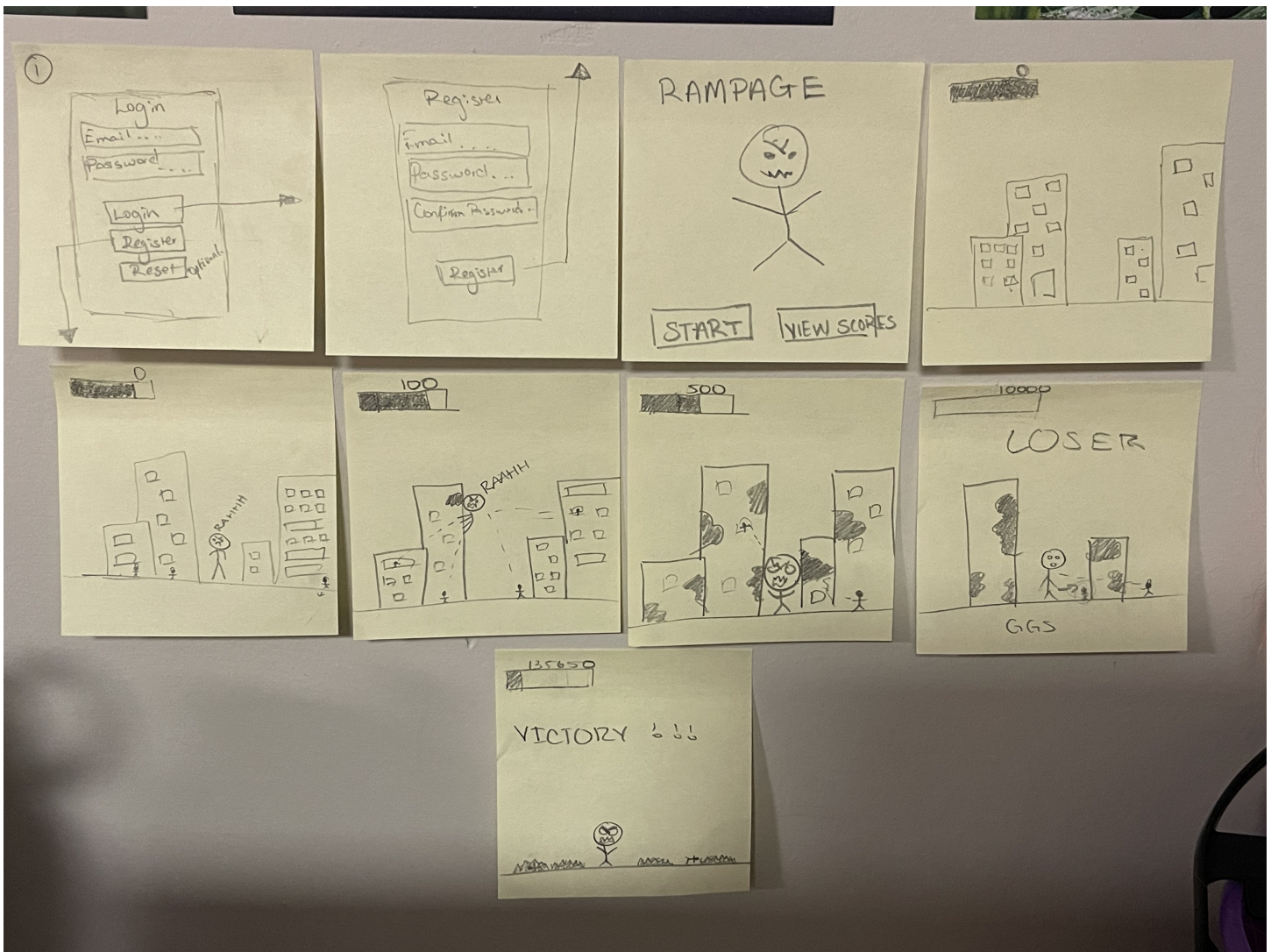
Figure 2: Image of storyboard.

We chose to sketch out the façade diagram since we only recreated 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 much avoid the soldiers, tanks and helicopters in order to live and destroy, thus collecting points. If the player
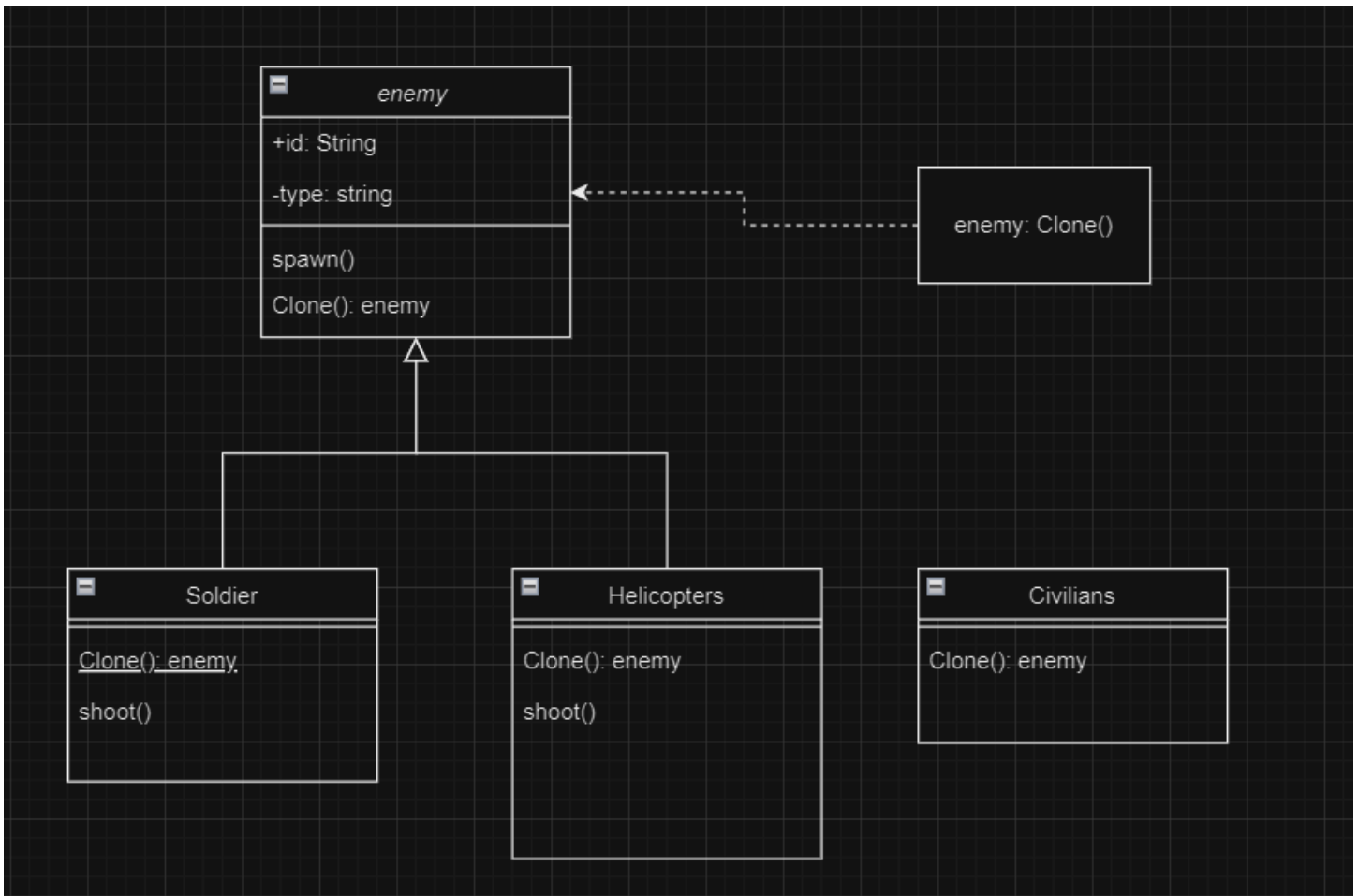
Figure 3: Prototype diagram showing how most of our

Figure 4: Observer diagram shows

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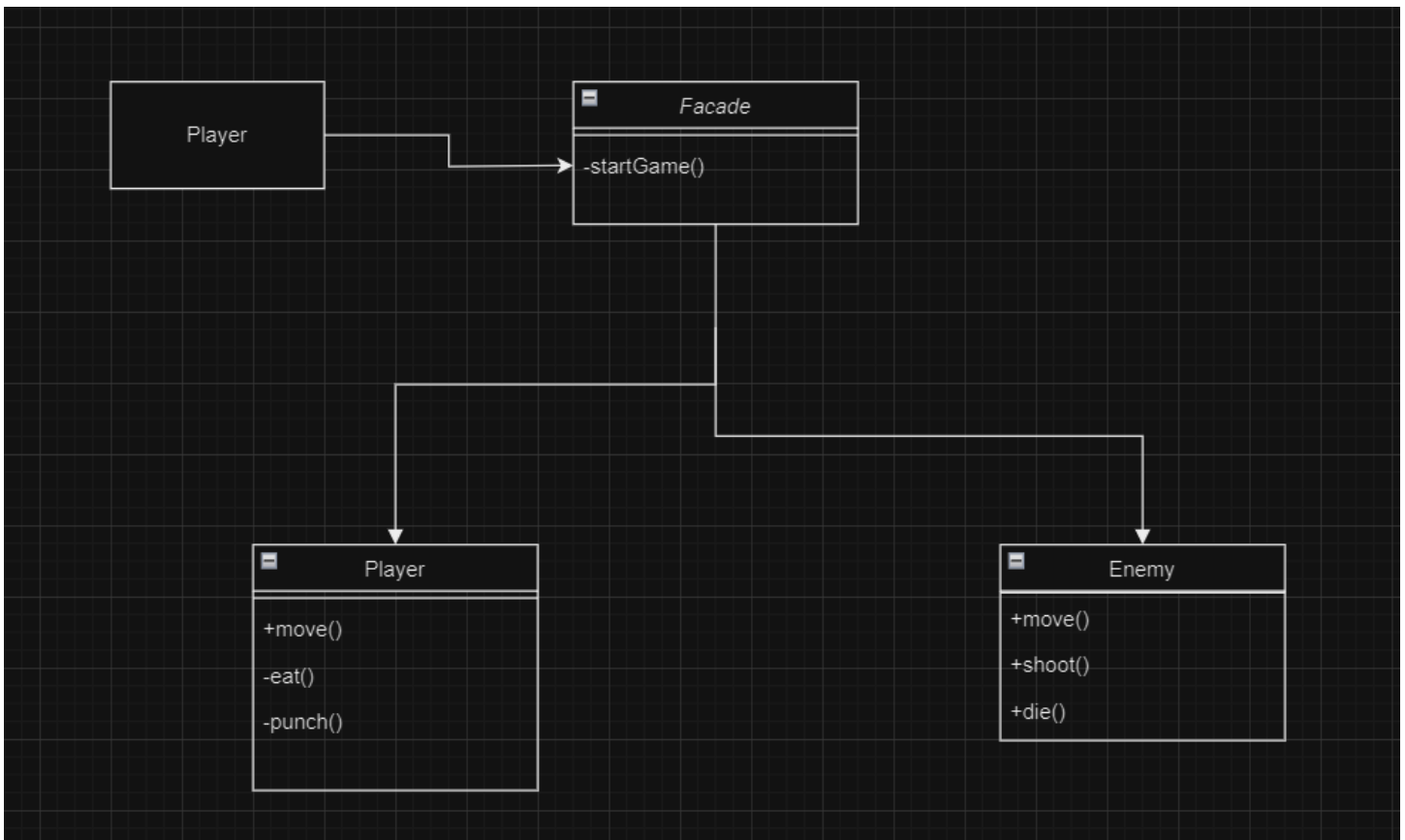
Figure 5: Façade diagram shows the relationship between the player and enemies.

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.

2.3.6   Component Diagrams

Title: Game Aspects Component Diagram
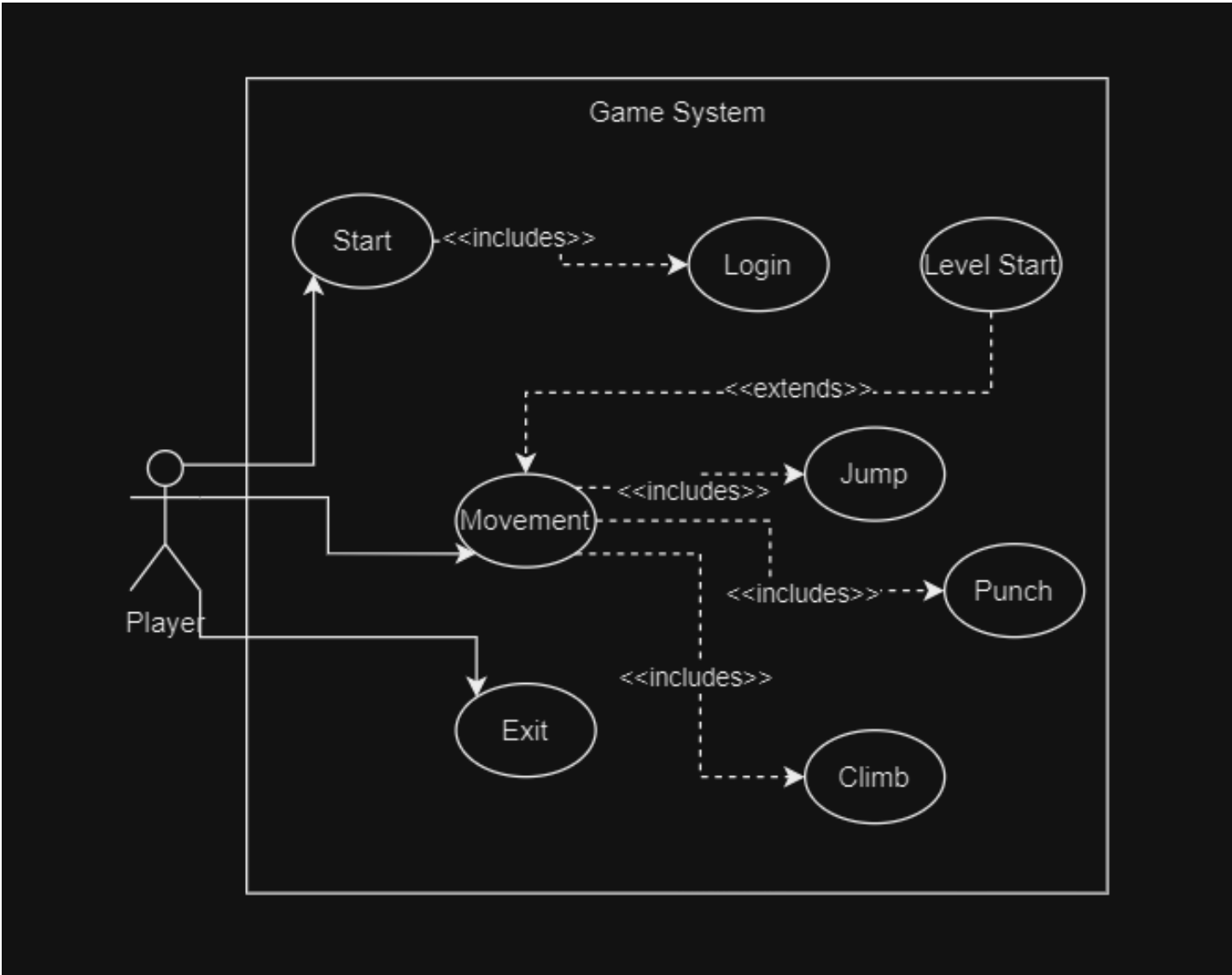
Figure 6: Use case diagram showing the player interactions with the game.

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.

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

	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

component, and provides an update on their current status. At the time of project completion, all sections should be completed.

10
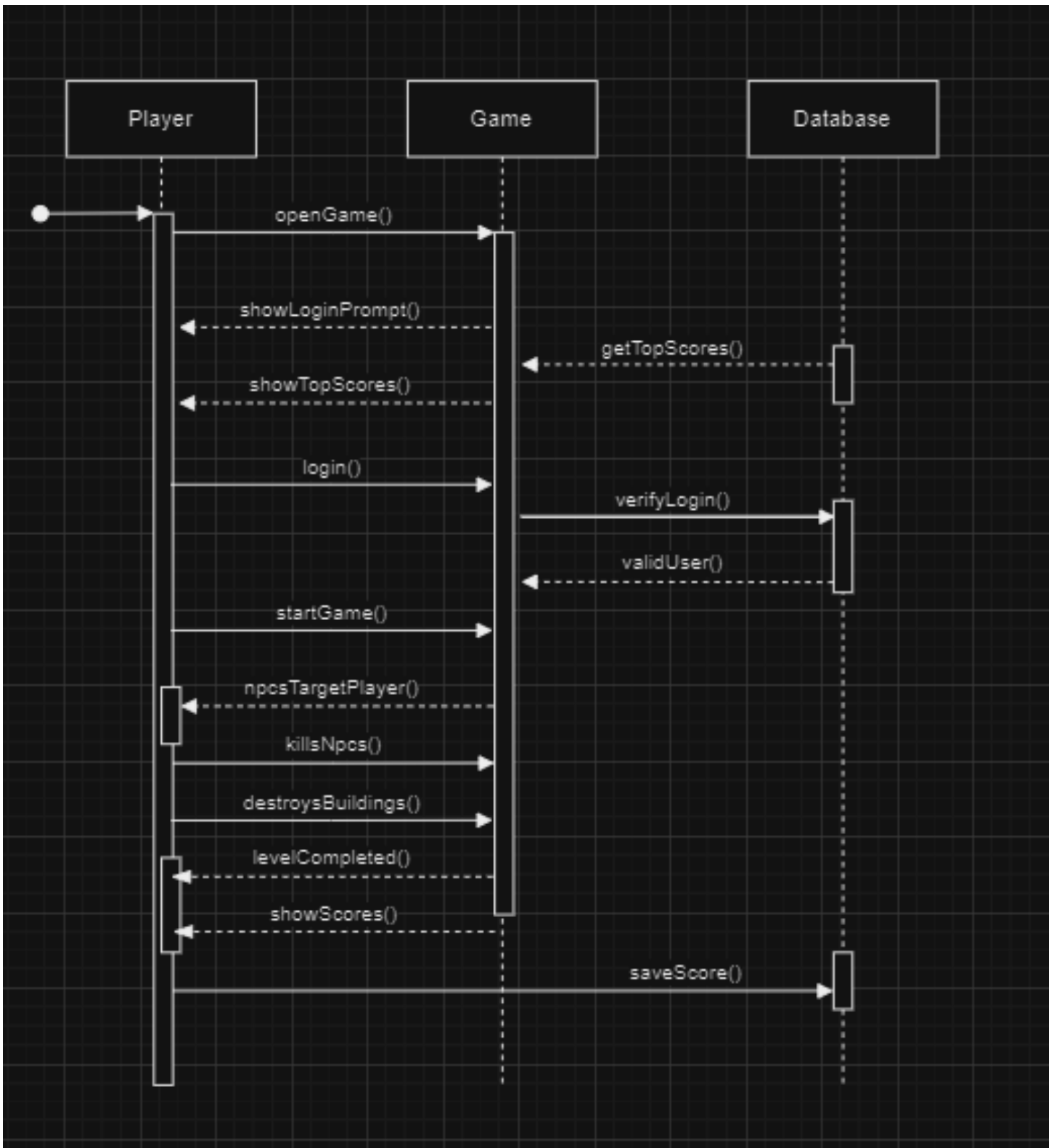
Figure 9: Sequence diagram on the overall software function.

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
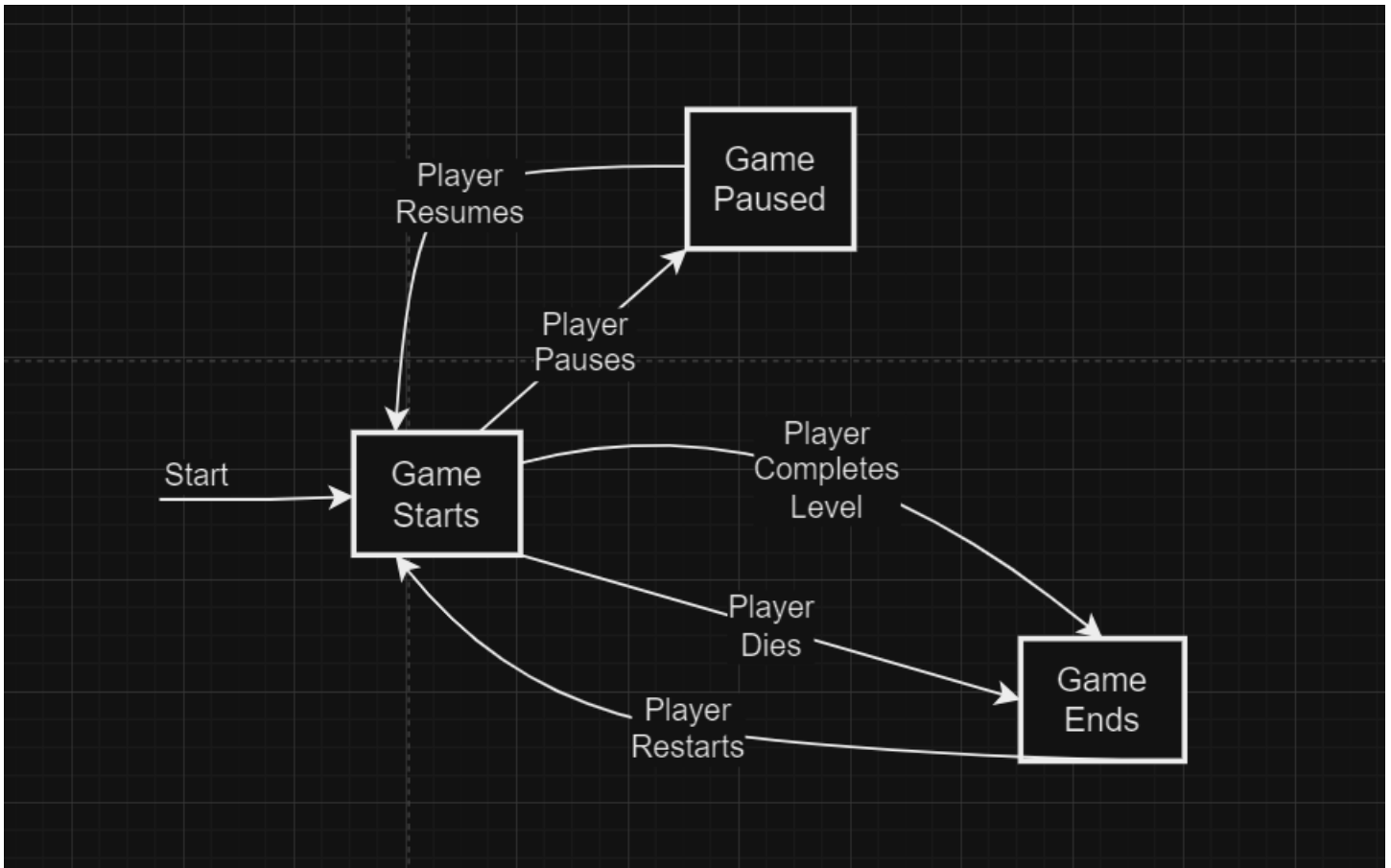
Figure 10: State diagram on the overall game state.

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

We usually made commit changes to the repository after we had completed a big section of our task for the project or when we were done working on it momentarily. We named our commits based on what we were working on or had completed in those changes. We used the GitHub desktop that is connected to the repository, and create changes through that. We all worked 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 without intervention.

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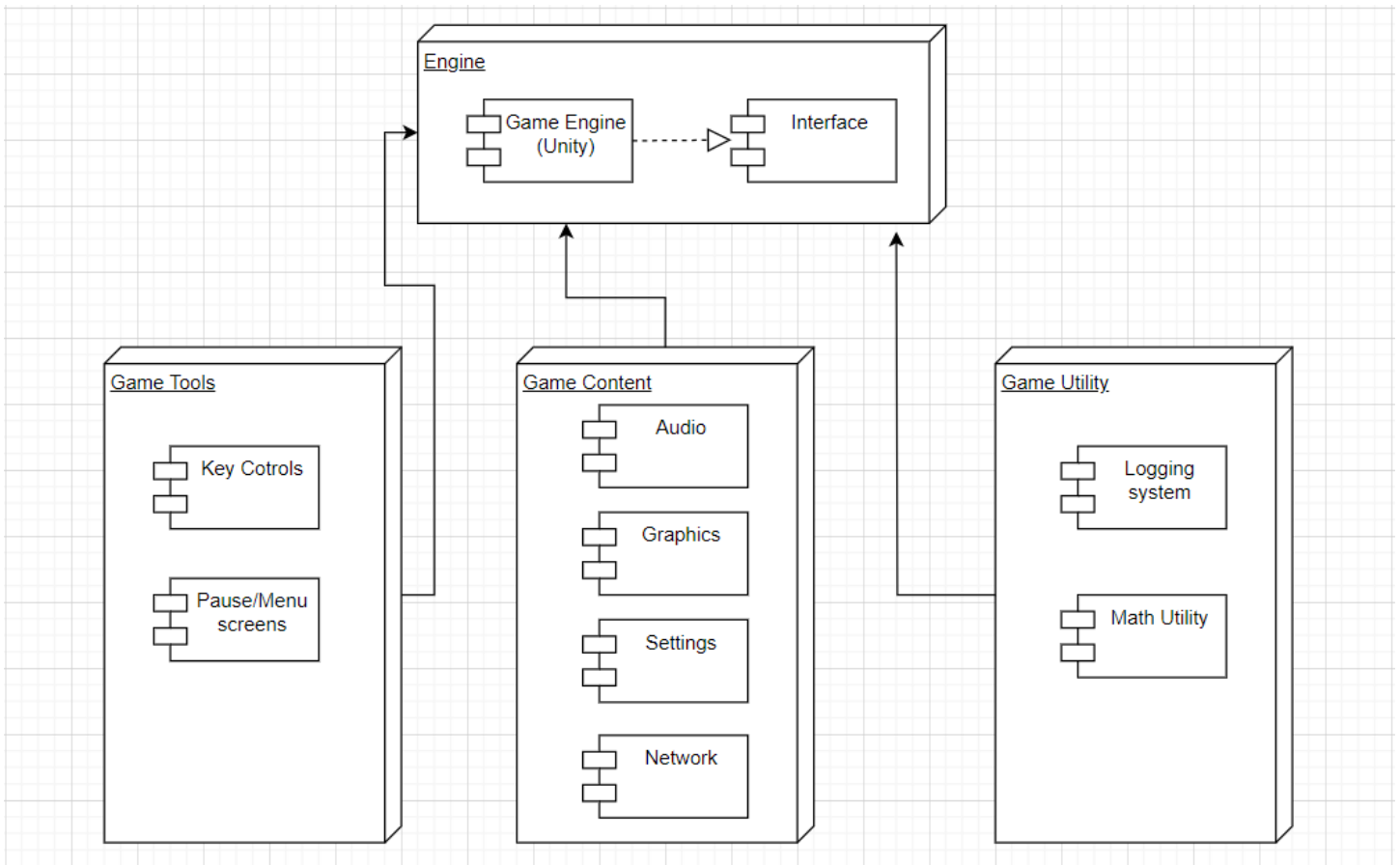
Figure 11: Component diagram showing the how aspects of our game feeds into Unity and interfaces.

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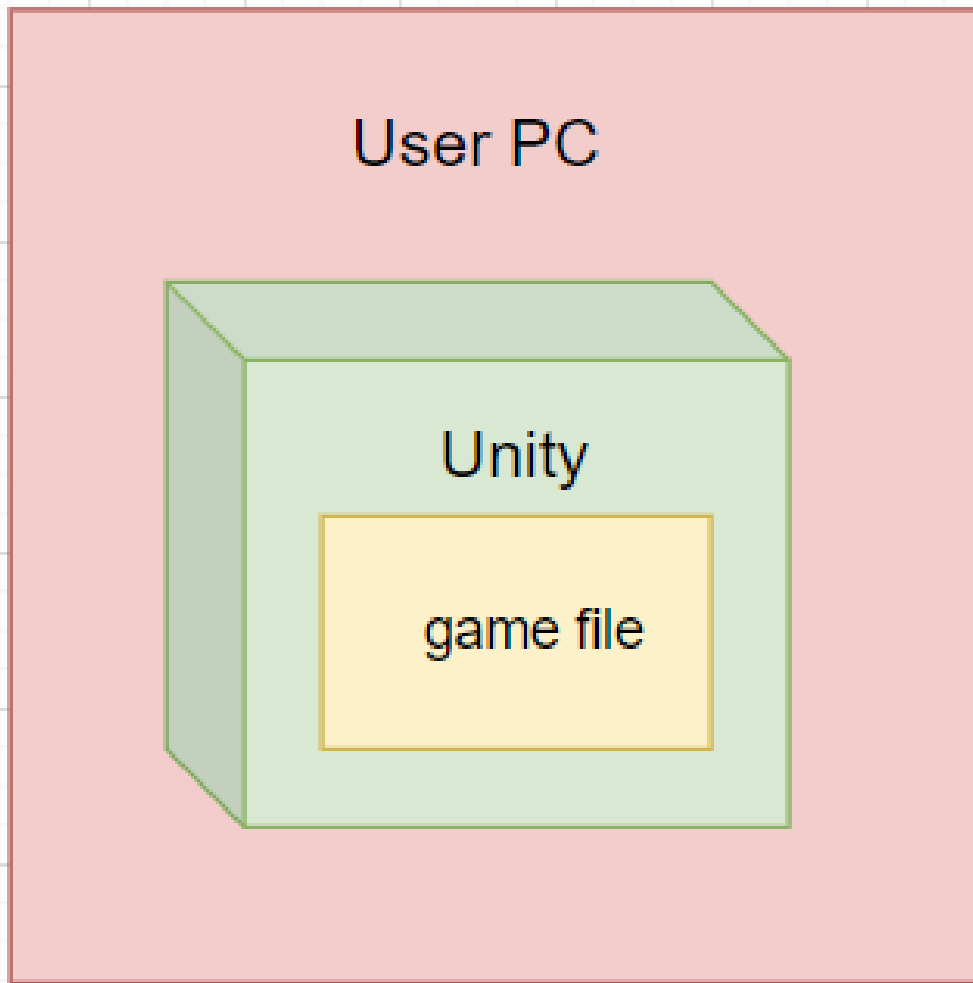
Figure 12: Deployment diagram showing how our game is deployed.

fly. They need to strategize how to destroy the buildings without coming into contact with the NPCs near them. If they come into contact with an NPC, they lose a life so they need to avoid them as much as they possibly can. Playing this game will keep their intellect and reflexes sharp as they move throughout the game and destroy the buildings in order to gain all available points without losing all of their lives in their efforts.

2.7.3 User Skillset

The user is 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 need to develop strategies as they play and use quick reflexes to avoid losing their lives coming into contact with enemies. The user needs 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 lives depletes to zero they lose; they only have three lives. The rules are as such: the player can destroy buildings by finding the collision boxes built into them. They do this by jumping around the building. The entire time they must avoid enemies. There is no way to replenish lost lives. The user can move to the enemy and jump on them to kill them – but coming into contact with them will still take a life so they must use this strategically. There is not multi-player funcitonailty so they cannot team up with someone to divide and conquer or race against. It is simply them vs. the enemies until they either destroy all of the buildings or lose all of their lives.

2.7.5 Gameplay Items

The user does not encounter any health tokens or powerups in our game. There are no coins, loot, or character upgrades that can be made. The user remains 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 is 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, 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 amazinggggggggg :D	Pixels

Figure 14: Data dictionary table game objects.

2.7.7 Gameplay Menu Screens

The first screen the user sees 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 fill in their email and password in that login screen and hit "Register and Login" and their account has been created. Once they login, they see the main menu screen. Once here, they see two options. Option one is to see the leaderboard, which takes them to the leaderboard scene and then they come back to the main menu. The second option is to start the game, which takes them into the game. Once the game is over, they will see either a win screen with a congratulatory message via audio, or they will see a lose screen that declares that they are, in fact, a loser. Both end case screens show "Game Over" and instruct the user on how to either play again, or logout. Screenshots of screens we have completed are attached.

2.7.8 Gameplay Heads-Up Display

Our heads up display is extremely simple. The user sees their score in the top left corner of their screen. If the score hits three, they win the game. The player sees their lives in the upper right hand corner. The lives start at three and count down each time they come into contact with an enemy NPC. These are texts that stay on screen

16

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.

throughout gameplay; there are no images, bars, or hearts implemented within our heads up display. 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 were 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 enemy NPCs will be basic pixelated drawings of soldiers. George himself is simple ape, and doesn't rotate - he just moves side to side and jumps. There are simple animations for walking and jumping, and he flips when the player walks in different directions.

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 gave us permission to use custom audio in our game. Our background music is two of our

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.

members humming and beatboxing together. There are gun sounds eminating from the soldiers. The win and lose scenes contain a cappella music pertaining to either winning or losing done by one member and someone outside of the group. Each member also did various ambient sounds like buildings breaking and gun 'pew's. These audios were attached to the appropriate sprites and scripts for each scenario, and layer with each other well when needed. These audios were created/recorded on Andrea's phone and compiled into a folder for the game to utilize. They are 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 are 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 lo-

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.

gin/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 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.
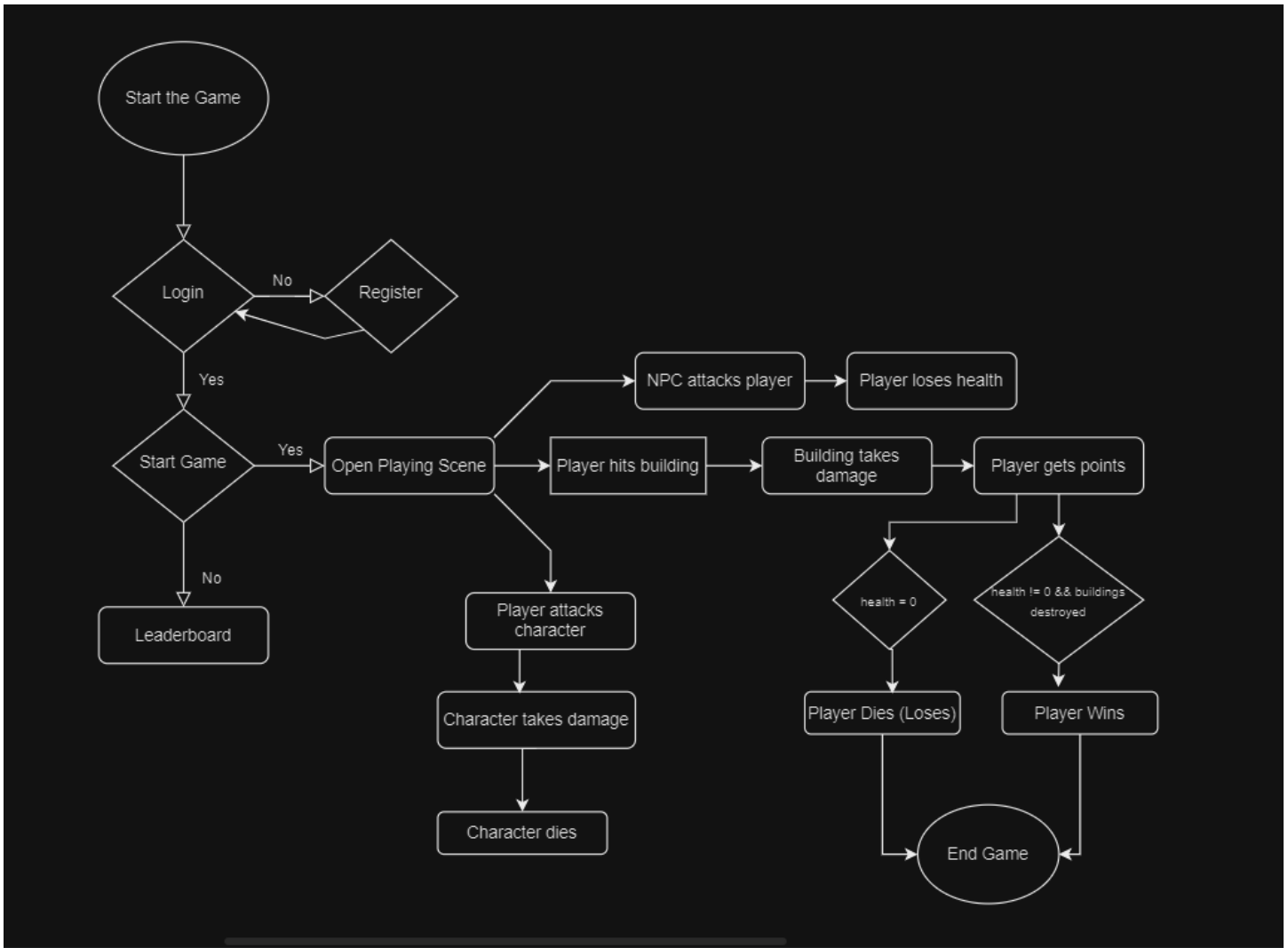
Figure 19: Diagram of Rampage Gameplay.

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 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
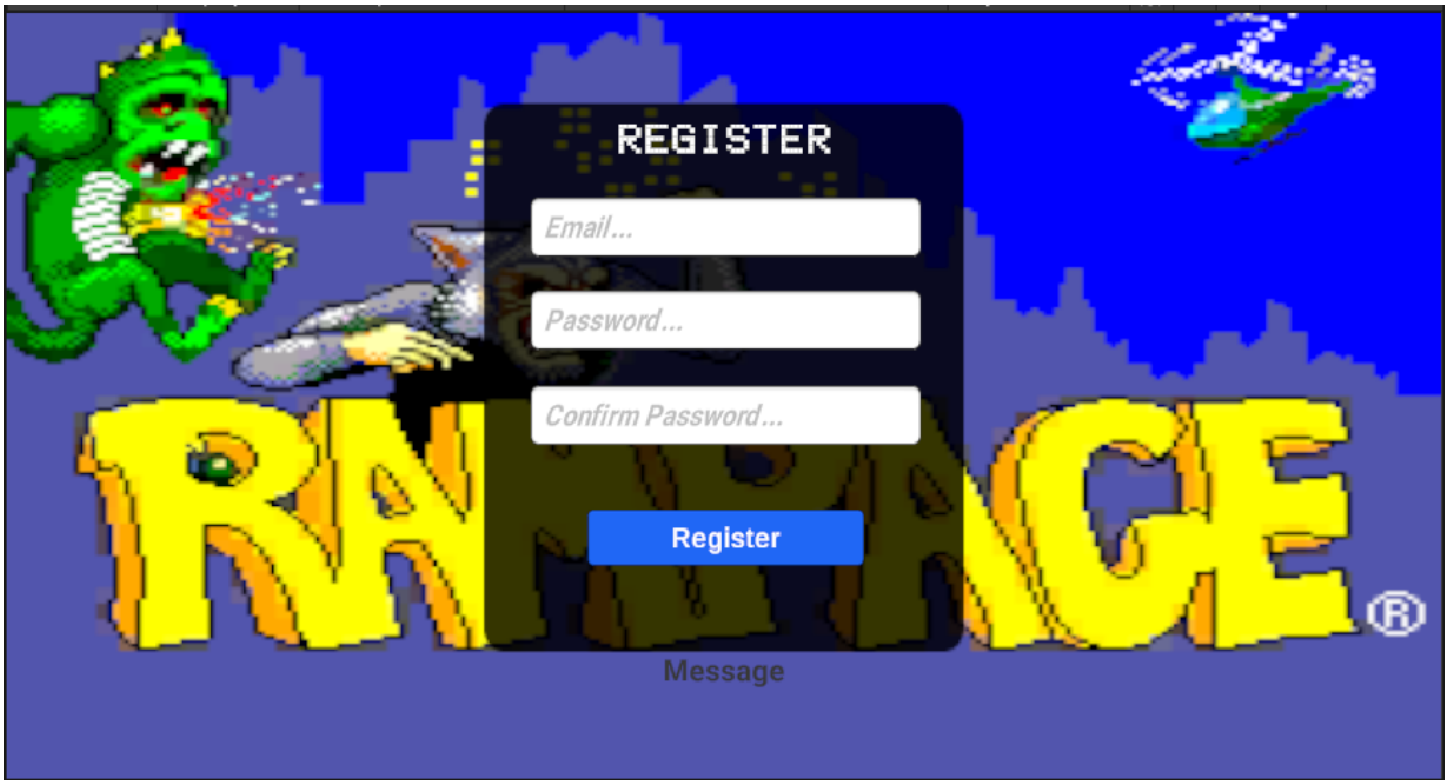
Figure 20: Image of Register Screen.

Figure 21: Image of Login Screen.
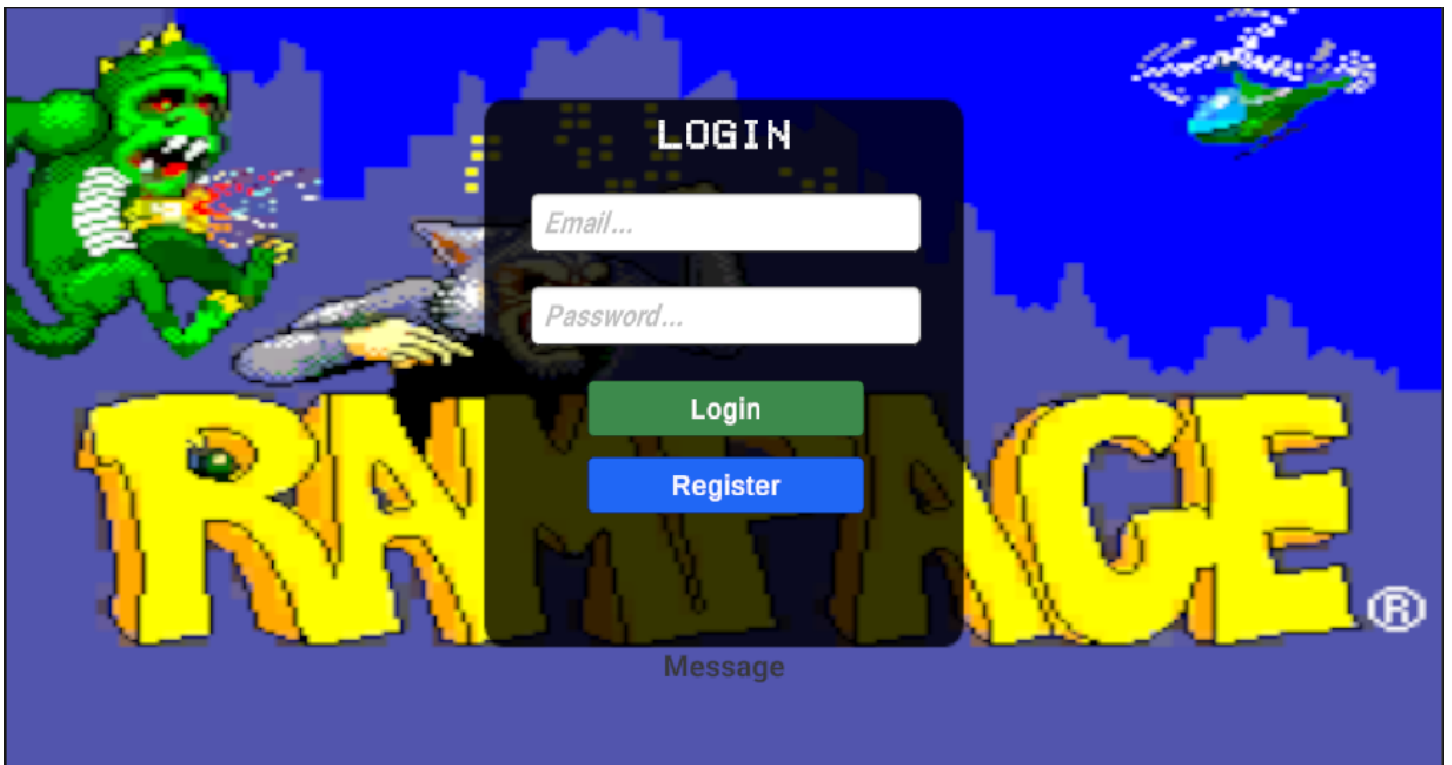
would allow them to cheat or steal vital user information.

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 are only two types of users. We have the administrators, which are our team members, and the player, which are 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, 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 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 is at least 70% at best. The operating system version for the player to play the game is 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 CPU needed for the game is 15 execution time but it sits at least 10 seconds in execution time. Analyzing and monitoring the software every few minutes also helps 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 is 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 was 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 were 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 also decided to look up YouTube videos to help give a guide on how to program the game. The main platform that the game is available on is the PC so to make things easier, the game is made with the control scheme of the keyboard. The next goal for the project was to go all into programming more complex parts of the code as well as doing more animations and making the NPCs functional. Sound was also be implemented at the same time as the other scripts were programed and there were ideas of having actual people doing the voices instead of just relying on sound effects.

3.2.3 Application Workload

The user generally spends about 5% of the game on the menu to at least start the game. About 75% of the user's time is spent in the actual game - destroying buildings and avoiding enemies. The login takes up at least up to another 5%, it does 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 is less than 10 seconds. The application for the game is mostly the one thing that is running on your computer for the best performance the user desires.

Many applications not related to the game is more than likely cause bottlenecks in the background because they will cause either uneven CPU bottlenecks or GPU bottlenecks. They are running the software or on a hard drive if one possessed it. The loading time to get from one part of the game to another is also short and takea less than 10 seconds to load. The use of settings in the game depends on the user, but the user would likely spend another 5% on settings. 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

Test Plan Identifier: 001-Acceptance

Introduction: The goal of the acceptance test is for our client to test the product and decide if it has been completed to their standards. The client was provided with a table of test cases and a column to mark as a pass, and a column to mark it as a fail. The objective is that the client is satisfied with the product and the development phase ends, moving forward into the maintenance of the product in the future.

Test item: The test item is the software as a whole. Each scene is tested as a fully integrated system, with all audio, scripts, prefabs, etc. Each component of the system should be tested to ensure that the whole thing works properly and to the level the client desires. The start scene, login, register, playing scene, win, and lose scenes should all interact with each other as desired.

Features to test/not to test: All features of the gameplay will be tested in the acceptance testing. Out of scope features are not tested but all features that are in the scope are tested. Character movement, scene changes, leaderboard functionality, login functionality, register functionality, and the win/lose conditions. That includes ensuring that the NPCs disappear when the player comes in contact with them and a life goes down. In turn, when the player touches the collision box on the buildings, the buildings disappear and the player gains a point.

Approach: The approach to the acceptance testing is to bring up the frst scene for the client and allow them to play through the game a few times to test different features. First, the client will test registering and logging into an account on the game. They will then test accessing the leaderboard and getting back to the main menu. Once back to the menu, they will play through the game and fulfill the winning conditions. After the win scene occurs, they will logout and try to register using the account they already created to see that it will not let them do that. They will then login with the account they made instead of registering it, and play the game again. This time they will fulfill the losing game conditions and see that the losing scene occurs. Throughout this, the client sees that the audio works, as well as each function we provided in the test case pass/fail table.

Test deliverables: The deliverable for this test is a table that contains three columns. The first column is the test case to be performed. The second column is to mark a check if the test case passed, aptly named "Pass". The last column, "Fail", is where the client will mark an X if the test was failed. The entire chart is completed at the final client/team meeting.

Item pass/fail criteria: Each test case involving moving from scene to scene requires that the client is able to move from one scene to another using the appropriate buttons/key presses wihtout issue in order to pass. If the client cannot do that then the test case failed. The testing of the player movement requires that the client is able to use AD or the left/right arrows keys to move George from side to side, and that the space bar makes George jump. If those actions occur as listed, the test is passed; otherwise it is a fail. For audio tests, if the audio plays without stopping, glitching, or beginning late then the test is passed.

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. Beyond that, there are no other infrastructure requirements.

Responsibilities: It is the responsiblity of the team to bring a laptop containing the game/game environment to the client meeting for the client to test with. It also the responsibility of the team to bring a chart containing test cases and space to mark them as pass/fail for the client to use as they test. It is the responsibliity of the client to check the test cases against what is occuring as they move through the game and mark them as pass or fail; it is not the responsiblity of the team to check behind the client.

Staffing and training needs: There is no staffing need for the acceptance testing. The training needs are those of understanding basic gameplay and gameplay mechanics. The client should have an understanding of game win and lose conditions, where to watch for those conditions to change, and how to navigate account creation, logins, and a leaderboard.

Schedule: The acceptance test will occur at the final client/team meeting of the semester. The meeting will take place on the twenty-eighth of November at three-thirty pm. The meeting is not expected to last any more

than thirty minutes; at maximum the meeting could last to forty minutes until completetion of the acceptance testing.

Risks and Mitigation: There are many risks within the acceptance testing. At any point a file could corrupt, which has the potential to derail the entire project. If a scene script corrupts that could entirely halt a section of gameplay and lead to multiple failed test cases, which would produce an overall failed product that does not meet the standards the client set forth. To avoid this, we have the project on a shared GitHub repository so that anyone can bring up the game on their laptop and switch it out if need be. Another risk that we face coming into the acceptance testing is that the internet is not working correctly. If the internet is not working, we cannot pull our project off of GitHub desktop to present to the client. Even if we could, our database could not function without the internet. If this does occur, we will use a phone from the group to start a hotspot and connect the laptop being used to that hotspot. This would allow us to pull the project off of GitHub and run it, as well as ensure that the database will be able to connect and function properly.

Approvals: The client, Dr. Galloway, approved of the acceptance test on November 28th. The team accepted the write-up on the acceptance test on December 1st.

4.5.1 Acceptance Tests and Results

5 Conclusion

The purpose of the technical document for this project, Rampage, is to ensure a complete understanding of how the project has been designed and created. This ensures that should anyone new come onto the project they could be completely up to date by reading this documention, as well as ensuring that should the project ever need to be replicated again that it can be. This document includes all of the design patterns, creation information, source code, etc., that would be needed to completely replicate this project. Truthfully, the real outcome of creating this documentation thus far has been a deeper understanding across the team of what this project is meant to accomplish – the scope, the functionality requirements, etc. It has also been a great way to check that we are building the product the way that we should be. While creating this documentation, we would check our designs and plans for implementation against what the client was expecting. That really enabled us to keep ourselves on that designated track. In terms of the team's shortcomings, there were a few that cropped up in sprints three and four. We found that not everyone on the team had a strong grasp on how to navigate and use the Unity game engine for development, which caused issues at every point along the way. We also found that not everyone was familiar with C Sharp; we hadn't planned on that and thus had not allocated ourselves enough time to learn it as thoroughly as needed. The last shortcoming that seemed to have a stronger impact on both the team and this project was a lack of necessary and thorough communication. Without clear and proper communication, portions of the project were harder to integrate together, and the expectations of each member were not always clear. If we were to continue this work in the future, we would need to target these issues before continuing on to ensure that they would not stand in the way of creating the best product as possible for our client.

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
```

```csharp
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)
{
```

```csharp
        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)
    {
```

```csharp
        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>();
    }
```

```csharp
// 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;
    }
```

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


}
```

6.3.6 ScoreManager.cs

```csharp
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);

    }
}
```
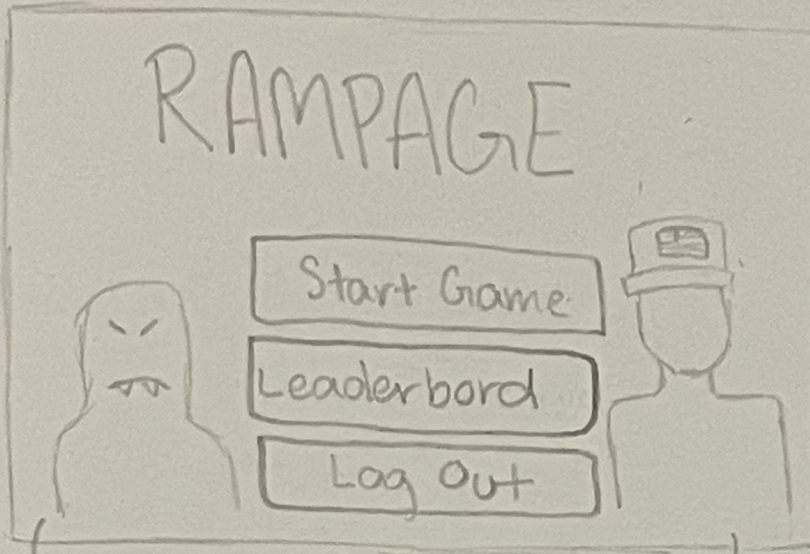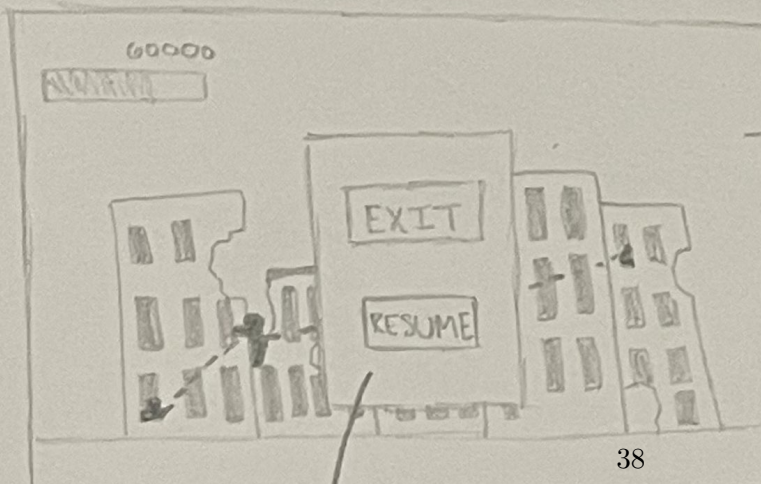
main menu

RAMPAGE

Start Game

Leaderbord

Log Out

↳ George

enemy
NPC

pause menu

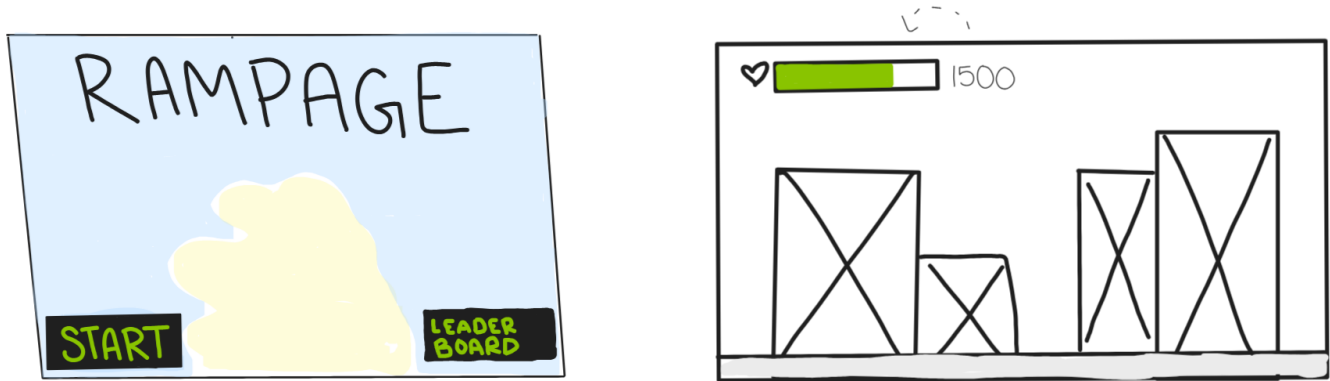60000

↳ game frozen
in background

EXIT

RESUME

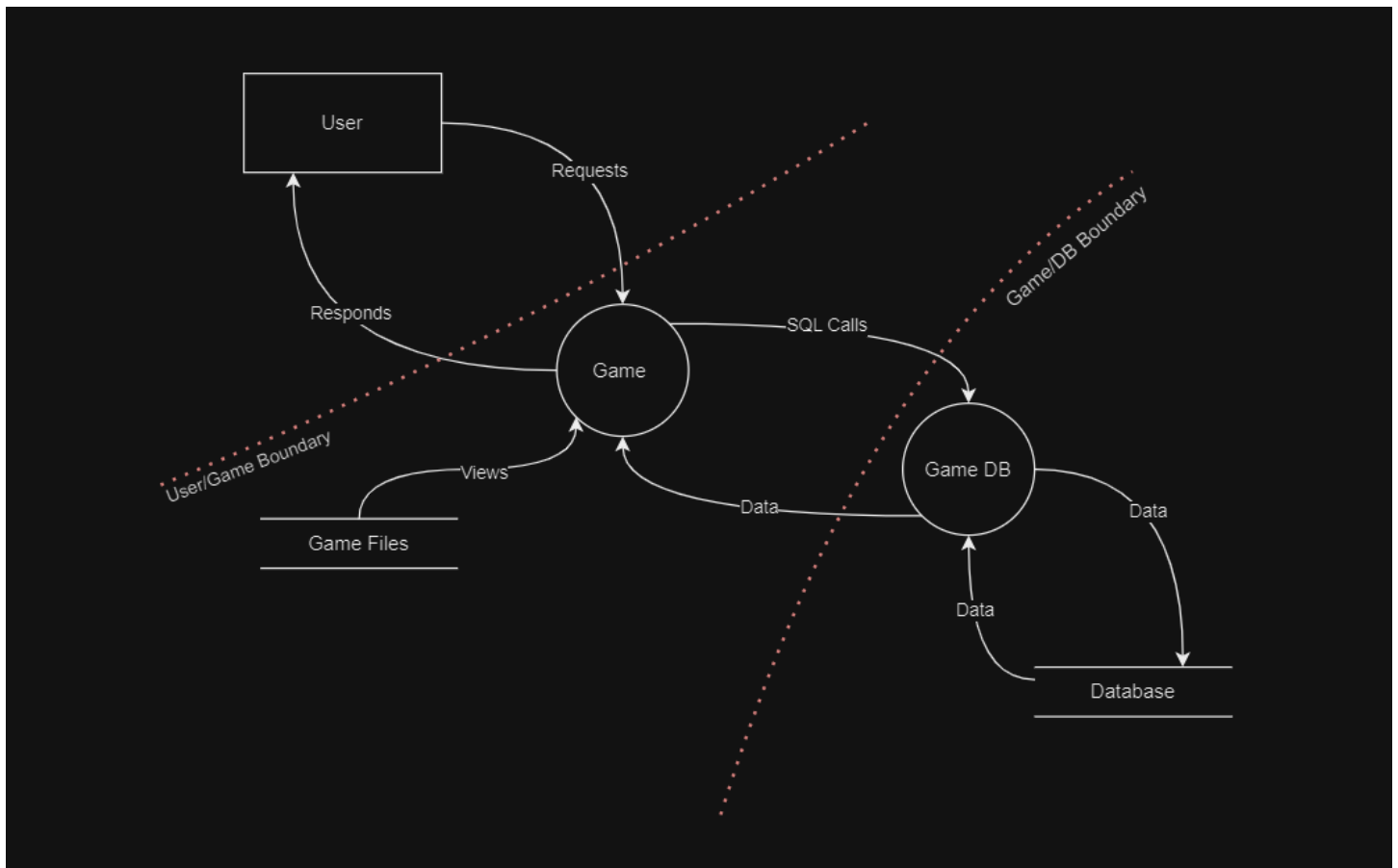Figure 23: Images of the wireframes for main menu and gameplay.

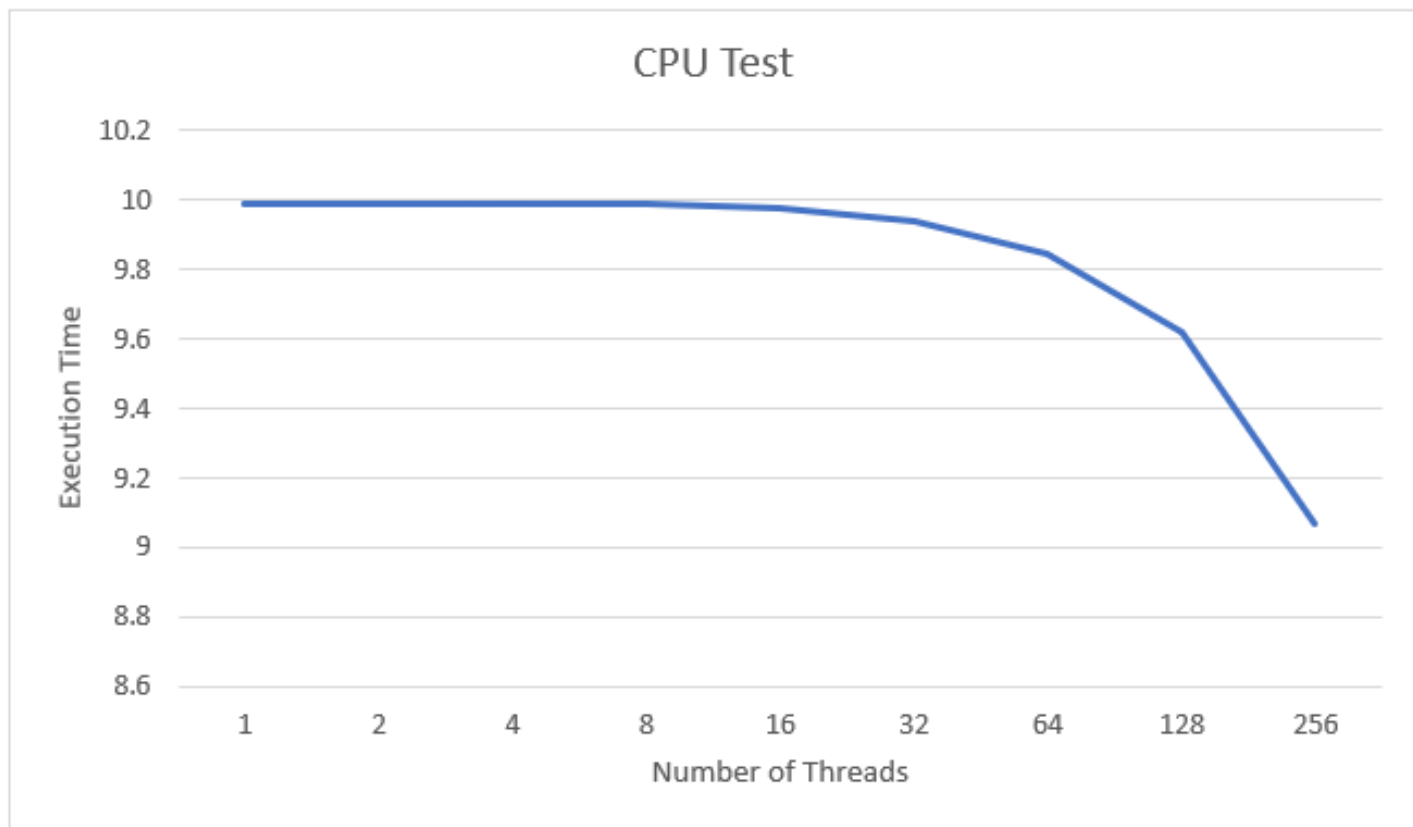Figure 24: Model showing trust boundaries and security risks.

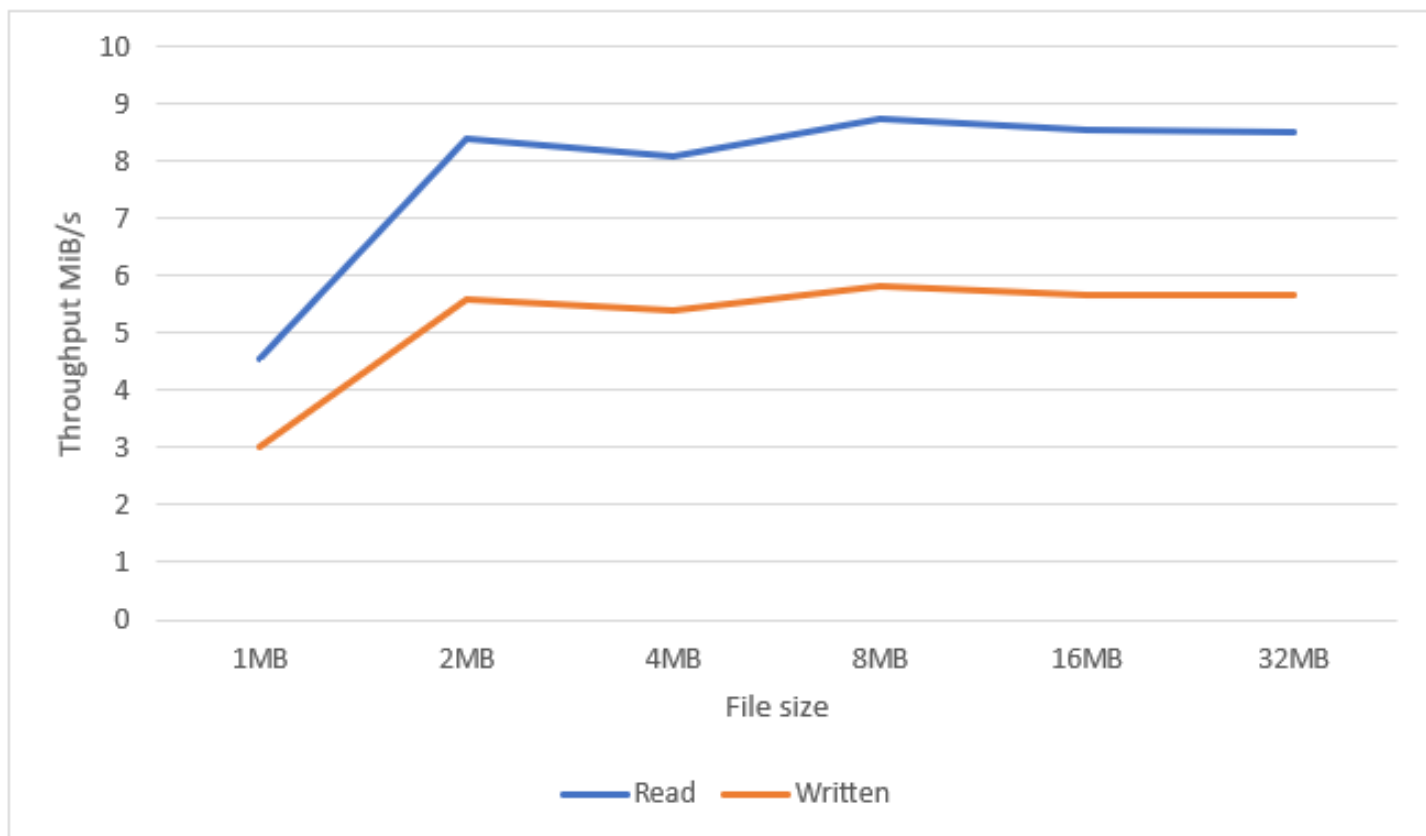Figure 25: Graph depicting the number of threads vs execution time.

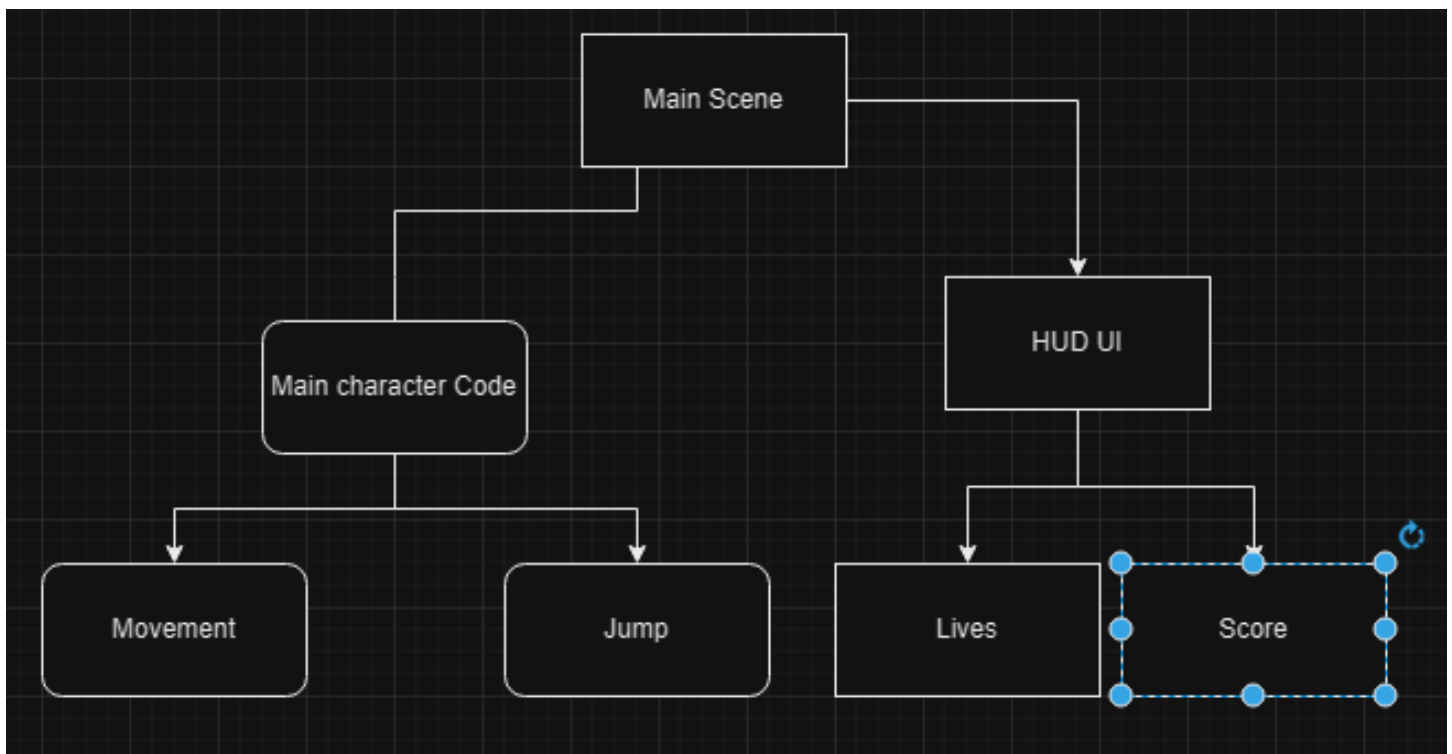Figure 26: Graph depicting file size vs throughput.

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.

41

Test	Pass	Fail
Cannot register an account that already exists.	✓	
Can login with an account that was previously registered.	✓	
Leaderboard shows high scores from previous gameplays.		✗
Login scene leads to game menu.	✓	
User can get to leaderboard scene and back to game menu.	✓	
Player can move George back and forth across the scene	✓	
Player can make George jump.	✓	
Player loses a life when they come in contact with an NPC.	✓	
Player gains one point with each building they destroy.	✓	

Figure 29: Acceptance test table showing what test cases passed and failed.

If player loses all lives, the lose scene appears.	✓	
If player destroys all buildings the win scene appears.	✓	
All audio plays as expected.	✓	
Buildings disappear when players interact with them.	✓	

Figure 30: Acceptance test table showing what test cases passed and failed.