Star Wars

Sprint 3
November 3rd, 2023

Name	**Email Address**
Matthew Irizarry	matthew.irizarry745@topper.wku.edu
Zach Vance	zachary.vance141@topper.wku.edu
Keimon Bush	keimon.bush105@topper.wku.edu
Jeremiah Harris	jeremiah.harris978@topper.wku.edu

CS 360
Fall 2023
Project Technical Documentation
Galloway Games, Inc.

Contents

List of Figures

1 Introduction

1.1 Project Overview

This project will attempt to produce a recreation of the 1991 Star Wars game that was produced for the SEGA Master System. While we have not recreated every level and aspect of the game, the aim is to create a fun, quality homage to the original. This means that at the end of delivery, the game should have minimal bugs, be performant on the targeted systems, and have gameplay that is smooth and fun. Moreover, the gameplay should be true and as close to the original as possible. The target audience is someone who already has some gaming experience, but the gameplay should be friendly and welcoming to a beginner.

1.2 Project Scope

This section will discuss the scope of the project. It is important to our client that our game replicate the original SEGA Master System Star Wars game very closely. In order to accomplish this goal, we must first understand what the scope of our project is. Our client does not need every single level, but needs one level that is high in quality. This means that we will likely iterate on scope items several times before their final delivery.

1.2.1 Title and Splash Screen

It is important to the client that a player of this recreation be able to recognize the branding associated with Lucasfilms and the Star Wars Franchise. This is the player's first impression of the game, so it is important that we get this part right. Much like in the movies, this includes a sliding wall of text that introduces the player to the story.

1.2.2 Textures and Art

In consideration of the original game, we have used the same art style. The art style used by the original game was a result of the hardware limitations at the time, which was an 8-bit computer. Therefore, we have used an 8 bit art style. We have either downloaded textures from open-source locations and utilize them, or we have needed to create our own. In consideration of the varying amount of textures, this will take up a significant amount of time for development and proper integration.

1.2.3 Player Input

It is important that the player has a smooth, and responsive playing experience. In order to accomplish this, we must effectively implement player controls for the player sprite. This includes functionality such as: jumping, crouching, and moving left to right.

1.2.4 Projectiles and Collision

There are a few distinct weapons in the game, but lots of enemies use projectiles. For example, the Jawa enemy, and storm troopers.

1.2.5 Enemies

There are several enemies in the game, with a few different abilities. This should be relatively easy to implement, and we can take advantage of the C# object oriented programming paradigm to share similar traits amongst enemies and players, such as health, XY location, and more.

1.2.6 Level Design

There are many unique components of this level design. Yes, it is a basic 2D platformer style, however there are some unique components of the environment such as jump boosting. These components allow the player to have a more interactive playing experience, and elevates the environment beyond just the basic 2D movement. For example, there is a conveyor belt component that horizontally speeds the player up. Moreover, there are up

and down platforms that players must jump on at the proper time in order to progress through the level. If the player gets caught beneath the platform, they should take damage. Another thing the environment offers are little health boosts that are placed throughout the map.

1.2.7 Sounds

The sounds in this game are synonymous with an 8 bit game, and as such we were able to find the original sounds themselves to implement, or we were easily able to reproduce them ourselves. This is much like the textures, and is a part of the detailing phase of this project.

1.3 Technical Requirements

1.3.1 Functional Requirements

Mandatory Functional Requirements
Character input with keyboard or mouse
Support for multiple players (account creation)
Login system for users and administrators
A save game system
Splash screen

Functional requirements are an important component of proper project planning and coordination. They allow for the client and development team's expectations to meet so there are not any miscommunications regarding expectations. Proper expression of product requirements, including functional requirements allows for the development team to produce the product to the specifications of the client. The mandatory functional requirements that the client has expressed to the development team includes: controlling the main character via keyboard or gamepad, account creation for multiple players, a login system for users and administrators, a save game system, and a splash screen.

For the scope of this project, the development team will only be implementing keyboard controls. Using the keyboard, a player will be able to jump, walk left and right, and fire projectiles at enemies. Each execution of a control will interact with the environment, play a sound clip, execute an animation, and/or move the character. The team will source as many of these assets (sounds clips, animations, sprites) online and will not be creating them in house.

A login system has been requested to be added to the recreation of the base game, which will allow both users and admins to login. With a username and login, players and admins alike will be able to login to the system. Admins will be able to access and edit account info for players.

In addition to the login system, the client has requested that the state of the game be saved for all players and allow players to continue once they've logged back into their accounts. The player will be able to simply pause the game and logout and the game will automatically be saved. When logging back in users will either be able to continue from their previous game or be able to start a new game. Both the login credentials and the save states of the game will be stored in a database.

The splash screen will display an appropriately themed backdrop image when the user launches the game. The splash screen will also function as a highscore display and a login screen.

For sprint two, we did end up adding a few functional requirements to the project.

New Functional Requirements
Entity Collision
Level design
Power ups/health orbs
Audio and Visuals
Scoring and Progression
Restart and Respawn handling
Game Over / Level complete conditions

1.3.2 Non-Functional Requirements

Mandatory Non-Functional Requirements
Project reliability
Smooth and responsive player experience
Quality UI/UX, meaningful controls
Accessible textures and visual aspects

This section will talk about the non-functional requirements of our project and their importance. When it comes to non-functional requirements they are not mandatory for something to operate but are still very much important, as they help the overall experience of a user. These requirements will help ensure that we make a project that can meet a user's expectations and overall improve the quality of said project.

One of the first important non-functional requirements we must think about for our project is how reliable the project is. For example if the game we create is slow or is prone to crashing this will hurt the overall project's quality. When creating our project we will need to ensure that we make the game reliable to where it is expected to be able to run with very few to no crashes. This is because a user will expect this project to be reliable in the sense that they do not have to fear or worry about crashes. If a project is seen as prone to crashing the user will be unable to trust in the project as a whole and will reject it.

As well as needing to make the game able to run at a good enough speed that the gameplay feels fluid. This means that the game will be able to take in and execute commands at high speeds. The output of the game will not seem laggy or incomprehensibly too fast for a user. All of this ensures the user is able to react and respond to actions or events that are taking place in the game. Vastly improving the quality as the user will not feel that any losses are cheap, unfair, and were out of their control.

Another non-functional requirement to be sure we implement is button layout and good feeling controls. What I mean by this is that something that is overlooked until it becomes an issue is the use of buttons and their corresponding actions. For example most games use the space bar or A for xbox, X for playstation, for the simple jump mechanic. Generally depending on the console or system of play the name of the button can change but games released for the system will stick to this button. The reason for this is because most people prefer certain buttons to perform certain actions and a jump mechanic being put as these buttons have no backlash from players. But if one were to put the jump button in an awkward position the user would not be happy. This issue can even become worse if all the buttons are placed awkwardly and in non-user friendly positions.

The last non-functional requirement to talk about is about the presentation of the game, or in other words the visuals of said project. We need to ensure that the visuals we use are clearly visible for the player as well as not painful on the eyes. For example if everything is too cluttered or if the visuals just look poor overall the player will have a hard time enjoying the game. This can even make it difficult to play if the visuals are not properly thought out.

For sprint 2, we did also have to add some new non-functional requirements. First we must make sure the ai has a good feel to it. By making sure that the ai feels like it is an actual decent challenge. Not walking into walls or walking off ledges to their doom, or aimlessly wandering throughout the map leaving the user disconnected from the gameplay.

Next, we want the items and drops to be balanced. It shouldn't heal to little or too much, but it should heal just the right amount to be challenging but playable. Finally, we must make sure that the animations are clean and fun!

New Functional Requirements
Quality Enemy AI
Balanced Items
Clean Animations

1.4 Target Hardware Details

1.4.1 General Requirements

When making software it's important to understand what hardware you are targeting and how to cater your software towards the ideal client's device while maximizing hardware coverage reach other clients. This section will go over the general hardware requirements we are targeting and then will go over some specific requirements that will be covered as the game is developed.

Unity is a universally accessible software that can target phone devices, gaming consoles, Linux, Windows, Macintosh, and other platforms as well. Consequently, that means we don't necessarily, development-wise, have to make any significant accommodations to target most of our desired platform since clients can most likely run the game on virtually any machine. Even with that being the case, there are some preliminary requirements that some machines have to abide by in terms of what versions of said OS or devices are supported by the game.

Unity offers developers the freedom to choose what version of the Unity Engine to develop, test, and build their game with. This offers a lot of flexibility for developers to use older libraries or plug-ins that are deprecated or only support specific versions of the Engine. As a result, different versions of the Unity Engine have certain requirements for supported devices. For example, the Unity 2022 version supports Windows 7, Windows 10, and Windows 11 versions, while the Unity 2023 version supports Windows 10, and Windows 11, but does not support Windows 7. There are other requirements such as GPU, CPU, and various miscellaneous requirements for each hardware to follow.

For this project, we are generally targeting low-end desktops and laptops that meet at least the minimum requirements for Unity. This makes sure that our client who most likely surpasses the minimum requirements can play the game as well as other potential clients as the game has as much hardware coverage as possible.

1.4.2 Specific Requirements

The game could potentially have many input devices but we only focused on keyboard and mouse for this scope. The game will feature simple controls as it is a simple "run and gun" platformer that requires no complex controls or mechanical skills as the character is either jumping, going in one direction, or shooting a projectile. With that being the case, a keyboard and mouse will be necessary to navigate the menu and play the core game as we initially made controls for the keyboard, but Unity has an input manager that can also let us make other control schemes for other input devices, but the only mandatory input requirement is the keyboard and mouse.

As output devices are concerned, there are requirements that computers will meet naturally as most devices come with a monitor, sound system/, etc., but we will restate them for clarity. A monitor is necessary for the game as it will let our users interact with the game. Additionally, we can target most monitor resolutions as Unity has options for the UI and characters can scale up or down to meet requirements for specific resolutions, but to match the original game the game it will be planned to play in full screen, making use of all of the monitor screen space. On the alternative, we have sounds in the game, but an audio system isn't required as sounds only look to immerse the player and not necessary for core gameplay.

As far as the memory components of the hardware, they should be low end providing no strain on the user's device. Concerning the CPU(x64) and RAM(4+GBs) if the computer meets the minimum requirements for Unity we have no issue running our game. Also, storage-wise we predict the game to be less than 1.5 gigabytes so the users will not need much space to have the game on their device.

1.5 Software Product Development

1.5.1 IDE/Programming Language

The development team will be using C# primarily as the core language for this project to develop the game. C# has plenty of IDEs that support it, but we will most likely use Microsoft's Visual Studio and/or Visual Studio code. Both IDEs work very well with C# and have packages that will allow you to work with Unity and its various libraries. It also allows for AI auto-completion of lines of code, syntax error markup, and many other utilities that will allow for the development of the game to go as efficiently as possible.

1.5.2 Version Control with GitHub

GitHub is a superb tool that not only allows for version control but also collaboration with its website features. GitHub will allow us to manage our game and make sure that all devs have access to the correct version of the development space. This is significant because if there is a desync in the development space it could cause workflow problems and code conflicts that can hinder development. GitHub is also accessible in many ways as well such as through IDEs, desktop applications, terminals, and many other channels. The software also allows us to collaborate by offering features such as code reviews that will allow us to double-check new code and scripts being added, as well as a comment system so that for every pull request the developers can explain what they are editing in the project, and project branches so that the main branch of the project doesn't get messed up with many changes at once.

1.5.3 Software Frameworks / Asset Generation

The Unity software has many tools that we can use within it to code certain functionality. One of those tools is the physics engine, which allows users to move the player character and other entities as well as non-player objects through game space. Additionally, it will have collision detection which is key for letting the game know when certain events need to happen, for example, when the bullet hits the player we need to let the game know at that moment to take health points from the player. There are visual functionality such as the animator, canvas system, and tilemap system that will allow the developers to create an immersive visual experience which is fully customizable by the devs. There is also the UnityEngine class that most code will be developed for even further control of characters, visual elements, physics, and various other parts of the game. Furthermore, if there are any assets or plug-ins we need Unity asset store and package manager that will allow developers to import and download any asset for the completion of the project.

2 Modeling and Design

2.1 System Boundaries

2.1.1 Physical

The physical system boundary similarly defines a scope of functions and actions and how they relate to each other. In this case, however, we are analyzing how the systems contribute to the physical hardware of the client. In this example, when looking at the visual aspect of the development we have to use Unity's various visual aid systems like the animation system, scene manager, and UI engine to send data to the client's monitor. Similarly, when taking player input we rely on the user's controller and keyboard to send requests to the input manager to do various things like move the character using Unity's built-in physics system or to control and navigate through the UI. There are other examples of physical hardware interaction with the software with the Unity Engine, but I only noted the most important features that would be the most occurring in the game.

2.1.2 Logical

The logical system boundary defines a scope of functions and actions and how they relate to each other. This is often represented in a diagram that has the various systems and functions and how they connect from front-end to back-end. The outside of the box represents the front-end and the back-end which in this case is the user and their machine and the back end is the database manager. The inside of the diagram represents various libraries and systems that help the front-end and back-end communicate and perform many different actions. Additionally, the inside also includes annotation labels to show how exactly different systems and/or actions connect. To explain the diagram above, we have a user who wants to send their user account and their game state data to the database where the admin can modify the data. Moreover, we have the player using the player and input manager classes to influence game state data being sent to the database.

2.2 Wireframes and Storyboard

Text goes here.

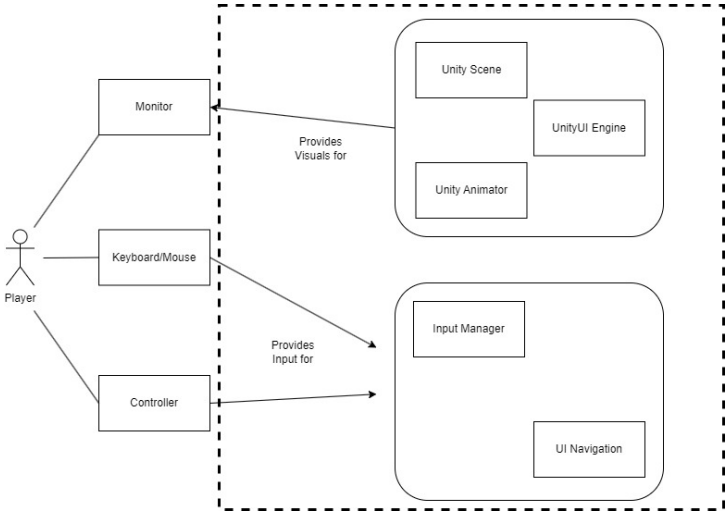

Figure 1: Physical Boundaries

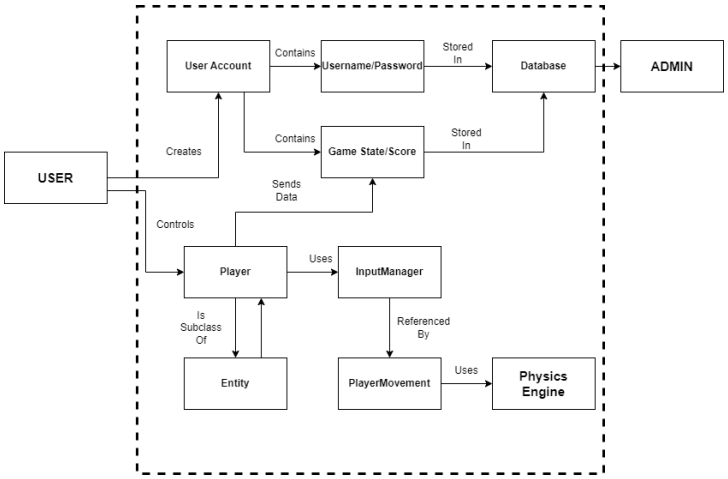

Figure 2: Logical Boundaries

2.3 UML

2.3.1 Class Diagrams

For the Creational Object Oriented Design Pattern our project is implementing the Factory pattern. The factory pattern provides a generic interface for creating objects and upon creation attributes are assigned to the generated object. In this project the Factory pattern was used to solve the problem of creating multiple types of non-player character enemies by basing multiple NPC enemy classes on one main enemy interface. This design pattern is applicable because each enemy NPC have differing health and damage amounts as well as different skins.

The Behavioral Object Oriented Design Pattern this project will apply is the state design pattern. The state pattern diagram allows an object to change the behavior based on state changes. This diagram covers the states of an enemy NPC which includes the Patrol, Attack, Damaged, and Dead states. This allows us to implement behaviors as a collection of self-contained components and will cause a reduction in condition code blocks in the project. This pattern is applicable to this class because enemy NPCs have different behavior states which can be easily encapsulated and organized via this design pattern.

The Structural Object Oriented Design Pattern this project will implement is the Decorator design pattern. The decorator design pattern is used to attach additional functionalities/responsibilities to an object dynamically - allowing for classes to be open for extension but closed for modification. In the project, the decorator pattern is applied to health orb power-ups that the player character can collect. The health orbs have different skins and heal the player for different amounts based on which variant is collected. This pattern is applicable as it allows for the health orb interface to be dynamically applied to a variety of health-orb powerups.

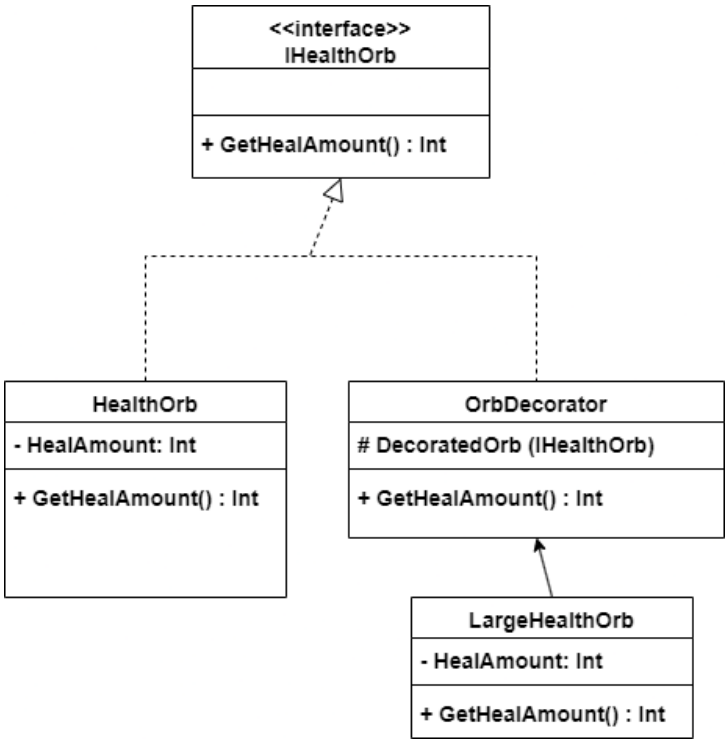

Figure 3: Decorator Diagram. This is located in Appendix listing 2.

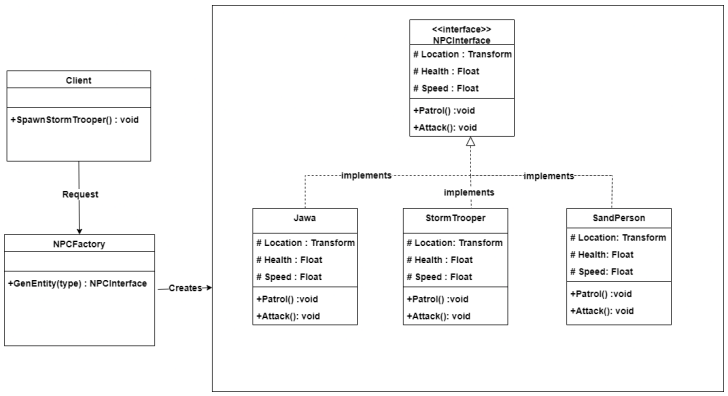

Figure 4: Factory Diagram. This is located in Appendix Listing 24, lines 1-11.

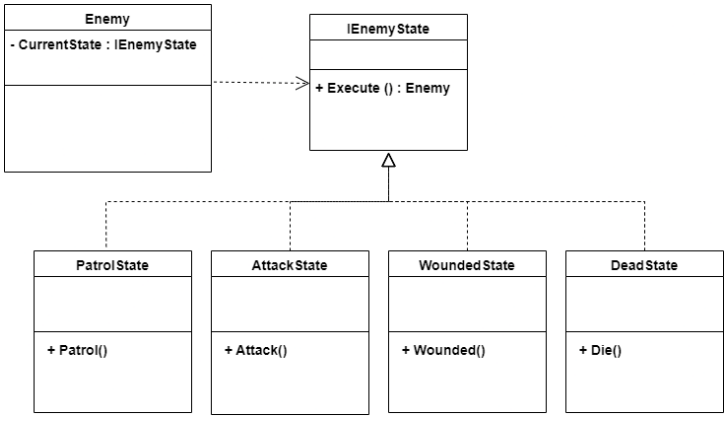

Figure 5: Model of EnemyStates. This is located in Appendix Listing 20, in its entirety.

2.3.2 Use Case Diagrams

This use case diagram maps the higher level functionalities of the game, such as loading, saving, user input, and the user accounts functionality. This diagram will be really beneficial to us when we start breaking out issues into smaller tasks, and will likely serve as the foundation for our "epics," or overarching themes of an issue set.

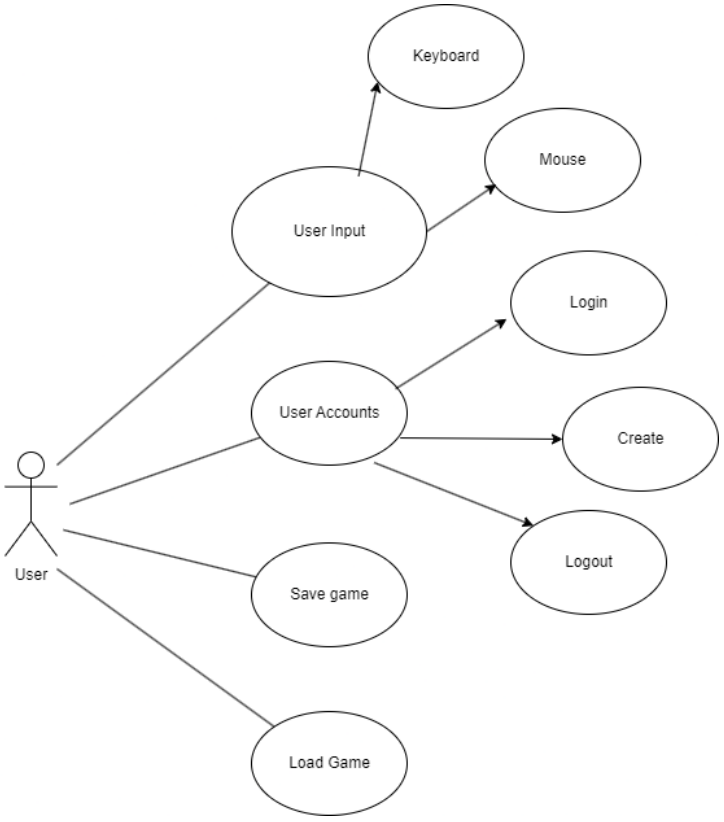

Figure 6: Use case diagram. Appendix, Listing 31, full file

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

2.3.4 Use Case Scenario 1 (Based On 1st Use Case Diagram)

ACTORS: USER COLLISION HANDLER

GOAL: To detect collisions between different game objects that either player-owned or enemy entity game objects

SYSTEMS: PHYSICS ENGINE ENTITY TRANSFORM COLLIDER

This scenario depicts how the user will interact with the physics engine in order to detect collectable items or enemies. This can be further applied to other objects like boost pads and hitbox detection for the speeder level.

2.3.5 Use Case Scenario 2 (Based On 2ND Use Case Diagram)

ACTORS: GAMEMANAGER ENTITY CLASS

GOAL: To Allow the game manager to handle the spawning and despawning of entity objects through the entity class

SYSTEMS: UNITY CLASS SYSTEM ENTITY TRANSFORM

The scenario depicts the interaction between the entity class and the game manager. The game manager will handle any level specific interactions like spawning and collectable placement.

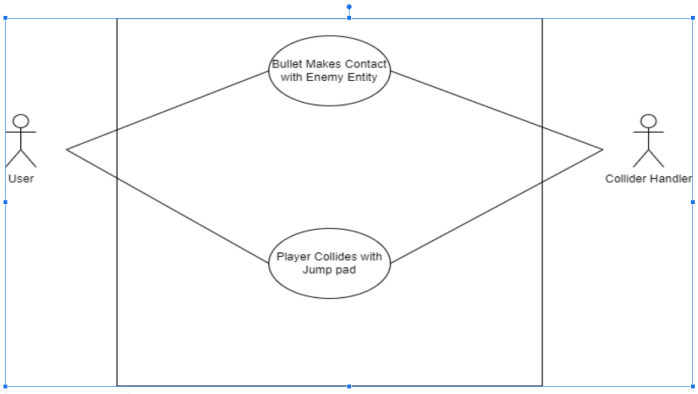

Figure 7: This use case diagram shows the relationship between the User actor and the collision handler. The actor will do actions such as doing damage to an enemy or using a jump pad which all use colliders and the collision handler will manage and send data back to the user. Appendix Listing 17, full code.

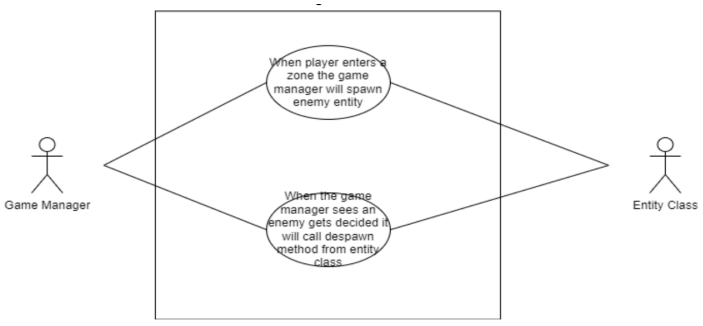

Figure 8: In this use case, we see the game manager as an actor and how it interacts with the entity class. The game manager when an event like an enemy dying or the player enters a zone will call functions from the entity class to spawn or despawn enemies.Appendix Listing 24 (full file), Listing 25, lines 70-75

2.3.6 Sequence Diagrams

The diagram above describes how we intend on implementing our authentication flow. We recognize that there may be additional steps necessary as we move down the line, but it is pretty straightforward. User will click the login button, which will take them to the login screen. Another way to do this is to just load the login screen first thing to the user, but I digress. After the user fills out the form, then the form is validated. Once we do this, we will compare the password to the one that is stored in the database and return a response to the user. Once the user is logged in, they will then be able to go to the main menu. Appendix

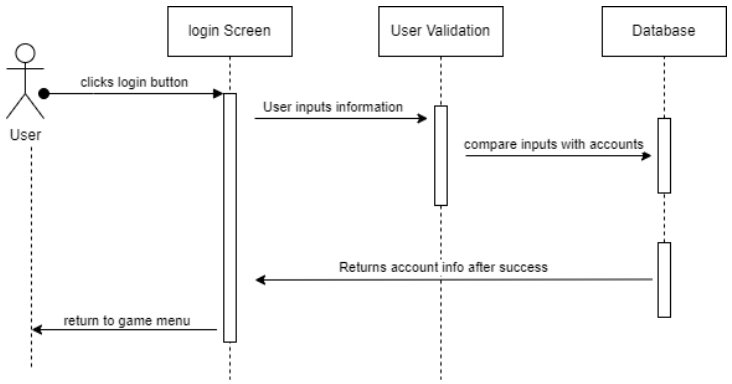

Figure 9: Auth Flow Sequence Diagram

2.3.7 State Diagrams

This diagram describes how the user will press certain buttons to control their characters. It is a pretty straightforward diagram, and most modern games will follow a similar style of implementation. The only thing that could be improved is that instead of giving a hardcoded key value combination, it could be an enum instead. This shows that we are thinking ahead to user control customization.

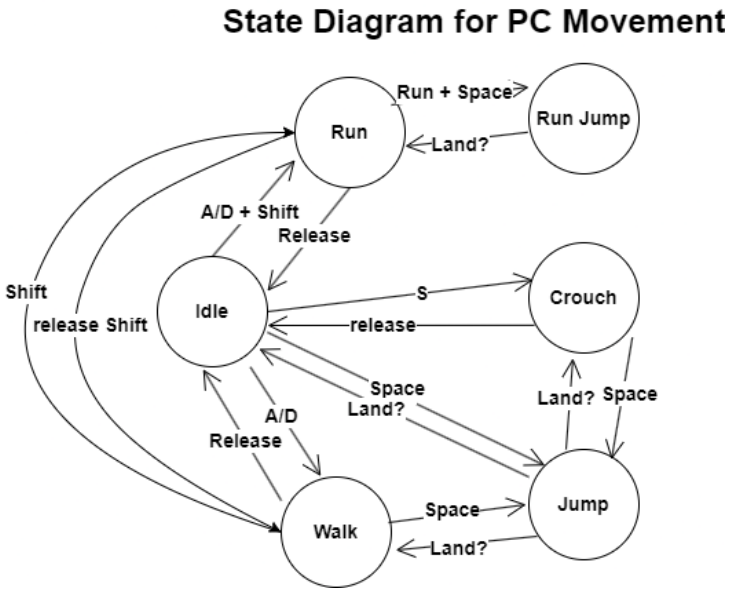

Figure 10: State Diagram for PC Movement. Source code is in Appendix Listing 20

2.3.8 Component Diagrams

This component diagram represents how the data models will be broken up, and how they will communicate with the database as a whole. The web server will be responsible for handling the interactions between the models and the database.

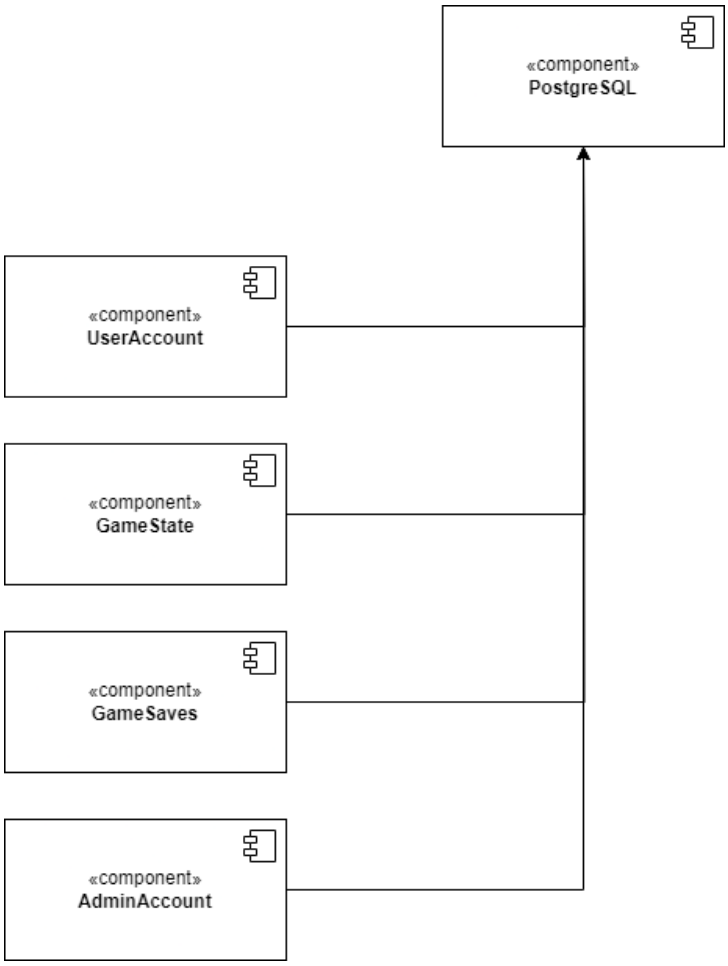

Figure 11: High Level Database Model

2.3.9 Deployment Diagrams

2.4 Traceability Table

2.5 Version Control

Our approach to version control is to use GitHub as our single source of truth for everything related to our project. This allows us to have control and consistency over our project management and codebase at all times. Moreover, it is also beneficial for keeping separate branches, thus always being able to have something presentable for the client.

As far as when we make commits to GitHub, we haven't made too many commits other than to store files such as our organization.tex and technical.tex, but we plan to make commits to our GitHub anytime that we make changes to the codebase of our upcoming Unity project. The way we will be doing so will adopt a master ¡- develop ¡- feature hierarchy, in that all changes will be first made in a feature branch, and then merged into the develop branch. This is where we will QA new features and PRs. Once they have passed the QA and testing stage, we will merge into our main branch.

As far as keeping track of issues, we have decided to adopt the Epic ¡- User Story ¡- Feature / Bug model of issues. This allows us to make higher level goals (epics), break them into medium sized bodies of work (stories), and then break those up into bite size pieces of work (feature / bugs). This allows us more granular insight into how our project is moving along relative to the timeline of our sprints.

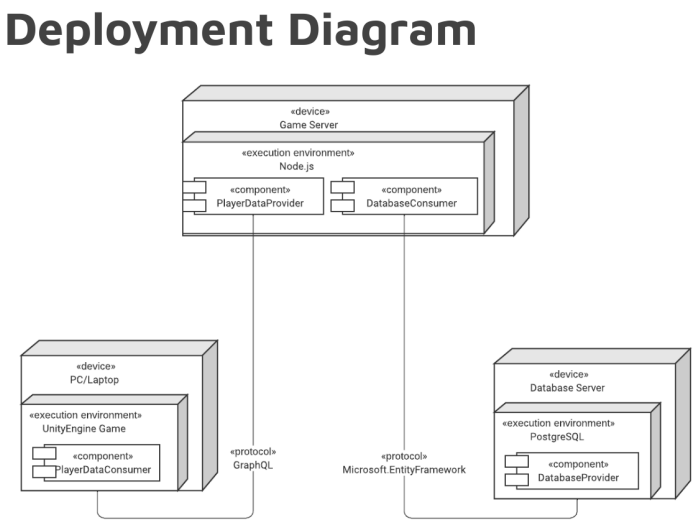

Figure 12: Deployment Diagram

Traceability Trace

Tasks	Requirements Description	Developer	Status
Product Performance	Test the game methods and sweep for performance issues that effect frame rate ,response times, memory usage, cpu time	Zach	Complete
Product Security	Make sure that the user can be authenticated and make sure inputs are secured to make sure the player cannot exploit the game	Matt	In Progress
Player Development	Handles the development of player mechanics, animations, and interactions	Keimon	In Progress
Database/Login Screen Dev	Develops the database using a Django backend, allows for the creation of a user account that can login	Matt	In Progress
Enemy Development	Handles the development enemy animations, hitboxes, and implementing enemy states	Zach	In Progress
Level Design	Creates level tilemap and colliders along with the game manager	Jeremiah	In Progress

Figure 13: Traceability Table

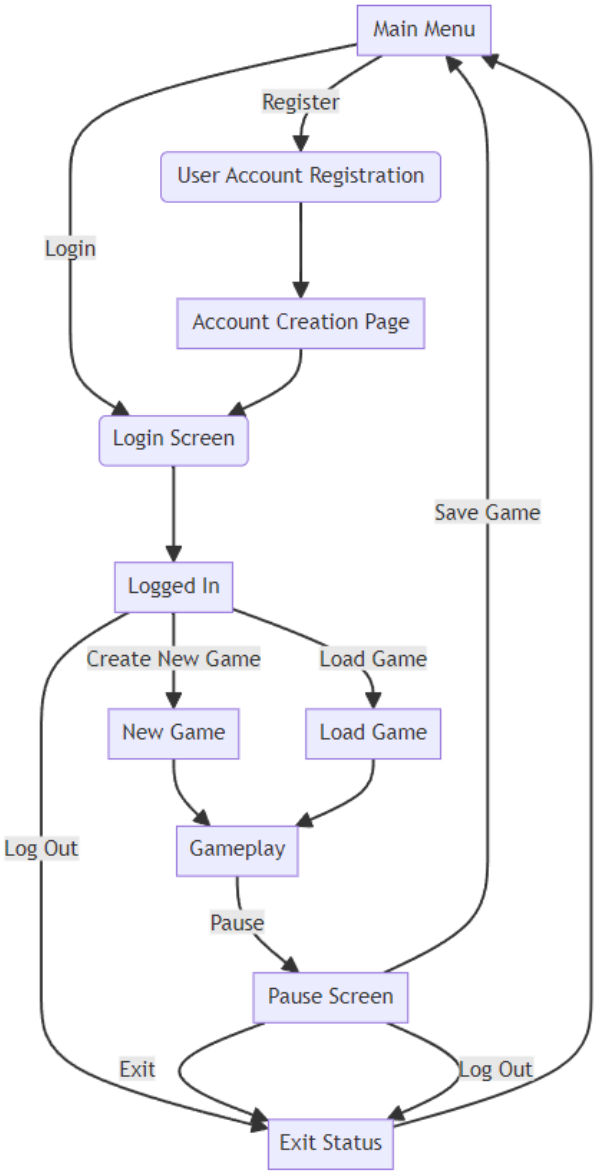

Figure 14: Gameplay Diagram

2.6 User Experience

2.6.1 Gameplay Diagram

This section of the document will talk about a gameplay flowchart and describe it. As one can see in the diagram the base of the flowchart is that the player is moving through the level. We have multiple things that can appear. For instance if there is an enemy. If there is one they can shoot or avoid said enemy. Then they see if there is an obstacle or a gap in the platform. If it is a yes they jump over said obstacle or gap. If not they keep moving forward. There is also a part that says was an item picked up and if one was an effect is applied based on what the item was. If not the player gains nothing and keeps moving. And finally a check on if the player has died. If yes they restart the level if they have an extra life or they get a game over if they do not have an extra life. If they don't die they keep playing.

2.6.2 Gameplay Objectives

This section will talk about the gameplay objectives, which will be about the goal that the player is striving for. As well as talking about why someone should play this game. To start off I would like to discuss why someone should play our game. I feel that people should play our game because it is a fun platformer that takes place in the Star Wars universe. In this game you play as the famous character Luke Skywalker as he attempts to save the galaxy. Another thing is that this story allows the player to play through the same story from the movies which allows the player to live through the story from a more interactive place.

The overall goal of the game is like that from the movies. You are trying to defeat Darth Vader and save the galaxy from the Empire. How you will do this is that you will have to dodge, fight, and avoid enemies of each level. Ultimately making it to the end of said level and moving on to the next. You will also have to traverse through dangerous spikes and other environmental dangers.

2.6.3 User Skillset

This section will discuss the user skillset. This means this section will be about the necessary user abilities necessary to effectively play this game. One of the most important skills the player should have is that they will need to have quick reflexes. This game will have enemies attacking you as you try to platform. If you are unable to react quick enough you will get killed many times by the enemies. Another necessary skill the user should have is timing. The reason for this is that you will need to know when to hit the jump button to make it to another platform, as well as when it is okay to attack the enemy back as if you try and do it at the wrong time you will get hit by an enemy blaster. The player will also need to effectively use the keyboard as if they do not know where keys are or are not used to keyboard controls they might not be able to perform the required actions effectively.

2.6.4 Gameplay mechanics

This section will be about the game mechanics for this game. The mechanics that this game is based on are platformer mechanics and a bit of some run and gun. The platformer part comes from the fact that you are tasked with getting to the end of the level all the while you avoid terrain dangers and enemies. You will jump between different platforms and try your best to not fall off or be killed by the previously explained dangers. This is where a bit of the run and gun comes from. The player will have a weapon to defend yourself and beat the enemies as you traverse the level. The player will be able to shoot their blaster at the enemies and at some point will have a lightsaber to attack as well. As well as looking out for life orbs to heal the player after getting hurt by something or even getting an extra life. The overall package being that you will run, jump, crouch behind cover, shoot, slice with your lightsaber, and pick up helpful items to help you get through the game.

2.6.5 Gameplay Items

This section of the document will involve the gameplay items. This entails the smaller components of our game that will slightly change the flow of the game as it is played. The first gameplay item that will be discussed will be the small health orb. The small health orb will recover the player's health by a small margin. This won't be a whole lot of health but this will allow the player to survive a few more hits and help them out when they are in a pinch. An item that is like the first but better would be the large health orb as this item will greatly recover the players health. Meaning that they could be on the brink of death and after picking up a large health orb they are doing okay, and are now able to take a few more risks. Another item would be the extra life as this will allow the player to make a mistake and have another chance at completing the level before a game over takes place.

2.6.6 Gameplay Challenges

This section will discuss the gameplay challenges of our game. This will entail the challenges a player is expected to face as well as how they are expected to get past said challenges. For the challenges that the player is going to come up against it will be against dangerous terrain and enemies. For the terrain it can be spikes, falling objects, and even bottomless falls. The player will be expected to learn the patterns for falling objects through some observation and thought out movement. We will also ensure that the placement of said objects are fair. For the spikes and bottomless pits the player will handle these through well placed jumps and observations. We will

also ensure placement of these things are fair as well. For the enemies the player will have to deal with these by using their wits and the weapons they are supplied to deal with them. For instance the player can duck to evade a blaster or they can jump. Based on the environment will depend on what works better for the current instance as well. We will ensure that the enemies are well placed throughout the level.

2.6.7 Gameplay Menu Screens

This section will go through the menu screens that will be seen in this game. The first screen that you will find when you enter the game will be the login/signup screen. These will be found on the left side of the screen as the star wars logo is shown on the right. After logging in you will see the main menu screen which will have your account on the upper left part of the screen and the high scores on the right. Under the account info you will find a load game, new game, settings, and controls button. Clicking the load game button will take you to a menu that will have all the loads that are owned by your account and each will have a resume button. The new game will just start the game and you will start to play. The settings function will take you to a menu that will allow you to change certain aspects of the game. The controls button will take you to a menu that will allow you to change your button layout and allow the player to change what button does what. While playing in the game if you press a certain button you will be taken to a pause menu which will allow the player to save the game, quit, or resume. Quitting will then take you back to the main menu screen.

2.6.8 Gameplay Heads-Up Display

This section will talk about the heads up display that will be in the game. This generally entails life bars, scores, and anything that is on the screen that the player uses to understand the current game state. For our game we will have three different displays, and these displays will be a life bar, a score, and extra life counter. These can be seen in the diagram but to explain a bit more in depth is that the life bar will be vertical and will be in the left upper corner. For the extra life counter we will put it in the right upper corner of the screen and it will display a small sprite of the player's character with a number beside it. This number will signify the amount of lives the player currently has left. For the score it will be placed in the bottom right of the screen and this will show the current score that the player has.

2.6.9 Gameplay Art Style

In this section of the documentation the art style of the game will be the topic of discussion. The art style that was chosen for this game was the use of pixel art. This is an art style that reflects the older days of gaming but can still look very good when everything is well detailed. This means all in-game things will have a more cartoon look to it. Every character will have a decently detailed sprite using this pixel art. As well as the scenery and menus. The terrain and scenery will have one sprite and not change its look. The enemies and characters though will use multiple sprites so that all their actions can be well animated. For any cutscenes even though the pixel art style is being used it will have a cartoon look to it but characters will look close to what is seen in the movies.

2.6.10 Gameplay Audio

This section of the document will talk about the gameplay audio. This will entail any sounds that the player will hear while playing the game. First thing will be damage noises and death sounds. These will be played when the player is hit or when an enemy is hit and these will be unique. The same goes for the death sounds as both the player and enemies will have a unique death sound. These will be auditory cues to help the player identify if an enemy has died and who has taken a hit. Another important audio feature that will be included will be the background music. The plan for this is to be engaging but not overbearing. As its name implies it is for background noise. This music will be something that relates to the Star Wars franchise. Basically it will be music that is from Star Wars. Another piece of audio that will be played will be a sound that will go over top of the player's gun. Whenever the player shoots it will play a quick sound. Finally item pick ups will have a confirmation jingle. Basically on pick they will play a sound that will inform the player it was successfully picked up.

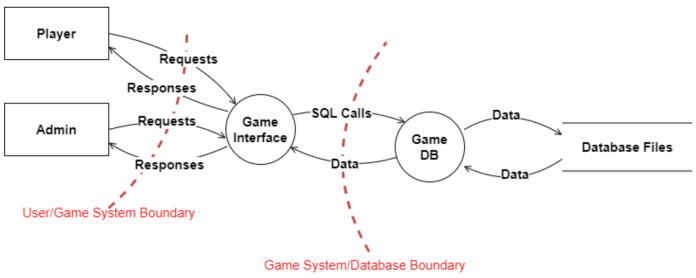

Figure 15: Enter Caption

3 Non-Functional Product Details

3.1 Product Security

3.1.1 Approach to Security in all Process Steps

When considering security in a game, there are a couple of ways to do so. First, what can the client change about their game? Can they make their player move in ways they should not be able to? Can they glitch through parts of the level? Can they cheat and get an unrealistic high score? These are client side risks, that are generally a concern in games that are singleplayer. Yes, this game is single player, but basically everything other than the actual gameplay itself must be authenticated on the Django game server.

When a user logs in, the game will store their credentials (JWT key) for future use. Whenever a user wants to perform any action such as changing their settings, saving or loading a level, or logging in, they MUST provide credentials or the Django server will decline the request.

We perform our API requests using the System.Net.Http class provided by C#. This ensures that our requests are done via HTTPS, which will automatically encrypt any data that is sent over it. Finally, Django also has safeguards in place for common attacks such as CSRF forging, XSS, and more. Choosing Django has allowed us to already have security features such as securely storing user passwords via hashing. Moreover, it allows us to extend these default user attributes and add some of our own. This helps by making it easier for us to implement user data, and it does so more securely as we are not trying to implement hashing ourselves. Doing this removes a possible avenue for attack.

Overall, we are really happy with the security of our product. I am sure there are plenty more things we should be doing, but for now this is a good start.

3.1.2 Security Threat Model

Our security threat model has three areas: 1. the User, 2. the Game server, 3. the database. The boundary between the user and the game server is one that can only be accessed with user credentials. Even then, they only have access to modify their own data, no one else. The boundary between the game server and the database is a more touchy one especially if we were directly interacting with the database instead of using a web framework like Django. This removes risks such as SQL injection because we never make a SQL call to the database, we only interact with the data via models in Python.

The biggest risk would be if somehow a Django secret key got leaked after production, and people were able to spoof API requests to a phony Django server. However, this is not easily done, and can be easily mitigated by using environment variables for any secret keys. This way, your repository never stores any sensitive data.

3.1.3 Security Levels

In security, you should follow the practice of give NO permissions by default, and give only the ones that are required. So, we are going to follow this rule. Default users will not even have access to the Django admin panel what so ever. They can only use the UI provided in the game to update their data such as settings, saves, user data, and more.

In contrast, the admin user will have full destructive access over the database and app itself. The admin will have access to add / remove high scores, add / remove user saves, delete users, and everything an admin should have access to. If we were handing this off to an end user, we would likely not give them full access, but just the permissions they would need. This removes the risk of having a careless manager who leaks their admin credentials, and minimizes the would be damages.

The way that a user is authenticated is by sending a POST request to cs360.irizarry.dev/accounts/login and if the login is successful, it will return a user object with the user credentials in it. The credentials are stored and then will be used in subsequent requests that require authentication. The app also supports /accounts/register.

3.2 Product Performance

3.2.1 Product Performance Requirements

The performance requirements should be set to ensure a smooth and satisfying user experience. If the performance of a product is not adequate for a consistent and fluid user experience then users will be unsatisfied and will not want to utilize the software.

Our target hardware is one of the team member's laptops, which has an AMD Ryzen 7 processor with an integrated graphics card and 16GB RAM, this should be more than sufficient for this project. For this product, we will track frames per second (FPS), CPU time, memory usage, duration of method calls, and time elapsed on database calls. The aim is to maintain a high frame rate (FPS) at equal or greater than 60 frames per second. The target CPU time for gameplay is less than 16ms. It should be acknowledged that at the lower end of the spectrum relative standard deviation is a faulty metric because even a change of a couple ms can drastically increase this measurement; as such this measurement will only be considered as performance approaches the minimum requirements. It is essential that our database operations be quick to ensure a positive user experience; therefore, database operations such as user authentication and save/load game states should take a maximum of 2 seconds. Finally, the game should be very responsive to user inputs and should respond to user input within 50 milliseconds, for this method calls will be observed for their duration. As development continues requirements will be set to set limits on scene switching times, throughput, load testing, and stress testing.

By keeping defining the performance requirements before development, it helps ensure that the development process will result in a good product. Additionally, it is important to measure and track these requirements throughout development so that mistakes or low performance can be caught and remediated.

3.2.2 Measurable Performance Objectives

To adequately meet product performance requirements it is essential that performance be measured throughout development. Measuring performance allows the team to monitor defined metrics and find bottlenecks and correct them. To track the gameplay metrics of FPS, CPU Time, and memory usage the team created an empty game object named 'pTest' and placed it in the scene. A script was then attached to 'pTest' which outputs the CPU time, FPS, and memory usage to a .log file every frame. As stated above the goals of 60 FPS, a maximum of 16 ms CPU time, and less than 2 GB memory usage have been set to ensure a smooth user experience. The goal for CPU time is 16ms or less, but the target is really consistent performance meaning the game should avoid major spikes in CPU usage. Relative standard deviation of CPU time should be kept under 5-10%, particularly as CPU time approaches 16ms. Currently it is not possible to implement testing for scene loading time because the game currently has only one scene implemented and no menus to load from. Stress and load testing are also not very feasible given our limited implementation, but these requirements will be set and tested as development progress is made.

3.2.3 Application Workload

Application workload analysis is important so developers can track how users are interacting with the product. This can showcase potential difficulties in the menu system, allowing the team to modify the user interface to be more straightforward. Additionally, this type of testing will highlight the most used parts of the game (perhaps monitoring the top scores) which would help the team allocate resources to enhance the overall user experience. At this time in the development process, the user interface is non-functional and has not yet been implemented in

the design. Therefore, we do not have direct historical data to demonstrate accurate workload measurements and percentages. To accomplish this in the future timers will be created on all major UI features, which will allow the team to monitor time spent on each feature. Some of these timers will include but are not limited to: main menu timer, gameplay timer, scoreboard timer, save/load game timer, login timer. In addition to timers, it would be useful to add even trackers to record specific actions, such as the number of times menus are accessed, how often a game is saved/loaded, number of times the game is paused and resumed. In a real world scenario, it would also be useful to deploy surveys and feedback forms during the testing phases, which would provide valuable feedback and insights on UI and workload analysis. A hypothetical workload might look like 10% of time spent in main menu interactions, 80% of time playing games, 3% of time saving and loading the game, 2% of time logging in, and 5% of time looking at highscore menus. These hypothetical numbers can serve as a loose goal for the future.

3.2.4 Hardware and Software Bottlenecks

Hardware and software bottlenecks are parts of a system where performance capacity is significantly lower than other points, creating a gap in capacity which can affect the overall efficiency of the system. It is essential to identify and address these bottlenecks to prevent system failures, delays, user dissatisfaction, and increased costs. This is a key portion of system optimization and is essential to monitor in order to create and maintain a responsive product.

Major potential hardware bottlenecks include reading data from the disk, shortage of memory, moving data from memory to CPU, and network capacity. In terms of our project this would look like database read and writes for save games, highscores, and logins. Additionally, because the team has opted for a live Django server to host the database, network connectivity could be a major bottleneck as well.

Possible software bottlenecks include poorly written algorithms that do not scale well with the scope of the program. This is a risk we are currently monitoring via method call timers and code reviews. The performance testing that is being conducted would also highlight potential bottlenecks. Overall, the largest performance bottleneck our project may face is the Unity Engine because Star Wars SMS is a very light 2D platformer that will be very light performance wise.

So far, the product is well within the defined requirements and there are no glaring bottlenecks or issues that are going to cause performance issues as of yet. After monitoring with testing nothing arose and the performance has been within expected areas.

3.2.5 Synthetic Performance Benchmarks

Synthetic performance benchmark testing is important for developing software for a target system because it allows developers to see how well a system can handle demands that its application might pose. The benchmark tests were performed on a device that is within the target hardware specifications, which was a laptop with AMD Ryzen 7 5825U processor with Radeon Graphics (2.00 GHz), and 16.0 GB RAM.

The CPU performance tests were run using the following line 'sysbench cpu –threads=X –cpu-max-prime=15000 run' using thread counts of 1, 2, 4, 8, 16, 32, and 64. Each thread count was run five times on the target machine to demonstrate an average metric which can be used to establish a trend in CPU performance.The execution time increases linearly with the increasing number of threads, this is expected because more time is dedicated by the CPU to manage thread management and switching. The total number of events increases drastically as the number of threads increases from 1 to 16, suggesting the CPU is able to handle much more work as more threads are added up to a certain point. From 16 threads on the average elapsed time increases but the number of events does not, meaning that the overhead cost of managing that number of threads outweighs the added performance (which in this case is not much).

The file input/output tests were run using this line 'sysbench fileio –file-total-size=X –file-test-mode=rndrw –threads=8 run'. These tests were run using a variety of different file sizes ranging from 512 MB to 32 GB. Each test was run a total of five times to establish a trend in system performance. The execution time of each file size was fairly constant, only increasing significantly for larger file sizes. The throughput of read and writes remains fairly constant across file sizes, except for the 32GB file size, which shows a higher throughput for both read and write.

The results of the synthetic benchmark tests allow the team to see potential bottlenecks in system performance and potential areas of concern. In the case of this software product and the target software, we determine that our

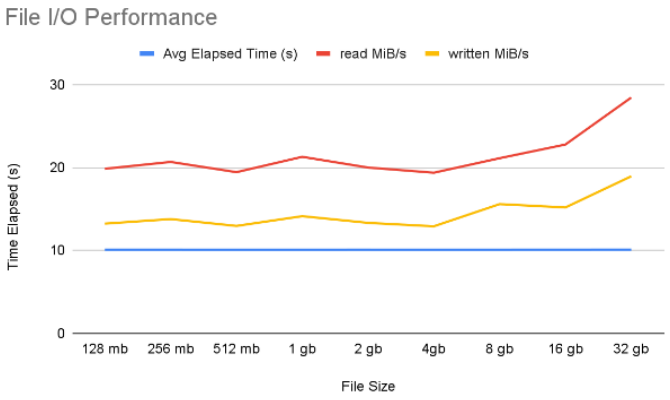

Figure 16: FILE IO Perf

Figure 17: CPU Perf

hardware performance will not be an issue in the performance of our product. In the future it would be useful to implement testing on the database using sysbench or another similar performance software to establish boundaries in the database performance as well.

3.2.6 Performance Tests

Performance testing is important in game development to ensure a smooth user experience. Gameplay should be monitored for consistent performance, without lag or crashes, which is essential for user satisfaction. So far, FPS, CPU Time, Memory usage, and method call duration are what have been implemented and monitored. These metrics will allow for monitoring of basic game performance in the scene we have implemented and would allow us to detect any hindrances in performance if there were any. Method call monitoring highlights possible inefficient code and helps give the team possible sections of code that need to be refactored. The method calls that have been implemented so far perform within our expectations and do not slow the game in any capacity. Additional areas of testing include throughput testing, stress testing, and load testing which could be designed and implemented in the future to determine the performance resilience of the game. Game scene transitions as well as database read/write speeds are planned on being monitored once those features are implemented. In previous sections, the methodology for our performance testing was discussed in detail, as well as future plans to implement said tests. So far the performance of the game exceeds our performance requirements and does not showcase any areas for improvements or any bottlenecks. These areas will continue to be monitored to showcase any issues that may arise.

3.2.7 Method Time Tests

In order to perform a performance test of the in-game c sharp functions, a script that would allow for the functions start and finish times to be exported to a log file document with the ability to check how many milliseconds

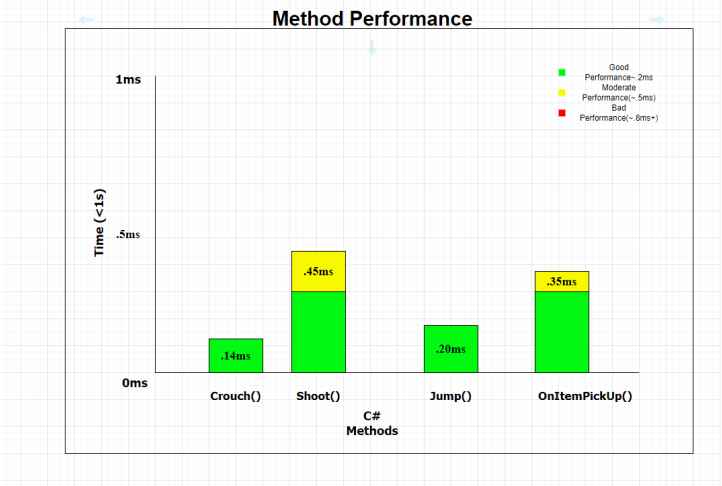

Figure 18: Method Time Graph

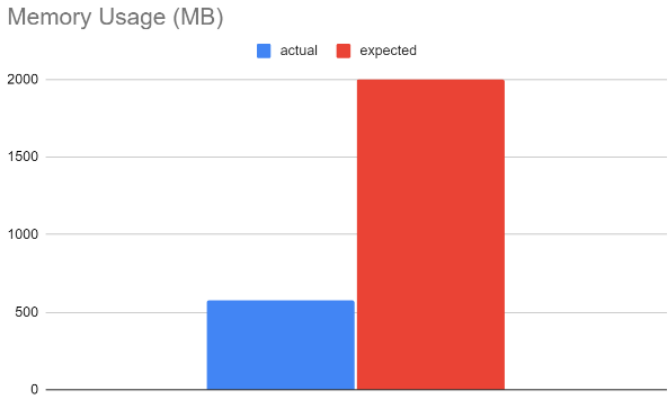

Figure 19: MEM Use exp v act

has passed since the native Unity console log does not check for milliseconds. We also did this so that we could store the log as unity logs resets every replay and we would like to archive data for a longer time period than that.

We rate a method performance wise as follows: 0s - .2s is good, .2s - .5s is moderate, and .5s ¡ is bad performance. For the crouch method a rating of good performance was expected as it only changed the crouch boolean and starts an animation, the shoot method got a moderate rating which is also expected as it is instantiating a projectile prefab and launching it for a duration, the jump method got a good since it is only adding force to the rigidbody and playing an animation, and lastly the most unexpected method for item pick up got a good performance which is surprising as it has to do a collision check then destroy a game object which I thought would take longer.

Overall, the performance averages out to good which is great for a platformer as our response times are near unnoticeable which helps with fast paced gameplay and making sure the user has little down time when it comes to waiting for a function or method call to be complete.

4 Software Testing

4.1 Overall Software Testing Plan

Test Plan Identifier: 00

Introduction: This comprehensive test plan outlines the unit, integration, system, and acceptance testing of the Sega Master StarWars game recreation. Each phase of testing focuses on different aspects of the game to ensure integrity and functionality, integration, reliability, and alignment with client expectations.

Test item: The SUT encompasses all components of the recreated Sega Master StarWars game, including player controls, enemy AI, UI elements, login functionality, and game mechanics. Developed in Unity (version

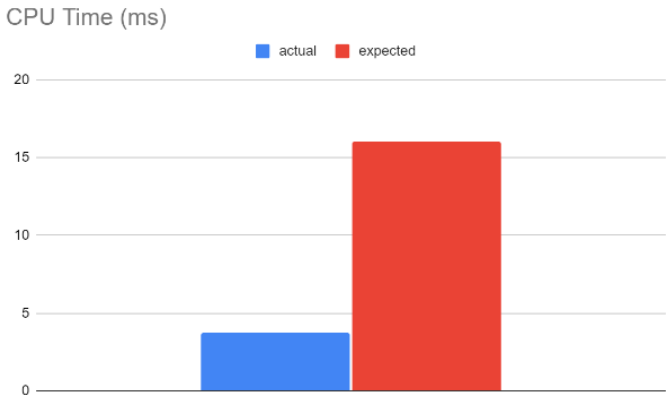

Figure 20: CPU Time exp v act

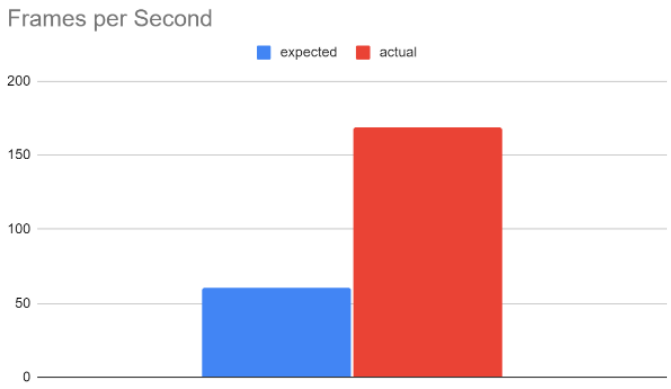

Figure 21: FPS exp v act

2022.3.11f1) with C sharp, the testing ensures each component functions individually and together.

Features to test/not to test: Testing covers player movement, enemy interactions, UI functionality, combat mechanics, and environmental interactions. Out of scope are features like background music, scoring screens, and multiplayer elements, deemed lower priority for current testing phases as those are lower priority or out of scope for our current needs and implementation.

Approach: The approach is majorly manual testing and review by the team. Unit tests will focus on components individually to test mechanics and functionality in a test environment. Sometimes this will involve printing results to the console to ensure its functioning correctly. Integration tests focus on the combination of individual components and ensure they work cohesively. System tests ensure overall functionality. Finally, acceptance tests verify that the product meets client requirements.

Test deliverables: Deliverables include test plans, explicit test cases and results, cyclomatic complexity calculations, flow graphs, and the final documentation. These together provide comprehensive coverage of testing and its outcomes.

Item pass/fail criteria: Pass/fail criteria are defined for each rest case based on expected behavior and outcomes, functionality, and performance. Failures are deviations of that, bugs, and/or crashes. Pass/Fail criteria vary on the different test cases and testing types but overall expected results will be compared with the actual functionality.

Environmental needs: Testing requires a stable machine and Unity environment, test scenes for testing components individually, an internet connection to test database connectivity with Firebase.

Responsibilities: The development team handles unit and integration testing, focusing on their code segments. System and acceptance tests involve the entire team, with responsibilities including test case development, execution, and documentation.

Staffing and training needs: The team requires proficiency in Unity, C Sharp, and testing methodologies. Training in specific testing techniques and tools, as well as in resolving merge conflicts and debugging in Unity, is essential.

Schedule: The testing spans from November 9 to November 29, 2023, with distinct phases for unit, integration, system, and acceptance testing. Key milestones include test planning, execution, and final client presentation.

Risks and Mitigation: Risks include code complexity, integration challenges, technical issues, and missing client requirements. Mitigation strategies involve thorough planning, continuous communication, regular code reviews, and adaptive test case development. Additionally, there is the possibility of one testing type taking too long and holding up the schedule so that we get behind. Integration testing could be the primary culprit of this especially as we work to get separate components integrated.

Approvals: Approved by Zach Vance, Jeremiah Campbell, Matthew Irizarry, Keimon Bush on November 11, 2023.

Unit Testing Checklist

☐ Code Reviews

- Review code for clarity and maintainability
- Ensure adherence to coding standards

☐ Unit Testing

- Test individual game components
- Validate functionality of each unit independently

☐ Creating Flow Graphs

- Develop flow graphs for complex code segments

☐ Calculating Cyclomatic Complexity

- Calculate complexity for each unit
- Identify potential areas for simplification

Integration Testing Checklist

☐ Prepare Test Environment

- Set up a controlled integration testing environment

☐ Component Integration

- Sequentially integrate components
- Test interaction between integrated components

☐ Bug Identification and Resolution

- Identify and document integration issues
- Collaborate with developers for bug fixes

System Testing Checklist

☐ System Functionality

- Test the complete system for overall functionality
- Validate game performance and stability

☐ Interoperability Testing

- Ensure compatibility with external systems

☐ Boot and Startup Testing

- Verify proper game startup and initial loading

Acceptance Testing Checklist

☐ Client Requirements Verification

- Ensure all client-specified requirements are met

☐ Acceptance Testing

- Conduct acceptance testing meeting with client

4.2 Unit Testing

Unit Software Testing Plan
Test Plan Identifier: 01
Introduction: This test type is unit testing, which is a kind of dynamic testing. For this test type we tested some of the basic components and functionalities of the game individually. Additionally we created flow graphs and calculated cyclomatic complexity which helped maintain a low level of code complexity and helped us generate test cases for the unit tests.

Test item: The software being tested is some of the base components of the recreated Sega Master StarWars game. These base components were tested separately from the rest of the system so we could ensure basic functionality. The code tested was written in C Sharp and is intended to be run in Unity version ¡2022.3.11f1¿.

Features to test/not to test: It was determined that the components to be tested would include player movement, the login function, the pause menu, and player collision. These are vital components that make up the base of the game and meet many of the functional requirements established by the client. Out of scope features such as background music, scoring screens, exact level design.

Approach: The strategy to test the software was to generate test cases and expected results by completing flow graphs and then isolating the component to be tested. The outcome will be tested by checking for proper output either via the console or by observing the expected effect onscreen.

Test deliverables: The deliverables from unit testing will be the test cases executed, the results, the flow-graphs, and the cyclomatic complexities for the code to be tested.

Item pass/fail criteria: For player movement the pass and fail criteria are whether the player PreFab completes the correct action based on the user input (A,D for lateral movement, space for jump, ctrl for crouch). If the player does not complete the action upon user input that would be a failed case. Login function will be tested by clicking buttons and if the appropriate action was completed. If nothing happens then that would be a failed test case. The pause menu will be tested in a test scene and will be checked that a splash screen is displayed and that time is paused. If nothing happens or one of the expected outcomes does not occur those will be failed test cases. Player collision will be tested that the player can interact correctly with environmental variables.

Environmental needs: A simple test scene was used to conduct unit tests. The scene was mostly blank but included a small platform that allowed the player to check for collisions with the player. The scene also allowed for a place for the pause screen to function. Console logging statements were placed in some of the code, particularly to test for Login function (which would print the corresponding action of the button pressed).

Responsibilities: Unit tests were primarily conducted by the team members who developed the corresponding code as it was fairly integrated in the development process, and those team members were more familiar with the code themselves. During code reviews and similar all team members were responsible for reviewing codes and approving it during pull requests and similar.

Staffing and training needs: The team had to learn how to construct flow graphs and compute cyclomatic complexity and generate test cases and expected results from those. Otherwise the team was fairly accustomed to testing basic functionalities of their code from previous projects.

Schedule: Unit testing took place between the dates of November 9, 2023 and November 17, 2023. Most of this testing was concluded before integration testing began. Ideally flowgraphs were constructed before unit tests were begun, but that was not always the case due to time constraints and were sometimes constructed after the fact. However, familiarity with the code was strong enough that the graph was not always necessary to identify test cases and desired results.

Risks and Mitigation: Risks of testing players movement included not having collision properly set, during testing actually it was observed that Luke fell through the platform or hovered a little - both issues were able to be fixed through refactoring the code. Risks of testing the pause menu was the time not being stopped entirely or the player still being able to move- none of which was encountered during testing but there were additional steps in coding that method that could have been implemented were that the case. Finally, there was a risk that upon collision the player would not interact correctly or the collision methods would not trigger.

Approvals: Approved by: Zach Vance, Jeremiah Campbell, Matthew Irizarry, and Keimon Bush on November 18, 2023

4.2.1 Source Code Coverage Tests

Player Movement
Cyclomatic Complexity:

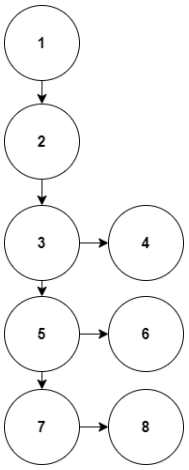

Figure 22: Flow graph for player movement function. Appendix listing 20, lines 8-19

CC = E- N + 2P
CC = 7 - 8 + 2(1)
CC = 1

Basis paths: 1, 2, 3, 4 //Min - player jumps
1, 2, 5, 6 // Avg - player crouches
1, 2, 3, 4, 7, 8 // Max - stop crouching

Login Function
Cyclomatic Complexity:
CC = E- N + 2P
CC = 33 - 34 + 2(1)
CC = 1

Basis paths:
1, 2, 3, 29, 30, 31, 32, 33, 34 -(Min) Successful Login, main scene loaded
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 -missing email
1, 2, 3, 4, 5, 6, 7, 8, 12, 13, 14, 15 -missing password
1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 17, 18, 19 - (Avg) Wrong Password
1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 21, 22, 23 -invalid email
1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 25, 26, 27 - (Max) Account does not exist

Pause Function

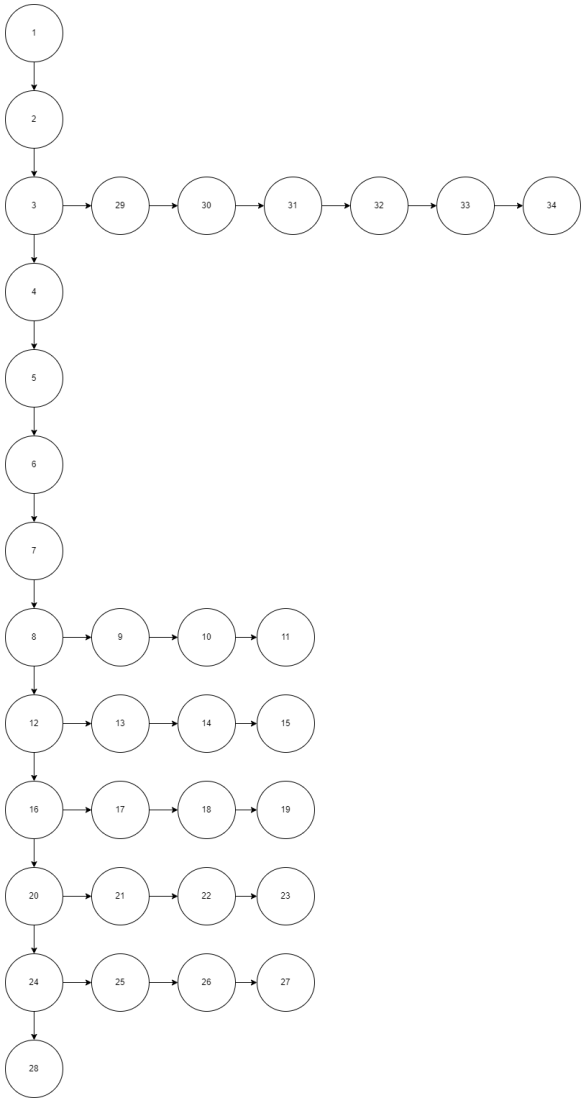

Figure 23: Flow graph for Login functionality. Appendix Listing 31, Lines 33- 67

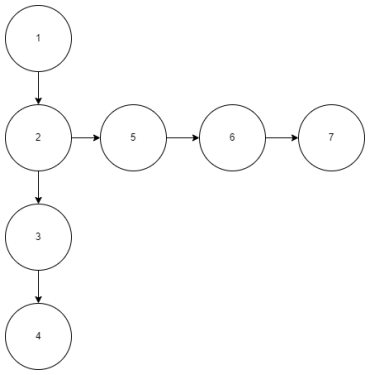

Figure 24: Flow graph for pause menu functionality. Appendix, Listing 25, lines 103- 115

Cyclomatic Complexity:
CC = E- N + 2P
CC = 6 - 7 + 2(1)
CC = 1

Basis paths:
1, 2, 3, 4 //Min - Resume

1, 2, 5, 6, 7 // Max - Pause

Player Collision
Cyclomatic Complexity:

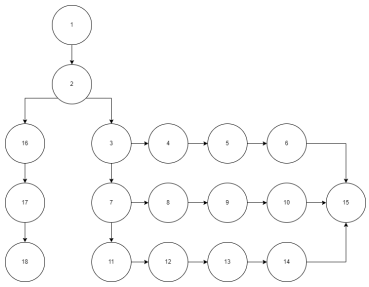

Figure 25: Flow graph for player collision. Appendix Listing 17

CC = E - N + 2P
CC = 19 - 18 + 2(1)
CC = 3
Basis paths:
1 2 3 4 5 6 15 //pickup health
1 2 3 7 8 9 10 15 //Avg - pickup extra life
1 2 3 7 11 12 13 14 15 //Max- pickup blaster upgrade
1 2 16 17 18 //Min- collision with enemy damage

4.2.2 Unit Tests and Results

Player Collision Test Cases		
Test Case	**Expected Result**	**Pass/Fail**
Collide with Health Orb	Health Orb will disappear	Pass
Collide with Extra Life	Extra Life will disappear	Pass
Collide with Blaster Upgrade	Blaster Upgrade will disappear	Pass
Collide with Enemy Damage	“Luke Takes Damage” is printed	Pass

Player Movement Test Cases		
Test Case	**Expected Result**	**Pass/Fail**
Player Jumps when pressing space	Player sprite will jump	Pass
Player crouched when ctrl is pressed	Player sprite will crouch	Pass
Player stops crouching when ctrl is released	Player sprite will stand	Pass

Login Function Test Cases		
Test Case	**Expected Result**	**Pass/Fail**
Successful login when using correct credentials	"Email successfully logged in"	Pass
Missing Email, return error when user doesn't input an email	"Email Missing"	Pass
Missing Password, return error when user doesn't input a password	"Password missing"	Pass
Wrong password, return error when user inputs the wrong password	"Wrong password"	Pass
Invalid Email, return error when user doesn't input a valid email	"Invalid Email"	Pass
Account nonexistent, there is no account associated with the email	"Account does not exist"	Pass

Pause Menu Test Cases		
Test Case	**Expected Result**	**Pass/Fail**
Execute the pause menu by pressing escape	The game pauses and pulls up the pause menu panel	Pass
Exit Pause menu by pressing escape	The game will resume and the pause menu panel will vanish	Pass

4.3 Integration Testing

Test Plan Identifier: 2

Introduction: This test plan focuses on the integration testing of the software product, the recreation of the SegaMaster StarWars video game. The objective is to validate the integration of the player, enemies, the main scene, environmental interactions, and UI components. The goal is to ensure all these separate components function together as intended.

Test item: The Software Under Test includes player movement mechanics, enemy behavior, combat mechanics, environmental interaction, and UI components such as the login and pause screens. All these aspects are part of the StarWars game that has been developed in the Unity environment.

Features to test/not to test: Features to test include player movement, enemy interactions, combat functionality, full scene integration, collision and environmental damage, and UI functionality. Features not tested at this stage are database functionality, music, and sound effects. We focused mostly on major implementations that would have the greatest effects on the single player experience.

Approach: This testing strategy will include manual testing for testing the user experience and interactive elements. The components will be integrated one by one and tested at each step at integration to make sure the components work well together. This testing was mostly straightforward in ensuring that our separately developed assets worked well together.

Test deliverables: Deliverables will include this very comprehensive test plan, the test cases, and the test results from our testing. This document and the final documentation will provide a comprehensive coverage of this testing in providing how we completed these tests and what exactly was done.

Item pass/fail criteria: Pass criteria are defined by the successful completion of the test cases without bugs. Fail criteria include crashes, bugs, or deviation from expected behaviors. Passing will involve proper interaction between the separate components once they are integrated. If there are issues the team will backtrack and find out the specific issue.

Environmental needs: The hardware needs include a machine that can run the game and unity is needed. Software requirements include the Unity environment and all of the separately developed components, assets, and scripts.

Responsibilities: Team members will oversee and assist with merge conflicts, as there have been many in the past this could be an extensive task that requires some training. Additionally beyond the merging, testing will need to be defined and conducted.

Staffing and training needs: The team will need training in resolving merge conflicts, in testing software, and in resolving errors in Unity's 2D environment. Additionally, skills in CSharp are needed to be able to attempt debugging and resolving unintended behaviors or similar issues.

Schedule: This testing will begin once Unit testing has been completed and the components have been individually developed. This testing took place during 11/12 to 11/22, which included the planning, development of test cases, resolving merge conflicts, and completing the integration test cases.

Risks and Mitigation: Risks include improper integration, unexpected program behavior, game crashes, compile errors, and other unexpected technical issues. If merging the game goes poorly there is a risk of a loss of work progress and further time loss due to needing to recoup lost work. There is a risk of a large amount of compile errors on integration and needing to spend lots of time debugging and resolving those errors which will lead to more time being spent on resolving those issues.

Approvals: November 22, 2023 Matthew Irizarry, Jeremiah Campbell, Kiemon Bush, Zachary Vance

4.3.1 Integration Tests and Results

Integration Test Cases		
Test Case	**Description**	**Pass/Fail**
Player Movement Test Cases		
Forward Movement	Test forward movement of the player character.	Pass
Backward Movement	Test backward movement of the player character.	Pass
Jump	Test jumping ability of the player character.	Pass
Enemy Integration Test Cases		
Enemy Spawn	Test if enemies spawn correctly in the environment.	Pass
Enemy Movement	Test movement patterns of enemies.	Pass
Combat Mechanics Test Cases		
Player Attack	Test player's ability to attack enemies.	Pass
Enemy Damage to Player	Test if the player takes damage from enemies.	Pass
Full Integration Test Cases		
Integrated Movement	Test player and enemy movement in the full game scene.	Pass
Integrated Combat	Test combat mechanics in the full game scene.	Pass
Collision and Environmental Damage Test Cases		
Collision Detection	Test if collisions between player, enemies, and environment are detected correctly.	Pass
Environmental Hazards	Test player interaction with environmental hazards (like spikes).	Pass
UI Functionality Test Cases		
Pause Menu	Test the functionality of the pause menu in full integrated scene.	Pass
Game Over Screen	Test the functionality of the game over screen in full integrated scene.	Pass
Login Screen	Test the functionality of the login screen.	Pass
Start New Game	Launch full game scene from UI.	Pass

4.4 System Testing

Text goes here.

4.4.1 System Tests and Results

Text goes here.

4.5 Acceptance Testing

Test Plan Identifier: 4

Introduction: This acceptance testing is designed to check the functionality and performance of the first recreated level of the StarWars video game. This testing aims to verify that the specified requirements by the client, Galloway Games, Inc.

Test item: The Software Under Test (SUT) includes various features of a typical video game and most of the functional requirements as outlined by the client. These features include player accounts and login, player controls, state saving, splash screens, a pause screen, and game over screens. The game is expected to accurately and correctly respond to user inputs and display proper UI components under their appropriate circumstances.

Features to test/not to test: Features to test include player controls, login functionality, state saving, and UI components. These aspects of the game are critical for user experience and game progression. Out of scope for testing include scoreboards, account management, and multiplayer functionalities as those were dropped from the scope of this project.

Approach: The strategy of testing this software is to do a simple playtest with our client and allow them to verify the progress of the game and see all the functionalities that have been implemented. They will be able to verify if the criteria are adequately met and check to make sure that the correct product was created.

Test deliverables: The deliverables for this testing will be well-defined test cases and the testing plan. As well as the game for the client to test and check the test cases with. The client will be able to verify the test cases themselves or view us complete the test cases and demonstrate functionality.

Item pass/fail criteria: The pass fail criteria are well defined in the test cases individually. If the video game performs as expected it will be a pass. If the feature was not implemented or if it does not perform as expected that will be a failed test case.

Environmental needs: The infrastructure required to run these test cases is a machine that can run the software adequately, in the case for this testing it will be the machine used previously for performance testing (which is one of the development team's laptops).

Responsibilities: The responsibilities of the team will be to develop test cases that are matched with the outlined client's requirements. The team will work together to determine test cases that are in-depth enough to verify that the game was developed as the client intended and help showcase the functionalities that have been implemented.

Staffing and training needs: The team will need to be familiar with acceptance testing and running test cases. For the client the game and UI is designed with the user experience in mind so it should be noted that it is unnecessary for the client to be trained to play this game or utilize this software.

Schedule: The test schedule for acceptance testing is to develop test cases between November 16 and November 27. Then on November 29 during the final team/client meeting the actual acceptance testing will take place with the client. After the 27th, final documentation will take place and results will be recorded and finalized.

Risks and Mitigation: Potential risks include missing requirements outlined by the client, which would cause insufficient coverage by the testing. Another risk would be technical difficulties of the game not launching or functioning as intended, such as encountering new bugs or errors. The lessen the chances of such risks occurring proper planning and test case generation is essential. Communication between the team in determining test cases and making sure the game is ready for presentation to the client is essential. Minimizing the number of failed test cases is a high priority so the client will be pleased with the product that is being delivered.

Approvals: November 29, 2023 Dr. Galloway, Matt Irizarry, Zach Vance, Kiemon Bush, Jeremiah Campbell

4.5.1 Acceptance Tests and Results

Acceptance Test Cases	
Player Control:	
Verify that the character responds accurately to keyboard inputs	Pass
- Left arrow key move left	Pass
- Right arrow key move right	Pass
- Space jumps	Pass
- Left click is shoot	Pass
- Ctrl is crouch	Pass
Verify that Luke's health increments when colliding with health orb	Pass
Health orbs and other items disappear upon collision	Pass
Verify that Luke loses health when	Pass
- Colliding with enemies	Pass
- Colliding with environmental hazards (spikes)	Pass
- Being shot by enemies	Pass
Luke can kill enemies by shooting them	Pass
Multiple Player Accounts:	
Confirm that the project allows multiple players to create separate accounts	Pass
Ensure that each player account is distinct and can be accessed independently	Pass
Login System:	
Validate the implementation of a login system for regular users	Pass
State/Progress Saving:	
Confirm that the project saves the state and progress of players	Fail
Going to be a failed test case	Fail
Verify that players can continue from where they left off upon logging into their accounts	Pass
Splash Screen:	
Check that a splash screen is displayed upon launching the project	Pass
Ensure the splash screen includes a login prompt	Pass
Validate that users can interact with the login prompt	Pass
Pause Screen:	
Test Case 1: Confirm that pressing the 'Escape' key results in the display of a pause screen	Pass
Test Case 2: Verify that the pause screen allows users to resume or exit the game	Pass
Score Board:	
Check that a scoreboard is displayed when a player completes a level	Pass
Ensure the scoreboard shows the player's score accurately	Pass
Validate that the scoreboard allows the user to exit the level	Pass
Game Over Screen:	
Confirm that a game over screen is displayed when a player fails to complete a level	Pass
Validate that the game over screen shows the player's score	Pass
Validate that the game over screen provides allows the user to exit the level	Pass

5 Conclusion

In conclusion this document will allow for our client as well as all group members to see an overall view of the creation of our product. As they will be able to see a roadmap of what we were planning to accomplish with the overview. As well as the ability to see the functional and non-functional requirements we specified. The document will also allow people to view what was put into the product as it was being created and some of the difficulties the team faced as we were working on said product. This will show any viewer of the designs that were used in our product with the use of looking at the UML diagrams. This document will show the overall quality of the final product as the client or others will be able to view the tests we had put onto our product and see the failures and passes of said final product, and the use of the section that goes in depth of the product's performance. For our shortcomings we were sadly unable to get a save function added into the game that would allow the player to save their game and then continue from where they left off from later. as well as a leaderboard for the ability to see the list of high scores and allow players to try and get their name or initials into the top spots. For future works we would like to implement a save function since we were not able to incorporate it in this sprint as well as a leaderboard. Then we would implement more enemies for the player to fight against. As well as a weapon upgrade allowing the player to use a lightsaber. We would like to add in more levels for the game and even give it a possible two player function.

6 Appendix

6.1 Software Product Build Instructions

For continuation of the project, there is a repository made that includes all documentation, assets, and source code. Once the repository has been cloned you can open up the Unity Hub and add a project from disk and go to the github folder and click on the cloned project. Unity Hub should install the correct version of Unity for you and open the project.

6.2 Software Product User Guide

The user experience has been streamlined as the user will have the build product of the game/software already ready for them. Once they open the build with the correct version of Unity, they will be prompted to create a user account, login, and then be able to enjoy the remake of Star Wars SMS edition. Additionally, the build should work on any platform that can run Unity and no additional settings have to be changed in order to do so.

6.3 Source Code with Comments

Listing 1: BlasterUpgradeCollectable

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

public class BlasterUpgradeCollectable : MonoBehaviour
{
    public int fireRate = 10;
    public bool isCollected = false;

    public float GetFireRate() {
        return fireRate;
    }
}
```

Listing 2: CollectableDecorator

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


public abstract class CollectableDecorator : MonoBehaviour
{
    //all can be changed to Player class when that gets implemented
    public abstract void Collect(Entity player);
    /* Needs to be added to player/entity class
    void OnTriggerEnter2D(Collider2D other)
    {
        CollectableDecorator collectable = other.GetComponent<CollectableDecorator>();
        if (collectable != null)
        {
            collectable.Collect(this);
            Destroy(other.gameObject);
        }
    }
    */
}
```

Listing 3: LargeHealthOrb

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

public class ExtraLifeCollectable : MonoBehaviour
{
    public int numLives;

    public int GetLifeAmount(){
        return numLives;
    }
}
```

Listing 4: HealthOrb

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

public class HealthCollectable : MonoBehaviour
{
    public int healthAmount = 30;


    public int GetHealAmount() {
        return healthAmount;
```

```
    }
}
```

Listing 5: SmallHealthOrb

```
public class SmallHealthCollectable : CollectableDecorator
{
    public int healthAmount = 10;

    public override void Collect(Entity player)
    {
        //player.IncreaseHealth(healthAmount);
    }
}
```

Listing 6: BulletScript

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

public class BulletScript : MonoBehaviour
{
    public float lifetime;

    public float speed;

    public bool isRight;

    // Start is called before the first frame update
    void Start()
    {
        StartCoroutine(LifeTimeCheck(lifetime));
    }

    // Update is called once per frame
    void Update()
    {
        if (isRight)
        {
            transform.position += transform.right * Time.deltaTime * speed;
        }
        else {
            transform.position += -transform.right * Time.deltaTime * speed;
        }
    }

    void Despawn() {
        Destroy(this.gameObject);
    }

    IEnumerator LifeTimeCheck(float time) {
```

```
        yield return new WaitForSeconds(time);
        Despawn();
    }

    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.transform.tag == "EnemyCollider") {
            Despawn();
        }
    }

}
```

Listing 7: CameraFollowScript

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

public class CameraFollowScript : MonoBehaviour
{
    public Transform playerTransform;
    private Vector3 cameraOffset;

    // Boundary coordinates
    private float minX = -7f;
    private float maxX = 184f;
    private float minY = -5f;
    private float maxY = 24f;

    private Camera cam; // Camera reference

    void Start()
    {
        cameraOffset = transform.position - playerTransform.position;
        cam = GetComponent<Camera>();
    }

    void LateUpdate()
    {
        Vector3 newPosition = new Vector3(
            playerTransform.position.x + cameraOffset.x,
            transform.position.y, // Keep the y position constant
            transform.position.z);

        // Calculate the visible width based on the camera's viewport size
        float horzExtent = cam.orthographicSize * Screen.width / Screen.height;

        // Adjust min and max values based on the camera's viewport size
        float adjustedMinX = minX + horzExtent;
        float adjustedMaxX = maxX - horzExtent;

        // Clamping the camera's x position
```

```
        newPosition.x = Mathf.Clamp(newPosition.x, adjustedMinX, adjustedMaxX);

        transform.position = newPosition;
    }
}
```

Listing 8: Entity

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

public class Entity : Object
{
    public int _health;



    public void Despawn(GameObject self) {

        Destroy(self);
    }
}
```

Listing 9: EntityDamageHandler

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

public class EntityDamage : MonoBehaviour
{
    [SerializeField]
    int _damage;


    public int GetEntityDamage() {
        return _damage;
    }


}
```

Listing 10: EntityHealthHandler

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

```
public class EntityHealthHandler : MonoBehaviour
{

    public int _health;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        healthcap();
    }

    public void healthcap() {
        if (_health > 100) {
            _health = 100;
        }
    }

    public int GetHealth() {
        return _health;
    }

    public void SetHealth(int val)
    {
        _health = val;
    }
}
```

Listing 11: DebugToFile

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

public class WriteDebugToFile : MonoBehaviour
{
    string filename = "-";
    private void Awake()
    {

    }
    private void OnEnable()
    {
        Application.logMessageReceived += Log;
    }
```

```
    private void OnDisable()
    {
        Application.logMessageReceived -= Log;
    }

    private void Start()
    {
        filename = Application.dataPath + "/LogFile.text";
    }

    public void Log(string logString,string stackTrace, LogType type) {
        TextWriter tw = new StreamWriter(filename,true);

        tw.WriteLine("[" + DateTime.Now.ToString("hh.mm.ss.ffffff") + "]" + logString);

        tw.Close();
    }
}
```

Listing 12: EnemyCollision

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

public class EnemyCollider : MonoBehaviour
{

    public EntityHealthHandler healthHandler;

    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "PlayerDamage")
        {
            Debug.Log("Detected Player Damage");
            var dmgInst = collision.gameObject.GetComponent<EntityDamage>();

            OnHit(dmgInst.GetEntityDamage());


        }


    }
    void OnHit(int dmg)
    {
        healthHandler.SetHealth(healthHandler.GetHealth() - dmg);


    }

}
```

Listing 13: EnemyDeathHandler

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

public class EnemyDeathHandler : MonoBehaviour
{
    public EntityHealthHandler healthHandler;

    private void Update()
    {
        if (healthHandler.GetHealth() <= 0) {
            Destroy(this.gameObject);

            // increment the enemy score in gamestate.cs

            GameManager.Instance.gameData.EnemyDefeated();

        }
    }
}
```

Listing 14: EnemyBehavior

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

public class EnemyBehavior : MonoBehaviour
{

    public float speed;

    public float movementTime;

    public float maxTime;

    public float horizontal;

    public bool goingRight;

    public bool shooting;

    public AnimationClip shootingAni;

    public AnimationClip walkingAni;

    public Animator anim;

    public GameObject bulletprefab;
```

```
public GameObject gunPosition;

public AudioSource blastersound;
// Update is called once per frame
void Update()
{
    // Moves the enemy to left or right
    if(shooting == false){

        anim.Play(walkingAni.name);

        if (goingRight)
        {
            transform.position += transform.right * Time.deltaTime * speed;
        }
        else {
            transform.position += -transform.right * Time.deltaTime * speed;
        }

        //tracks the time the enemy has been moving in a certain direction
        movementTime += Time.deltaTime;

    }

    // Will change the direction of where the enemy will move
    if(movementTime >= maxTime){

        if(shooting == false)
        {
             if(goingRight == true)
            {
                StartCoroutine(shootAnim(false));
            }
            else
            {
                StartCoroutine(shootAnim(true));
            }
        }
    }

}

private void Flip()
{

    // Multiply the player's x local scale by -1.
    Vector3 theScale = transform.localScale;
    theScale.x *= -1;
    transform.localScale = theScale;
}

private IEnumerator shootAnim(bool status)
{
```

```
        anim.Play(shootingAni.name);
        shooting = true;
        //shooting logic goes here

        yield return new WaitForSeconds(shootingAni.length);

        shooting = false;

        goingRight = status;
        movementTime = 0;
        Flip();
    }

    //method called in animation events and spawns bullet
    public void shoot()
    {
        blastersound.Play();
        if (transform.localScale.x < 0) {
            var clone = Instantiate(bulletprefab, gunPosition.transform.position, transfo
            clone.GetComponent<BulletScript>().isRight = false;
        }

        if (transform.localScale.x > 0)
        {
            var clone = Instantiate(bulletprefab, gunPosition.transform.position, transfo
            clone.GetComponent<BulletScript>().isRight = true;
        }

    }

}
```

Listing 15: CharacterController

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

public class CharacterController : MonoBehaviour
{
    [SerializeField] private float m_JumpForce = 1000f;                          // Amou
    [Range(0, 1)][SerializeField] private float m_CrouchSpeed = .36f;            // Amoun
    [Range(0, .3f)][SerializeField] private float m_MovementSmoothing = .05f;    // How m
    [SerializeField] private bool m_AirControl = false;                          // Wheth
    [SerializeField] private LayerMask m_WhatIsGround;                           // A mas
    [SerializeField] private Transform m_GroundCheck;                            // A pos
    [SerializeField] private Transform m_CeilingCheck;                           // A pos
    [SerializeField] private Collider2D m_CrouchDisableCollider;                 // A col

    const float k_GroundedRadius = .2f; // Radius of the overlap circle to determine if
    private bool m_Grounded;            // Whether or not the player is grounded.
    const float k_CeilingRadius = .2f; // Radius of the overlap circle to determine if t
```

```
    private Rigidbody2D m_Rigidbody2D;
    private bool m_FacingRight = true;  // For determining which way the player is curre
    private Vector3 m_Velocity = Vector3.zero;

    [Header("Events")]
    [Space]

    public UnityEvent OnLandEvent;

    [System.Serializable]
    public class BoolEvent : UnityEvent<bool> { }

    public BoolEvent OnCrouchEvent;
    private bool m_wasCrouching = false;

    private void Awake()
    {
        m_Rigidbody2D = GetComponent<Rigidbody2D>();

        if (OnLandEvent == null)
            OnLandEvent = new UnityEvent();

        if (OnCrouchEvent == null)
            OnCrouchEvent = new BoolEvent();
    }

    private void FixedUpdate()
    {
        bool wasGrounded = m_Grounded;
        m_Grounded = false;

        // The player is grounded if a circlecast to the groundcheck position hits anyth
        // This can be done using layers instead but Sample Assets will not overwrite yo
        Collider2D[] colliders = Physics2D.OverlapCircleAll(m_GroundCheck.position, k_Gr
        for (int i = 0; i < colliders.Length; i++)
        {
            if (colliders[i].gameObject != gameObject)
            {
                m_Grounded = true;
                if (!wasGrounded)
                    OnLandEvent.Invoke();
            }
        }
    }


    public void Move(float move, bool crouch, bool jump)
    {
        // If crouching, check to see if the character can stand up
        if (!crouch)
        {
            // If the character has a ceiling preventing them from standing up, keep ther
            if (Physics2D.OverlapCircle(m_CeilingCheck.position, k_CeilingRadius, m_Wha
```

```
        {
            crouch = true;
        }
    }

    //only control the player if grounded or airControl is turned on
    if (m_Grounded || m_AirControl)
    {

        // If crouching
        if (crouch)
        {
            if (!m_wasCrouching)
            {
                m_wasCrouching = true;
                OnCrouchEvent.Invoke(true);
            }

            // Reduce the speed by the crouchSpeed multiplier
            move *= m_CrouchSpeed;

            // Disable one of the colliders when crouching
            if (m_CrouchDisableCollider != null)
                m_CrouchDisableCollider.enabled = false;
        }
        else
        {
            // Enable the collider when not crouching
            if (m_CrouchDisableCollider != null)
                m_CrouchDisableCollider.enabled = true;

            if (m_wasCrouching)
            {
                m_wasCrouching = false;
                OnCrouchEvent.Invoke(false);
            }
        }

        // Move the character by finding the target velocity
        Vector3 targetVelocity = new Vector2(move * 10f, m_Rigidbody2D.velocity.y);
        // And then smoothing it out and applying it to the character
        m_Rigidbody2D.velocity = Vector3.SmoothDamp(m_Rigidbody2D.velocity, targetVe

        // If the input is moving the player right and the player is facing left...
        if (move > 0 && !m_FacingRight)
        {
            // ... flip the player.
            Flip();
        }
        // Otherwise if the input is moving the player left and the player is facing
        else if (move < 0 && m_FacingRight)
        {
            // ... flip the player.
```

```
                Flip();
            }
        }
        // If the player should jump...
        if (m_Grounded && jump)
        {
            // Add a vertical force to the player.
            m_Grounded = false;
            m_Rigidbody2D.AddForce(new Vector2(0f, m_JumpForce));
        }
    }


    private void Flip()
    {
        // Switch the way the player is labelled as facing.
        m_FacingRight = !m_FacingRight;

        // Multiply the player's x local scale by -1.
        Vector3 theScale = transform.localScale;
        theScale.x *= -1;
        transform.localScale = theScale;
    }

    public bool getGrounded() {
        return m_Grounded;
    }
}
```

Listing 16: AnimationManager

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

public class PlayerAnimation : MonoBehaviour
{
    public Animator animator;

    public CharacterController controller;

    public PlayerMovement moveScript;

    public AnimationClip walkClip;
    public AnimationClip idleClip;
    public AnimationClip jumpClip;
    public AnimationClip crouchClip;

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

    }
```

```
    // Update is called once per frame
    void Update()
    {
        if (Input.GetAxisRaw("Horizontal") != 0 && controller.getGrounded() && !moveScri
        {
            animator.Play(walkClip.name);
        }

        if (!controller.getGrounded() && !moveScript.crouch) {
            animator.Play(jumpClip.name);
        }

        if (Input.GetAxisRaw("Horizontal") == 0 && controller.getGrounded() && !moveScri
            animator.Play(idleClip.name);
        }

        if (moveScript.crouch) {
            animator.Play(crouchClip.name);
        }
    }
}
```

Listing 17: CollisionHandler

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

public class PlayerCollision : MonoBehaviour
{
    public SpriteRenderer spriterender;

    public Rigidbody2D rb;

    public float power = 1000f;

    public bool invuln;

    public PlayerShoot shoothandler;

    public EntityHealthHandler healthHandler;

    public PlayerDeathHandler deathhandler;

    // Start is called before the first frame update
    void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.tag == "EnemyDamage" || collision.gameObject.tag == "Tr
            var dmgInst = collision.gameObject.GetComponent<EntityDamage>();

                OnHit(dmgInst.GetEntityDamage());
```

```
    }


}

void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.tag == "PickUp")
    {
        if (collision.gameObject.GetComponent<HealthCollectable>()) {
            var healInst = collision.gameObject.GetComponent<HealthCollectable>();
            healthHandler.SetHealth(healthHandler.GetHealth() + healInst.GetHealAmou

            // if this is a health collectible, we need to increment the GameState s

            // we can do this by calling the GameManager's IncrementScore method

            GameManager.Instance.gameData.CollectableCollected();
        }

        if (collision.gameObject.GetComponent<ExtraLifeCollectable>())
        {
            var lifeInst = collision.gameObject.GetComponent<ExtraLifeCollectable>()
            deathhandler.SetLife(deathhandler.GetLife() + lifeInst.GetLifeAmount());

            GameManager.Instance.gameData.CollectableCollected();
        }

        if (collision.gameObject.GetComponent<BlasterUpgradeCollectable>())
        {
            var ratelifeInst = collision.gameObject.GetComponent<BlasterUpgradeColle
            shoothandler.SetRate(shoothandler.GetRate() - ratelifeInst.GetFireRate()

            GameManager.Instance.gameData.CollectableCollected();
        }

        Destroy(collision.gameObject);
    }

    if (collision.gameObject.tag == "EnemyDamage")
    {
        var dmgInst = collision.gameObject.GetComponent<EntityDamage>();

        OnHit(dmgInst.GetEntityDamage());
    }
}

void OnTriggerStay2D(Collider2D collision) {
    if (collision.gameObject.tag == "Trap")
    {
        var dmgInst = collision.gameObject.GetComponent<EntityDamage>();
```

```
            OnHit(dmgInst.GetEntityDamage());


        }
    }

    void OnHit(int dmg) {
        if (!invuln) {
            healthHandler.SetHealth(healthHandler.GetHealth() - dmg);
            rb.AddForce(Vector2.left * power * 2f);
            rb.AddForce(Vector2.up * power * .25f);
            StartCoroutine(Flicker());
        }
    }

    IEnumerator Flicker() {
        if ( healthHandler.GetHealth() > 0) {
            spriterender.enabled = true;
            invuln = true;
            yield return new WaitForSeconds(.15f);
            spriterender.enabled = false;
            yield return new WaitForSeconds(.15f);
            spriterender.enabled = true;
            yield return new WaitForSeconds(.15f);
            spriterender.enabled = false;
            yield return new WaitForSeconds(.15f);
            spriterender.enabled = true;
            yield return new WaitForSeconds(.15f);
            spriterender.enabled = false;
            yield return new WaitForSeconds(.15f);
            spriterender.enabled = true;
            invuln = false;
        }
    }
}
```

Listing 18: PlayerDamageHandler

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

public class PlayerDamage : MonoBehaviour
{
    public Rigidbody2D rb;

    public float power = 1000f;

    public bool invuln;
    // Start is called before the first frame update
    void OnCollisionEnter2D(Collision2D collision)
    {
```

```
        if (collision.gameObject.tag == "EnemyDamage") {
            OnHit();
        }
    }

    void OnHit() {
        rb.AddForce(Vector2.left * power * 4f);
        rb.AddForce(Vector2.up * power * 1.25f);
    }
}
```

Listing 19: DeathHandler

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

public class PlayerDeathHandler : MonoBehaviour
{
    public int playerLives;

    public GameObject playerSpawner;

    public EntityHealthHandler healthHandler;

    public SpriteRenderer spriterender;

    public Rigidbody2D rigidbody;

    public bool isDead;
    // Start is called before the first frame update
    void Start()
    {
        playerLives = 0;
        isDead = false;
        playerSpawner = GameObject.FindGameObjectWithTag("PlayerSpawn");
    }

    // Update is called once per frame
    void Update()
    {
        StartCoroutine(ExecuteDeath());
    }

    IEnumerator ExecuteDeath() {
        if (healthHandler.GetHealth() < -1 && !isDead)
        {

            if (playerLives - 1 < 0) {
                GameManager.Instance.GameOver();
            }
```

```
            playerLives = playerLives - 1;
            rigidbody.constraints = RigidbodyConstraints2D.FreezePosition;
            spriterender.enabled = false;
            isDead = true;
            yield return new WaitForSeconds(1f);
            transform.position = playerSpawner.transform.position;
            yield return new WaitForSeconds(1f);
            if (playerLives >= 0) {
                rigidbody.constraints = RigidbodyConstraints2D.None;
                spriterender.enabled = true;
                healthHandler.SetHealth(100);
                isDead = false;
            }
        }
    }

    public int GetLife() {
        return playerLives;
    }

    public void SetLife(int val)
    {
        playerLives = val;
    }
}
```

Listing 20: PlayerMovement

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
/// <summary>
/// This script controls player movement
/// </summary>
public class PlayerMovement : MonoBehaviour
{
    public CharacterController contoller;

    public float speed;

    public float horizontal;

    public bool jump;

    public bool crouch;

    public Rigidbody2D rigidbody;

    private void Start()
    {

    }
```

```
    void Update()
    {

        rigidbody.freezeRotation = true;


        horizontal = Input.GetAxisRaw("Horizontal") * speed;



        if (Input.GetKeyDown(KeyCode.Space) && !crouch) {

            jump = true;
        }

        if (Input.GetKeyDown(KeyCode.LeftControl) && !crouch)
        {
            Debug.Log("crouch");
            crouch = true;
        }
        else if(Input.GetKeyUp(KeyCode.LeftControl))
        {
            Debug.Log("Exit crouch");
            crouch = false;
        }
    }

    void FixedUpdate()
    {
        contoller.Move(horizontal * Time.fixedDeltaTime,crouch, jump);
        jump = false;
    }

    public void setSpeed(int val) {
        speed = val;
    }

}
```

Listing 21: PlayerShoot

```
using System.Collections;
using System.Collections.Generic;
using UnityEditor;
using UnityEngine;
using static UnityEngine.UI.Image;
/// <summary>
/// This script handels shooting controls and makes bullet projectlile
/// </summary>
public class PlayerShoot : MonoBehaviour
{
```

```
public float cooldown;

public bool canShoot;

public GameObject bulletprefab;

public AudioSource blastersound;

public GameObject gunPosition;
// Update is called once per frame
private void Start()
{
    canShoot = true;
}
void Update()
{
    if (Input.GetKeyDown(KeyCode.Mouse0) && canShoot) {
        Debug.Log("Start shoot command");
        ShootBullet();
        blastersound.Play();
        StartCoroutine(StartCoolDown());
    } else if (Input.GetKeyUp(KeyCode.Mouse0) && !canShoot) {
        Debug.Log("Exit shoot command");
    }
}

void ShootBullet() {
    if (transform.localScale.x < 0) {
        var clone = Instantiate(bulletprefab, gunPosition.transform.position, transfo
        clone.GetComponent<BulletScript>().isRight = false;
    }

    if (transform.localScale.x > 0)
    {
        var clone = Instantiate(bulletprefab, gunPosition.transform.position, transfo
        clone.GetComponent<BulletScript>().isRight = true;
    }
}

IEnumerator StartCoolDown() {
    canShoot = false;
    yield return new WaitForSeconds(cooldown);
    canShoot = true;
}

public float GetRate() {
    return cooldown;
}

public void SetRate(float val)
{
     cooldown = val;
```

```
    }
}
```

Listing 22: PlayerUIHandler

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

public class PlayerUIHandler : MonoBehaviour
{
    public EntityHealthHandler healthhandler;

    public PlayerDeathHandler deathhandler;

    public Slider healthslider;

    public Text lifecounter;
    private void Start()
    {
        healthslider.maxValue = healthhandler.GetHealth();
    }

    private void Update()
    {
        lifecounter.text = deathhandler.GetLife().ToString();
        healthslider.value = healthhandler.GetHealth();
    }
}
```

Listing 23: BackgroundMusicManager

```
using UnityEngine;

public class BackgroundMusic : MonoBehaviour
{
    private AudioSource audioSource;


    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        audioSource.loop = true;  // Enable looping
        PlayMusic();
    }

    public void PlayMusic()
    {
        audioSource = GetComponent<AudioSource>();
        audioSource.loop = true;  // Enable looping
        if (audioSource != null && !audioSource.isPlaying)
```

```
        {
            audioSource.Play();
        }
    }

    public void StopMusic()
    {
        if (audioSource != null && audioSource.isPlaying)
        {
            audioSource.Stop();
        }
    }

}
```

Listing 24: EnemyFactory

```
// EnemyFactory.cs
using UnityEngine;

public class EnemyFactory : MonoBehaviour
{
    public GameObject enemyType1Prefab;
    public GameObject enemyType2Prefab;


    public GameObject CreateEnemy(string type)
    {
        switch (type)
        {
            case "EnemyType1":
                return Instantiate(enemyType1Prefab);
            case "EnemyType2":
                return Instantiate(enemyType2Prefab);
            default:
                return null;
        }
    }
}
```

Listing 25: GameManager

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

public class GameManager : MonoBehaviour
{
    public int CurrentLevel { get; private set; }
```

```
 public Vector2 PlayerPosition { get; private set; }
 public int PlayerHealth { get; private set; } //maybe float
 public int PlayerLives { get; private set; }
 private Dictionary <string, int> EnemiesDefeated;
 private Dictionary <string, int> CollectablesCollected;
 public GameObject scorecardPanel; // Assign this in the Unity Editor
 public GameObject pauseMenuPanel; // Assign in Unity Editor
 public TextMeshProUGUI scoreText; // Assign this in the Unity Editor
 public TextMeshProUGUI gameOverText;
 public GameObject playerPrefab; // Assign in Unity Editor
 public GameObject gameOverPanel; // Assign in Unity Editor
 private GameObject currentPlayer;
 private EnemyFactory enemyFactory;
 public static GameManager Instance { get; private set; }  // Singleton pattern imple
 public GameState gameData;
 public LevelEndMusic levelCompleteAudio;
 LevelData currentLevelData;
 public GameOverSound gameOverSound;
 public BackgroundMusic bgMusic;
 void Start()
{
     CurrentLevel = 1;
     StartLevel();
     PlayerHealth = 100;  //player.getHealth() or player.setHealth()
     PlayerLives = 1;     //player.getLives() or player.setLives()
     EnemiesDefeated = new Dictionary<string, int>();
     enemyFactory = FindObjectOfType<EnemyFactory>();


}
 void Awake()
 {
     if (Instance == null)
     {
         Instance = this;

         DontDestroyOnLoad(gameObject);

         gameData = new GameState(); // Initialize gameData here
         playerPrefab = GameObject.Find("Luke");
     }
     else if (Instance != this)
     {
         Destroy(gameObject);
         return;
     }
 }

 void Update()
 {
     if (Input.GetKeyDown(KeyCode.Escape))
     {
         TogglePause();
```

```
    }
}

void StartLevel()
{
    currentLevelData = FindObjectOfType<LevelData>(); // Find the LevelData for the
    if (currentLevelData != null)
    {
        SpawnPlayer(currentLevelData.GetSpawnPosition());
        bgMusic.PlayMusic();
        //needs to be replanced by below method
        //SpawnPlayer(currentLevelData.GetPlayerPosition());
        SpawnEnemies(currentLevelData.GetEnemyPositions());
        SpawnItems(currentLevelData.GetItemPositions());

    }
}

void ResumeLevel()
{
    LevelData currentLevelData = FindObjectOfType<LevelData>(); // Find the LevelDat
    if (currentLevelData != null)
    {
        bgMusic.PlayMusic();
        SpawnPlayer(currentLevelData.GetSpawnPosition());
        //needs to be replanced by below method
        //SpawnPlayer(currentLevelData.GetPlayerPosition());
        SpawnEnemies(currentLevelData.GetEnemyPositions());
        SpawnItems(currentLevelData.GetItemPositions());

    }
}

void SpawnEnemies(Dictionary<GameObject, Vector3> enemyPositions)
{
    //Dictonary has already been checked if defeated
    foreach (var entry in enemyPositions)
    {
        Instantiate(entry.Key, entry.Value, Quaternion.identity);
    }
}

void SpawnItems(Dictionary<GameObject, Vector3> itemPositions)
{
    //Dictonary has already been checked if collected
    foreach (var item in itemPositions)
    {
        Instantiate(item.Key, item.Value, Quaternion.identity);
    }
}

// Method to be called when player reaches the end of the level
private void OnTriggerEnter2D(Collider2D other)
```

```
{
    if (other.gameObject.tag == "Player")
    {
        EndLevel();
    }
}

public void AdvanceToNextLevel()
{
    CurrentLevel++;
    //LoadLevel(CurrentLevel);
}

public void LoadLevel(int level)
{
    // Logic to load the specified level scene
    // Example: SceneManager.LoadScene("Level" + level);
}

public void EndLevel()
{
    // Debugging to check for null references
    if (gameData == null)
    {
        Debug.LogError("GameManager: GameData is null.");
        return;
    }
    if (scoreText == null)
    {
        Debug.LogError("GameManager: ScoreText is null.");
        return;
    }

    Debug.Log("GameManager: Ending Level...");
    bgMusic.StopMusic();
    levelCompleteAudio.Play();

    // Update the score text
    scoreText.text = gameData.CalculateScores().ToString();

    scorecardPanel.SetActive(true);
    //AdvanceToNextLevel();
}


public void ResetLevel() {
    CurrentLevel = 1;
    PlayerPosition = new Vector2(0, 0);
    PlayerHealth = 100;
    PlayerLives = 3;
    EnemiesDefeated = new Dictionary<string, int>();
}
```

```
public void QuitGame() {
    SceneManager.LoadScene("MainMenu");
}

public void TogglePause() {
    if (Time.timeScale == 0f)
    {
        // Resume game
        Time.timeScale = 1f;
        pauseMenuPanel.SetActive(false);
    }
    else
    {
        // Pause game
        Time.timeScale = 0f;
        pauseMenuPanel.SetActive(true);
    }
}

public void ResumeGame()
{
    TogglePause();
}

public void SpawnPlayer(Vector3 spawnPos)
{
    if (playerPrefab == null)
    {
        Debug.LogError("Player prefab is not assigned in LevelManager.");
        return;
    }
    if (currentPlayer == null)
    {
        currentPlayer = Instantiate(playerPrefab, spawnPos, Quaternion.identity);
        // Set additional player properties if needed
    }
}

public void KillPlayer()
{
    Destroy(currentPlayer);
    currentPlayer = null;
}

public void RespawnPlayer(Vector3 spawnPos)
{
    if (PlayerLives > 0)
    {
        // Optionally add delay or respawn animation
        currentPlayer.transform.position = spawnPos;
        gameData.UpdatePlayerLives(-1);
```

```
            Debug.Log("Player Respawned");
            // Reset player state as needed
        }
        else
        {
            // Handle game over scenario
            Debug.Log("Game Over");
            GameOver();

        }
    }

    public void GameOver()
    {
        // Optionally add delay or game over animation
        gameOverText.text = gameData.CalculateScores().ToString();
        gameOverPanel.SetActive(true);

        if (gameOverSound != null)
        {
            gameOverSound.PlayGameOverSound();
        }
        else
        {
            Debug.LogError("GameOverSound script is not assigned in GameManager.");
        }
    }

    public void SpawnEnemy(string type, Vector3 position)
    {
        GameObject enemy = enemyFactory.CreateEnemy(type);
        if (enemy != null)
        {
            enemy.transform.position = position;
            // Initialize enemy-specific properties if needed
        }
    }


    public void SaveGame()
    {
        //update player health
        //gameData.PlayerHealth = playerPrefab.getHealth();

        //update player lives
        //gameData.PlayerLives = playerPrefab.getLives();

        //update player pos
        //gameData.PlayerPosition = playerPrefab.getPosition();

        //save player health,lives, pos
        //save collectable list
```

```
        //potential logic to save and load enemy and collectables
        //could also track enemy health if desired, but probably unnecessary for our sco
        //foreach (EnemyInfo enemy in currentLevelData.enemies)
        //{
        //    if (enemy.enemyPrefab.getHealth() <= 0 || enemy.enemyPrefab.isDefeated())
        //    {
        //        enemy.isDefeated = true;
        //    }
        //}

        //foreach (CollectableInfo item in currentLevelData.items)
        //{
        //    if (item.collectablePrefab.isCollected())
        //    {
        //        item.isCollected() = true;
        //    }
        //}

        //save num items collected and num enemies defeated

        //save enemy list
        //save level #

        //saveGameData???

    }

    public void LoadGame()
    {
        //load player health,lives, pos
        //playerPrefab.setHelth();
        //playerPrefab.setPosition();
        //playerPrefab.setLives();


        //load collectable list
        //load enemy list
        //set player health, lives, pos
        //set collectable lsit
        //set enemy lsit
        //load level #

        //load enemy data

        //load item data



    }
```

```
}
```

Listing 26: GameOverSoundScript

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

public class GameOverSound : MonoBehaviour
{
    private AudioSource audioSource;

    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        // Removed the PlayMusic call from Start and loop setting
    }

    public void PlayGameOverSound()
    {
        if (audioSource != null && !audioSource.isPlaying)
        {
            audioSource.Play();
        }
    }

    public void StopMusic()
    {
        if (audioSource != null && audioSource.isPlaying)
        {
            audioSource.Stop();
        }
    }
}
```

Listing 27: GameState

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


public class CollectableInfo
{

    public int collectibleID;
    public GameObject collectablePrefab;
    public Vector3 position;
    public bool isCollected;

    // Constructor
```

```
    public CollectableInfo(int id, GameObject prefab = null, Vector3 pos = default(Vecto
    {
        collectibleID = id;
        collectablePrefab = prefab;
        position = pos;
        isCollected = collected;
    }
}
public class EnemyInfo
{
    int enemyId;
    public bool isDefeated = false;
    public GameObject enemyPrefab;
    public Vector3 position;

    public EnemyInfo(int id, GameObject prefab = null, Vector3 pos = default(Vector3), b
    {
        enemyId = id;
        enemyPrefab = prefab;
        position = pos;
        isDefeated = defeated;
    }

}

public class Scores {
    public int healthScore;
    public int collectableScore;
    public int enemyScore;
    public int levelScore;
    public int totalScore;

    public Scores(int health, int collectable, int enemy, int total) {
        healthScore = health;
        collectableScore = collectable;
        enemyScore = enemy;
        totalScore = total;
    }

    public override string ToString() {
        return "Health Score: " + healthScore + "\nCollectable Score: " + collectableSco
    }
}


public class GameState
{
    public int CurrentLevel { get; set; }
    public int PlayerHealth { get; set; }
    public int PlayerLives { get; set; }
    public Transform PlayerPosition { get; set; }
    private int EnemiesDefeated;
    private int CollectablesCollected;
```

```
    public Scores CalculateScores()
    {
        PlayerHealth = GameManager.Instance.playerPrefab.GetComponent<EntityHealthHandle

        Debug.Log("Player Health: " + PlayerHealth);

        int collectableScore = 10 * CollectablesCollected;
        int enemyScore = 50 * EnemiesDefeated;

        int healthScore;

        if (PlayerHealth < 0) {
            healthScore = 0;
        } else {
            healthScore = PlayerHealth * 10;
        }

        int score = healthScore + collectableScore + enemyScore;

        return new Scores(healthScore, collectableScore, enemyScore, score);
    }



    public void UpdatePlayerLives(int numLives)
    {
        PlayerLives += numLives;
    }

    public void EnemyDefeated()
    {
        //struggling to figure out where this needs to be called from, probably enemy cl
        //not sure how, maybe like below
        //GameManager.Instance.gameData.EnemyDefeated();
        EnemiesDefeated++;
        //enemy.isDefeated = true;
    }

    public void CollectableCollected()
    {
        //struggling to figure out where this needs to be called from, probably on colli
        //not sure how, maybe like below
        //GameManager.Instance.gameData.CollectableCollected();
        CollectablesCollected++;
        //item.isCollected = true;

    }
}
```

Listing 28: LevelData

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


public class LevelData : MonoBehaviour
{
    //public List<EnemyData> enemies;
    //public List<ConsumableData> consumables;
    // Start is called before the first frame update
    public List<CollectableInfo> items = new List<CollectableInfo>();
    public List<EnemyInfo> enemies = new List<EnemyInfo>();
    public Transform playerSpawnPoint;
    public Transform enemy1Pos;
    public Transform enemy2Pos;
    public Transform item1Pos; //smallHealthOrb
    public Transform item2Pos; //healthOrb
    public Transform item3Pos; //extraLife
    public Transform item4Pos; //blasterUpgrade
    //public GameObject stormCloackPreFab;
    //public GameObject otherEnemyPreFab;
    public GameObject smallHealthPreFab;
    public GameObject healthOrbFreFab;
    public GameObject extraLifePreFab;
    public GameObject blasterUpgradePreFab;


    private Dictionary<GameObject, Vector3> enemyPositions = new Dictionary<GameObject,
    private Dictionary<GameObject, Vector3> itemPositions = new Dictionary<GameObject, V




    void Start()
    {
        CollectableInfo smallHealth = new CollectableInfo(1, smallHealthPreFab, item1Pos
        CollectableInfo healthOrb = new CollectableInfo(2, healthOrbFreFab, item2Pos.pos
        CollectableInfo extraLife = new CollectableInfo(3, extraLifePreFab, item3Pos.pos
        CollectableInfo blasterUpgrade = new CollectableInfo(4, blasterUpgradePreFab, ite
        items.Add(smallHealth);
        items.Add(healthOrb);
        items.Add(extraLife);
        items.Add(blasterUpgrade);


        //EnemyInfo stormTrooper1 = new EnemyInfo(1, stormCloackPreFab, enemy1Pos.positio
        //EnemyInfo stormTrooper2 = new EnemyInfo(2, stormCloackPreFab, enemy2Pos.positio
        //enemies.Add(stormTrooper1);
        //enemies.Add(stormTrooper2);

        SetItemAndEnemyPositions();

    }
```

```
    void Awake()
    {
        SetItemAndEnemyPositions();
    }

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

    }
    public void SetItemAndEnemyPositions()
    {
        foreach (var enemy in enemies)
        {
            if (!enemy.isDefeated)
                enemyPositions.Add(enemy.enemyPrefab, enemy.position);
        }

        foreach (var item in items)
        {
            if (!item.isCollected)
                itemPositions.Add(item.collectablePrefab, item.position);
        }
    }
    public Vector3 GetSpawnPosition()
    {
        return playerSpawnPoint.position;
    }

    public Dictionary<GameObject, Vector3> GetEnemyPositions()
    {
        return enemyPositions;
    }

    public Dictionary<GameObject, Vector3> GetItemPositions()
    {
        return itemPositions;
    }

    public Vector3 GetPlayerPostition()
    {
        //needs to be replaced by position getter Method for player!
        //return player.getPos()
        return playerSpawnPoint.position;
    }
}
```

Listing 29: LevelEndPlayMusic

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

```
using UnityEngine;

public class LevelEndMusic : MonoBehaviour
{
    // Start is called before the first frame update
    private AudioSource audioSource;
    void Start()
    {
        audioSource = GetComponent<AudioSource>();
        // Removed the PlayMusic call from Start and loop setting
    }
    public void Play()
    {
        if (audioSource != null && audioSource.clip != null)
        {
            audioSource.Play(); // Plays the assigned AudioClip
        }
        else
        {
            Debug.LogError("AudioSource or AudioClip is missing!");
        }
    }
}
```

Listing 30: LevelEndTrigger

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

public class LevelEndTrigger : MonoBehaviour
{
    void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.tag == "Player")
        {
            if (GameManager.Instance == null)
            {
                Debug.LogError("LevelEndTrigger: GameManager instance is null.");
                return;
            }

            Debug.Log("LevelEndTrigger: Player has triggered the end level.");
            GameManager.Instance.EndLevel();
        }
    }

}
```

Listing 31: LoginManager

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

public class LoginManager : MonoBehaviour
{

    [Header("Firebase")]
    public DependencyStatus dependencyStatus;
    public FirebaseAuth auth;
    public FirebaseUser User;

    [Header("Login UI")]
    public TMP_InputField emailField;
    public TMP_InputField passwordField;
    public TMP_Text warningText;
    public TMP_Text confirmLoginText;

    [Header("Register UI")]
    public TMP_InputField registerEmailField;
    public TMP_InputField registerPasswordField;
    public TMP_Text registerWarningText;

    // Declare next scene
    public string nextScene;

    private void Awake() {
        FirebaseApp.CheckAndFixDependenciesAsync().ContinueWith(task => {
            dependencyStatus = task.Result;
            if (dependencyStatus == DependencyStatus.Available) {
                InitializeFirebase();
            } else {
                Debug.LogError("Could not resolve all Firebase dependencies: " + depende
            }
        });
    }

    private void InitializeFirebase() {
        Debug.Log("Setting up Firebase Auth");
        auth = FirebaseAuth.DefaultInstance;
    }

    public void LoginButton() {
        Debug.Log("Login Button Pressed");
        StartCoroutine(Login(emailField.text, passwordField.text));
    }

    public void RegisterButton() {
```

```
        StartCoroutine(Register(registerEmailField.text, registerPasswordField.text, reg
    }

    private IEnumerator Login(string _email, string _password) {
        var LoginTask = auth.SignInWithEmailAndPasswordAsync(_email, _password);
        yield return new WaitUntil(predicate: () => LoginTask.IsCompleted);

        if (LoginTask.Exception != null) {
            Debug.LogWarning(message: $"Failed to register task with {LoginTask.Exceptio
            FirebaseException firebaseEx = LoginTask.Exception.GetBaseException() as Fir
            AuthError errorCode = (AuthError)firebaseEx.ErrorCode;

            string message = "Login Failed!";
            switch (errorCode) {
                case AuthError.MissingEmail:
                    message = "Missing Email";
                    break;
                case AuthError.MissingPassword:
                    message = "Missing Password";
                    break;
                case AuthError.WrongPassword:
                    message = "Wrong Password";
                    break;
                case AuthError.InvalidEmail:
                    message = "Invalid Email";
                    break;
                case AuthError.UserNotFound:
                    message = "Account does not exist";
                    break;
            }
            warningText.text = message;
        } else {
            User = LoginTask.Result.User;
            Debug.LogFormat("User signed in successfully: {0} ({1})", User.DisplayName,
            warningText.text = "";
            confirmLoginText.text = "Login Successful!";

        // redirect to the game scene

            yield return new WaitForSeconds(1.5f);

            SceneManager.LoadScene("MainMenu");
        }
    }

    private IEnumerator Register(string _email, string _password, string _confirmPasswor
        if (_email == "") {
            registerWarningText.text = "Missing Email";
        } else if (_password != _confirmPassword) {
            registerWarningText.text = "Passwords do not match";
        } else {
            var RegisterTask = auth.CreateUserWithEmailAndPasswordAsync(_email, _passwor
            yield return new WaitUntil(predicate: () => RegisterTask.IsCompleted);
```

```
            if (RegisterTask.Exception != null) {
                Debug.LogWarning(message: $"Failed to register task with {RegisterTask.E
                FirebaseException firebaseEx = RegisterTask.Exception.GetBaseException()
                AuthError errorCode = (AuthError)firebaseEx.ErrorCode;

                string message = "Register Failed!";
                switch (errorCode) {
                    case AuthError.MissingEmail:
                        message = "Missing Email";
                        break;
                    case AuthError.MissingPassword:
                        message = "Missing Password";
                        break;
                    case AuthError.WeakPassword:
                        message = "Weak Password";
                        break;
                    case AuthError.EmailAlreadyInUse:
                        message = "Email already in use";
                        break;
                }
                registerWarningText.text = message;
            } else {
                User = RegisterTask.Result.User;
                if (User != null) {
                    UserProfile profile = new UserProfile { DisplayName = _email };
                    var ProfileTask = User.UpdateUserProfileAsync(profile);
                    yield return new WaitUntil(predicate: () => ProfileTask.IsCompleted)

                    if (ProfileTask.Exception != null) {
                        Debug.LogWarning(message: $"Failed to register task with {Profile
                        FirebaseException firebaseEx = ProfileTask.Exception.GetBaseExce
                        AuthError errorCode = (AuthError)firebaseEx.ErrorCode;
                        registerWarningText.text = "Username Set Failed!";
                    } else {
                        Debug.LogFormat("Username set successfully.");
                        registerWarningText.text = "";
                    }

                Login(registerEmailField.text, registerPasswordField.text);
                }
            }
        }
    }

    public void ExitToDesktop() {
        Application.Quit();
    }
}
```

Listing 32: MainMenuManager

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

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

    }

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

    }

    public void StartNewGame() {
        SceneManager.LoadScene("TutorialLevelScene");
    }
}
```