Psycho Fox

Sprint 4
12/01/2023

Name	**Email Address**
Nicholas Johnson	nicholas.johnson769@topper.wku.edu
Cy Dixon	cy.dixon656@topper.wku.edu
Matthew Polak	matthew.polak034@topper.wku.edu
Harsha Suresh	harshavardhan.suresh593@topper.wku.edu

Dr. Galloway CS 360-001
Fall 2023
Project Technical Documentation

Contents

List of Figures

1 Introduction

1.1 Project Overview

This project was centered around recreating the first level of the game Psycho Fox in the Unity game engine for our computer science course, CS360. Pyscho Fox was created in 1989 and originally realeased for the Sega Master System. For background information, the plot focuses on a group of fox priests who worship the Inari Daimyojin (Fox Deity). Among this group of fox priests, one evil fox named Madfox Daimyojin infiltrated his way to the highest ranks and took over the shrine. After seizing power, Madfox corrupted the land and created hordes of creatures. One young fox, who would earn the name Psycho Fox, has been chosen by his fellow people to rid the land of this evil deity.

The creation of Psycho Fox as a course-long project provided our team members with a lot of experience in different areas of project development and engineering. These areas consisted of working together to succeed in a team environment, creating clear documentation that followed the steps of the software engineering processes that were taken to create Psycho Fox, presenting our work and progress to our client through presentations, and completing team member evaluations. Keeping a focus on the client was one of the most important aspects in the creation of our project. Attending client meetings has allowed us to develop a deeper understanding for the different requirements and functionalities that were expected of us. This documentation outlines these requirements and discusses the different methods and tools that our team has utilized to create our project. Certain risks needed to be addressed as well and could have potentially affected the progress of our work. These risks are identified in later sections, in addition to how we dealt with and monitored them over the course of the Fall 2023 semester. Considering all of this, our progress was divided into four separate sprints that spread the workload of the project across sixteen weeks. Within the first sprint (approximately the first two to three weeks) we gathered our group members and created a plan for how we wanted to approach our project. This involved considering the schedules of everyone in the group, learning about the different requirements that are present, figuring out how each of the requirements tie into the finished product, organizing the information that we have collected, considering risks that may affect our project, and deciding how feasible our ideas were. Within the second sprint, we focused more on the design of the project. Here, we considered the layout of the first level and how the player interacts with different objects and environments. Next, we moved on to the third sprint and really began the creation process. This period focused on the actual programming and asset creation which will allow us to show the client what they can expect from the finished product. Finally, the fourth sprint focused on testing our game and finalizing any code or logic that needed to be implemented. Within this sprint, we took the time to go back and review any steps that we thought still needed to be worked on. This whole process followed a modified waterfall diagram, which is a type of software production model. There are many different software production models that can be followed, but for the sake of this document, we relied on the modified waterfall. The modified waterfall model allows the development team to go back and change certain aspects that may not have worked in the creation process while still following a coherent well-structured process.

1.2 Project Scope

Our first main deliverable of this project included recreating the Psycho Fox game. Due to this just being a semester project, we limited our scope to the first level of Psycho Fox. While recreating Psycho Fox's first level, we made other Unity scenes that encompass the game. These other scenes include a login screen, main menu, loading screen, and a death screen. The project scope also consists of all of the original game mechanics that exist on the first level of Psycho Fox. Some of the included game mechanics are the movement of the character, the enemies that can harm the protagonist,the character animation, and the first level's design.

Since this is a course-long project, the game's progress was broken into 4 main sprints across a 16-week period. These sprints were broken into 3 to 4-week sections. Additionally, these sprints align with the software engineering process. The first sprint included the feasibility study, the second sprint included the software modeling/design, the third sprint included software implementation, and the fourth sprint included testing and bug fixing. The project deliverables also included 2 in-depth documents. For each sprint, an eight-page technical document and a two-page organizational document was added to the documentation to give a broader understanding of the program and

its application in the software development process. The technical document consists of the information that was related to the current sprint we are working on. The organizational document included a Gantt Chart, a progress visibility section, a software process model, and a summarization of the risks that can appear in development. It also included 4 presentations that went along with the completion of each sprint that was shown directly to our client. This project also included CATME evaluations of our peers throughout each sprint to ensure that each group member was completing their fair share of the work.

With the tasks and deliverables outlined above, this course project was very demanding. Due to this, we met weekly with our client, dedicated many out-of-class in-person meetings, and created a Discord channel dedicated to this project to discuss further if we were not face to face. We meet weekly with our client to ensure that we stayed on track with the tasks entrusted to us in each deadline and keep project requirements in sight. Our team also met several times after classes throughout the week to continue making steady progress towards our goal at the end of each sprint. These meetings added up to anywhere from 4 to 6 hours by the end of each week in a sprint. We often met between classes and talked frequently through Discord. In addition to this, if we needed to meet more outside of already scheduled times, we had plans in place to meet with each other if needed. Study rooms within the library are provided that gave us a meeting place to work. We have also made a Discord channel to keep all team members on track and keep an open line of communication. This Discord Channel also allowed for the sharing of files that may include code, documents, or charts that might have been used in other places. The Discord Channel also included a links section that allowed people on the channel to quickly access important documents and information that's useful in any contributions that were made towards the project.

There were several due dates for presentations, documents, and CATME evaluations. Below is a chart showing when the previously listed sprints are due.

Sprint Number	**Sprint Number Due Date**
Sprint #1	Due the week of September 3rd - 9th
Sprint #2	Due the week of October 8th - 14th
Sprint #3	Due the week of October 29th - November 4th
Sprint #4	Due the week of November 26th - December 2nd

1.3 Technical Requirements

1.3.1 Functional Requirements

Mandatory Functional Requirements
1. The main character can be controlled using the keyboard.
2. The project should allow multiple players to create separate accounts.
3. The project should implement a login system for users and administrators
4. The project should save the state/progress of all players and allow players to continue once they've logged into their account
5. The project should display a splash screen when launched which also shows top scores and the login prompt
Extended Functional Requirements
1. Losing a Life
2. Item Usage
3. Enemy AI
4. Obstacles
5. Audio Files and Music

Table 1: Functional requirements

Table 1 shows the different mandatory and non-mandatory functional requirements that were given to us by our client to be in our project. Each requirement is explained in further detail within the following sections.

1. Keyboard Inputs and Character Control

One of the main functions that needed to be implemented into our game involved control over the main character. Movement was defined using the arrow keys to move the player in the respective relative direction. Holding either the left or the right arrow key for an extended period of time causes the player to start running. Their speed increases and they're able to clear larger obstacles and get to the end of the stage faster. Running also allows the player to skid on top of water. Pressing the down arrow will cause the player to duck, which can be useful for getting underneath low-hanging objects. The Space key will be used to allow the player to jump over obstacles and enemies throughout the level. Holding the jump button for an extended period of time will cause the player to jump higher than if they were to quickly tap it. Some obstacles require the player to run and jump in order to get over them. Several eggs are also scattered throughout the level that contain objects that the player can interact with. Pressing the left click button will cause the player to punch or throw whatever object they're holding.

2. Storing Game States and Player Information

Our team plans to implement a database interface that allows different players to login and store game information that can be saved and revisited at a later time. Once players have created a username and password, they'll have the ability to save how far they've progressed on a specific level, how many lives they have, and how many power-ups they've collected. Generally speaking, it could be said that the overall game state is saved to the database. An SQL server will be started to hold all of this information and a GUI will be created for the user to interact with. The player will be able to click on text boxes to input their information with the keyboard, and pressing the enter key will input everything into the database. The database will be capable of holding the information of multiple users that will be able to create multiple accounts.

3. Login System

Users will be able to login and access their information through the database. Such information will include, their game progress, score, item count, and life count. Administrators will also be able to create separate accounts that are capable of not only accessing, but also managing specific accounts and information. This involves editing, removing, and adding certain entries. Admin accounts should also be able to start the game from any save state, modify how many lives they have, modify how many items they have, and modify their score.

4. Save States

Once users or administrators create an account in the database, they'll be able to save their progress that's achieved within the level. After logging into the database, users will also be able to load information that they've saved in the past. This allows the functionality of being able to pick up the game after closing out of it. The progress that's achieved in the level, the number of lives the player has, the player's score, and the number of items they're holding will be recorded into the database.

5. Splash Screen

When the game is launched, the user will be greeted with a splash screen that will prompt the user to start the game. The title of the game, as well as title art will be displayed on this screen. Pressing the Z key will load a separate game scene and begin the gameplay. Starting music will play while this screen is active and will remain active until the user starts the game. From the screen, we will likely implement the database functionality, since it would be difficult to incorporate it over the gameplay. It doesn't make sense to have the user encounter multiple/overlapping screens. Pressing a separate key will bring up the GUI panel and the user will be able to load/save their progress. The game version will also be displayed on this screen and it'll match the version that we upload to the GitHub account. After the user starts the game, a loading screen will appear that will connect to the splash screen. It will be a solid black screen with the word loading across the middle. Three white dots will follow the word loading, and an animation will play to indicate that loading progress is being made.

1. Losing a Life

When the level begins, the player will see a screen displaying the number of lives they have (the screen will have an animation of the fox running in place and the number of lives remaining, separated by a multiplication sign). The screen then transitions into the game screen, and the player will control the fox and try to get to the end of the map. There are two ways a player can die: By touching/running into an enemy and by falling into the water. In the event that either of these happens, the fox will stop moving, have a sprite animation showing its face in

shock, will shoot up into the sky, and will drop straight down. After this, the screen will transition back into the screen displaying the number of lives (this time, the lives count will have decreased by one), and the game screen will appear again. The player will then be able to play the game again, and this process repeats.

2. Item Usage / Player Leveling Up

As previously mentioned, there are eggs placed throughout the entirety of the first level. These eggs contain items and power-ups that the player is able to interact with. Punching enemies also presents the chance to drop certain items. All of the different power-ups and their functionalities are listed in this section. By opening the pause menu, the player is able to view and use the different items that they've collected. The player is also able to stack nine of each item. The psycho stick is an item that allows the player to change which character they play as. The psycho stick's functionality is explained in further detail in the next functional requirement section listed. Only eggs can drop the psycho stick. The straw effigy is an item that destroys all the enemies that are present on the screen. The player has the ability to hold onto this item and can use it at their disposal. Magic medicine is an item that grants the player temporary invincibility. Stars hover around the player when the item is in effect and cannot be held to be used at a later time. Enemies and eggs can drop magic medicine. Money bags can be dropped by eggs and act as currency to be used in a bonus game that can be played at the end of the level. Since we're only recreating the first level, the bonus game won't be included, but money bags will still be present for the player to interact with and view. Extra lives can also be dropped by punching an egg open. Extra lives are represented as a smaller sprite of the character that's currently on-screen. Once an extra life appears, the sprite will begin to move away from the player, and if the player touches the sprite, then they'll be granted an extra life. Finally, the bird power-up can be collected and can be thrown by the player to destroy enemies. The bird can only be collected by breaking open eggs and becomes part of the character's sprite through the addition of a small bird being present on the back of the player. When the bird power-up is active, the punch functionality is replaced with the throwing mechanic. While the bird is thrown and away from the player, punching is restored until the player picks up the bird again. Over time, the bird will automatically be returned to the player after it's thrown. The bird protects the player from taking damage once. After the player has taken damage, they go back to dying on the next instance of damage.

3. Character Selection

When the psycho stick is used, a screen will appear that allows the player to select a different character to play as. Originally, when the first level begins, the player is forced to start as the fox. Later on, they'll have the ability to choose between the fox, the hippopotamus, the monkey, or the tiger. Each character has different abilities and changes the way the controls affect jump height, movement speed, and punch strength. The fox has what would be considered average movement speed, average jump height, and average punch strength. Compared to the fox, the hippo has slower movement speed, lower jump height, and stronger punch strength. The increased punch strength allows the hippo to break special blocks that make levels less challenging and reward the player. The monkey is also slower than the fox, but his jump height is the greatest amongst all the characters. This allows the monkey to reach areas that would be impossible to reach with the other animals. Finally, the tiger has a low jump height, but his running speed is the fastest amongst all the characters. Like the monkey, he's able to reach certain areas by jumping, but only after he's gotten a running start first.

4. Enemy AI

Enemies are able to move back and forth throughout all levels in the game. Programming this aspect of the game will allow the user to find difficulty in getting past enemies. Without this functionality, a major part of the game is missing and the experience isn't the same. Some enemies move around on the ground, and some are able to jump towards the user. The team will need to develop a program to allow enemies to track the user's location and follow the player's inputs.

5. Obstacles

There are many different obstacles that are present throughout the first level. Platforms are arranged in a particular order to challenge the player and add difficulty to the stage. Since this game could be categorized as a platformer, the team will need to pay attention to how platforms are arranged and viewed by the user. Another obstacle that stands out in the game that needs to be implemented is a pole that the player is able to hang onto. When the player jumps towards this pole, a swinging animation plays. Pressing the Z key allows the user to jump off the

pole higher than they normally would. Another obstacle that the user will have to face, and that the team will have to implement, is pits containing water. When the player walks over a pit, they fall through the bottom of the screen and they lose a life. This obstacle is unique because it will have to be separately recognized from the ground. If the player is walking across the level, the program will need to recognize when the user walks across a pit so that the sprite updates correctly and the life counter changes. One of the ways we can implement this is by tracking the vertical position of the player. If we establish a death barrier, the program will be able to check whether or not the player is above or below this imaginary line. Sectioned platforms will implement a similar logic outcome compared to pits and will be strictly above this death barrier.

6. Audio Files and Music

Music is a major part of our project's development. A soundtrack will play continuously throughout the first level and serve as background music. Audio will also be played when the player presses the right arrow to move forward, when the player presses the left arrow to move backward, when the player presses the z button to jump, when the player presses the x button to punch or throw an object, when the player skids on water, when the player goes into a menu to select another character (unsure if we will implement this yet), when a player dies, and when a player makes it to the end of the level. Depending on if we make more progress, we may add more audio for login and other small additional screens the client has asked of us. Audio files will be pulled from a reputable source and will allow us to replicate the game more accurately.

1.3.2 Non-Functional Requirements

Mandatory Non-Functional Requirements
1. The project should be implemented using the Unity game engine and development environment.
2. The project team should create a complete replica of the game.
3. The project can use pre-developed images, artwork, audio, and other related media.
4. All project source code must be developed by the CS 360 project team.
5. The project must use a database.
6. Performance metrics should be gathered and optimized
7. Security metrics should be gathered and optimized
8. User interface metrics should be gathered and optimized
Extended Non-Functional Requirements
1. Refresh Rate
2. Resource Management

Table 2: Non-functional requirements

Table 2 outlines mandatory and non-mandatory non-functional requirements outlined by our client, Dr. Galloway. This list will also include some additional requirements outlined by the developers which will be completed if we find extra time during the process of this course-long project.

1. Using Unity

The first non-functional requirement that we need to accomplish is utilizing the Unity game engine and development environment to make our game. This is a non-functional requirement because our client, Dr. Galloway, wants the game developed in the Unity game engine. Unity is a very powerful game engine that can handle some of the largest games made, so recreating a small 1980's 2D side-scrolling platformer will not be an issue. Unity will also allow for easy scalability in the future if the scope of the project were to increase. The Unity game engine will also allow for future cross-platform capability if we, the developers, decide to increase the number of systems this game could be played on. To ensure that there are no issues with the Unity game engine across all parties that touch the game, we will be using the most recent LTS version of the Unity game engine. This Long Term Support package will help with concerns our client may fear about reliability issues. This package is discussed more in the Target Hardware Details section, 1.4.

2. Replicating the Game

The second non-functional requirement involves making a complete replica of the 1989 retro game, Psycho Fox. This 2D platformer game has several different scenes and other aspects that will need to be recreated for the replica to be completed. With the previously described time constraint, our client has limited this scope to simply the first level, 1-1, which will allow us to focus on all other aspects of the game. These scenes that need to be recreated consist of the main menu screen, the loading/death screen, and the main game-level view. A few other scenes that will need to be implemented will be discussed in other sections that were not in the original version of the game that our client has asked us to implement, like the login view. These originally implemented scenes are integral to the creation of the game for the experience of the player. The main menu will showcase the cover of the game with the ability to start the game. The loading/death screen allows the player to view the number of lives and the current character being used. The main game level allows the player to see the output of their actions in the game. Another aspect of recreating this game includes adding enemies with basic AI that act as an obstacle. Recreating the first level exactly as it is in the original game is an important step in making this game a replica of Psycho Fox as well.

3. Asset Creation

The third non-functional requirement required in this project is utilizing pre-developed images, artwork, audio, and other related media to remake the game. To be able to meet all mandatory requirements before the course-long project due date, taking advantage of already developed assets and other sources to remake Psycho Fox is a must. Utilizing these pre-made assets will also give us the opportunity to polish up other gameplay aspects, mainly those listed inside the functional requirements. Making use of these assets will also give the client a better replica of the game. This is mostly due to the fact that remaking images of the loading screen of Psycho Fox will simply not look like a replica game that we are aiming to make. This same statement goes along with the audio. Utilizing the same audio that was used inside of the original Psycho Fox game will allow us to make a near-perfect replica if we can correctly get the functional requirements up to the standards set by our client.

4. Source Code Development

The fourth non-functional requirement being utilized in this course-long project is utilizing our code and our code only. While this project will mainly use our own code from this CS360 course, we will also utilize other sources to help create the Psycho Fox game. These other sources that may help us create the source code for the game will be through utilizing the Unity User Manual, using YouTube tutorials to help us with major concepts in our code, and referencing websites that refer to troubleshooting Psycho Fox.

5. Database Implementation

The fifth non-functional requirement our team is utilizing is the use of a database. The database in this project will be a fundamental requirement needed to store a lot of information regarding the user. This information will be made available to the user by calling certain inputs and running them through a program that formats it into a GUI. In addition to the system calls being made, we will also need to write a program that writes data to the database and manages it in an efficient manner.

6. Performance Metrics

Performance metrics will be stored in the database to give the development team an idea as to how efficient the system calls are. These metrics will allow us to make improvements and potentially reduce lag that the user could experience. If the system is run through a server, this information will be valuable in analyzing how we structure our database. We will analyze whether different characters/actions/animations result in drops in frame rates, different player/environment interactions/animations lead to any crashes, users are able to log in and save data in a smooth and responsive manner, users are able to access saved data without having to wait on delays or issues with pulling their data, and if multiple user accounts affect gameplay or performance. Based on the data we collect, we will look to provide solutions to these issues and look to provide a stable, smooth experience for the user and the client.

7. Security Metrics

Security metrics will be implemented so that users can access their information consistently. We need users to be

able to know that their information isn't being tampered with, so certain protocols will be put in place to provide this assurance. This information will also be stored in the database for administrators to view and analyze.

8. User Interface Metrics
User interface metrics will allow the development team to see how people are interacting with the database. We plan on recording how often people are logging on and off and seeing any potential improvements that can be made. This information will also give the development team an idea as to how effective the GUI design is. If users are having a hard time logging on or saving data, this information will allow us to tweak certain aspects of our program.

1. Refresh Rate
This project should maintain a consistent frame rate, which means there shouldn't be any visual or input lag. A realistic goal to have would be for the game to continuously run at sixty frames per second. We can achieve this by paying attention to the assets we use in the game's creation so that they don't overwhelm the system in any way. This also ties into maintaining a secure connection with servers and utilizing our performance metrics effectively.

2. Resource Management
Effectively managing the project's resources will heavily contribute to how the game performs. Handling the assets within the game correctly will allow us to create a finished product that is much more enjoyable to play. This involves quickly loading in sprites, source code, music, save states, and game logic. This project will pay attention to the amount of processing power and RAM space that are present on a given machine so that assets are able to load efficiently.

1.4 Target Hardware Details

Our game is not very computationally demanding given that it is older and relatively simple. Since we are using Unity, which is a much more advanced engine compared to the hardware that our project originally ran on, our game only requires the minimum hardware specifications to run a simple two-dimensional game in Unity. This includes but is not limited to, a processor capable of running at 1 GHz, 2 gigabytes of RAM, any form of integrated graphics, which are far less powerful than dedicated graphics cards, and enough storage to be able to handle the game's assets and source code. As our project development continued and we gathered a more accurate understanding of how our project ran, we could update these specifications so that the client knows what their machine should be capable of before playing. Our project is also designed so that it can run on multiple versions of Windows. The version of Unity that our team is using is able to run on systems with Windows 7 or later, so this will be the requirement for our game. It'd be fully possible to have our project exported to run on different operating systems, but for the sake of simplicity, and in accordance with what the client is expecting, our game will be running everything in Windows. This also relates to whether or not our project will be available to play on consoles. For the same line of reasoning, porting to consoles is beyond the scope of our project.

To ensure that the end user, our client, and the developers have no issues down the road, we utilized the most recent LTS version of Unity to create our project. At the time of making our game, this release version is Unity LTS 2022. Utilizing this package/version is important to us because these Unity packages typically have maximum stability and support. This will allow minimal problems for all parties that deal with this game. This Unity version also allows the end user the best performance possible with their machine.

1.5 Software Product Development

The development of our game Psycho Fox included the use of Visual Studio Code. This IDE is great for game development and C-sharp coding due to the abundant amount of plugins and add-ons that allowed us to make forming the code improved overall. Within Visual Studio Code we used the included C-sharp add-ons that allow us to use the debugging console and terminal to help us with the coding of the scripts for our game. This allowed us to create a more well-polished product. As mentioned we made good use of the C-sharp software language. C-sharp is the default language that the Unity game engine utilizes, that being so it is substantially easier to

implement than implementing an alternative coding language. C-sharp also has wonderful documentation so that any hiccups can be ironed out in the process. This leads back to using VS code due to its optimization for C-sharp.

Onto the asset creation and generation tools. Due to the nature of our game remake being an almost one-to-one recreation of the first level, as stated in our scope, most of the assets for our game were downloaded from online web repositories that include many game assets for older console games such as the Sega Master System. Therefore, we did not utilize many asset creation tools outside of Unity. We mainly pulled things like sprite animations and audio files to create our project. That being said, a small risk was if we could not find needed assets, we will use the in-engine creation tools provided within the Unity game engine. In the worst-case scenario if we could not use the in-engine creation tools, we would have put Blender to use in creating small textures or models for our game. This was unlikely but still worth mentioning.

The main tools for collaboration that our team used are as follows: Discord for chatting and discussing our plan and general comments, as well as some smaller file sharing and links, GitHub for major project management and task allocation between team members, and finally Overleaf and the cloud Google apps for documentation and writing. The GitHub repository will be used to keep track of the different versions of our project. Each person in our group has similar permissions so that everyone can equally contribute to the project versions.

2 Modeling and Design

2.1 System Boundaries

2.1.1 Physical

Physical system boundaries relate to logical boundaries and involve how the hardware is affected by the software being created. We are utilizing the Unity game engine that connects our code to the hardware we're working on and creates a physical boundary for our work. Since we are also deciding to use a remote database, this will allow us to transfer save states of the users that have made an account in the database. Also, this will allow us to keep high scores from the users who have made accounts and display those names. The machine we are working on also sets a physical boundary. Our code only can perform as well as the hardware we are working on.

2.1.2 Logical

Logical system boundaries focus on mechanics that take place within the game. The technical limitations are not a concern when identifying these. The core elements of the gameplay represent the logical limitations of the system. This includes things like player movement, saving the game, collecting a power-up, and killing an enemy. How we will we implement these components will determine our logical boundaries. These boundaries are related to the functional requirements of our project which is in the introduction section of our project.

2.2 Wireframes and Storyboard

The image below of our storyboard shows the progression that the user will face when playing Psycho Fox. This storyboard also contains wireframes that make up our storyboard. There are 3 main separate wireframes that we pieced together in this storyboard. The first wireframe mainly consists of the main menu. When the user first starts the game, the user will first see the boot screen. Users will then be able to log in, register, or view the scoreboard for previous scores that are kept based on registered users. Once the user logs in, they will be able to start playing the game, log out, or be able to view the scoreboard. The second part of the storyboard contains the original Psycho Fox game screens. Once the user hits the start game button after the login page, the original start screen from Psycho Fox will appear. If the user wants the view the scoreboard here too they can. They can also hit the start game button and the level will load. Once the player completes the level, they will be able to go back to the "Hi, Player Name" screen. The next part of our wireframe deals with the game over and death screens. These screens only will appear if the player dies inside of the Psycho Fox game. If the player loses all lives, the player will be pushed back to the start screen button.

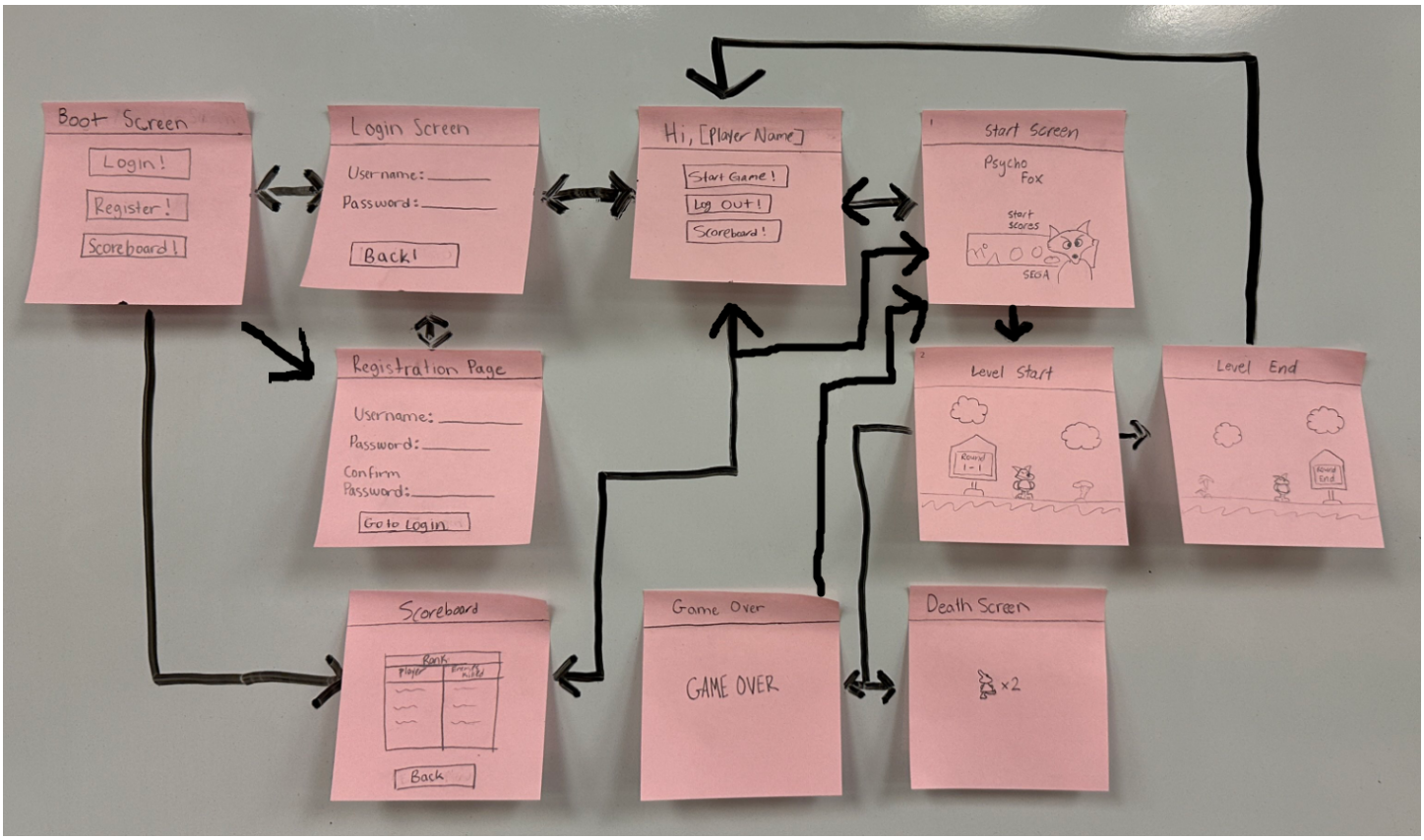

Figure 1: Storyboard of Psycho Fox that contains wireframes.

2.3 UML

2.3.1 Class Diagrams

Behavioral Design Patterns:

In Figure 1, the High Score System diagram functions by having the High Score Observer constantly check for updates on the high score by calling the updateHighScore() function. The Concrete Observer takes data from the game and uses it in the Observer and Concrete Subject nodes to create information that's uploaded to the database. The Concrete Subject node constantly checks using logic to see if changing the high score is necessary. Lastly, the Subject node is how the user is informed of changes to the high score. The Subject node communicates with the user and maintains contact with the Observer.

Creational Design Patterns:

Singleton Creational Design Pattern for the Player Manager:

Figure 2 demonstrates how a single PlayerManager class will handle the user's health, movement, and position. Other classes will use the PlayerManager class as a global access point to update certain values as the user plays the game. This simplifies dependencies and improves gameplay performance. The instance variable represents the PlayerManager class. The getInstance method instantiates the class and creates the access point. The jump and move functions impact the player's location, while the remaining functions control the player's health and state. The state diagrams that we have created relate to this class and extend out to create a more intricate system.

Singleton Creational Design Pattern for the Enemy Manager:

The EnemyManager class is handled in a similar way. Referring to Figure 3, the Player calls an instance of the class by using the getInstance method associated with it. Different enemies that are created are passed through

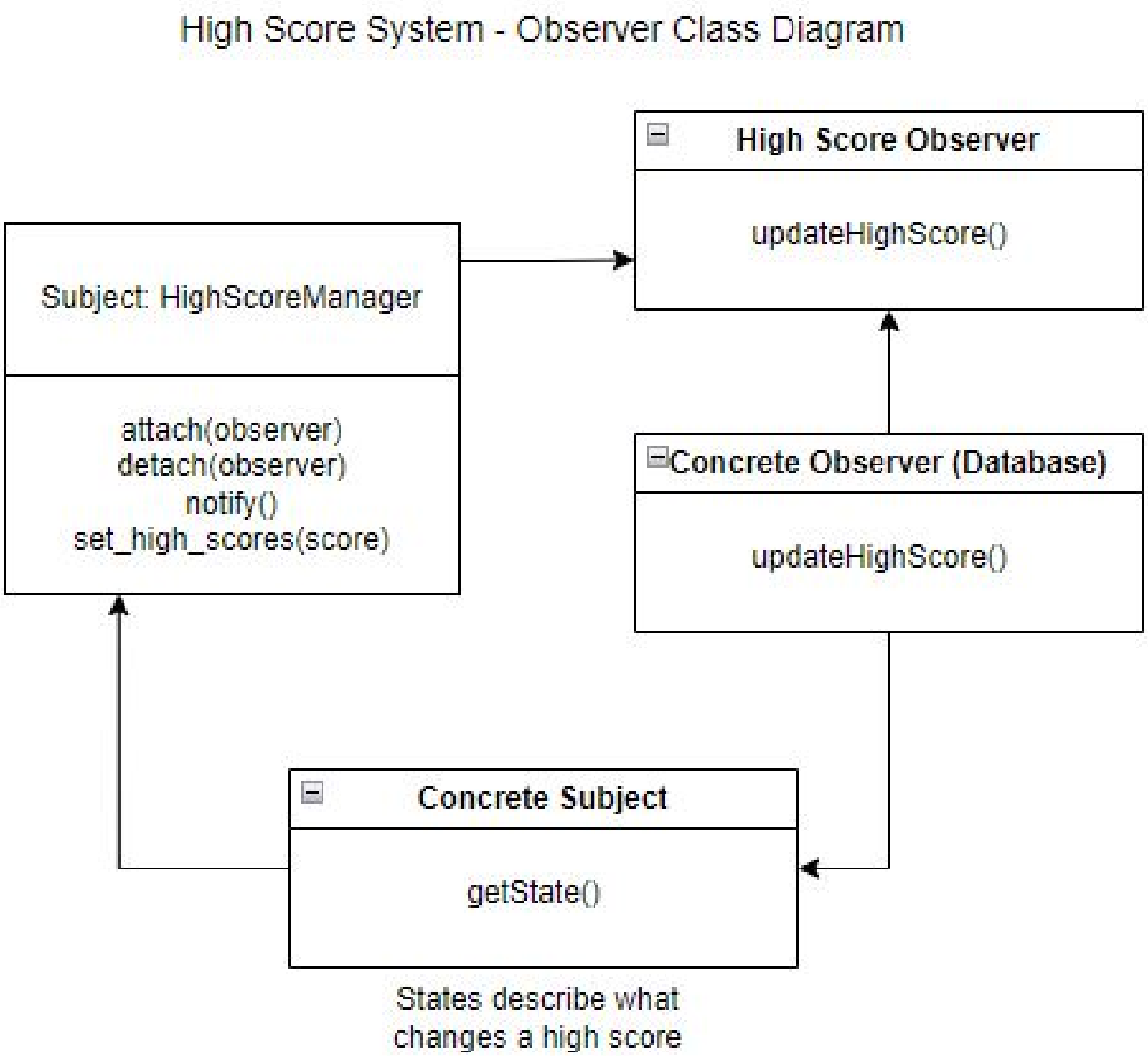

Figure 2: Observer Behavioral Design Pattern for the High Score System.

the EnemyManager and are stored in a list of objects. Enemies contain a sprite, as well as location, movement, and damage variables. These variables are changed through the changeEnemyState method. Enemies that are created are processed through the manager as well. When an enemy is defeated, the defeatEnemy method is called and the enemy object is removed from the list. All of this occurs through a single instance of the EnemyManager class which creates a global access point and maintains simplicity.

Structural Design Patterns:

For the general UI and world as well as how the game works, we have chosen to use a structural Adapter OO Design Pattern. With this adapter pattern, we can create different parts and link them together to the rest of the game easily. We start with the user player interacting with the Main Menu/Login screen. The user is presented with options to play, quit, and view the scoreboard of the game. From there, we have modules to load the and view the game scene which moves into the loading of enemy and player movement as well as loading in the rest of the terrain. This Adapter module system design pattern helps out a ton in the creation of the game, as we can easily see how parts of the project work together and can adapt to each other. With all these parts if there is ever

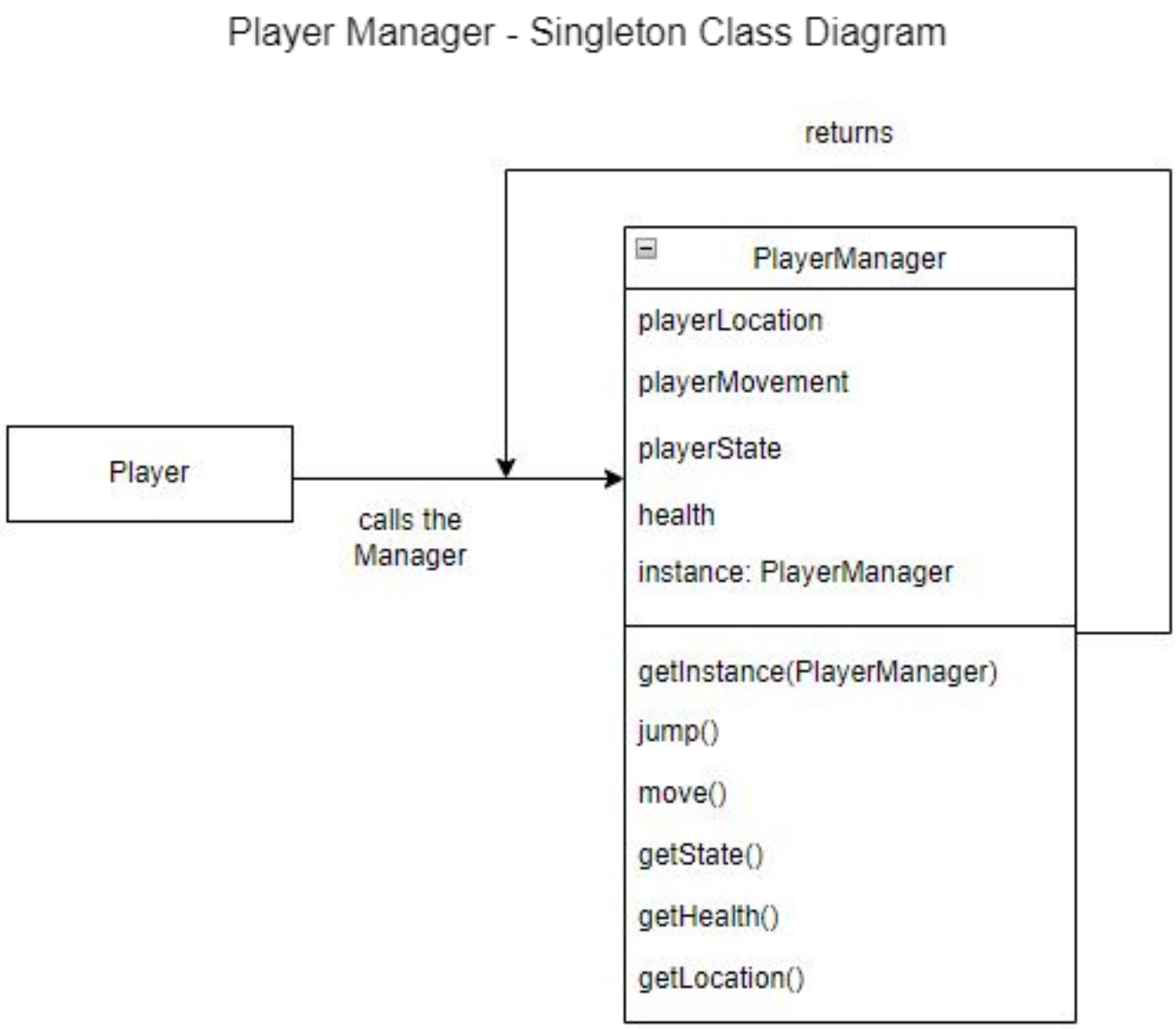

Figure 3: Singleton Creational Design Pattern for the Player Manager.

something that needs to be added, we can easily adapt it to the current design without much issue.

2.3.2 Use Case Diagrams

Use Case Diagram description for Psycho Fox Main Menu:

This use case diagram focuses on the main menu of our recreation of Psycho Fox. Essentially, the actor (user in this case) will have the options to either log in to an existing account, register for an account, or open the scoreboard to view the high scores of other players. If the user opens the scoreboard, they have to return to the main menu to be able to actually play the game. If the user does not already have an account, they can register for an account and then log in to their account after that. If the user already has an account, they can just log in to their account. Once the user has logged in, they can either start the game, log out, or open the scoreboard (similar to the beginning, if the user opens the scoreboard, they must return to the previous screen to be able to either log out or play the game.

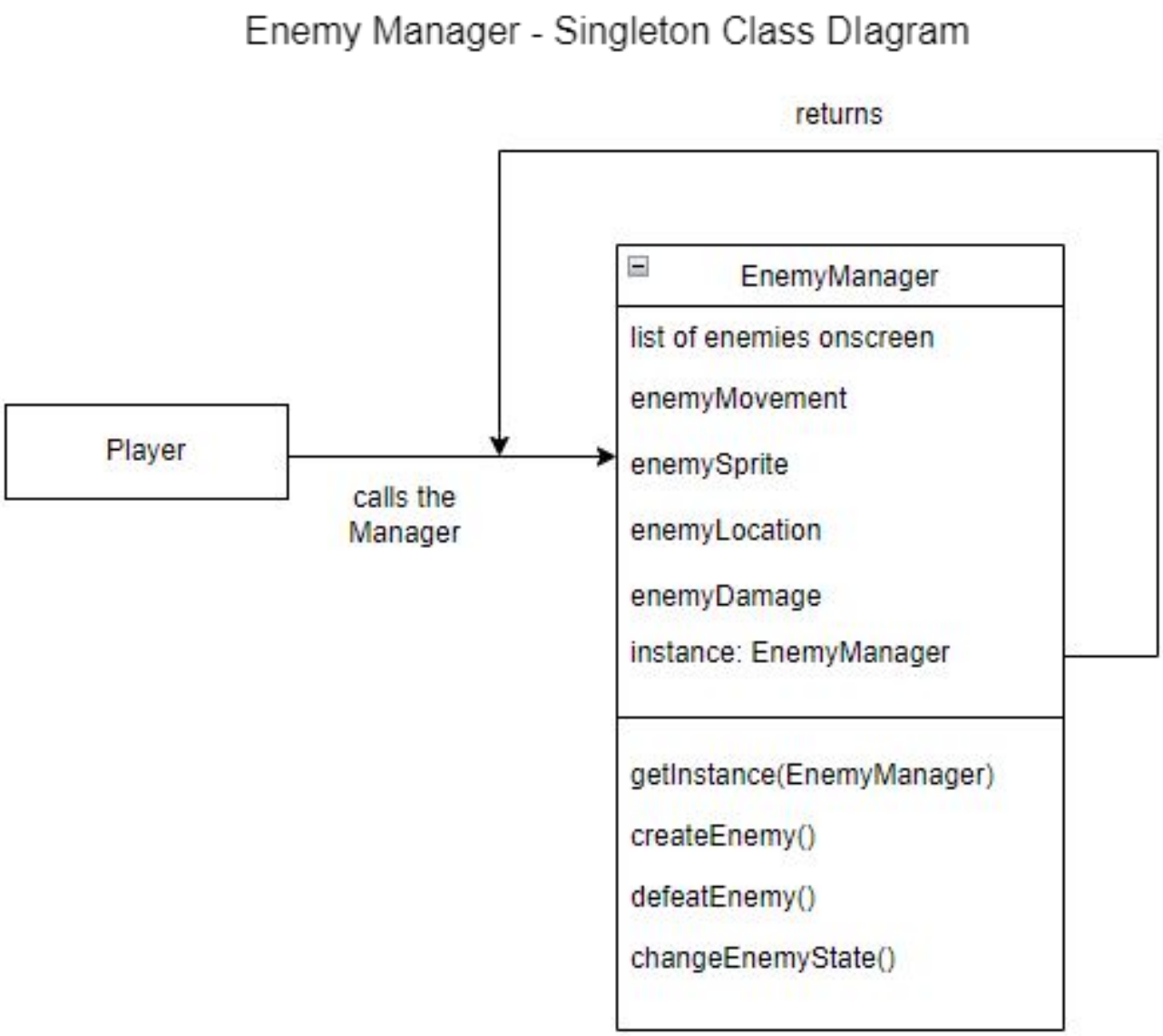

Figure 4: Singleton Creational Design Pattern for the Enemy Manager.

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

Use Case Scenario for Psycho Fox Main Menu:

Use Case: Main Menu

Actor: User

Preconditions: The user has launched the Psycho Fox game

Normal Flow: The user will either log in, register, or open the scoreboard. If the user opens the scoreboard, they can see the current standings of previous users with an account who have played the game. The user must then return to the main menu screen and either log in or register for an account. If the user decides to register an account and successfully does so, they will be taken to the login page and can log in. If the user gets to the login page as a result of creating an account or if they already have an account, they can log in to their account.

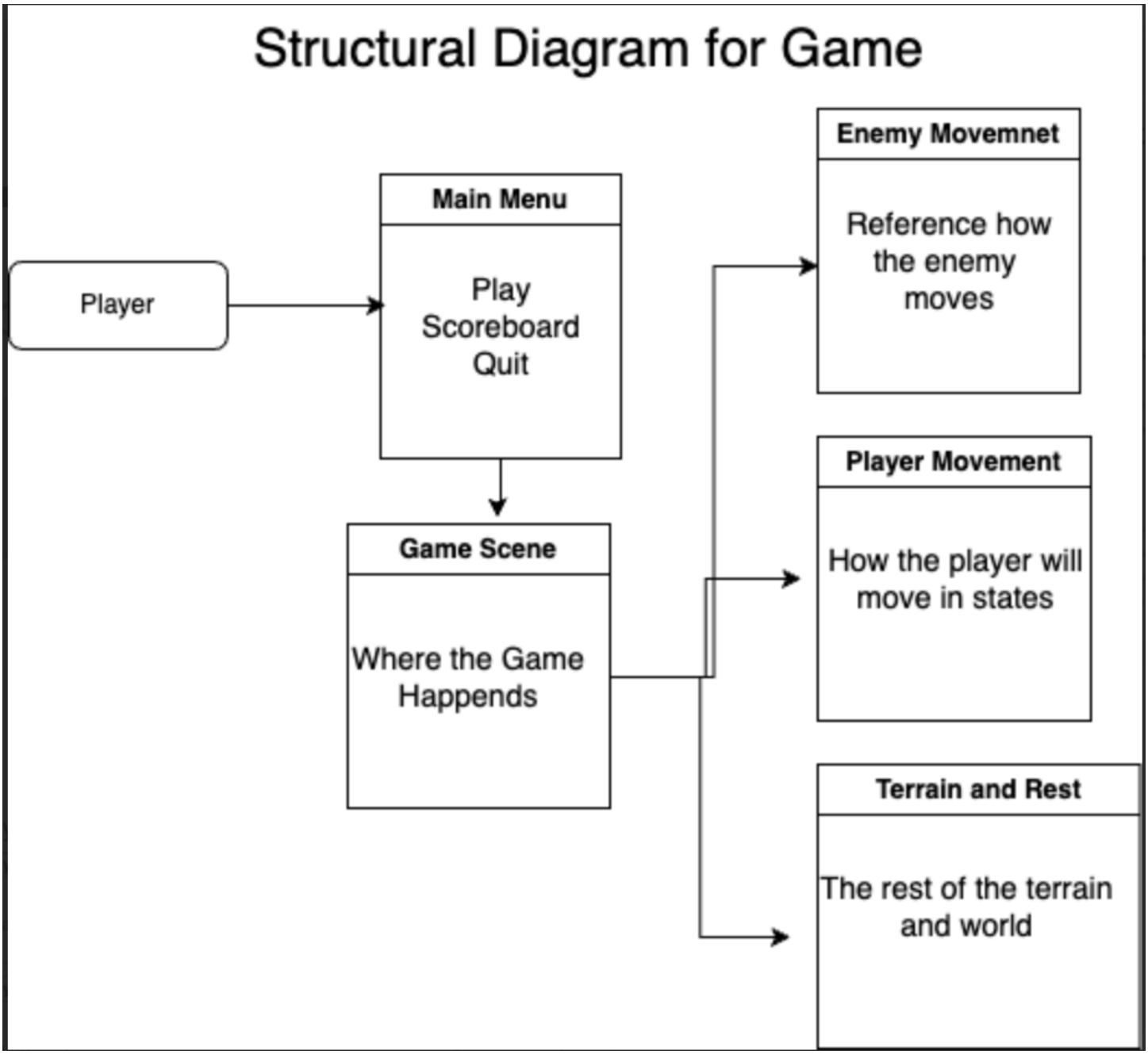

Figure 5: Adapter Structural Design Pattern for Overall Game Functions.

If they successfully log in to their account, they can either start the game, open the scoreboard, or log out. If the user opens the scoreboard, they must then return to the login page (the "Main Menu" use case ends).

Exceptional Flow: If the user has any issues during the registration or login process, they will continue to stay on those respective screens until they resolve the errors, or they can go back to the previous step (either the main menu if they initially attempted to log in to their account or the registration screen if they created an account (the "Main Menu" use case ends).

Postconditions: If the process was successful, the user will be able to either start the game, open the scoreboard and return to the login page, or log out. If the registration process did not go successfully, the user can retry the process or return to the main menu screen without having permission to advance. If the login process did not go successfully, the user can either retry the process, return to the main menu screen if they already had an account, or return to the registration page if they just created an account without having permission to advance.

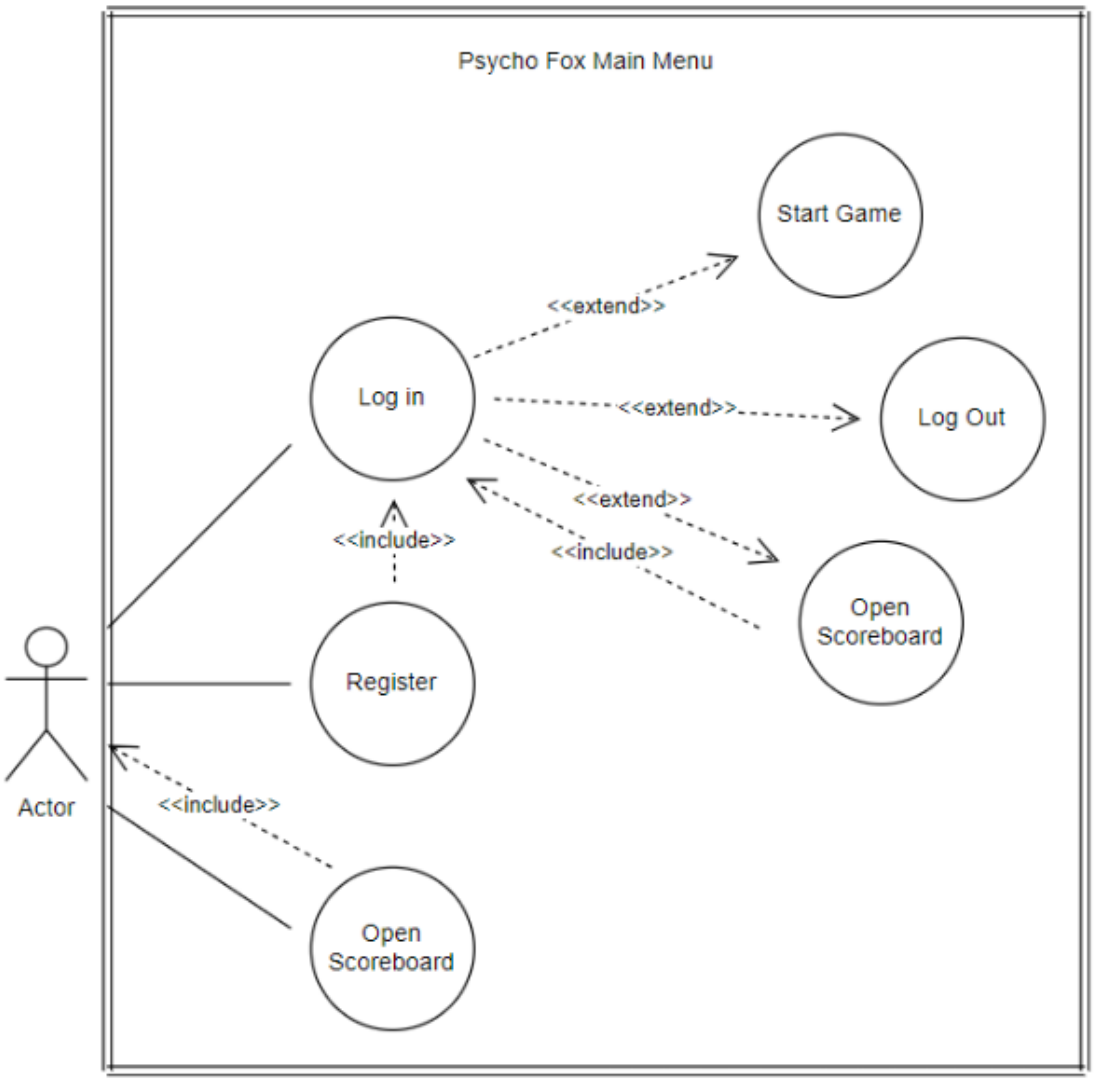

Figure 6: Use Case Diagram for Psycho Fox Main Menu.

2.3.4 Sequence Diagrams

Sequence Diagram for Psycho Fox Birdfly Acquisition:

This sequence diagram focuses on the Birdfly acquisition process in our Psycho Fox game. Psycho Fox can use Birdfly as either a weapon or armor. It can be launched at enemies to damage them and can also be used as a shield against attacks from enemies (each Birdfly allows for Psycho Fox to survive on enemy hit, but Psycho Fox loses Birdfly after this). Based on this, it is clear that Birdfly is a crucial aspect of the game. The sequence diagram shows the process to attain Brdfly, which is to approach an egg and punch it. Once Psycho Fox punches the egg, the egg hatches and turns into Birdfly that Psycho Fox can now pick up.

Sequence Diagram for Psycho Fox Player-Enemy Interaction:

This sequence diagram focuses on the player-enemy interaction in our game. The actor (user) will press the forward and backward moving keys to control Psycho Fox's movement and the game system will react by traversing through the map either in the left or right direction based on the key press. Enemies will spawn on the map, and Psycho Fox will either try to dodge them or kill them. If Psycho Fox successfully kills the enemy either by stomping on them multiple times or by punching them, the enemy will die, an enemy death animation will take place, and the player will continue the game. If Psycho Fox comes in contact with the enemy without properly

dodging it or killing it (essentially just running into the enemy), the Psycho Fox death animation will take place, the player will lose a life, and the level will restart.

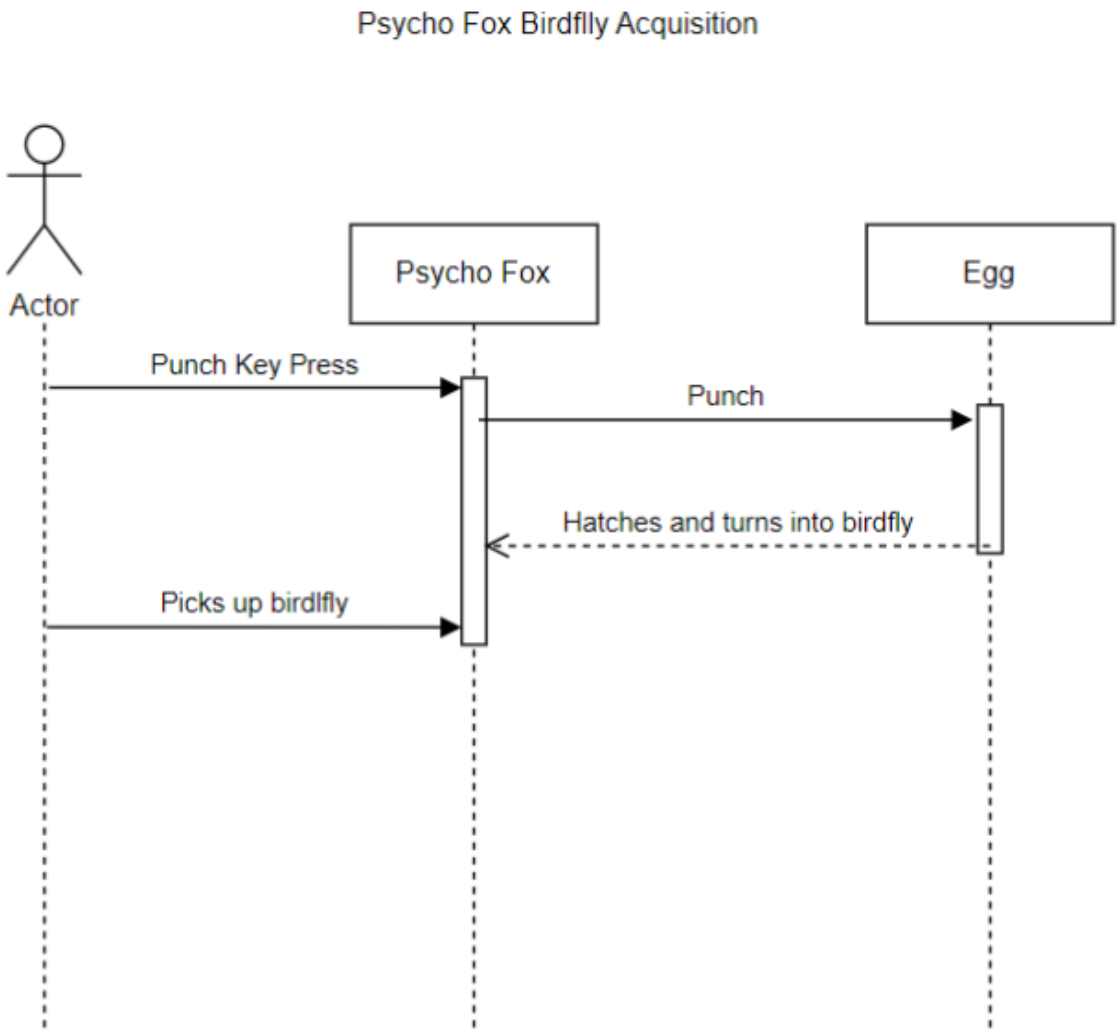

Figure 7: Sequence Diagram for Psycho Fox Birdfly Acquisition.

2.3.5 State Diagrams

State Diagram for Player Movement:

This State diagram shows the different states the player can move in and out of. For the player character in our game, Psycho Fox, he can jump, walk, run, and punch. This diagram is used to show the different states that he can go into based on user input. The states listed are Walk, Run, and Jump. These are all associated with functions within the code so that the user input will change the state of the character. The wait function shown in the diagram is used to wait for user response and input. Then based on what the user decides to do, the player character will move accordingly. We have chosen to use this diagram type because it can easily be adapted to the player use case. With the different actions that the player can do, using different states for the player character is perfect. Within these states, we can then have other objects and world logic work around what state the player is in.

State Diagram for Psycho Fox Life Status - Falling:

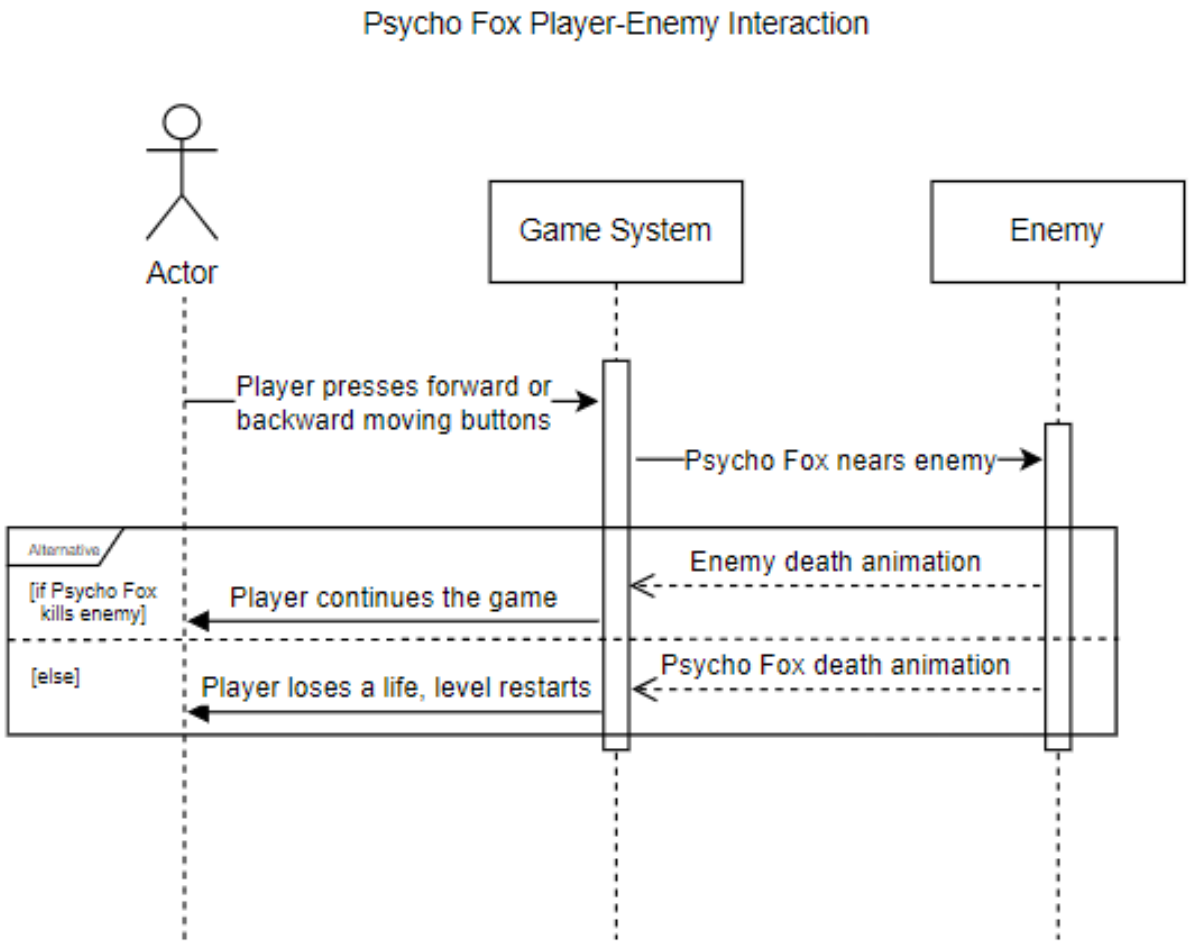

Figure 8: Sequence Diagram for Psycho Fox Player-Enemy Interaction.

This state diagram focuses on the life status of Psycho Fox when attempting to jump over a water pit. Initially, the user begins the existing level at whatever life count they have, and Psycho Fox is alive. If Psycho Fox begins the jump, is in the air, and successfully lands on a platform, it is still alive. If Psycho Fox begins the jump, is in the air, and falls into the water pit, the user loses a life. If the current life count is equal to 0, Psycho Fox permanently dies and the user has to restart the level. If the current life count is greater than 0, Psycho Fox respawns in the game.

2.3.6 Component Diagrams

In the component diagram below, enemy interaction is shown. In this diagram, there are 3 main nodes, the Psycho Fox character, an enemy object, and the database. The enemy interaction as shown in the diagram works by starting with the Psycho Fox character node. If the Psycho Fox character rigidbody has an enemy object collides with it, then more logic will occur. In Psycho Fox, Birdfly acts as a second life for the character. In the diagram, we showcase this. If the Psycho Fox character has Birdfly, then it will change the Psycho Fox character node boolean value. If Psycho Fox does not have Birdfly, then inside the database, the player will lose a life and then have to restart the level.

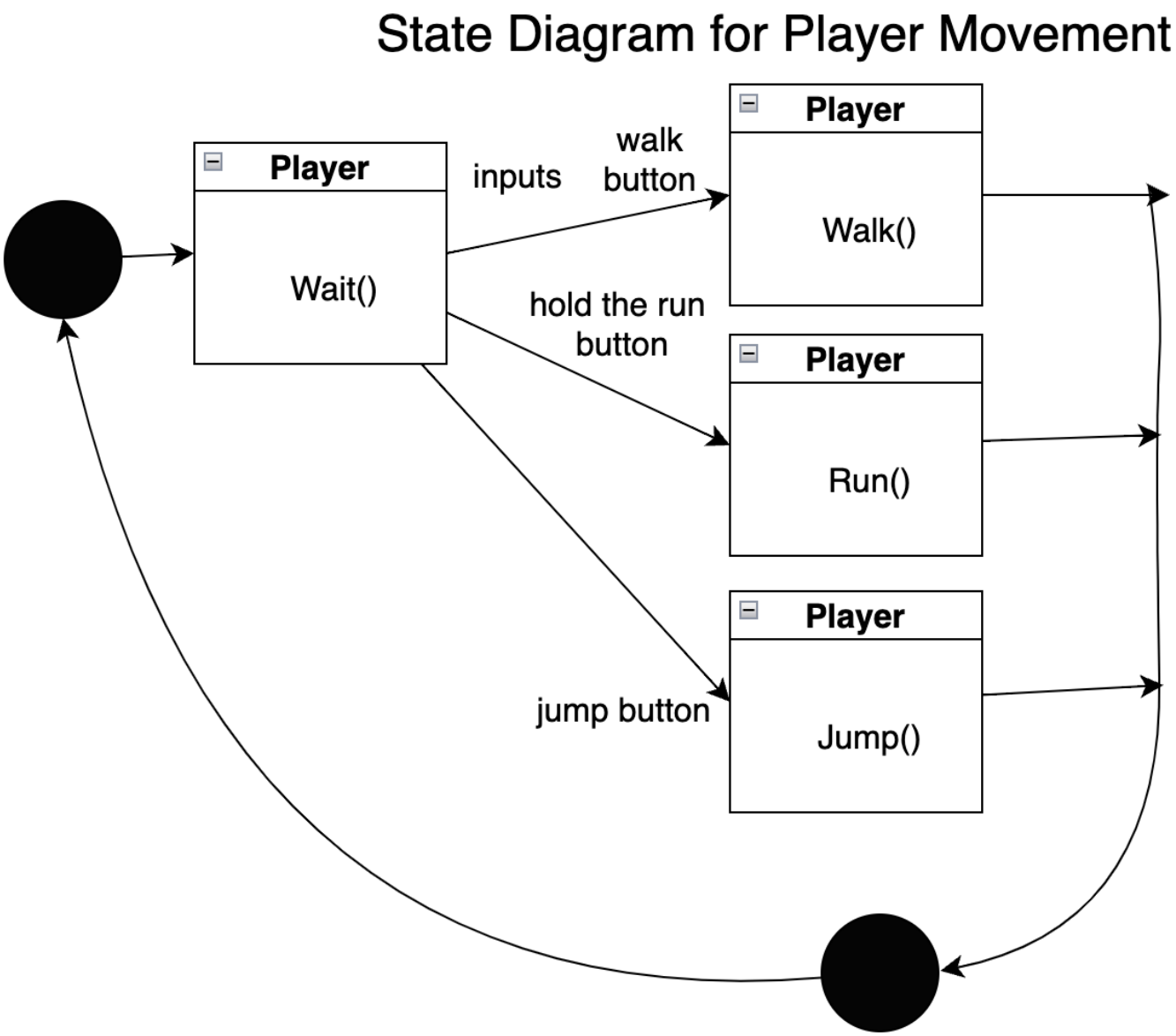

Figure 9: State Diagram for Player Movement.

2.3.7 Deployment Diagrams

In the deployment diagram below, all the basic functionality of how nodes will interact with each other are lined out. The player, or person who controls the computer, interacts with the Unity game, in our case, Psycho Fox. These game functions will either produce game output or lead to modifying the database on that user's system. It will update or get database information and feed that back into the Unity game as needed. This loop continues until the player decides to stop running the game on their system.

2.3.8 Requirements Traceability Table

The below table shows the requirements(diagrams) in a traceability table.

2.4 Version Control

So that there are no complications with Unity versions, we all decided to use the same LTS version of Unity. The specific unity version used is 2022.3.8f1. If we did not use the same version of Unity, we'd run the risk of being unable to collaborate effectively. Our team utilizes a shared Github repository to save and upload changes

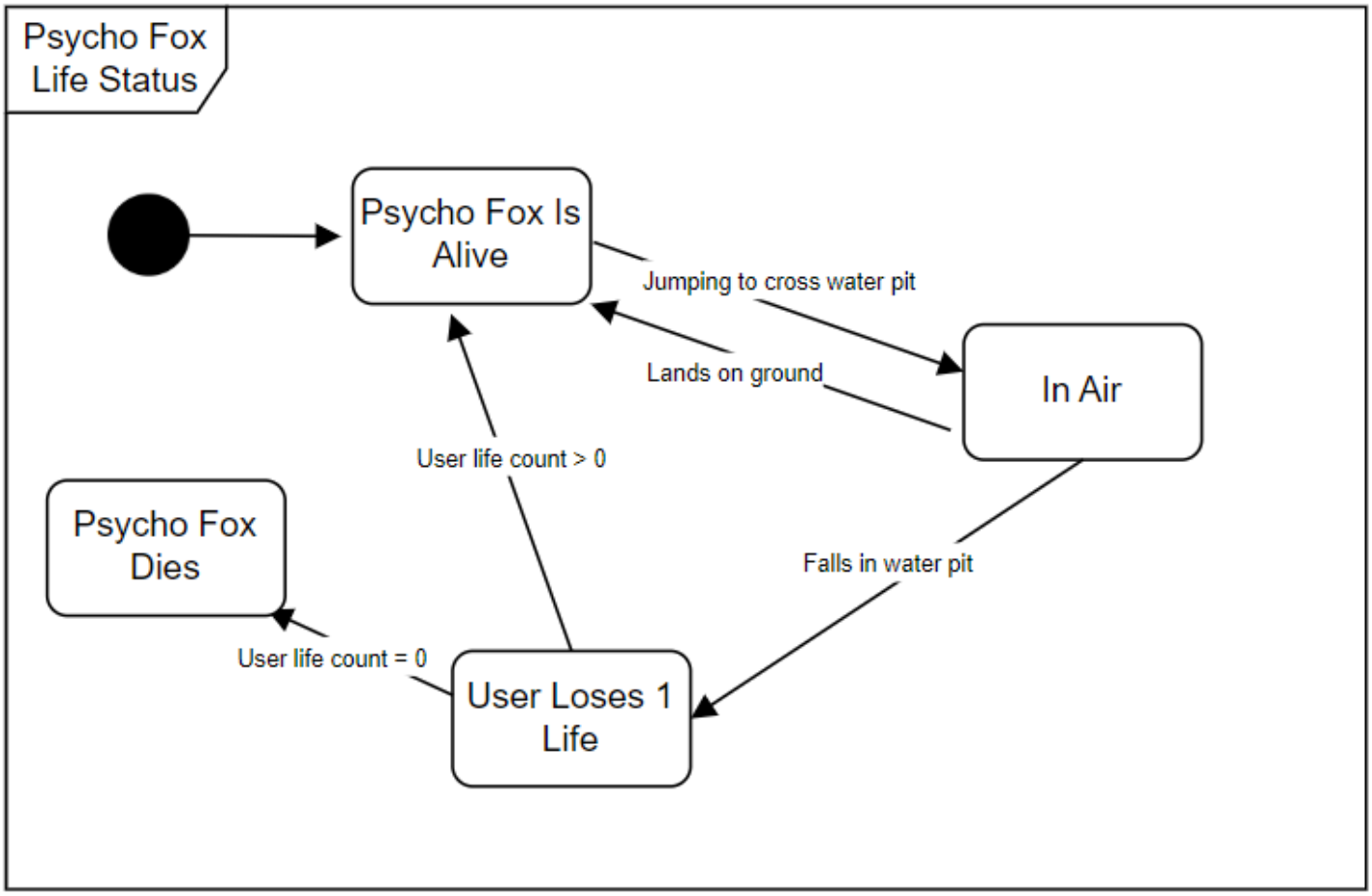

Figure 10: State Diagram for Psycho Fox Life Status - Falling

to Psycho Fox. We all have Github desktop downloaded onto our systems which allows us to make changes to our shared repository much easier. Github desktop is directly linked to our Psycho Fox Unity game which will allow us to easily make changes to our repository. To make the changes, we make a new branch from the main branch and alter the new features needed. Then, we commit/push the changes to the shared Github repository. After we have new branches, we can then merge them into a new main branch that we can all use to alter the program. If there are issues with making merges, we have previous main branches we can use to revert to previous versions of Psycho Fox. We can also pull old versions or the current version of main to modify or see old/current versions of the Psycho Fox game.

2.5 Data Dictionary

N/A

Requirements Description	Objective	Requested By	Design
UML Class Diagrams	Create Diagram for Code Development	Dr. Galloway	Finished
UML Case Diagrams	Create Diagram for Code Development	Dr. Galloway	Finished
UML Use Case Scenarios	Create Diagram for Code Development	Dr. Galloway	Finished
UML Sequence Diagrams	Create Diagram for Code Development	Dr. Galloway	Finished
UML State Diagrams	Create Diagram for Code Development	Dr. Galloway	Finished
UML Component Diagram	Create Diagram for Code Development	Dr. Galloway	Finished
UML Deployment Diagram	Create Diagram for Code Development	Dr. Galloway	Finished

Table 3: Diagram Requirements Traceability Table

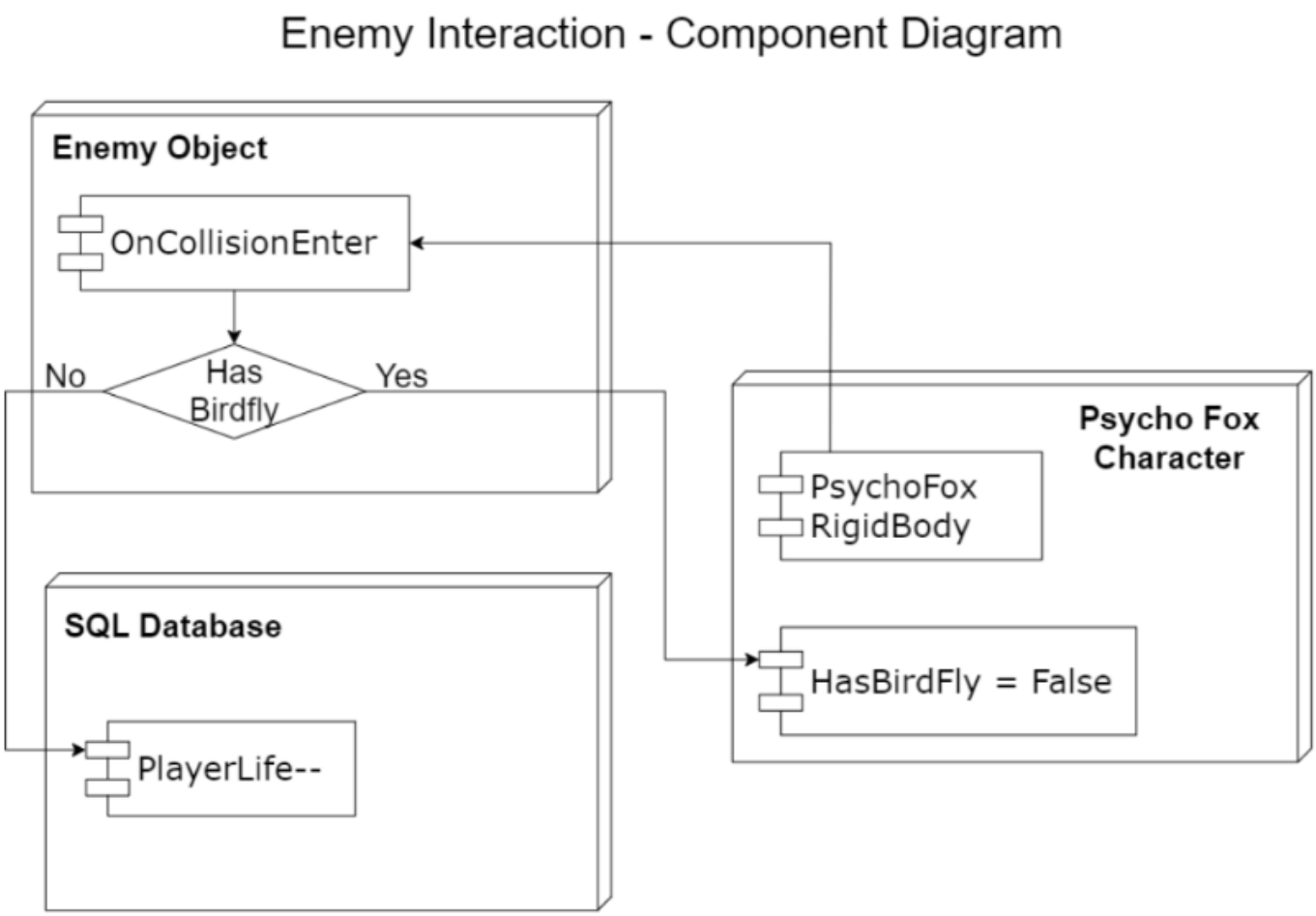

Figure 11: Component Diagram for Enemy Interaction.

2.6 User Experience

2.6.1 Gameplay Diagram

The 2 figures below showcase our gameplay diagram. These 2 diagrams show all of the controlled flow that will happen inside Psycho from the boot screen to the gameplay itself. This flow is also discussed in our storyboard/wireframe section and the following sections of the user experience below.

2.6.2 Gameplay Objectives

Our game, which is a rendition of the original Psycho Fox game, is designed for a few types of people: Fans of the original Psycho Fox game and who want to try our version of the beloved game, people who have played video games in this style (2D arcade style side scrolling platformer) but have not played Psycho Fox and are willing to give it a try, people who enjoy playing video games and want to give this classic a try, and people who have never played video games but are interested in doing so. As a result, it is a great game that is easily accessible to anybody as long as they have equipment that meets the hardware and software requirements, making it a great choice for a game to play at any time. In terms of the specific goals of the game, the objective is for the user to control a character called Psycho Fox and reach the end of the map. In our case, we will be recreating level 1 of the full game, so the goal of our game will be for Psycho Fox to make it to the end of that level. What makes this process challenging is that there will be enemies and obstacles that spawn around the map, so the user must dodge these in order to make it to the end of the level.

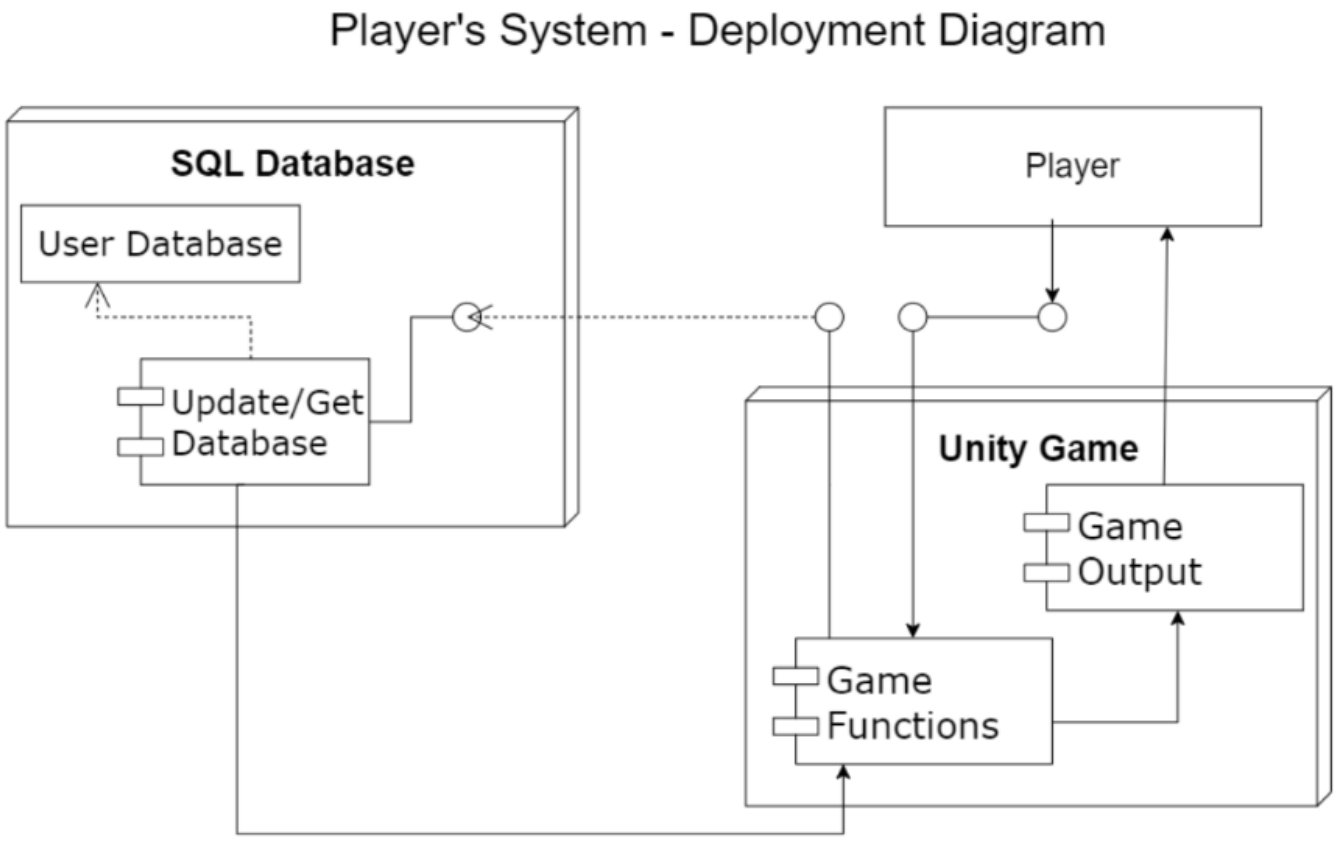

Figure 12: Deployment Diagram of Psycho Fox on a User's PC.

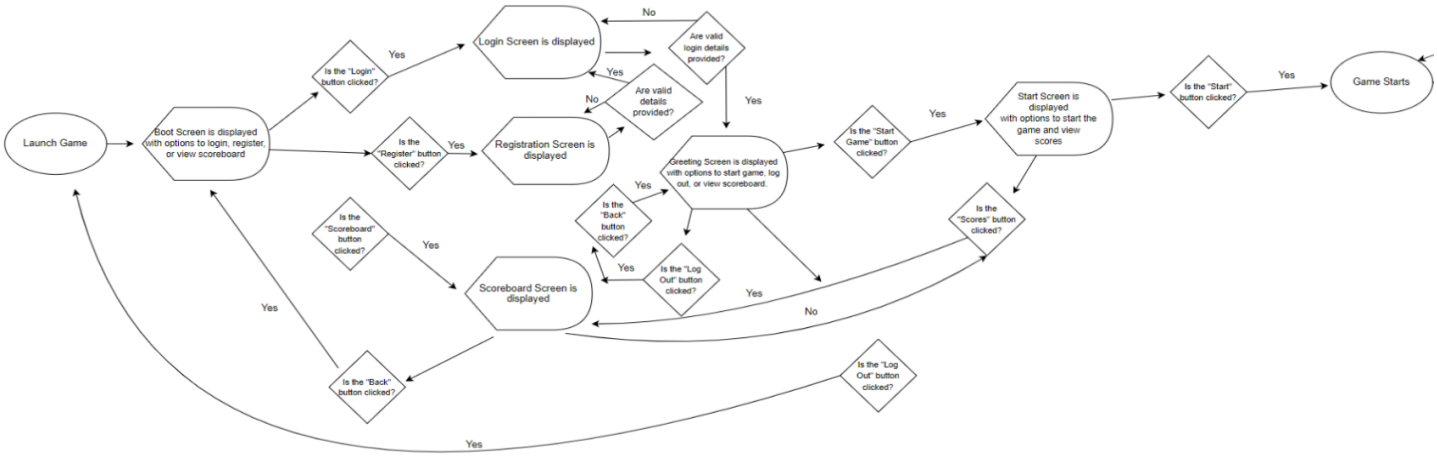

Figure 13: Psycho Fox Gameplay Diagram Part 1.

2.6.3 User Skillset

For our game, users will have to be proficient at using keyboard keys in order to navigate through the map and interact with objects along the way. The map primarily has "blocks" of terrain/platforms that Psycho Fox will have to jump on top of/around, bodies of water that Psycho Fox will have to jump over, enemies that Psycho Fox will have to kill, objects that Psycho Fox will have to interact with, and a swinging pole that Psycho Fox will have to grab on to in order to be launched farther in the map, so the user will also need to develop decently quick

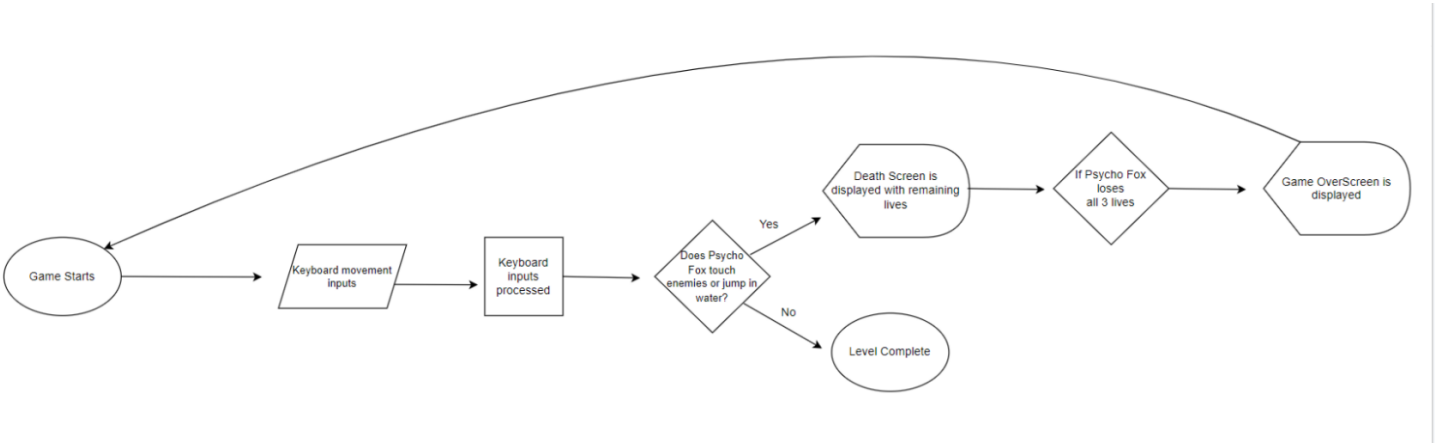

Figure 14: Psycho Fox Gameplay Diagram Part 1.

reflexes and have the ability to plan out their actions in advance so that they can pass these hurdles and progress through the level. Overall, our game does not require a high skillset in order to be completed, but it still provides users with a challenge, allowing them to be engaged and have a good time without feeling overwhelmed.

2.6.4 Gameplay Mechanics

Since our game is 2D, the user will utilize the arrow keys to control Psycho Fox and move around the map. By pressing either the left or right arrow key, the player to start running in those respective direction. As the game progresses, the speed of Psycho Fox increases. In addition, Psycho Fox can skid on water while running. The user can press the down arrow in order to duck, which can help navigate certain parts of the map. By pressing the Z key, the user can make Psycho Fox jump over obstacles and enemies. If the user holds the jump button, Psycho Fox will jump higher. Eggs are also placed around the map, and these contain objects that the player can interact with. If the user presses the X key, Psycho Fox will punch or throw whatever object they're holding. If Psycho Fox punches the egg, it will hatch and turn into something called Birdfly. Birdfly can be used as shield or be thrown as a weapon. In terms of the life system, the user will see a screen that shows the number of lives they have at tbe beginning of each level. This screen will have an animation of Psycho Fox running in place, followed by a multiplication sign, followed by the number of lives remaining. The ways Psycho Fox can die are by touching/running into spawned enemies and falling into the water. If either situation occurs, Psycho Fox will stop moving, have a sprite animation showing its face in shock, will shoot up into the sky, and will drop straight down. The screen will then transition once again to the screen displaying the number of lives, with the total number of lives being decreased by 1. After this, the game screen will appear again, and then the game will start. If the user is able to successfully traverse the map, the level has been passed. The time it took to pass the level will be recorded and stored in a database, which will be shown on the scoreboard.

2.6.5 Gameplay Items

There are eggs placed around the map that hatch Birdfly when punched by Psycho Fox. Birdfly can be used as a weapon, or it can be used as a shield/extra life for Psycho Fox. The bird power-up can be collected and can be thrown at enemies to kill them (this throwing mechanism replaces the punching mechanic, but the punching mechanic is restored either once Birdfly has been thrown and Psycho Fox is waiting for it to return like a boomerang, or once Psycho Fox loses Birdfly). Birdfly is positioned like a backpack, resting on the back of Psycho Fox. The bird protects the player from taking damage once. After the player has taken damage, they go back to dying on the next instance of damage and they will effectively lose/not have access to Birdfly. In terms of character selection, our game will focus primarily on Psycho Fox and the ability for the user to control the fox to complete the level, so we are not focusing on the implementation of other characters.

2.6.6 Gameplay Challenges

The primary challenges that the user/Psycho Fox will face around the map are platforms that Psycho Fox will have navigate around by jumping on top of or over, pits of water that Psycho Fox will have to jump over, enemies that will spawn around the map that Psycho Fox will have to kill, and a swinging pole that Psycho Fox will have to grab on to in order to launch across the map. Specifically about the enemies, they spawn around the map and have the ability to move back and forth. Some of these enemies will move on the ground while others will jump towards Psycho Fox, so the user will have to effectively dodge or kill these enemies in addition to making it past various other obstacles throughout the map in order to complete the level. Psycho Fox can kill these enemies by punching them, stomping them, or by throwing Birdfly at them, so the user should keep these in mind when nearing them around the map.

2.6.7 Gameplay Menu Screens

When the game is launched, the boot screen is displayed. This screen has a login button, register button, and scoreboard button that the user can click. If the user clicks the scoreboard button, the scoreboard screen is displayed with a chart with scores for all users who have played the game. By clicking the back button on this screen, the user will return to the boot screen. If the user clicks the register button, the registration page screen is displayed, and the user can provide a username and password in the given spaces (and confirm their password in the given space on that screen). Once they have inputted valid details, the user can click a button on that screen to get to the login screen. If the user already has an account, they can click on the login button on the boot screen and be taken to the login screen. Here, the user (whether they already have an account or if they just registered for one) can input their login details in the given spaces and hit the "Enter" button to advance to the greeting screen. The user can also click the back button on the login screen to be taken back to the boot screen. On the greeting screen, the user can click the start game button, logout button, or scoreboard button. If the user clicks the start game button, the start screen will be displayed. If the user clicks the logout button, they are taken back to the login screen. If the user clicks the scoreboard button, they are taken back to the scoreboard screen. Once the user is on the start screen, they can either click the start button or the scores button. If the user clicks the scores button, they will be taken back to the scoreboard page. If the user clicks the start button, the level start screen will be displayed, and the game will start. During the game, if Psycho Fox dies, the death screen will appear and will show the remaining number of lives (the initial count is 3 lives, and 1 life is lost each time Psycho Fox dies). Then, the user will be taken to the level start screen again and play the game. If the user keeps losing the level and uses up all 3 lives, the game over screen will be displayed, and the user will be redirected to the start screen. However, if the user eventually completes the level successfully, they will make it to the level end screen, which is just the end of the map.

2.6.8 Gameplay Heads-Up Display

The original Psycho Fox game does not utilize a heads-up-display, so our replica of the game will also not include a heads-up-display. We are planning on utilizing a timer for our scoreboard so will most likely be the only heads-up display on the screen.

2.6.9 Gameplay Art Style

Our rendition of Psycho Fox will have a very similar art style to that of the original. It will be a 2D arcade-style platformer, have bright colors, have a cartoonish feel, have a pixelated feel, have sprite animations similar to the original, and a very similar looking map.

2.6.10 Gameplay Audio

The audio we will use in our game will be very similar to that of the original Psycho Fox game. The main soundtrack will play throughout the game in the background. There will also be audio incorporated when Psycho Fox moves forward, when Psycho Fox moves backward, when Psycho Fox jumps, when Psycho Fox punches or throws an object, when Psycho Fox skids on water, when Psycho Fox dies, and when Psycho Fox makes it to the end of the level. Other than this, we may incorporate more audio into other parts of our game based on how much

progress we make with our main requirements. Because our goal is to replicate the aspects of our game that we will implement as close as possible to the original Psycho Fox game, we will get the required audio files from a credible resource to ensure that the quality is up to the mark.

3 Non-Functional Product Details

3.1 Product Security

3.1.1 Approach to Security in all Process Steps

Our approach to security in the main steps of our project is to keep only what needs to be seen sent to the player. This also means that we will only receive what is absolutely needed from the player as well. First, we will ensure that our databases store information in a safe way. We don't have a solid plan yet, but we plan to have database monitoring and other types of security implemented to keep our user's data safe and secure. We could also implement algorithms that make checks within the game and inside the database to ensure that the information does not leak. We also plan on implementing our security boundaries. These are boundaries that are placed between the user and the game, as well as between what the game and database have access to. This will keep game data secure. With these boundaries, we can make sure that there aren't any miscommunications within our systems, and that only the data that needs to get to certain places will reach its correct place. We ultimately decided that our approach to security should keep the user safe, the data reliable, and the experience great. We will continue to research the security methods to maintain a good working game, and with these security methods make sure to keep the player will have a good experience. The most important thing to know in our approach is that we use secure protocols and well-researched methods to ensure our user's safety. we will derive our security protocols from the OWASP Secure Coding Practices Quick Reference Guide.

3.1.2 Security Threat Model

The security threat model for our game has 3 main entities and 2 boundaries. The first entity that exists is the gamer, or person that is playing the game. The player can create inputs into the game. Once this happens, the game will output things onto the screen with various sound queues. The next entity that exists is the game. The game serves as a middleman that interacts with both the player and the game database. The game interacts with its files when completing game-related functions. The third entity that exists is the database. The game database holds player-related information like login information and score information related to players for a scoreboard. The game will interact with the database when database functions need to be called. The first boundary in this game is the user/game boundary. This boundary limits what the player can do when playing the game. The gamer will only be able to control the character, navigate through the menu, and login to the game. The second boundary is the game/database boundary. This boundary limits what the game has access to regarding the database. The game will only be allowed to call for certain data and write data regarding scores and registration. Users will not be allowed to communicate directly with the database and must pass information through the game.

3.1.3 Security Levels

The game we are recreating will not have more than one security level. The only type of user that will be able to be created is a player. The player will be able to register with a username and password. This username and password will also be used to log the user into their account. This account will also be used to track user scores similar to an old arcade scoreboard. The player must be logged into the game to play the game. Due to this, the player must first register before being able to log in. The user that is logged in will be able to complete the stage and upon level completion, the user's score will be saved inside of the database.

3.2 Product Performance

3.2.1 Product Performance Requirements

Our game is a simple remake of the first level of a 1980s game, therefore our performance requirements will be rather bare compared to modern games. Taking also into consideration that our game is made in the Unity game

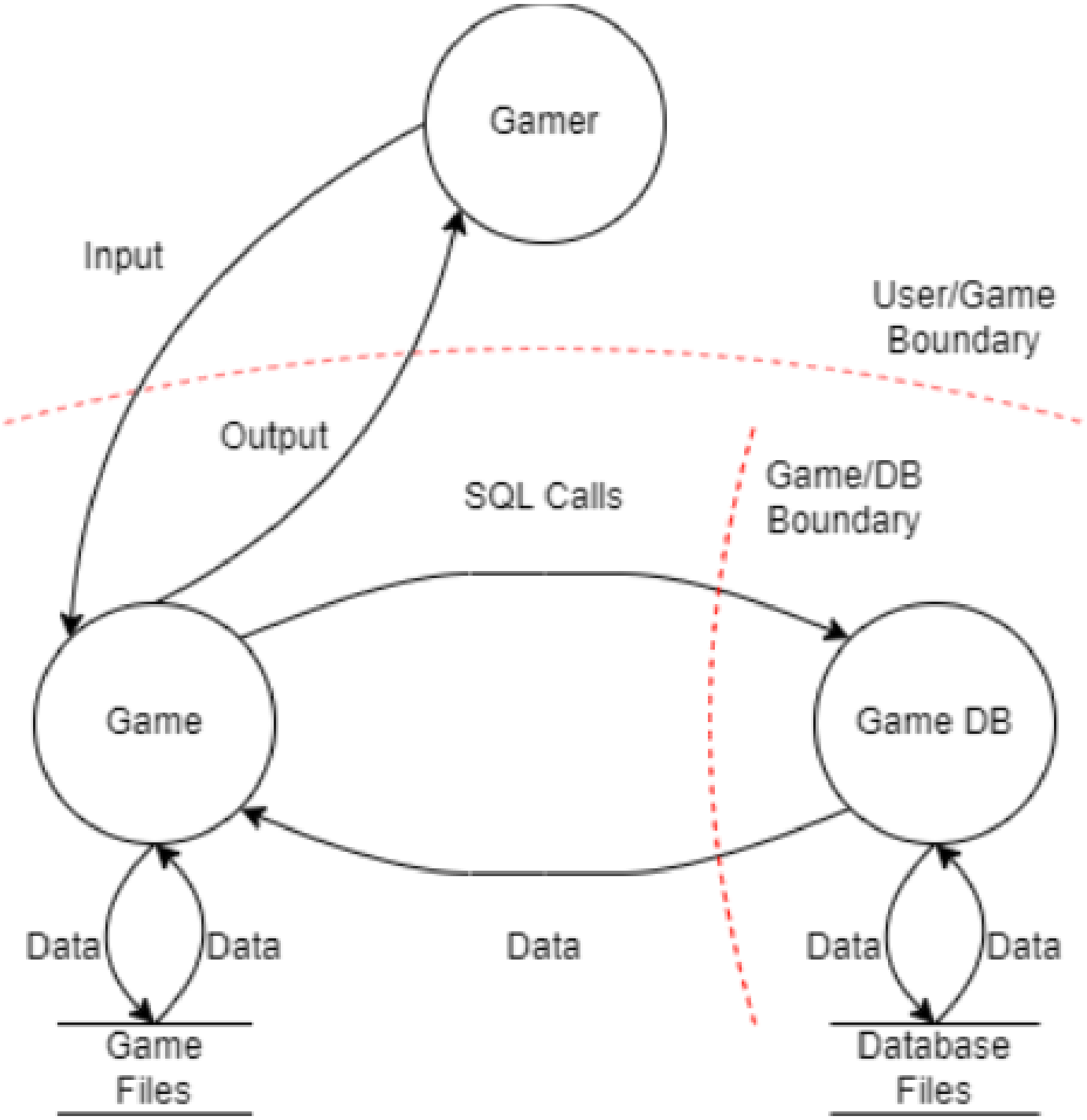

Figure 15: Security Threat Model for Psycho Fox.

engine, we can see that the minimum requirements will include whatever requirements are needed to run Unity. Knowing this we can see that the performance requirements include at least a Windows 7 operating system, or macOS High Sierra, any CPU with x64 architecture support, and direct X 10 or above supported with your GPU. These requirements are the base minimum to run our game on modern systems. After testing and measuring our game we can conclude that these requirements are true and when met can run our game fluently. With more performance tweaks and advances we can improve our game even more and possibly have it run on lesser power systems in the future. Systems with at least 4 core CPUs and 8GB of RAM allow users to comfortably play the game. Referring to our non-functional requirements, this will allow our game to run smoothly at 60fps and effectively load assets and audio files. Security and performance metrics are also being collected by utilizing timers that are set up before and after specific function calls.

3.2.2 Measurable Performance Objectives

The measurable performance objectives for our game include statistics such as hardware and software bottlenecks, and synthetic performance benchmarks that show performance based on file size and thread usage. The hardware bottlenecks that we are aware of include the fact that some of our hardware components run faster than others. Some machines are measurably slower than others and the hardware can become a bottleneck for the game. We have measured this with timers and other metrics that show how long the code within our game takes to execute or run. Our synthetic benchmarks that we have ran give us an idea of how a CPU will handle the files of our game. Tests prove that at 8 threads, our game will run as effectively as It can. From our collected data shown in the figures, we can also see that with an increased thread count, the execution time will exponentially increase. Finally, the file size vs throughput metric informs us that the write speed and read speed are most effective for files that are less than 16MB in size.

3.2.3 Application Workload

Rather than assuming how much time users spent on each screen in our application, we set up timers that are enabled and disabled when a page is opened or closed. When the user starts the application from the boot screen, they're presented with the menu that currently only contains a start button. When the user clicks the button, the menu screen time is recorded and the main game scene is loaded. When the user dies or quits the application, the game timer is recorded. In the future, we'd like to perform these tests again when we have our database implemented and other functions working. Between three people in our group, each member test played our game five times and each time interval was recorded. For each screen, the average amount of time was calculated and used to gauge the total amount of time each user spends on the application overall. From our data, we found that users typically spend 15% of their time using the program on the menu screen and 85% of their time playing the game. On average, users typically spend 3.68 seconds on the start menu screen and 22.17 seconds playing the game. This means users typically use our application for 25.85 seconds. The most time spent on the menu screen across all sessions was 20 seconds, and the most time spent playing the game was 47.04 seconds. For the system workload, some of the processes that are running are the Unity Editor and the different processes associated with it. This includes the compilation of assets, shader logic, and the system used to handle game scripts in C#. Our game is currently only running on one system which means these processes aren't being distributed; however, once our database is implemented we could choose to have some of these interactions spread across a remote server. The transfer of data between the user's system and our remote database would distribute the database calls across two different machines. While playing the game, there are different performance states that utilize the machine more than others. These can be described as peak and normal states. Using the analysis profiler that's built into the Unity Editor, we recorded the CPU and memory usage over a 10 minute interval to determine where the game's performance idles at and where it spikes. Considering the background processes that are running in sync with the Unity Editor, our CPU's normal activity state allows game processes to complete in 4.55ms on average, utilizing only 20% of the CPU. When our CPU peaks, processes take around 13.69ms to complete on average and utilize 45% of the CPU. Our memory data was much more consistent and utilized 2.3 GB of RAM throughout each playthrough with spikes that were unnoticeable. Considering this information, we were able to conclude that our data aligned with our performance requirements in the sense that the 2.4 Ghz processor with 4 cores and 8GB of RAM that were used was only partially utilized to handle all processes within the game

3.2.4 Hardware and Software Bottlenecks

In our testing for bottlenecks, we implemented a few things to be able to test for issues when it came to a component bottlenecking performance. For future reference in this section, the charts referenced can be seen in the Performance Testing section. We did notice slight bottlenecking with our game utilizing more CPU usage than GPU usage. In our testing, CPU usage was always higher than the GPU regarding ms(milliseconds). This was gathered through the Unity Profiler tool. The ms(milliseconds) were then converted into fps which can be seen in the chart below made in Excel. This shows the average fps and lowest fps seen based on the ms of the CPU and GPU. Because the CPU usage was higher, we will continue to watch this to make sure nothing hurts the performance of our system despite achieving exceptionally high fps numbers which is what we are aiming for with our performance requirements being minimal. We also saw very minimal usage of memory inside of our game.

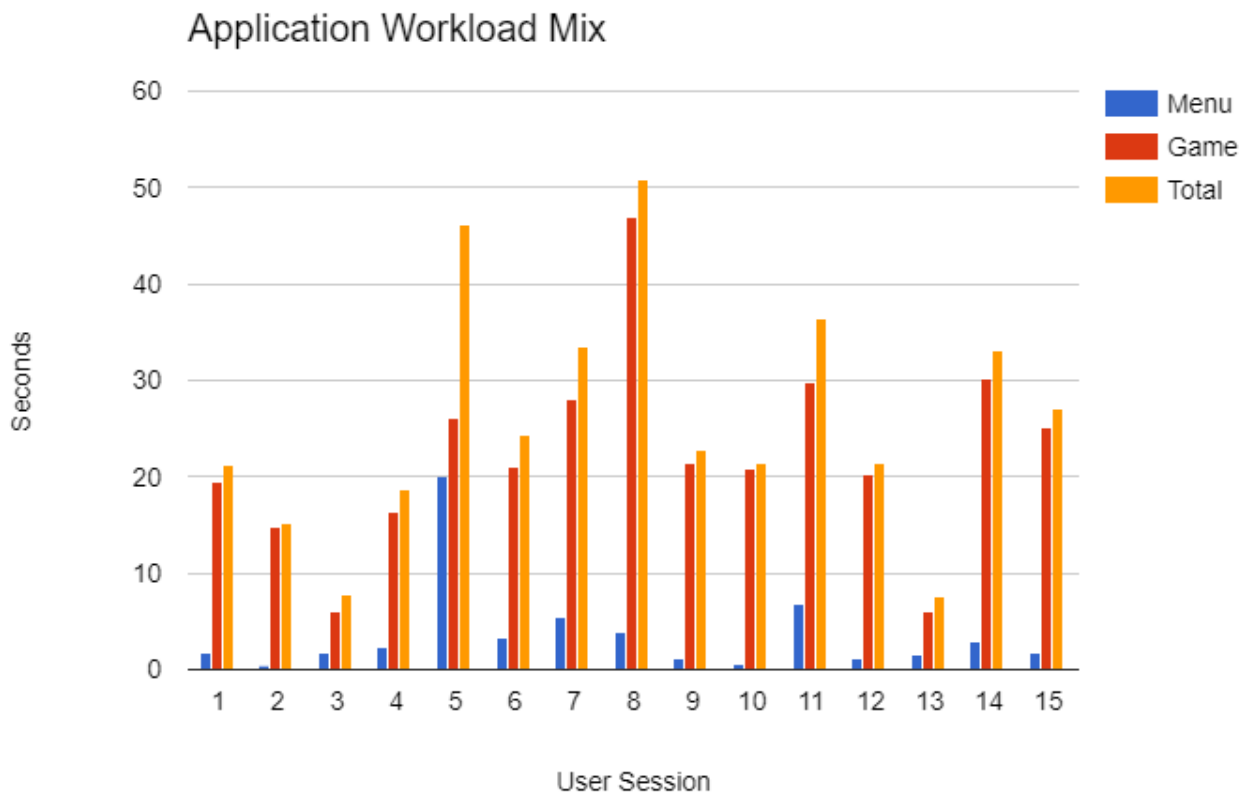

Figure 16: Application Workload Mix Chart

During our testing, there was around 1 GB of RAM used on average. Meanwhile, the system was only allocated at most 1.71 GB of memory. The reference to this data from the Unity Profiler tool can be found in the Performance Test section below. A whole breakdown of the memory being used in real-time can be seen as well.

3.2.5 Synthetic Performance Benchmarks

To test for synthetic performance benchmarks on our system we used Sysbench. Sysbench is a Linux program that stress tests different hardware components and allows users to pick out meaningful data that relates to their machine. From our tests we were able to derive data about our target hardware's CPU and Disk usage. To perform our tests, we used Sysbench's CPU program that calculates every prime number up to a specified number. In this case, we generated every prime up to 10,000. This stresses our CPU and allows us to gather information about its capabilities. From the graphs provided in Figures 16 and 17, we were able to find that our CPU is most effectively utilized when 8 threads are being processed. As more and more threads are used, our data caps out and allows us to identify a potential bottleneck that could relate to our software. The execution time and number of processes plateaus and provides no additional performance increase. For the file throughput tests, we found a similar trend of initially increasing performance that eventually plateaus after a certain limit has been reached. Once 16MB files were being transferred, our read and write speed capped out at 40MB/s and 28MB/s respectively. Once again, this data has the potential to relate to our software and allows us to gauge how taxing our program needs to be. This data also aligns with our performance requirements since the target hardware that our tests were performed on meet the same requirements.

3.2.6 Performance Tests

One of the main performance tests that we performed refers to how quickly each of our methods were running. To test the speed of our functions, we set up stopwatch objects that started before a function was called and

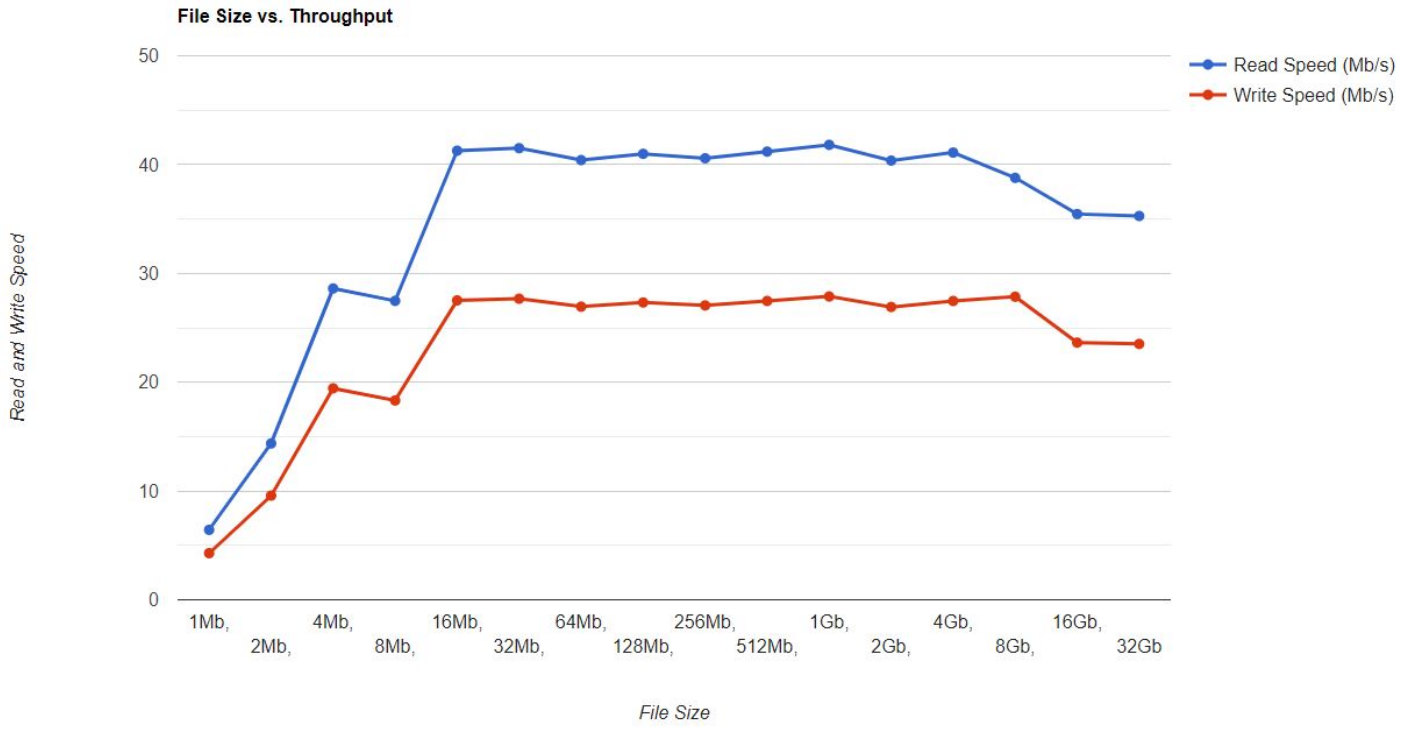

Figure 17: Graph of File Size vs. Throughput.

stopped after they finished executing. Timers were set up for our scene loading function, enemy spawner, enemy updaters, collider functions, and animation methods. Referring to Figure 20, we found that each of our functions were running within a fraction of a millisecond with the exception of our enemy spawning function. This data may not seem very useful, but having the timers set in place for when we begin to develop more complex methods will allow us to clearly see where our program experiences a performance break. What we have now is an indication that our systems are outperforming the software we have developed, which can provide us with information relating to bottlenecks.

Another performance test that we set up relates to how quickly our machines were able to manage file input and output. Using Sysbench, we set up a test that records the read and write speed of our disks for files of different sizes. As previously mentioned, we found that files less than 16MB are most effectively transferred considering the performance of our target hardware. The throughput of our machine with files over 16MB capped out and moved consistently at 40Mb/s.

Finally, we used Sysbench to measure CPU performance through stress testing. By setting up a prime number calculator, we were able to derive the ideal number of threads to handle our product considering our target hardware. By utilizing eight threads, we are able to handle the largest number of events in the least amount of time, allowing us to maximize our performance. This aligns with our first test since our functions were able to execute in an unnoticeable amount of time.

4 Software Testing

4.1 Software Testing Plan Template

Test Plan Identifier:
Introduction:
Test item:
Features to test/not to test:
Approach:

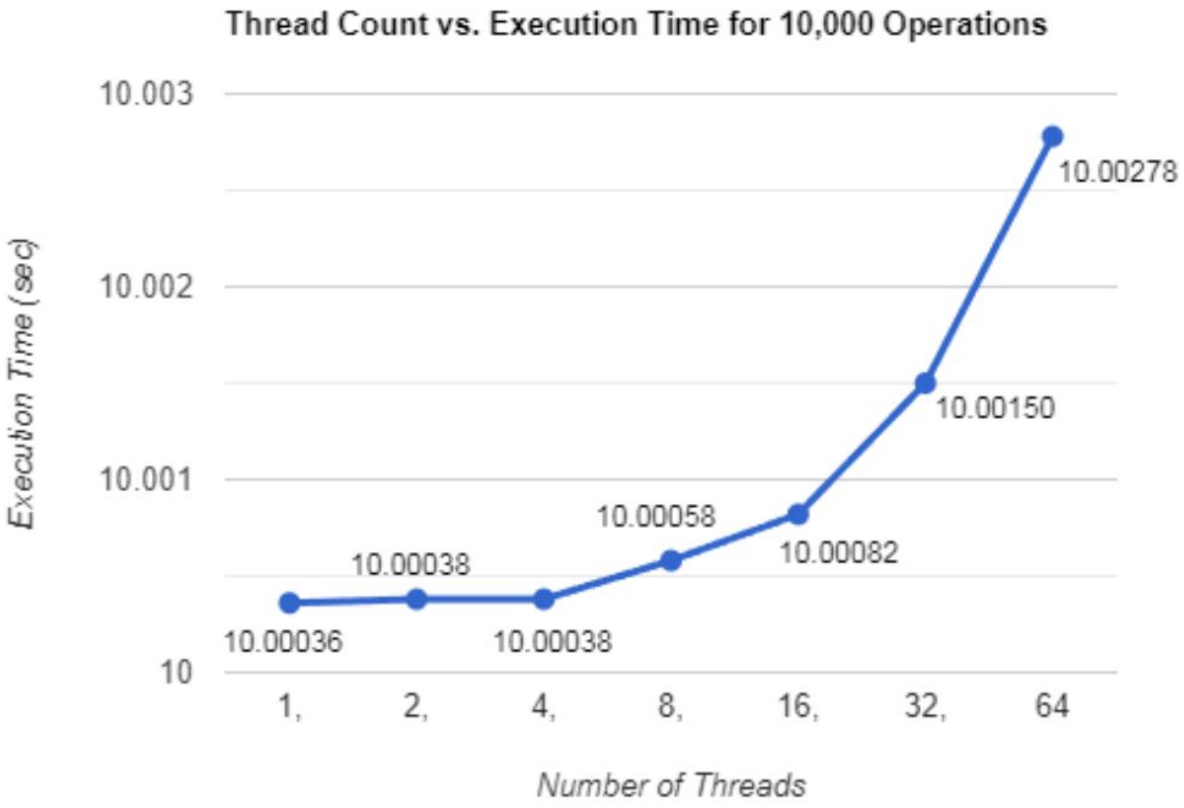

Figure 18: Graph of Thread Count vs. Execution Time

Test deliverables:
Item pass/fail criteria:
Environmental needs:
Responsibilities:
Staffing and training needs:
Schedule:
Risks and Mitigation:
Approvals:

4.2 Unit Testing

Test Plan Identifier: PF-UT- 001: Death Condition and Logic Snippet

Introduction: This unit test's objective is to test the complexity of the death conditions to optimize for functional correctness/completeness, enhance the algorithms/performance, and deal with any security concerns that deal with these programs. When Psycho Fox dies, several values are updated that deal with implementations that make it appear like the character is dying. Since we're performing a static test and not actually running our game, we are going to be looking at the entry and exit criteria for our script file and making sure that they align with our requirements.

Test item: The Software Under Test (SUT) for this specific test deals with the death script file and objects in Psycho Fox that handle how Psycho fox can die. There are several objects that can cause death depending on the tag of the object that Psycho Fox interacts with. For this test, we will be primarily focusing on the death class and its different data types, function calls, and structure.

Features to test/not to test: The features being tested in this Unit test deal with the interaction of the object that will kill Psycho Fox and make him go into the death state. All other objects that Pyscho Fox interacts

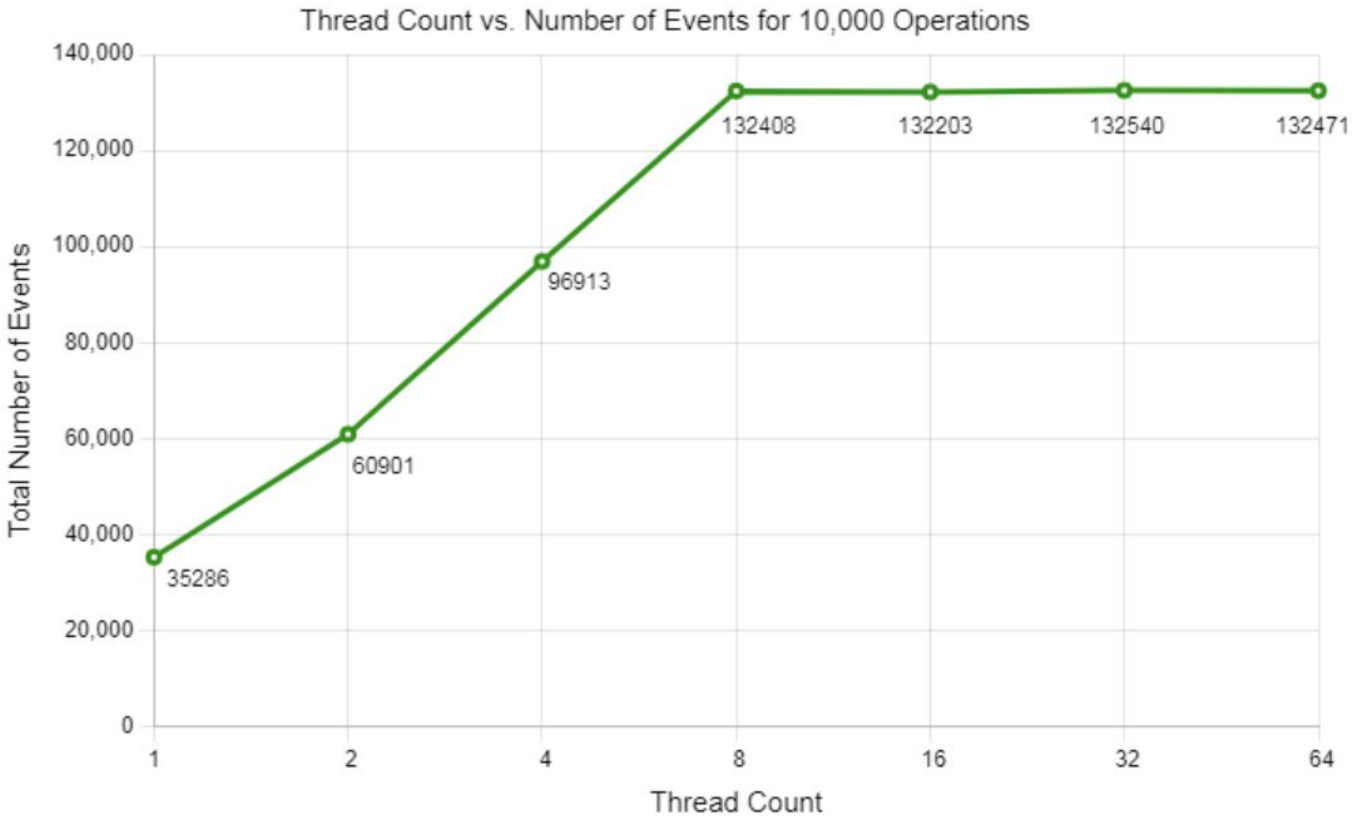

Figure 19: Graph of Thread Count vs. Number of Executions

with will not be tested since they're beyond the scope of this function. For this test, we'll be checking to make sure that audio cues, scene managers, and object vectors are set properly.

Approach: The approach strategy for this test deals with the functionality of the death.cs script file. We used both automatic and manual methods for testing everything. Each member of our group looked over and reviewed the file to check for what we believed to be a correct implementation. Visual Studio Code also made suggestions for what we could fix and allowed us to automatically perform testing. This allowed us to quickly determine errors that we made and speed up our code review.

Test deliverables: This test will be a pass/fail for if the Psycho Fox character properly interacts with enemies when struck with an object with the Death tag. This data will be gathered in a chart which will be located inside of the Unit Tests and Results section. One of the ways we'll test this implementation will be by comparing desired outputs to actual ones. This involves using the different basis paths that we created from our flow graph for this test plan.

Item pass/fail criteria: The Pass/Fail criteria is as follows: Pass: Audio Plays, Psycho Fox's animator triggers are updated properly, the player's alive state is changed to false, the sprite renderer is disabled. Print statements are also in place to allow us to integrate this unit properly once we start combining it with other classes or files. Fail: None or some of the things occur in the Pass section.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. This includes an IDE that is capable of compiling C# scripts since that's the programming language that our game is written in. For our tests, we used Visual Studio to compile our scripts.

Responsibilities: First and foremost, the tester needs to have the environmental needs implemented in their system. The tester must adequately run all tests / basis paths that interact with the Death script. This means that the tester would need to run every different entry and exit criteria pair for a proper test result to be generated. The tester then needs to record the values they come across and compare them to the pass/fail criteria for the

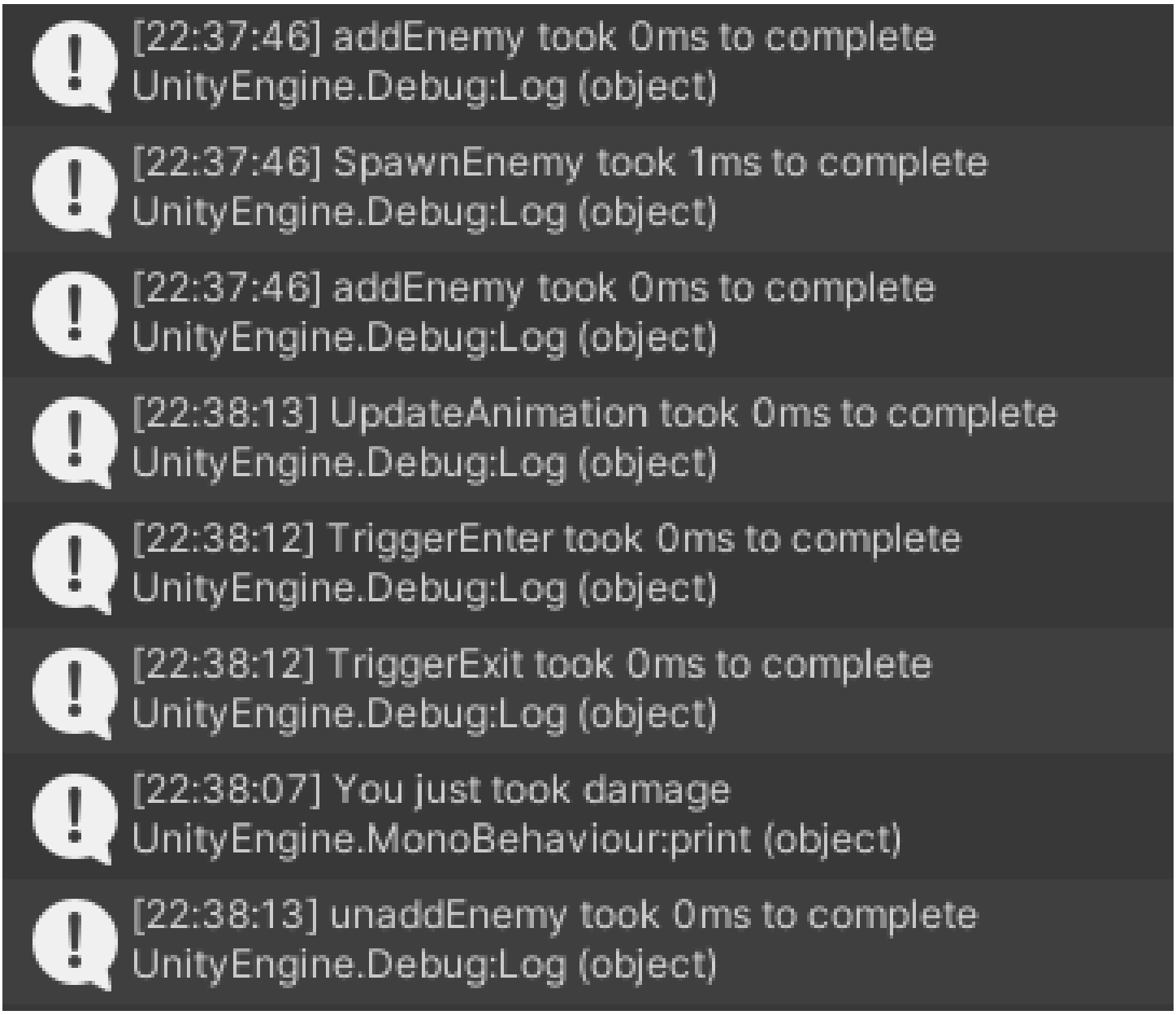

Figure 20: Function Timer Tests.

test case.

Staffing and training needs: Little training needs to be done to test this specific unit test. Some of the things that the tester will need to prepare to generate useful information include the audio files, scene managers, and player manager states. These objects align with the exit criteria for our test. The tester will need to compare desired outputs to actual ones to complete our unit test.

Schedule: This unit test, along with all other testing occurred during the "Testing" phase of our Gantt Chart. By the end of the testing phase, we plan to have a report generated that describes the outcome of our tests. This also includes charts that describe the number of passing test cases that we had and the cyclomatic complexities of each section.

Risks and Mitigation: There are a few risks that may appear when testing for this. Some issues that may arise may deal with the collisions not occurring for some reason. If this would occur, we would have to fix this bug and resolve this issue. Also, if the proper scenes do not appear when a collision with an object that causes death we would need to fix this as well. If our script properly works, we can assume that our code for death does work properly and as intended.

Approvals: As of 11/22/23, our group was ready to begin testing this.

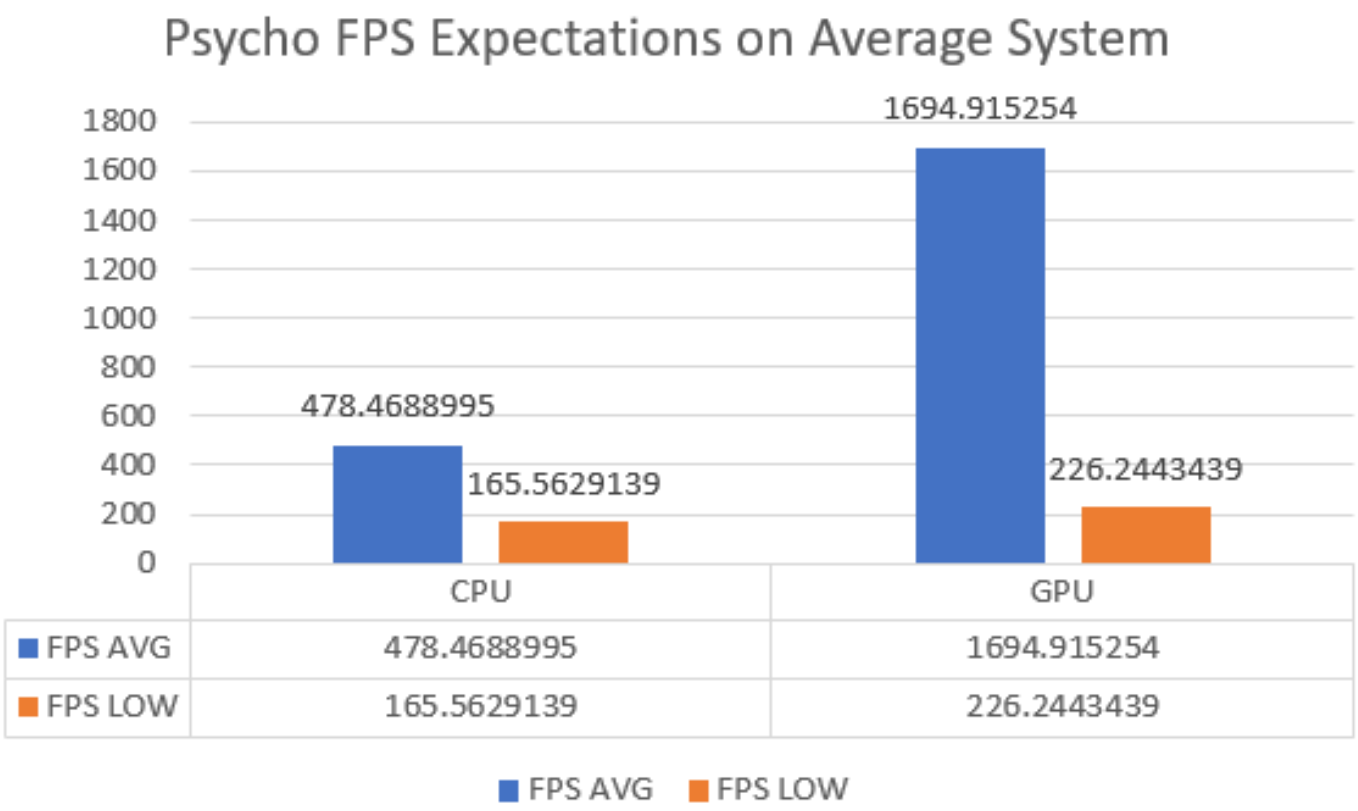

Figure 21: FPS Performance in Psycho Fox.

Test Plan Identifier: PF-UT- 002: Red Hopper AI class

Introduction: This test aims to determine if the movement script for the Red Hopper enemy produces the correct outputs. The unit being tested is the RedHopperMovement class. This enemy jumps continuously and moves towards the player based on their x location, so the output we'd be looking for would be the correct Vector2 or the lack of a force being applied. There are two separate functions that determine the class's output when execution starts. We tested the functionality of our code both manually and automatically through Visual Studio Code.

Test item: The SUT for this unit test would be the RedHopperMovement class itself. This class determines different forces that are applied to Red Hopper objects and which direction Red Hoppers should move in order to dynamically respond to player inputs. Considering this, this test only focuses on the RedHopperMovement class in particular and doesn't consider how enemy movements affect the player.

Features to test/not to test: One of the main features that we'll be testing involves how Red Hopper objects behave and respond to player inputs. We'll be checking to see if Red Hoppers appear to automatically jump after touching the ground and move slightly towards the player. This is the most important feature of this test. We won't be testing how this enemy interacts with the player since that's beyond the scope of the movement file.

Approach: We'll be doing functional testing both automatically and manually. We'll be testing to make sure that our outputs allow game objects to behave properly, so our group sat down to manually analyze this class file. We also used Visual Studio Code's code editor to automatically suggest corrections that affected our test results.

Test deliverables: By creating a flow graph of our class file, we were able to calculate a cyclomatic complexity to give us an idea as to how accurate our code was. This means the test case that we generated would relate to a minimum cyclomatic complexity. This allowed us to generate a report that suggests whether or not our file passed or failed the test case.

Item pass/fail criteria: The Pass/Fail criteria is as follows: Pass: The entry criteria for this class include the player's location and the Red Hopper's current rigid body characteristics. This test will pass if the exit criteria match the expected behavior described for Red Hoppers. This includes the correct Vector2 objects or the lack of them. Fail: The script file exits with Vector2 objects that move the Red Hopper object in a different direction.

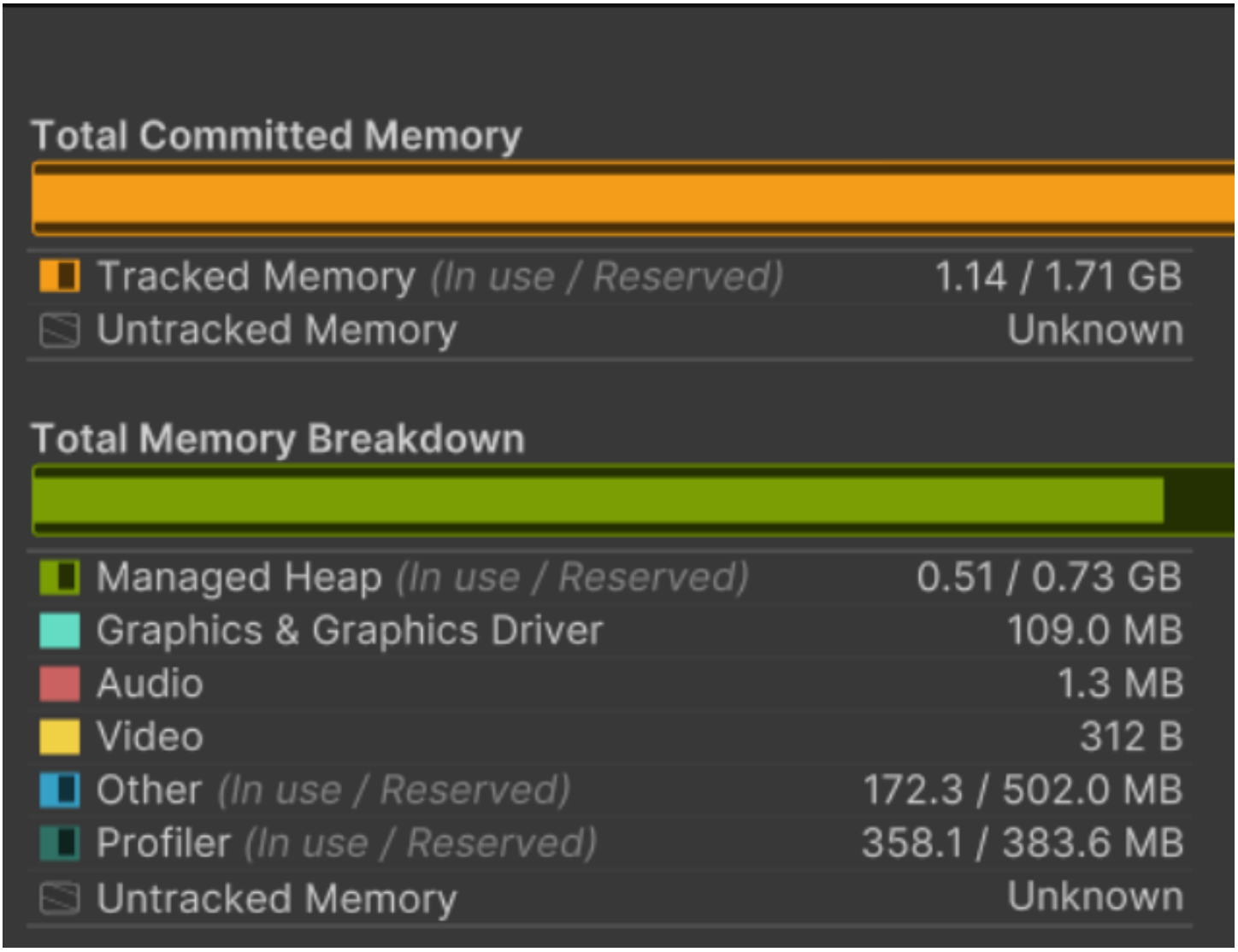

Figure 22: RAM Performance in Psycho Fox.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. This includes an IDE that is capable of compiling C# scripts since that's the programming language that our game is written in. For our tests, we used Visual Studio to compile our scripts.

Responsibilities: The tester for this plan needs to have access to the environmental needs before they can begin testing. After that, they need to be able to go through every basis path for the flow graph created to make sure that correct Vector2 objects are outputted as the exit criteria.

Staffing and training needs: The amount of training required to perform these tests is very low. The tester simply needs to be able to compare actual outputs to desired outputs and check that the vector objects are within an appropriate magnitude and direction. This is because these are the requirements for the Red Hopper object to move correctly.

Schedule: All of the testing that we performed for this plan was done during the testing phase outlined on our Gantt chart. We plan to set aside an hour to go through the different basis paths that we have created to generate results regarding the functionality of our unit.

Risks and Mitigation: One of the known unknown risks that we will encounter regard syntax errors that we'll find in our file. This is something that our team will be able to fix fairly quickly and is not a major priority. We assume that the enemy objects will behave as expected, but they could produce results that affect other game objects and be incorrectly implemented. This is a more important risk and will require more attention. Some of the things we're preparing to do in order to handle these risks and properly perform this test involve allocating time to debug certain functions and further reviewing our code.

Approvals: As of 11/26/23, our group agreed that this test was ready for execution.

Test Plan Identifier: PF-UT- 003: Spring box logic snippet

Introduction: This unit test's objective is to test the complexity of the spring box to optimize for functional correctness/completeness, enhance the algorithm's performance, and deal with any security concerns that deal with these programs. We will be performing a static test on our code snippet which means that we will be paying attention to the different exit criteria values and how they relate to our requirements.

Test item: The Software Under Test (SUT) for this specific test deals with the scripts and objects in Psycho Fox that deal with how the spring boxes throughout the map work. The player utilizes their collider to interact with an object holding a "spring" tag. The unit for this test is relatively small, but it makes up an important aspect of our game.

Features to test/not to test: The in-scope feature being tested deals with the interaction of the object that will boost/bounce Psycho Fox when it lands on the box. Essentially, we will be checking that the force or Vector2 object that's being applied to the player is correct. Other object interactions are out of the scope of this unit test and are a low priority compared to the actual spring box functionality.

Approach: The strategy to test the spring boxes utilizes functional testing. Functional testing is used to check whether the spring box is correctly boosting Psycho Fox in the air when it comes into contact with the top of the box. This means we'll be looking for correct Vector2 values.

Test deliverables: This test will be used to determine the pass or fail status of whether the spring box is correctly implemented. This involves collecting data regarding the outcomes from our different basis paths. This data will be gathered and displayed in a chart that will be located in the Unit Tests and Results section.

Item pass/fail criteria: Pass criteria: A vertical force is applied through a Vector2 to Psycho Fox after coming in contact with a spring collider. Fail criteria: No vertical force is applied through a vector force to Psycho Fox after coming in contact with a spring collider.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. This includes an IDE that is capable of compiling C# scripts since that's the programming language that our game is written in. For our tests, we used Visual Studio to compile our scripts.

Responsibilities: The tester must understand how the spring box works and perform tests accordingly. In this case, they should know that the spring box should activate and boost Psycho Fox when it comes into contact with anywhere on the top of the box. This means that when they're looking over our code, they should understand how Vector2 objects are formatted and how they interact with our player's rigid body component.

Staffing and training needs: A small level of training is required to ensure that the tester understands how to compare expected values to actual ones and record appropriate results. They should be able to identify incorrect forces being applied and recognize when the test cases fail.

Schedule: This unit test, in addition to all of the other testing, occurred during the "Testing" phase on our Gantt Chart. All of the following tests for this specific test were tested on the same day.

Risks and Mitigation: The main risk during this test is that no vertical force is applied through a vector force to Psycho Fox after coming in contact with a spring collider. In this case, it can be assumed that either the collider on the box is not working, the collider for Psycho Fox is not working, or both are not working. A mitigation plan that can be utilized to resolve this error is to review the collider implementation of both the spring boxes and Psycho Fox, test them individually to ensure that they are working as expected, and test them together once changes are made. If the vertical force is applied correctly, it can be assumed that the spring box is working as expected.

Approvals: As of 11/22/23, our group was ready to begin testing this.

4.2.1 Source Code Coverage Tests

Test Plan Identifier: PF-UT-001

Cyclomatic Complexity
E = number of edges
N = number of nodes
P = set of connected components

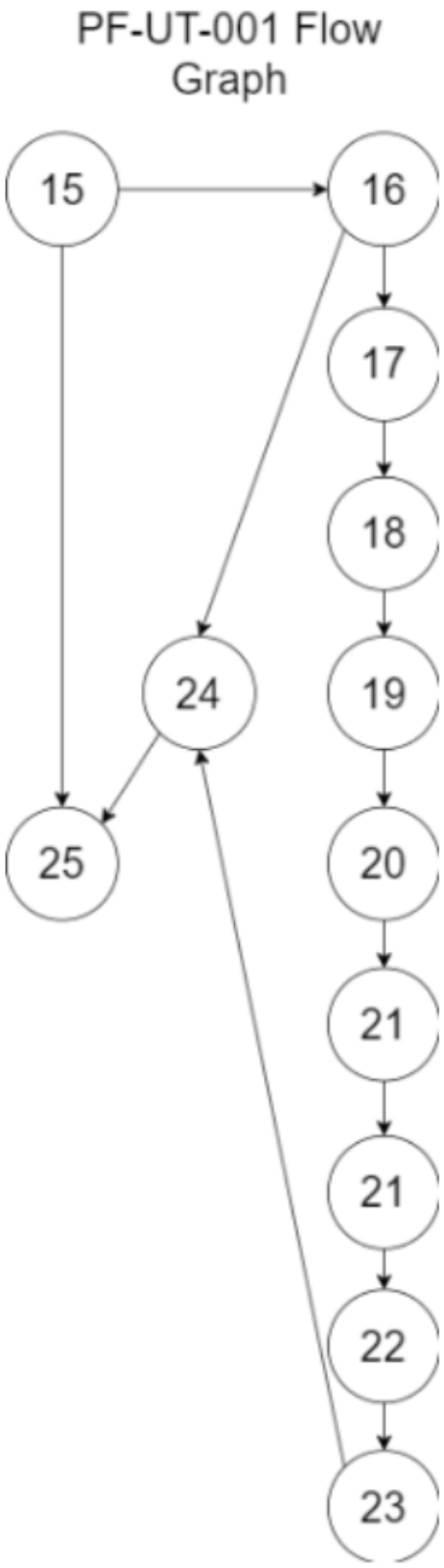

Figure 23: Flow Graph of PF-UT-001

CC = E - N + 2P
CC = 12 - 11 + 2(1)
CC = 3

Basis Paths
1. 15, 25
2. 15, 16, 24, 25
3. 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25

Unit Test Cases

This section identifies whether or not the tests meet the Pass criteria. More specifically, if Psycho Fox hits a "Death" tagged object, the appropriate logic scripts will be updated and Psycho Fox will die. The first basis path describes when the player isn't touching a death collider. The second basis path represents when the player makes contact with a death collider, but the player is already attacking. Essentially, this overrides the death script's main contents. Finally, the third basis path describes when the player makes contact with a death collider and they aren't attacking. This sets the values described in our test case to the appropriate values.

Test Plan Identifier: PF-UT-002

Cyclomatic Complexity
E = number of edges
N = number of nodes
P = set of connected components
CC = E - N + 2P
CC = 14 - 13 + 2(2)
CC = 4

Basis Paths
1. 1, 2, 3, 5, 8
2. 1, 2, 3, 4, 13, 14, 17, 8
3. 1, 2, 3, 4, 13, 15, 16, 17, 8
4. 1, 6, 7, 8

Unit Test Cases

The first basis path shows when the player is alive and within range of the enemy, but the enemy isn't touching the ground. The second basis path shows when the enemy meets all the requirements to jump and is to the right of the player. The third basis path shows when the enemy meets all the requirements to jump and is to the left of the player. The final basis path shows when the player isn't alive.

Test Plan Identifier: PF-UT-003

Cyclomatic Complexity
E = number of edges
N = number of nodes
P = set of connected components
CC = E - N + 2P
CC = 3 - 3 + 2(1)
CC = 4

Basis Paths
1. 81, 82, 83

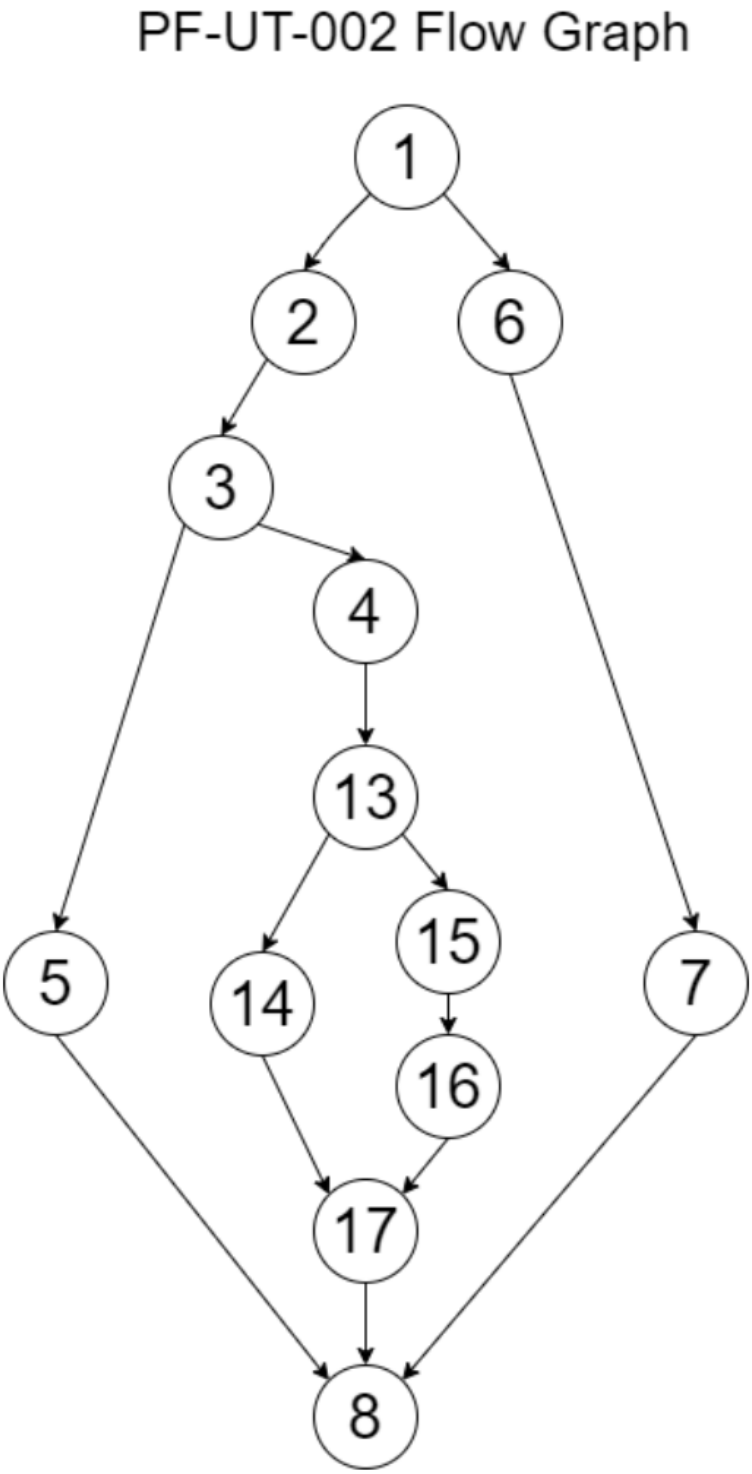

Figure 24: Flow Graph of PF-UT-002

2. 81, 83

Unit Test Cases

This is a relatively simple unit and only checks to see if the player collides with a spring object. With this in mind, the first base path checks to see if the player has entered a trigger with the spring tag and adds an appropriate force. The second basis path skips this because the player has not made

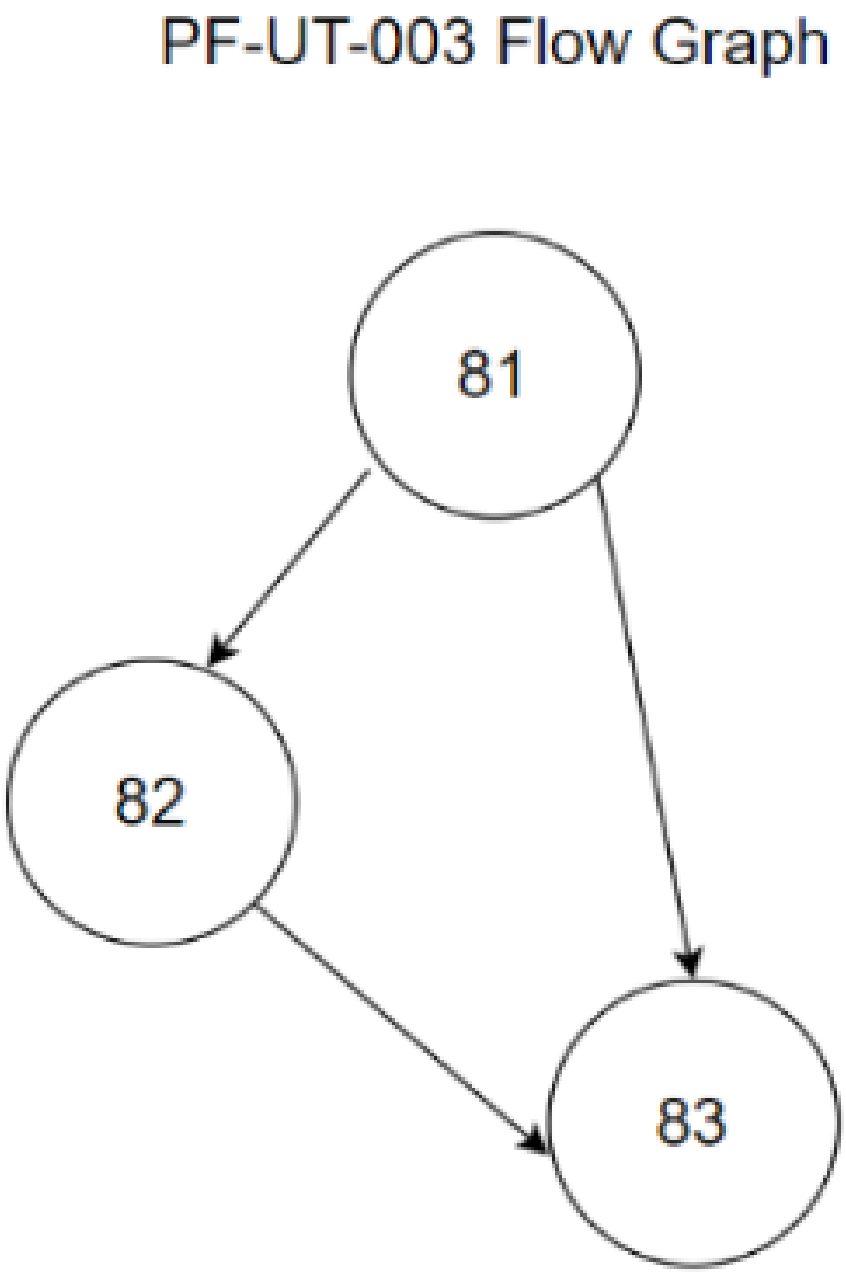

Figure 25: Flow Graph of PF-UT-003

contact with a spring collider.

4.2.2 Unit Tests and Results

Below is a table that shows a visualization of what tests passed and failed from the unit test plans. There is also a graph to show the differences in the cyclomatic complexities of the different Unit Tests.

Test Plan Identifier / Test Type	Pass	Fail
PF-UT- 001		
Death values are updated properly	X	
PF-UT- 002		
Red Hopper Vector is implemented properly	X	
PF-UT- 003		
Vertical Force is Applied to Psycho Fox	X	

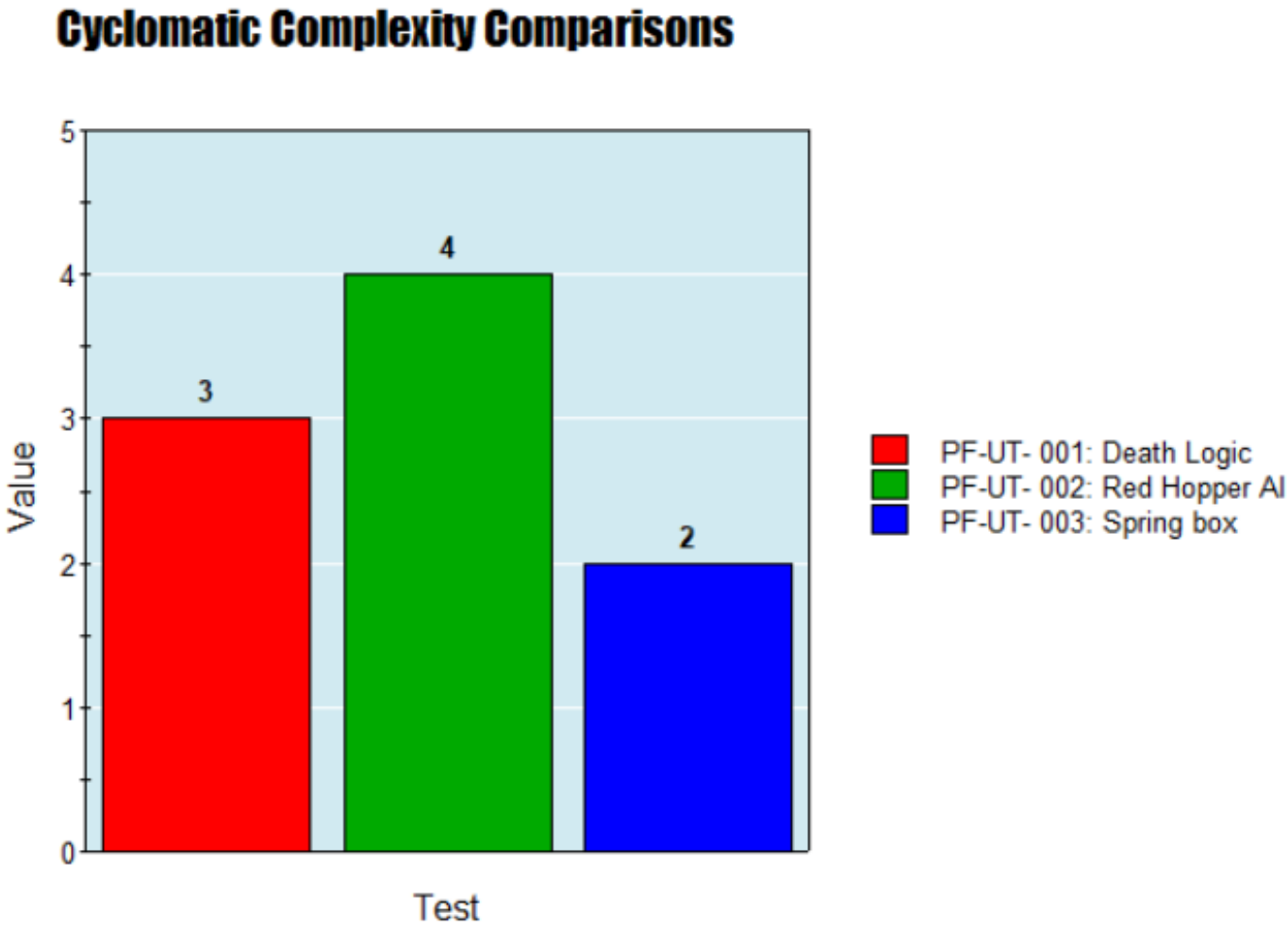

Figure 26: Cyclomatic Complexities of Unit Tests

4.3 Integration Testing

Test Plan Identifier: PF-IT-001: PF-UT- 001: Death Condition & Logic Snippet + PF-UT- 002: Red Hopper AI code snippet

Introduction: This test aims to determine if the individual software modules that were tested in the Unit Test sections work together as intended with other software modules. These 2 modules being tested together are unit tests PF-UT-001 and PF-UT-002.

Test item: The SUT for this unit test would be the RedHopperMovement class and the Death.cs script. The RedHopperMovement determines different forces that are applied to Red Hopper objects and which direction Red Hoppers should move in order to dynamically respond to player inputs. The Death.cs script works by testing for collisions with a certain object.

Features to test/not to test: One of the main features that we'll be testing involves how Red Hopper objects behave and respond to player inputs and how theses movements interact with the Death.cs script. We'll be checking to see if Red Hoppers appear to automatically jump after touching the ground and move slightly towards the player. If this jumping mechanic works properly and moves towards Psycho Fox, then the Death.cs script will trigger. This will be the main and only test.

Approach: We'll be doing dynamic testing inside of the Unity game engine for this test. For this test, Psycho fox will be moved next to the Red Hopper enemies. We will then start the game and test if the RedHopperMovement class is making the Red Hopper move towards Psycho Fox. If this happens, then the Death.cs script will activate due to the "Death" collider tag that the Red Hopper has. If Psycho Fox dies, then this test would be a pass.

Test deliverables: The test deliverables for this section will be testing the functionality between these 2 unit tests. This data will then be made into a chart with this data based on if these unit test work together or not.

Item pass/fail criteria: The Pass/Fail criteria is as follows: Pass: The pass for this test would be if the Red Hopper correctly uses its RedHopperMovement file to move towards the player object. Once the Red Hopper collides with Psycho Fox, the Death.cs script will then complete its logic and run. Fail: If the movement of Red Hopper, and the Death.cs script do not interact properly as described in the pass section, then this would be a fail. More specifically, if the Red Hopper enemies aren't able to interact with Psycho Fox due to incorrect movements.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. This includes an IDE that is capable of compiling Csharp scripts since that's the programming language that our game is written in. For our tests, we used Visual Studio to compile our scripts.

Responsibilities: The tester for this plan needs to have access to the environmental needs before they can begin testing. After that, they need to be able to modify the location of the Psycho Fox object in the Unity game next to a Red Hopper. After this, the tester needs to be able to run the game and test for the pass/fail criteria.

Staffing and training needs: The amount of training required to perform these tests is very low. The tester simply needs to be able to run the game after moving the Psycho Fox game object. The tester then needs to be able to run the game and understand the pass/fail conditions for PF-IT-001. They should then record whether or not Psycho Fox died from interacting with Red Hopper enemies and their movements.

Schedule: This integration test, in addition to all of the other testing, occurred during the "Testing" phase of our Gantt Chart. All of the following tests for this specific test were tested on the same day.

Risks and Mitigation: The main risks for this testing plan revolve around the RedHopperMovement class not properly working so that the Red Hopper does not properly move towards the Psycho Fox object. If this occurs, then there is no way to test these units properly. Another risk involves the collision not registering when the Red Hopper collides with Psycho Fox after the RedHopperMovement class moves the Red Hopper towards the Psycho Fox object. If these issues occur, we would need to go back and fix these issues.

Approvals: As of 11/27/23, our group agreed that this test was ready for execution.

Test Plan Identifier: PF-IT-002: PF-UT- 001: Death Condition & Logic Snippet + PF-UT- 002: Red Hopper AI code snippet + PF-UT- 003: Spring box logic snippet

Introduction: This test is used to determine if the individual software modules that were tested in the Unit Test sections work together as intended with other software modules. These 3 modules being tested together are unit tests PF-UT-001, PF-UT-002, and PF-UT-003.

Test item: The SUT for this is the implementation of colliders, specifically whether the collider set for the Red Hoppers, the collider set for the spring boxes, and the collider for Psycho Fox work correctly and do not conflict with each other.

Features to test/not to test: The in-scope features of this test check whether the script for Red Hoppers correctly provide them with colliders that result in the death of Psycho Fox if the fox comes into contact with it and the script for the the spring boxes correctly has colliders set that result in the boosting of Psycho Fox in the air after coming into contact with the top of the boxes. The respective code files for this implementation are RedHopperMovement.cs, Death.cs, and foxMovement.cs. Everything else is of lower priority compared to the purpose of this test, making it out of the scope.

Approach: Dynamic, functional, and automated testing will be applied in our Unity game for this test. For this test, Psycho Fox will come into contact with both the Red Hoppers and the spring boxes on separate occasions to ensure that the colliders work as expected. When Psycho Fox comes into contact with the Red Hoppers, the

colliders of both should result in the death of Psycho Fox. When Psycho Fox comes into contact with the spring boxes, the colliders of both should result in Psycho Fox being boosted into the air.

Test deliverables: This test will be used to see the pass or fail status of whether the colliders/interactions between the Red Hoppers and Psycho Fox and the spring boxes and Psycho Fox work as intended. This data will be gathered and displayed in a chart that will be located in the Integration Tests and Results section.

Item pass/fail criteria: Pass criteria: First, Psycho Fox should die when it comes into contact with the Red Hoppers. Second, Psycho Fox should be boosted into the air when it comes into contact with the top of the spring boxes. Fail criteria: First, Psycho Fox does not die after coming into contact with the Red Hoppers or the fox gets boosted into the air after coming into contact with the Red Hoppers. Second, Psycho Fox does not get boosted into the air after coming into contact with the top of the spring boxes or the fox dies after coming into contact with the top of the spring boxes.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. This includes an IDE that is capable of compiling C# scripts since that's the programming language that our game is written in. For our tests, we used Visual Studio to compile our scripts.

Responsibilities: The tester must understand how the interactions between Psycho Fox and the Red Hoppers and the interactions between Psycho Fox and the spring boxes should work in order to correctly perform tests.

Staffing and training needs: A small level of training is required to ensure that the tester is able to maneuver Psycho Fox through the map and obstacles to get to the point where the Red Hoppers and spring boxes are present on the map. They should also be able to interact with the Red Hoppers and the spring boxes correctly.

Schedule: This integration test, in addition to all of the other testing, occurred during the "Testing" phase of our Gantt Chart. All of the following tests for this specific test were tested on the same day.

Risks and Mitigation: The main risks during this test are that the interaction/collider contact between Psycho Fox and the Red Hoppers does not result in Psycho fox being boosted into the air and the interaction/-collider contact between Psycho Fox and the spring boxes does not result Psycho Fox dying. In this case, it can be assumed that either the Red Hopper colliders are not working, the spring box colliders are not working, the Psycho Fox collider is not working, or any combination of the respective colliders are not working. A mitigation plan that can be utilized to resolve this error involves reviewing the collider implementation of each object, testing them individually to ensure that they are working as expected, and testing them together once changes are made. If the game shows the intended results of the interactions between Psycho Fox, the Red Hoppers, and the spring boxes, it can be assumed that the colliders are working correctly and the test passes.

Approvals: As of 11/27/23, our group agreed that this test was ready for execution.

4.3.1 Integration Tests and Results

Below is a table that shows a visualization of what tests passed and failed from the integration test plans.

Test Plan Identifier / Test Type	Pass	Fail
PF-IT- 001		
Death and Red Hopper AI Interaction	X	
PF-IT- 002		
Psycho Fox Death with Red Hopper	X	
Psycho Fox is Boosted due to Spring	X	

4.4 System Testing

Test Plan Identifier: PF-ST-001: Basic Test

Introduction: This test is used to determine whether the system can boot up its software image from the supported boot options and if the system software can be upgraded or downgraded in a graceful manner.

Test item: The SUT for this is the system as a whole to ensure that it meets the requirements mentioned above (in the Introduction section).

Features to test/not to test: The in-scope features that will be tested are the game system's ability to boot up its software image correctly and upgrade or downgrade its software. Other aspects of the game are out of the scope for this test.

Approach: A dynamic, manual, performance-based test will be used for this. To fully perform this test, two steps will be done. First, the game will be opened/booted up to ensure that everything loads as expected and is ready to be played immediately after it launches. Next, different objects/scenes in the game will be modified/added/removed and played to ensure that the game is able to handle these changes.

Test deliverables: This test will be used to see the pass or fail status of the two aspects of basic testing (mentioned above). This data will be gathered and displayed in a chart that will be located in the System Tests and Results section.

Item pass/fail criteria: Pass criteria: The system is able to boot its software image (the full game loads when it is opened) and the system software can be upgraded/downgraded in a smooth manner (changes can be made to the game and it should still run correctly without many hiccups). Fail criteria: The system is not able to boot its software image (the game is not able to load or open properly), the system software cannot be upgraded/downgraded in a smooth manner (the game either cannot be modified at all or takes a while for changes to be reflected), or both.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. This includes an IDE that is capable of compiling C# scripts since that's the programming language that our game is written in. For our tests, we used Visual Studio to compile our scripts.

Responsibilities: The tester must open the Unity files and edit using the Unity UI in order to test this correctly.

Staffing and training needs: Not much training is needed to be able to launch the Unity files, but some training is required to be able to make various modifications to the game using the Unity UI.

Schedule: This system test, in addition to all of the other testing, occurred during the "Testing" phase of our Gantt Chart. All of the following tests for this specific test were tested on the same day.

Risks and Mitigation: The main risks during this test are that the system is not able to boot its software image (the game is not able to load or open properly) and the system software cannot be upgraded/downgraded in a smooth manner (the game either cannot be modified at all or takes a while for changes to be reflected). In this case, it can be assumed that either the game is corrupted or the system does not have the sufficient resources to open or modify the game. A mitigation plan that can be utilized to resolve this error is to examine what could be causing the error and fix it, simplify the game, or open an older version of it that works (if that access was given). If the game boots up correctly and is able to be modified, it can be assumed that the system and system software are working as expected.

Approvals: As of 11/27/23, our group was ready to start executing this test.

Test Plan Identifier: PF-ST-002: Smoke Test

Introduction: This test is used to determine whether there are any major overall issues within the game that need to be resolved before the final push (submission in our case).

Test item: The SUT for this is the game as a whole to see whether it works and has worked during different stages of the development process.

Features to test/not to test: Since this test focuses on the way the game functions as a whole, everything about the game is within the scope of the test.

Approach: A dynamic, automated, performance-based test will be used for this. To perform this test, all of the functions, features, interactions, transitions, scenes, controls, visuals/audio, and behaviors must be carefully analyzed to ensure that everything is working as expected.

Test deliverables: This test will be used to see the pass or fail status of whether the game is functioning correctly as a whole. This data will be gathered and displayed in a chart that will be located in the System Tests and Results section.

Item pass/fail criteria: Pass criteria: The overall functionality of the game, including the features, interactions, transitions, scenes, controls, visuals/audio, and behaviors, work as expected in its current form and during the development phase when new additions were made. Fail criteria: The game is not functioning as expected and changes have to be made to ensure that it works based on initial requirements.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. This includes an IDE that is capable of compiling C# scripts since that's the programming language that our game is written in. For our tests, we used Visual Studio to compile our scripts.

Responsibilities: The tester must understand the expected functionality of our game and must be able to test it in its current form to adequately check if the initial requirements have been met.

Staffing and training needs: A little bit of training is required to be able to interact with the Unity UI and test the current implementation of various features in our game.

Schedule: This system test, in addition to all of the other testing, occurred during the "Testing" phase of our Gantt Chart. All of the following tests for this specific test were tested on the same day.

Risks and Mitigation: The main risk during this test is that the game as a whole is not functioning correctly or as expected. In this case, it can be assumed that there are errors within the features, interactions, transitions, scenes, controls, visuals/audio, and behaviors that are in the game. A mitigation plan that can be utilized to resolve these errors is to go back and review the functionality of incorrectly working aspects of the game, testing them individually, and test them together once changes have been made. If these aspects of the game are working correctly, it can be assumed that the game as a whole is functioning as expected.

Approvals: As of 11/27/23, our group was ready to start executing this test.

4.4.1 System Tests and Results

Below is a table that shows results for the above system tests.

Test Plan Identifier / Test Type	Pass	Fail
PF-ST- 001		
Basic Test	X	
PF-ST- 002		
Smoke Test	X	

4.5 Acceptance Testing

Test Plan Identifier: PF-AT- 001: Psycho Fox Acceptance Testing

Introduction: This test include looking over the functional and non-functional requirements and determining if they were implemented, and if they work within the game.

Test item: This test tests the game and all the requirements within the game.

Features to test/not to test: For this test, we test all of the aspects of the game defined within the requirements of the documentation.

Approach: For each requirement, we ran a test case within the game to determine if the requirement was implemented, and whether or not it was implemented correctly.

Test deliverables: We have gathered the data using an Excel spreadsheet, which will be included within this documentation.

Item pass/fail criteria: The Pass/Fail criteria for this particular test depended on the requirement that is currently being tested. Each requirement will be tested with different pass/fail criteria.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. A current working version of the game is needed.

Responsibilities: The responsibilities for this testing requires us to pay attention to the requirement being tested and how we approach said requirement within the game.

Staffing and training needs: The only training needed would be how to correctly operate the game, which is very simple.

Schedule: This unit test, along with all other testing occurred during the "Testing" phase of our Gantt Chart. By the end of the testing phase, we plan to have an excel sheet generated that describes the outcome of our tests.

Risks and Mitigation: There are a few risks that may appear when testing for this. The only issues that would arise during this test would be a failure of a certain requirement.

Approvals: As of 11/30/23, our group was ready to begin testing this. We tested all and confirmed the results with the team on the same day.

Test Plan Identifier: PF-VVT-001: Psycho Fox Verification & Validation Testing

Introduction: This test will be conducted over our game based on the Verification and Validation test methods. This includes testing if we are implementing the system right, i.e. the game systems. This also includes testing if we are implementing the right system.

Test item: The test item would be the whole game, and all the systems within.

Features to test/not to test: We will be testing all of our implemented requirements and systems inside of our project.

Approach: For the approach to this testing type, we will be using the outlined requirements for V&V and testing the game systems based on those.

Test deliverables: We will deliver the test results and data gathered in a table.

Item pass/fail criteria: Based on the outlined requirements we will determine if the currently tested object or system pass or fails.

Environmental needs: Since this game is developed using the Unity framework, a current version of Unity is required to perform these tests. A current working version of the game is needed.

Responsibilities: The responsibilities for this testing requires us to pay attention to the requirement being tested and how we approach said requirement within the game.

Staffing and training needs: The only training needed would be how to correctly operate the game, which is very simple.

Schedule: This unit test, along with all other testing occurred during the "Testing" phase of our Gantt Chart. By the end of the testing phase, we plan to have an excel sheet generated that describes the outcome of our tests.

Risks and Mitigation: There are a few risks that may appear when testing for this. The only issues that would arise during this test would be a failure of a certain requirement.

Approvals: As of 11/30/23, our group was ready to begin testing this. We tested all and confirmed the results with the team on the same day.

4.5.1 Acceptance Tests and Results

The below figures represent the Acceptance, Validation, and Verification testing for each test plan above.

5 Conclusion

This document closely covers all of the steps we made to recreate Psycho Fox. This document is broken down into 4 main sections which cover the 4 main parts of the software engineering process: Feasibility Study, Software Modeling/Design, Software Implementation, and Testing/Bug Fixing. This document also helped our team better work towards breaking down the functionality of the project so that the recreation of the Psycho Fox would be easier when we get to the later steps of the software engineering process.

The final version of our project was not able to meet all acceptability testing that was initially set out by our team and our client in the functional and non-functional requirements section. However, we were able to meet 17 out of the 20 acceptability tests. Because we were able to meet 17/20 acceptability tests, we feel as if we made a relatively good project during the time allotted for this project without current skills. We also made a lengthy document that tracked the whole development process of the Psycho Fox game. This was more important to us because it allowed us to learn the software engineering steps.

In the future or if we were allotted more time, we would have liked to improve the game's functionality and be able to pass all tests in the acceptability section. More specifically, we would like to fully implement the database functionality, a scoreboard that directly interacts with user accounts, and more levels. There is also the ability to utilize items inside of the main version which we did not get to. There were also a few things outside of our scope that we could implement. An example of this could be the ability for the user to change characters which all have differences in how the main character works functionally.

Item #	Acceptance Criteria or Requirement	Test Type	Priority	Test Result	Test Date	Comments
1	Functional: The project should allow the main character to be controlled by the keyboard or gamepad.	Functional	HIGH	Pass	11/26/23	
2	Functional: The project should allow multiple players to create separate accounts.	Functional	MEDIUM	Pass	11/26/23	
3	Functional: The project should implement a login system for users and administrators.	Functional	MEDIUM	Pass	11/26/23	
4	Functional: The project should save the state/progress of all players and allow players to continue once they've logged into their account.	Functional	MEDIUM	Fail	11/26/23	We currently have the login/registration system working but need to finish implementing the save states.
5	Functional: The Project should allow players to lose a life by taking damage or hitting a death barrier.	Functional	HIGH	Pass	11/30/23	
6	Functional: Players should be able to obtain and use items within the game.	Functional	MEDIUM	Fail	11/30/23	Due to time constraint, items are not implemented and the player can not obtain or use them.
7	Functional: The enemy objects in the game should have a basic AI and pathfinding to attack the player.	Functional	MEDIUM	Pass	11/30/23	
8	Functional: The game should include obsticals that the player must navigate to complete the level.	Functional	HIGH	Pass	11/30/23	
9	Functional: The Game should include music and audio pertaining to the current scene.	Functional	MEDIUM	Pass	11/30/23	
10	Functional: The project should display a splash screen when launched which also shows top scores and the login prompt.	Functional	MEDIUM	Pass	11/26/23	
11	Nonfunctional: The project should be implemented using the Unity game engine and development environment.	Performance	HIGH	Pass	11/26/23	
12	Nonfunctional: The project team should create a complete replica of the game.	Performance	MEDIUM	Pass	11/26/23	
13	Nonfunctional: The project can use pre-developed images, artwork, audio, and other related media.	Performance	MEDIUM	Pass	11/26/23	
14	Nonfunctional: All project source code must be developed by the CS 360 project team.	Performance	HIGH	Pass	11/26/23	
15	Nonfunctional: The project must use a database.	Performance	MEDIUM	Pass	11/26/23	
16	Nonfunctional: Performance metrics should be gathered and optimized.	Performance	MEDIUM	Pass	11/26/23	
17	Nonfunctional: Security metrics should be gathered and optimized.	Performance	MEDIUM	Fail	11/26/23	We ran out of time and plan to implement this in our future works section.
18	Nonfunctional: The game should run at a consistent refresh rate.	Performance	MEDIUM	Pass	11/30/23	
19	Nonfunctional: The game should use the correct amount of resources that it is provided.	Performance	MEDIUM	Pass	11/30/23	
20	Nonfunctional: User interface metrics should be gathered and optimized.	Performance	MEDIUM	Pass	11/26/23	

Figure 27: Acceptance Testing Results

6 Appendix

6.1 Software Product Build Instructions

To copy the current state of our game to new computers for continued development, those computers must first meet the performance requirements. Windows machines should be running Windows 7 (SP1+) or Windows 10/11 in 64-bit versions only, have the X64 architecture CPU with SSE2 instruction set support, have a DX10/DX11/DX12 GPU, and have hardware vendor officially supported drivers. macOS machines should be running High Sierra 10.13+, have the X64 architecture with SSE2 instruction set support, have a metal-capable Intel or AMD GPU, and have Apple officially supported drivers. Linux machines should be running Ubuntu 16.04 or Ubuntu 18.04 or CentOS 7, have the X64 architecture with SSE2 instruction set support, be OpenGL 3.2+/Vulkan-capable or have Nvidia or AMD GPU, and have a Gnome desktop environment running on top of X11 windowing system or Nvidia official proprietary graphics driver or AMD Mesa graphics driver (other configuration and player environment provided by default with the supported distribution). The machines should also have the Unity 2022.3.8f1 LTS version and a code editor installed. Once these requirements are met, the file for our Unity game (including the game and scripts) can be shared to other computers through GitHub for example, and these

Validation Case	Pass/Fail
Are we implementing the right system?	Pass
Evaluating products at the closing of the development process	Pass
Objective is making sure the product is as per the requirements and design specifacations	Pass
Activities included: black box testing, white box testing, and grey box testing	Pass
Validates that the uesrs accept the software or not	Pass
Items evaluated: actual product or software under test	Pass
Checking the developed products using the documents and files	Pass

Figure 28: Validation Testing Results

Verification Case	Pass/Fail
Are we implementing the system	Pass
Evaluate product of development phase	Pass
Objective is making sure the product is as per the requirements and design specifacations	Pass
Activities included: reviews, meetings and inspections	Pass
Verifies the outputs are according to inputs or not	Pass
Items evaluated; plans, requirements specifacations, design specs, code and test cases	Pass
Manual checking of the documents and files	Pass

Figure 29: Verification Testing Results

computers can download the file locally. Once the file is downloaded, the computers can open it on Unity Hub and have full access. From here, the game can be played and modifications can be made.

6.2 Software Product User Guide

We do not have an administrative user for our game, so this section will instead focus on how a general user and our client will use our game. As mentioned in the previous section, the user and client will have to ensure that they have equipment that meets the requirements to run the game. Also mentioned in the previous section is how the user and client will get access to the game (the game file will be shared with them, for example through a repository like GitHub). Once they have the file downloaded locally on their machine, they can open it through Unity Hub. The key difference is that the file that the general user will receive is a build version that will allow them to play the game but not make any modifications. The client on the other hand will receive a file that grants them full access to the game (allowing them to make changes to it) in addition to a build version. Creating a build for the game will allow a general user to open the file directly and begin playing.

6.3 Source Code with Comments

UML Diagram Mapped

High Score System - Observer Class Diagram: C.23 UserData.cs
Player Manager - Singleton Class Diagram: C.20 PlayerManager.cs
Enemy Manager - Singleton Class Diagram: C.4 EnemyManager.cs
Game(Everything) - Structural Diagram: C.19 PanelManager.cs
Psycho Fox Main Menu - Use Case Diagram: C.19 PanelManager.cs
Psycho Fox Birdfly Acquisition: Not Implemented
Psycho Fox Player-Enemy Interaction Diagram: C.3, C.4, & C.20
State Diagram for Player Movement: C.16 foxMovement.cs
State Diagram for Psycho Fox Life Status - Falling: C.20 PlayerManager.cs
Enemy Interaction - Component Diagram: C.4 EnemyManager.cs
Player's System - Deployment Diagram: All Source Code Below

C.1 DestroyP.cs

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class DestroyP : MonoBehaviour
6  {
7      public GameObject bro;
8
9      public void Destroy() {
10         Destroy(bro);
11     }
12 }
```

C.2 Enemy.cs

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using System.Diagnostics;
5
6  public class Enemy : MonoBehaviour
7  {
8      private float speed = 0.05f;
9
10     private void OnEnable()  {
11
12         Stopwatch stopwatch = new Stopwatch();
13         stopwatch.Start();
14         EnemyManager.Instance.addEnemy(this);
15         stopwatch.Stop();
16         UnityEngine.Debug.Log("addEnemy took " + stopwatch.
               ElapsedMilliseconds + "ms to complete");
17     }
18
19     private void OnDisable() {
20         Stopwatch stopwatch = new Stopwatch();
21         stopwatch.Start();
22         EnemyManager.Instance.unaddEnemy(this);
23         stopwatch.Stop();
24         UnityEngine.Debug.Log("unaddEnemy took " + stopwatch.
               ElapsedMilliseconds + "ms to complete");
```

```
    }

    public void move(Vector3 direction) {
        transform.position += direction * speed * Time.deltaTime;
    }


}
```

C.3 EnemyKill.cs

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

public class EnemyKill : MonoBehaviour
{

    void OnTriggerEnter2D(Collider2D col) {
        if (col.gameObject.CompareTag("Enemy")) {
            Destroy(col.gameObject);
        }
    }
}
```

C.4 EnemyManager.cs

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

public class EnemyManager : MonoBehaviour
{
    public static EnemyManager Instance { get; private set; }

    public GameObject enemyOne;
    public GameObject enemyTwo;

    public GameObject fox;
    private CircleCollider2D colliderObj;

    private List<Enemy> allEnemies = new List<Enemy>();



    private void Awake() {
        if (Instance == null) {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        } else {
            Destroy(gameObject);
        }
    }

    private void Start() {

        if (fox != null)
        {
```

```
            colliderObj = fox.GetComponent<CircleCollider2D>();
        }
    }

    void Update() {

    }

    public void addEnemy(Enemy enemy) {
        if (!allEnemies.Contains(enemy))
            allEnemies.Add(enemy);
    }

    public void unaddEnemy(Enemy enemy) {
        if (allEnemies.Contains(enemy))
            allEnemies.Remove(enemy);
    }

    public void MoveEnemy(GameObject enemyType) {

    }

    public void triggerenter(Collider2D other)
    {
        if (other.tag != "Ground") {
            print("You just took damage");
        }
    }


    public void triggerstay(Collider2D other)
    {
        if (other.tag != "Ground") {

        }
    }


    public void triggerexit(Collider2D other)
    {
        if (other.tag != "Ground") {

        }
    }
}
```

C.5 PocklyMovement.cs

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

public class PocklyMovement : MonoBehaviour
{

    void Update()
    {
        if (PlayerManager.Instance.alive == true) {
            transform.Translate (-0.005f, 0f, 0f);
        }
```

```
13
14      }
15  }
```

C.6 RedHopperMovement.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class RedHopperMovement : MonoBehaviour
6   {
7       public Rigidbody2D rb;
8       public float jumpForce;
9       private bool isGrounded;
10      public Transform groundCheck;
11      public LayerMask groundLayer;
12      [SerializeField] Vector2 boxsize;
13      public float relPos;
14      public GameObject bro;
15
16      void FixedUpdate()
17      {
18          if (PlayerManager.Instance.alive && (PlayerManager.Instance.xlocation
                - transform.position.x > -2f)) {
19              isGrounded = Physics2D.OverlapBox(groundCheck.position, boxsize,
                    0, groundLayer);
20              if (isGrounded) {
21                  Jump();
22              }
23          } else if (!PlayerManager.Instance.alive){
24              Destroy(bro);
25          }
26
27
28      }
29
30      public void Jump() {
31
32              if (PlayerManager.Instance.xlocation < transform.position.x) {
33                  relPos = -0.0008f;
34              } else {
35                  relPos = 0.0008f;
36              }
37
38                  rb.AddForce(new Vector2(relPos, jumpForce), ForceMode2D.
                        Impulse);
39      }
40
41  }
```

C.7 Login.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4   using TMPro;
5
6   public class Login : MonoBehaviour
```

```
{
    public TMP_InputField usernameInput;
    private string username;
    public TMP_InputField passwordInput;
    private string password;
    public GameObject panel;
    private UserInfo ud;

    private void Start() {
        ud = panel.GetComponent<UserInfo>();
    }

    public void loginUser() {

        username = usernameInput.text;
        password = passwordInput.text;
        print("from input field:    " + username + ":" + password);

        foreach (var item in ud.userdata) {
            print("from database:   " + item.Key + ":" + item.Value);
            if (username == item.Key && password == item.Value) {
                print("Successfully logged in " + item.Key);
                return;
            }
        }
        print("User does not exist, you need to register first");
    }

}
```

C.8 UserData.cs

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

public class UserInfo : MonoBehaviour
{
    public IDictionary<string, string> userdata = new Dictionary<string,
        string>();
    public TMP_InputField usernameInput;
    private string username;
    public TMP_InputField passwordInput;
    private string password;


    public void registerUser() {
        username = usernameInput.text;
        password = passwordInput.text;
        userdata[username] = password;
        print("Successfully registerd " + username);
    }

    public void UpdateHighScore() {

    }
}
```

C.9 BackToMain.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Diagnostics;


public class BackToMain : MonoBehaviour
{
    public void BackToMainScreen()
    {
        Stopwatch st = new Stopwatch();
        st.Start();
        SceneManager.LoadScene("Boot Screen");
        st.Stop();
        UnityEngine.Debug.Log(string.Format ("BackToMain took {0} ms to
            complete", st.ElapsedMilliseconds));
    }
}
```

C.10 camera_script.cs

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

public class camera_script : MonoBehaviour
{
    public Transform player;
    public float cameraSpeed = 15.0f;
    public float maxCameraX;
    public float minCameraX;
    public float maxCameraY;
    public float minCameraY;
    void FixedUpdate() {

        if (player.position.x > transform.position.x) {
            Vector3 targetPosition = new Vector3(player.position.x, transform
                .position.y, transform.position.z);
            targetPosition.x = Mathf.Clamp(targetPosition.x, minCameraX,
                maxCameraX);
            transform.position = Vector3.Lerp(transform.position,
                targetPosition, cameraSpeed * Time.deltaTime);
        }

        if (player.position.y != transform.position.y) {
            Vector3 targetPosition = new Vector3(transform.position.x, player
                .position.y, transform.position.z);
            targetPosition.y = Mathf.Clamp(targetPosition.y, minCameraY,
                maxCameraY);
            transform.position = Vector3.Lerp(transform.position,
                targetPosition, cameraSpeed * Time.deltaTime);
        }
    }
}
```

C.11 DatabaseManager.cs

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

```
using UnityEngine;

public class DatabaseManager : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

C.12 Death.cs

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

public class Death : MonoBehaviour
{
    public  GameObject background;
    public  GameObject deathmusic;
    public Animator ghostanimator;
    public Animator actualanimator;
    public GameObject hitbox;

    private void OnTriggerEnter2D(Collider2D col) {
        if (col.tag == "Death") {
            if (hitbox.GetComponent<BoxCollider2D>().enabled == false) {
                background.GetComponent<AudioSource>().Stop();
                deathmusic.GetComponent<AudioSource>().Play();
                ghostanimator.SetTrigger("Death");
                actualanimator.SetTrigger("Death");
                PlayerManager.Instance.alive = false;
                gameObject.GetComponent<SpriteRenderer>().enabled = false;
                print("You just died");
            }

        }
    }
}
```

C.13 DisplayLives.cs

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

public class DisplayLives : MonoBehaviour {

    public GameObject text;
```

```
    private int lives = PlayerManager.Instance.lives;

    void Start() {
        PlayerManager.Instance.alive = true;
        PlayerManager.Instance.lives -= 1;
        // text.GetComponent<TextMeshProUGUI>().text = "X" + (lives);
        text.GetComponent<TextMeshProUGUI>().text = "X" + (lives - 1);
    }

    public void LoadGameScene() {
        // if (lives == 0)
        if (lives == 1) {
            SceneManager.LoadScene("Boot Screen");
        } else {
            SceneManager.LoadScene("Main Scene");
        }
    }
}
```

C.14 FinishGame.cs

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




public class FinishGame : MonoBehaviour
{
    [SerializeField]
    private string levelcompletedscene = "Level Completed Screen";
    [SerializeField]
    private string bootscene = "Boot Screen";

    void OnTriggerEnter2D(Collider2D col) {
        if (col.gameObject.name == "RoundEndSign") {
            SceneManager.LoadScene(levelcompletedscene);

        }

    }



}
```

C.15 foxController.cs

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

public class foxController : MonoBehaviour
{
    void Start()
    {
```

```
10
11      }
12
13      void Update()
14      {
15
16      }
17
18      //collider functions
19      private void OnTriggerEnter2D(Collider2D other)
20      {
21          Stopwatch stopwatch = new Stopwatch();
22          stopwatch.Start();
23          EnemyManager.Instance.triggerenter(other);
24          stopwatch.Stop();
25          UnityEngine.Debug.Log("TriggerEnter took " + stopwatch.
                ElapsedMilliseconds + "ms to complete");
26      }
27
28
29      private void OnTriggerStay2D(Collider2D other)
30      {
31          EnemyManager.Instance.triggerstay(other);
32      }
33
34
35      private void OnTriggerExit2D(Collider2D other)
36      {
37          Stopwatch stopwatch = new Stopwatch();
38          stopwatch.Start();
39          EnemyManager.Instance.triggerexit(other);
40          stopwatch.Stop();
41          UnityEngine.Debug.Log("TriggerExit took " + stopwatch.
                ElapsedMilliseconds + "ms to complete");
42      }
43  }
```

C.16 foxMovement.cs

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class PlayerMovement : MonoBehaviour
6   {
7       public float acceleration = 5.0f; // Acceleration rate
8       public float deceleration = 5.0f; // Deceleration rate
9       public float maxSpeed = 10.0f; // Maximum speed
10      public float jumpForce = 10.0f; // Jump force
11      public float springForce = 2.5f; // Spring force
12      public Transform groundCheck; // Transform to determine if the player is
            grounded
13      public float groundCheckRadius = 0.5f; // Radius for ground check
14      public LayerMask groundLayer; // LayerMask to determine what is ground
15      public Animator animator;
16      private Vector2 currentVelocity; // Current velocity of the object
17      private bool isGrounded; // Is the player on the ground?
18      private Rigidbody2D rb; // Rigidbody2D component of the object
19      public AudioSource jumpAudioSource;
20      public AudioClip jumpSound;
```

```
21
22      void Start()
23      {
24          rb = GetComponent<Rigidbody2D>();
25          if (jumpAudioSource == null)
26          {
27              jumpAudioSource = GetComponent<AudioSource>();
28          }
29      }
30
31      void Update()
32      {
33          if (PlayerManager.Instance.alive == true) {
34              // Check if the player is grounded
35              isGrounded = Physics2D.OverlapCircle(groundCheck.position,
                    groundCheckRadius, groundLayer);
36              animator.SetBool("isJump", !isGrounded);
37
38              // Get input from horizontal and vertical axes
39              float inputX = Input.GetAxis("Horizontal");
40              if (inputX > 0f) {
41                  GetComponent<SpriteRenderer>().flipX = false;
42              } else if (inputX < 0f) {
43                  GetComponent<SpriteRenderer>().flipX = true;
44              }
45              Vector2 inputDirection = new Vector2(inputX, 0).normalized;
46
47              // Apply acceleration or deceleration
48              if (inputDirection != Vector2.zero)
49              {
50
51                  // Accelerate towards the input direction
52                  currentVelocity += inputDirection * acceleration * Time.
                        deltaTime;
53                  animator.SetFloat("Speed", Mathf.Abs(inputX));
54              }
55              else
56              {
57                  // Apply deceleration when no input is given
58                  if (currentVelocity.magnitude > 0)
59                  {
60                      currentVelocity -= currentVelocity.normalized *
                            deceleration * Time.deltaTime;
61                      if (currentVelocity.magnitude < 0.1f) // To stop
                            completely if the speed is very low
62                      {
63                          currentVelocity = Vector2.zero;
64                      }
65                  }
66              }
67
68              // Clamp the velocity to the maximum speed
69              currentVelocity = Vector2.ClampMagnitude(currentVelocity,
                    maxSpeed);
70
71              // Apply horizontal movement
72              rb.velocity = new Vector2(currentVelocity.x, rb.velocity.y);
73              //update the player's location
74              PlayerManager.Instance.xlocation = transform.position.x;
75
```

```
            // Jumping
            if (Input.GetButtonDown("Jump") && isGrounded)
            {
                PlayJumpSound();
                rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
            }
        } else {
            rb.velocity = new Vector2(0, 0);
        }
    }

    void PlayJumpSound()
    {
        // Check if there is an audio clip assigned
        if (jumpSound != null && jumpAudioSource != null)
        {
            // Play the jump sound
            jumpAudioSource.PlayOneShot(jumpSound);
        }
    }
    private void OnCollisionEnter2D(Collision2D col) {
        if (col.gameObject.CompareTag("Spring")) {
            rb.AddForce(Vector2.up * springForce, ForceMode2D.Impulse);
        }
    }
}
```

C.17 GhostDeath.cs

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

public class GhostDeath : MonoBehaviour
{
    public void LoadMainMenu() {
        SceneManager.LoadScene("Lives");
    }
}
```

C.18 MenuTimer.cs

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

public class MenuTimer : MonoBehaviour
{
    private float menuTime = 0f;

    private void OnEnable() {
        StartMenuTimer();
    }

    private void OnDisable() {
        StopMenuTimer();
    }

```

```
    private void StartMenuTimer() {
        menuTime = 0f;
    }

    private void StopMenuTimer() {
        LogMenuTime();
    }

    void Update() {
        menuTime += Time.deltaTime;
    }

    private void LogMenuTime() {
        Debug.Log("User spent " + menuTime + " seconds in the menu.");
    }
}
```

C.19 PanelManager.cs

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

public class CanvasController : MonoBehaviour
{
    public GameObject loginPanel;
    public GameObject registrationPanel;
    public GameObject mainPanel;
    public GameObject scoreboardPanel;
    public GameObject bootPanel;
    public GameObject completionPanel;

    public InputField loginUsernameInput;
    public InputField loginPasswordInput;
    public InputField registerUsernameInput;
    public InputField registerPasswordInput;
    public Text errorMessageText;
    public Text loggedInUsernameText;

    private string loggedInUsername;
    private bool isLoggedIn = false;
    private List<UserData> scoreboardData = new List<UserData>();

    // Start is called before the first frame update
    void Start()
    {
        bootPanelActivate();
    }

    // Function to hide all panels
    void HideAllPanels()
    {
        loginPanel.SetActive(false);
        registrationPanel.SetActive(false);
        mainPanel.SetActive(false);
        scoreboardPanel.SetActive(false);
        bootPanel.SetActive(false);
        completionPanel.SetActive(false);
    }
```

```
42
43      //Panel activation
44
45      //loginPanel
46      public void loginPanelActivate() {
47          loginPanel.SetActive(true);
48          registrationPanel.SetActive(false);
49          mainPanel.SetActive(false);
50          scoreboardPanel.SetActive(false);
51          bootPanel.SetActive(false);
52          completionPanel.SetActive(false);
53      }
54
55      //registrationPanel
56      public void registrationPanelActivate() {
57          loginPanel.SetActive(false);
58          registrationPanel.SetActive(true);
59          mainPanel.SetActive(false);
60          scoreboardPanel.SetActive(false);
61          bootPanel.SetActive(false);
62          completionPanel.SetActive(false);
63      }
64
65      //mainPanel
66      public void mainPanelActivate() {
67          loginPanel.SetActive(false);
68          registrationPanel.SetActive(false);
69          mainPanel.SetActive(true);
70          scoreboardPanel.SetActive(false);
71          bootPanel.SetActive(false);
72          completionPanel.SetActive(false);
73      }
74
75      //scoreboard
76      public void scoreboardPanelActivate() {
77          loginPanel.SetActive(false);
78          registrationPanel.SetActive(false);
79          mainPanel.SetActive(false);
80          scoreboardPanel.SetActive(true);
81          bootPanel.SetActive(false);
82          completionPanel.SetActive(false);
83      }
84
85
86      //bootPanel
87      public void bootPanelActivate() {
88          loginPanel.SetActive(false);
89          registrationPanel.SetActive(false);
90          mainPanel.SetActive(false);
91          scoreboardPanel.SetActive(false);
92          bootPanel.SetActive(true);
93          completionPanel.SetActive(false);
94      }
95
96      //bootPanel
97      public void completionPanelActivate() {
98          loginPanel.SetActive(false);
99          registrationPanel.SetActive(false);
100         mainPanel.SetActive(false);
101         scoreboardPanel.SetActive(false);
```

```
        bootPanel.SetActive(false);
        completionPanel.SetActive(true);
    }
}
```

C.20 PlayerManager.cs

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

public class PlayerManager : MonoBehaviour
{

    public static PlayerManager Instance { get; private set; }
    public bool alive = true;
    public int lives = 2;
    public float xlocation;

    private void Awake() {
        if (Instance == null) {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        } else {
            Destroy(gameObject);
        }
    }

    private float menuTime = 0f;

    private void OnEnable() {
        StartMenuTimer();
    }

    private void OnDisable() {
        StopMenuTimer();
    }

    private void StartMenuTimer() {
        menuTime = 0f;
    }

    private void StopMenuTimer() {
        LogMenuTime();
    }

    void Update() {
        menuTime += Time.deltaTime;
    }

    private void LogMenuTime() {
        Debug.Log("User spent " + menuTime + " seconds in the game.");
    }


}
```

C.21 PlayerPunch.cs

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

public class playerPunch : MonoBehaviour
{
    public Animator animator;
    public GameObject hitbox;

    void Start() {
        hitbox.GetComponent<BoxCollider2D>().enabled = false;
    }

    void Update()
    {
        if (Input.GetMouseButtonDown(0) | Input.GetKeyDown("x")) {
            animator.SetTrigger("Punch");
        }
    }

    public void EnableCol() {
        hitbox.GetComponent<BoxCollider2D>().enabled = true;
    }

    public void DisableCol() {
        hitbox.GetComponent<BoxCollider2D>().enabled = false;
    }
}
```

C.22 StartGameButton.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;
using System.Diagnostics;

public class StartGameButton : MonoBehaviour
{
    public void StartGame()
    {
        PlayerManager.Instance.alive = true;
        Stopwatch st = new Stopwatch();
        st.Start();
        SceneManager.LoadScene("Main Scene");
        st.Stop();
        UnityEngine.Debug.Log(string.Format ("StartGame took {0} ms to
            complete", st.ElapsedMilliseconds));
    }
}
```

C.23 UserData.cs

```
using SQLite4Unity3d;

public class UserData
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    public string Username { get; set; }
    public string Password { get; set; }
```

```
9       public int Score { get; set; }
10
11      public override string ToString()
12      {
13          return string.Format("[UserData: Id={0}, Username={1}, Password={2},
                Score={3}]", Id, Username, Password, Score);
14      }
15  }
```