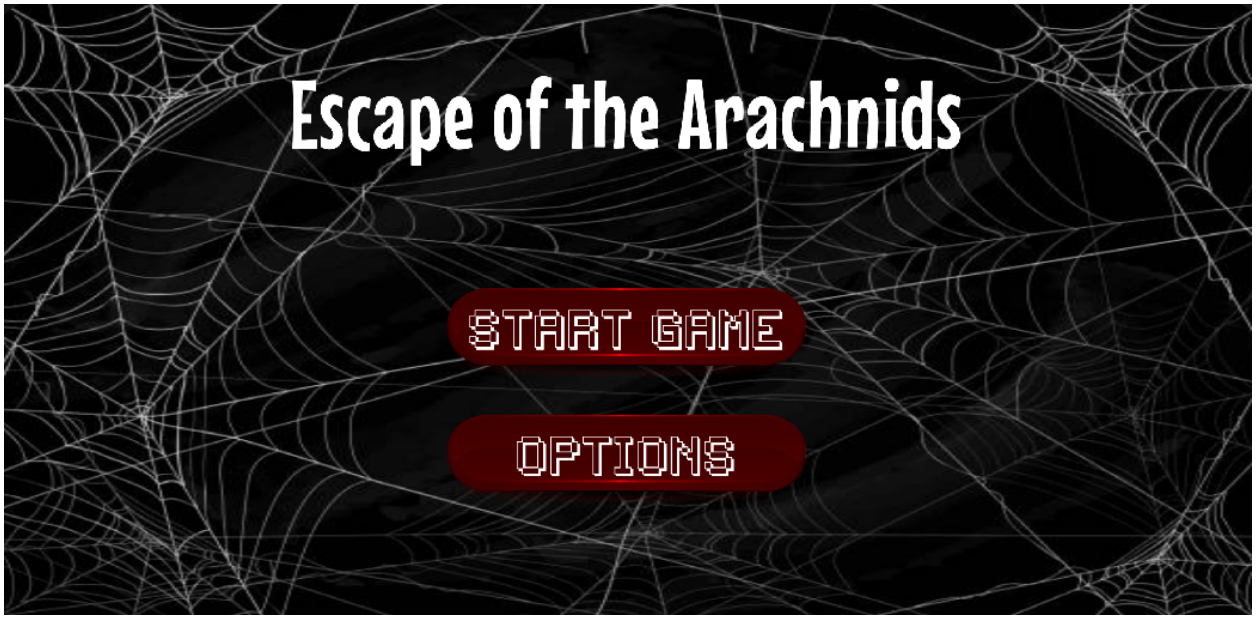
Escape of the Arachnids - UnityEngine 2D Mobile Game

Cole Robinson, Dr. Hugh Smith
Computer Engineering Senior Project 2023
California Polytechnic State University San Luis Obispo
Github: github.com/crobin27/escape-of-the-arachnids



Introduction

Making use of the Unity Game Engine and under the supervision of Dr. Hugh Smith at Cal Poly San Luis Obispo, I was able to further my knowledge of game development techniques through the creation of this mobile 2D Android game. Broadening my knowledge, I focused on including important aspects of the software and game development process such as version control, object-oriented programming, storyboarding, design flow, and user testing. The Unity Engine's cross platform build compatibility combined with its beginner friendly design made it the perfect choice to build the game on.

The primary goals of this project include deepening my understanding of game design principles and familiarizing myself with the Unity Game Engine. Focusing on industry standard practices such as different version control options, mobile input best practices, and professional code style helped to prepare myself for the level of code expected in the workforce. After completing this project, I feel that I have gained the tools necessary to continue creating improved applications.

In this document, I detail the evolution of my project – from background and development environment to design, game mechanics, and implementation. Each aspect of the game, including touch system, player controller, enemy system, UI setup, sound, animation, and camera, is discussed in detail, along with the challenges faced and the solutions implemented. The intention is to provide an in-depth view into the thought process and practical application behind the development of a mobile 2D Android game.

Background And Development Environment

Unity Engine

Through the development process of my 2D game, I gained a comprehensive understanding of both the Unity Editor and the underlying game engine. The Unity Editor served as my primary tool for constructing game scenes, implementing and testing gameplay features, and adjusting game settings. Although an extensive and intimidating tool at first and after struggling with many of the features in the first few weeks of development, I managed to get a hold on the different features of the Unity Editor and Game Engine. Eventually learning keyboard shortcuts and nearly full navigability of the editor, my development process sped up rapidly. I used the Prefabs system to create reusable game objects that could be placed multiple times in the scene or across different scenes, learning how to instantiate and manipulate these at runtime using C#.

Version Control - Github LFS

Throughout the development of the game, I utilized Github to version control the project. As my project grew in complexity and size, being able to revert to previous versions, branch off for experimental features, and merge changes back into the main line was invaluable. In order to

keep track of the larger files in my project such as graphics, audio, and tilesets, I tested out Git LFS(Large File Storage) for the first time. With such large project files, Git LFS made it easy to determine which files should be stored on a remote server in order to minimize space on my remote repository.

C# And OOP

Prior to this project, I had used C# only a few times in a previous Unity project. However, as the development of the mobile 2D Android game progressed, I was able to delve deeper into the nuances of the language and its application in a game development environment. I explored the robust features of C# that offer an upper hand over traditional languages such as C and C++, particularly in terms of memory management, type safety, and advanced runtime capabilities.Considering Unity and C# work together so well in an Object Oriented fashion, I was able to create an efficient arrangement of interfaces and class extensions to encapsulate my objects.

C# was employed as the primary programming language for this project, and the Unity Engine is intrinsically designed to work in harmony with C#. The language's deep integration with the Microsoft .NET framework made it a seamless fit to work in an object-oriented environment within the Unity engine. The project also provided a platform for me to reinforce my knowledge of object-oriented programming (OOP) principles, which are critical for software development, particularly for designing game characters and scenarios.After this project, I plan to look into other applications involving the C# language and the .NET framework.

Code Review Process

Despite working on the project alone, I managed to maintain a relatively healthy code base. The most important aspect of my code review process was the practice of regular commits, facilitating an organized development process. By committing changes often, I ensured that each commit represented a logical and self-contained unit of work. This approach allowed me to track the progression of my project effectively, making it easier to identify and revert changes if something went wrong. I also leveraged the use of AI code review tools for suggestions and potential improvements. To keep the codebase optimized and scalable, refactoring was a part of my regular coding workflow. As my proficiency with Unity and C# increased over the course of the project, I revisited older scripts from earlier development stages and restructured them to align better with the current state of the game. This practice of constant code improvement helped in maintaining the project's performance and reducing technical debt.

Testing

Ensuring the game performs as expected was crucial throughout the development process. For this purpose, I employed a multi-tiered testing approach.

Simulator View Testing: Testing the game in conditions that simulate the final user experience as closely as possible was essential. For this, I used the Simulator View, which allowed me to test the game in a simulated Android environment. This was particularly useful for testing user interface elements and game features that depended on the specific screen size and resolution of the target devices. Below you can see a screenshot of the Simulator view within the Unity Editor.

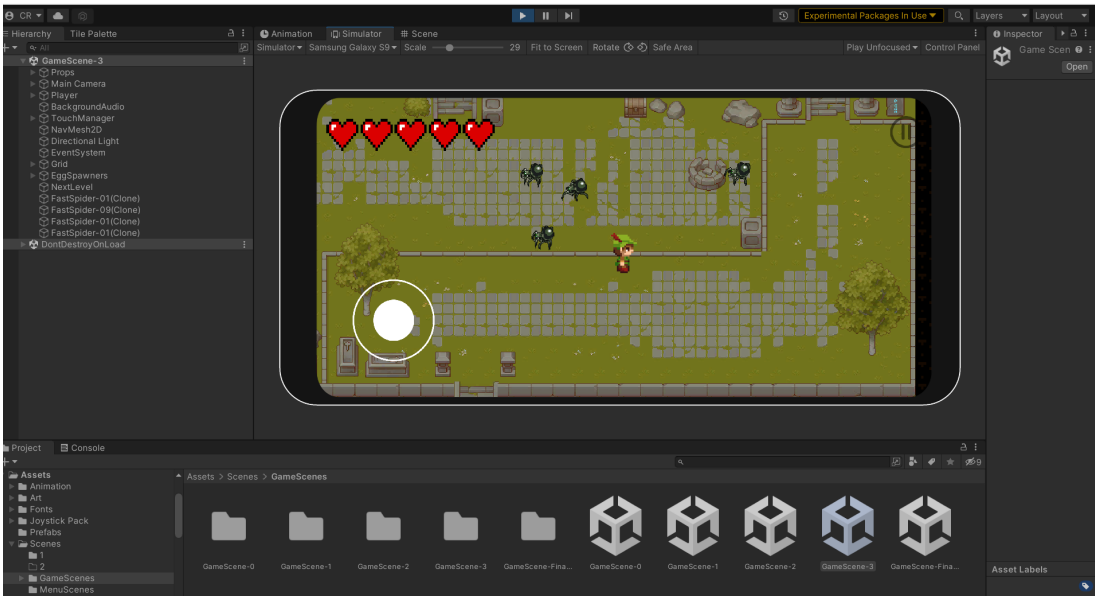


Figure 1: Simulator testing view within the Unity Editor

Unity Remote with Android Device: The Unity Remote app allowed me to run the game on my Android device directly from the Unity Editor. This was critical for testing touch input and how the game felt and responded in real-world conditions. The view is similar to that of the simulator above, but with the added effect of controlling touch on my own device.

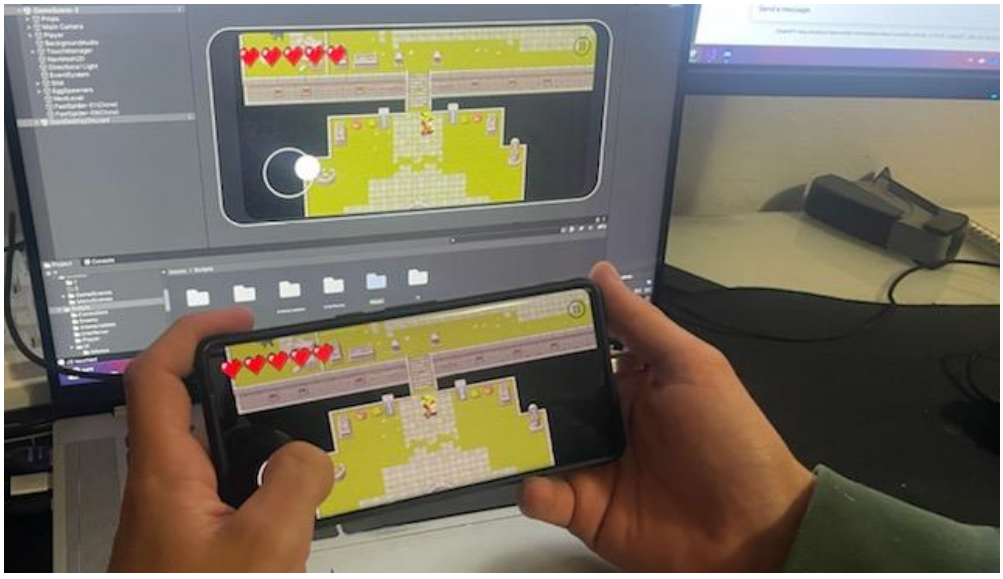


Figure 2: Utilization of the Unity Remote app to test touch input

Build Testing: Unity's build process packages the game into an APK file, which can be installed and run on an Android device just like any other app. This was the final, critical step in the testing process. Given that a build reflects the final product, I also used these testing sessions to review the game's pacing, mechanics, and overall fun factor from the player's perspective. By going through the game in its entirety, I was able to catch and improve aspects of the player experience that could only be evaluated in the context of the full, built game. Considering the build process is lengthy (~10min), this testing phase was used sparingly to test final versions of levels.

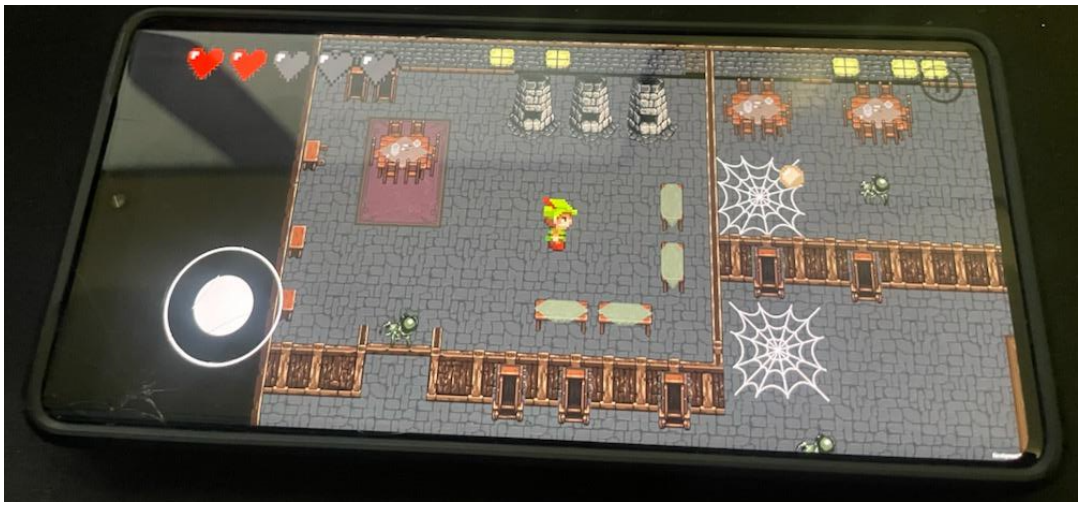


Figure 3: Example of a fully built game on an Android device

User Testing: While it was crucial to test the game in various technical scenarios, it was equally important to gather feedback from real users. This perspective allowed me to understand how players would interact with and perceive the game, which in turn significantly influenced the iterative refinement of the gameplay experience. Although the user testing was informal, consisting primarily of roommates and friends, it provided a fresh and valuable perspective distinct from my own as the developer.These user testers played the game in its various development stages and their feedback led to a range of improvements, from minor tweaks to significant changes in game mechanics. They helped uncover issues with user interface clarity, gameplay bugs, and overall game engagement that I may not have noticed due to my closeness to the project.

Continuous Deployment Options

Unity Cloud Build was not free, and I didn’t feel the need to spend money for this feature. But in industry, it is standard to set up Unity Cloud Build to create a remote build of your Unity game for every push to a given branch. This helps to provide continuous integration and ensure that any recent changes did not have a serious impact on the build of the game. This is also extremely

convenient as the files are built on the cloud, meaning that you don't need to wait for builds to complete locally, which can take up to 10-15 minutes to complete. Nonetheless, learning about this valuable tool helped to reinforce the industry standard features and plugins for development on the Unity Engine.

Design

Storyboard

The storyboarding technique is a common game development practice that was used to illustrate the user flow of the game as a whole. Considering the user throughout the design process was crucial to ensure that the game is intuitive to play for the first time. Game developers often become so detached from the users during the development process, the storyboard serves as a way to ensure the game flow is logical and doesn't have any unintentional complications.

A link to my storyboard can be found here, and some example frames from the storyboard can be seen below..

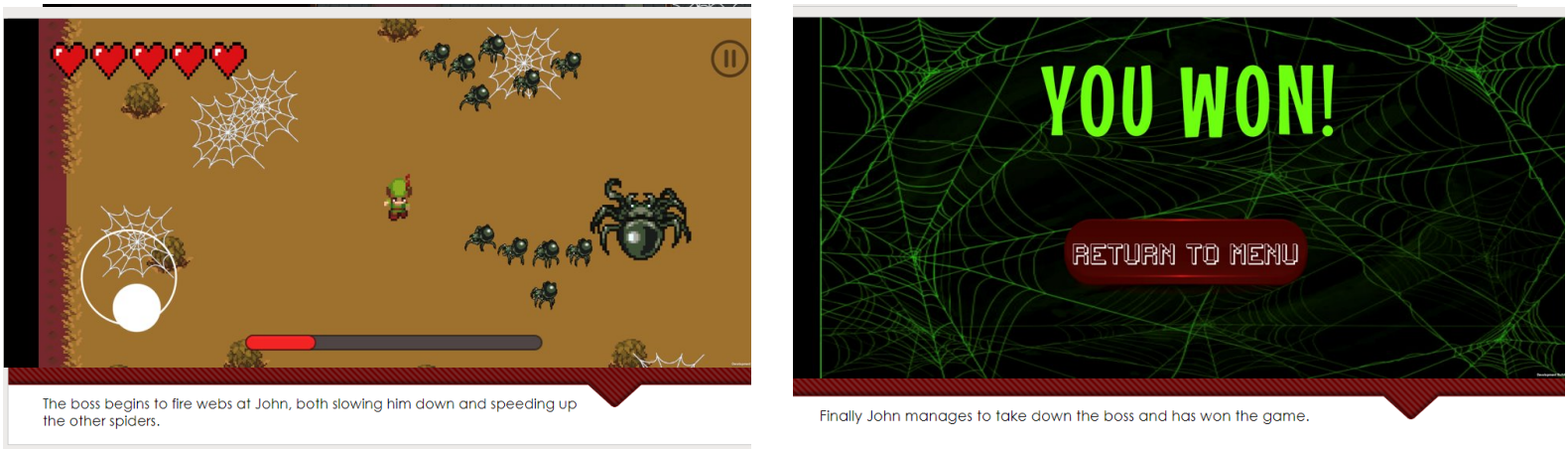


Figure 4: Example scenes from the storyboard

UML Diagram

Throughout the development of the game, a UML diagram proved instrumental in maintaining an organized, modular approach to the project. It served as a visual guide for the structure of my code, making it easier to understand and communicate the relationships and dependencies between different game components. When developing new features or debugging, the diagram provided quick insight into the code architecture. This tool not only facilitated better design decisions but also made refactoring a more efficient process. Below you can see the UML Diagrams for different components of the project and the organization of the codebase as a whole.

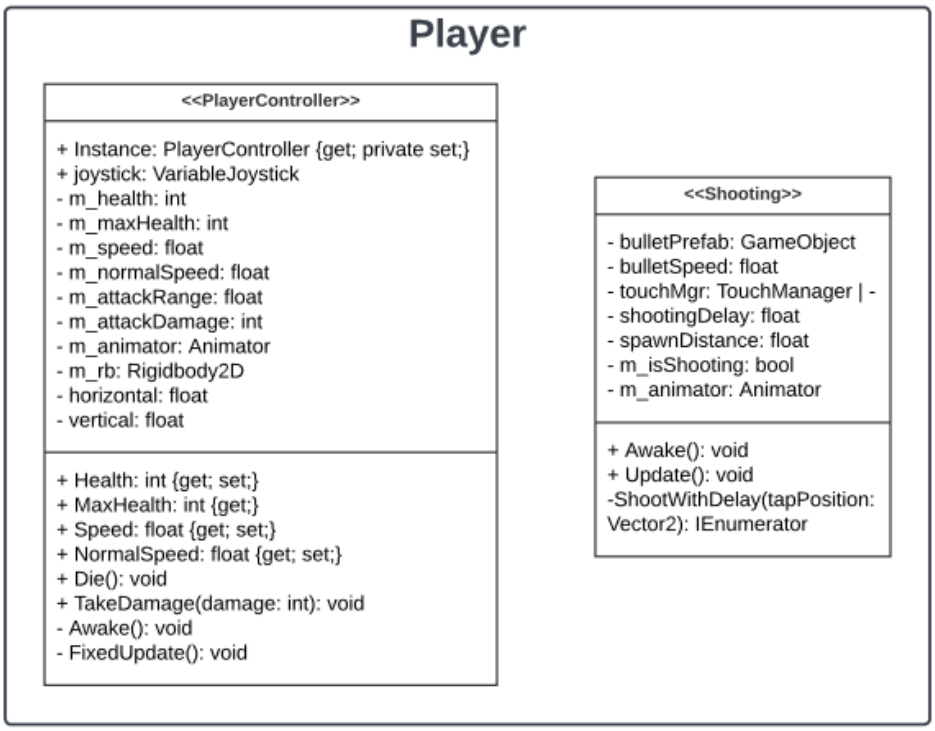


Figure 5: The Player namespace consists of the general PlayerController that works alongside the Shooting script to deal with the user's character

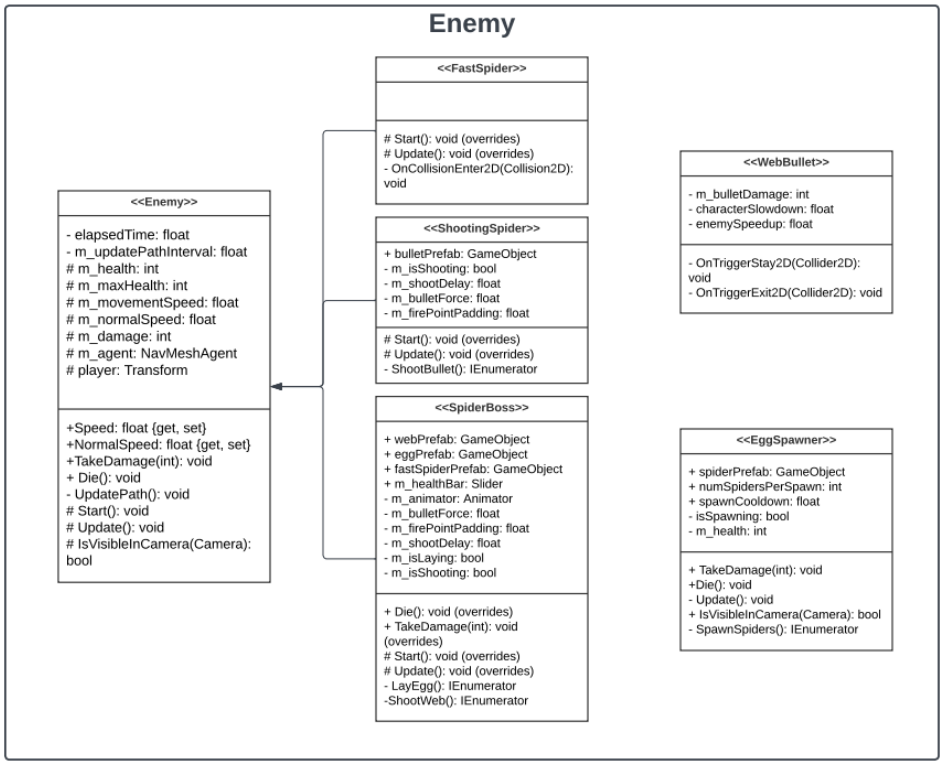


Figure 6: The Enemy namespace deals with all aspects of enemies, including spawning and enemy bullets

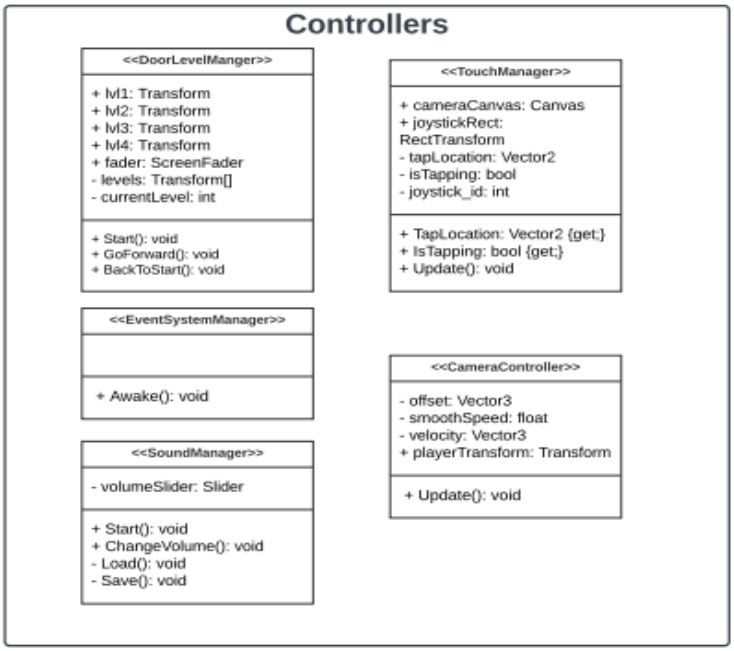


Figure 7: The Controllers namespace consists of various controller components used throughout the game.

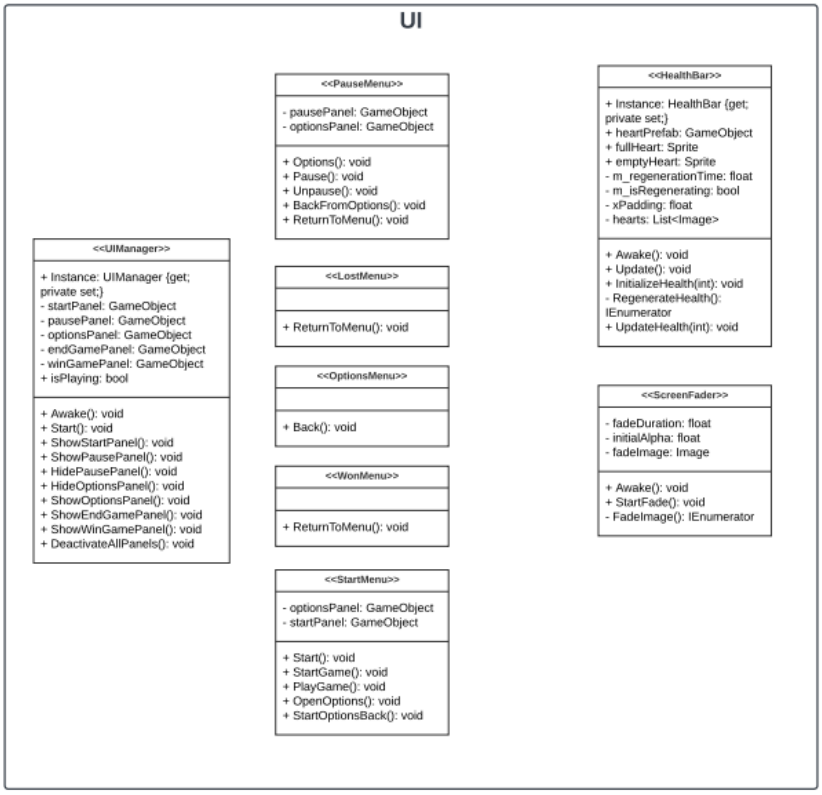


Figure 8: The UI namespace contains all of the helper functions for the UI pages as well as definitions for the HealthBar

Game Flow State Diagram

The flow diagram below illustrates the possible user interactions with the entire game. Although basic, having a drawing was helpful while programming the scene flow into the game. The user begins at the start scene and can either begin the game or visit the options menu to adjust the difficulty and volume level. Upon beginning the game, the user must advance through each level without dying, or else the DeathScene is played and the user returns to the start. Assuming the user wins the game, the WinScene is played and the user can return to the StartScene. Throughout all of the GameScenes, the user can access the PauseMenu as well, which has access to the options menu, the option to return to the StartScene, or return to the current GameScene.

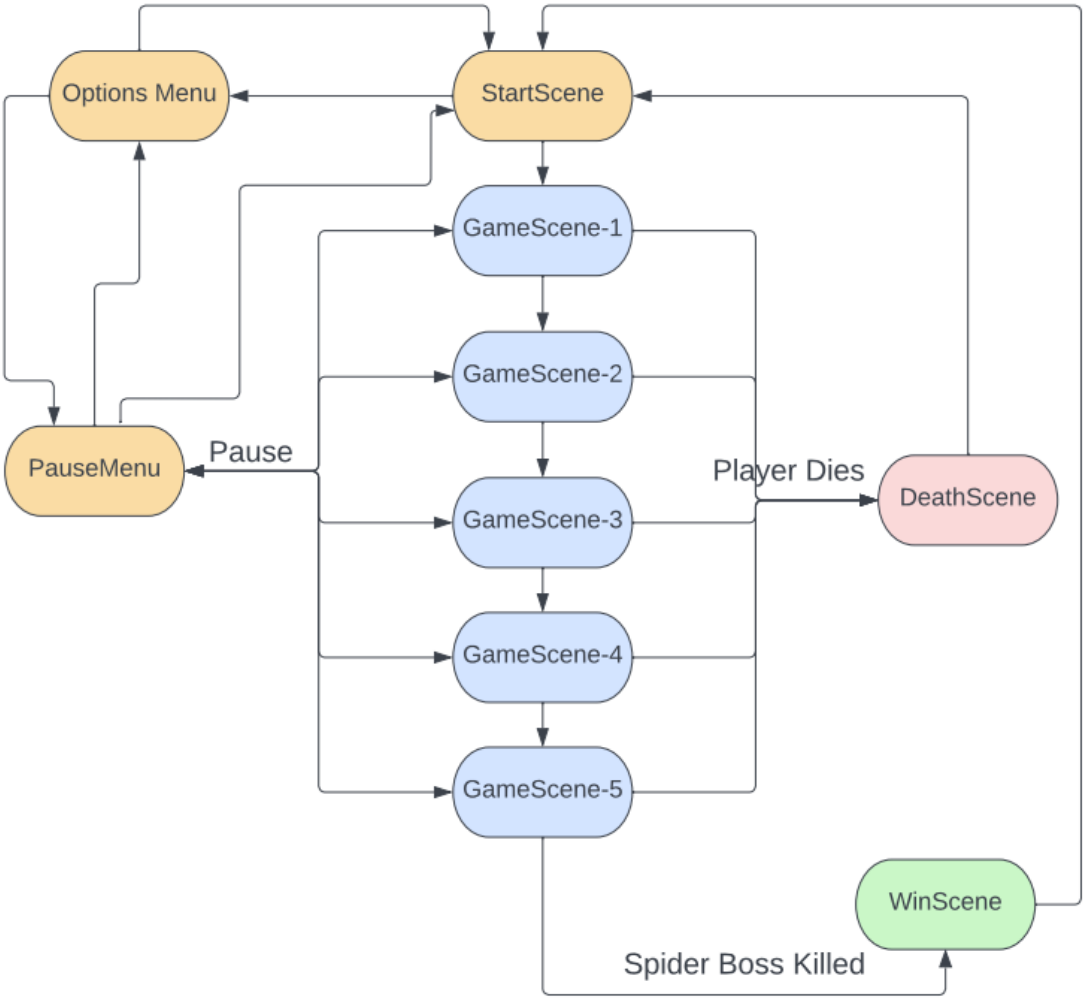


Figure 9: Game Flow State Diagram

Art Design

Rather than creating the game's artwork from scratch—an undertaking that would have required significant time and artistic skill—I chose to use pre-existing assets from the Unity Asset Store. This approach not only saved time but also enhanced the aesthetic appeal and professionalism of

my game, as I was able to incorporate visually impressive assets created by skilled artists. Additionally, I pulled artwork from several different artists to give each level its own feel. In retrospect, it may have been worthwhile to make use of entirely open source art and graphics so that I would have no issues in the publication process. The artwork in this game consisted mainly of 32x32 pixel sprites for both the tilemap and the different characters.

Game Mechanics and Implementation

All source code can be found at the following github repository: link

Touch System

- The touch system is composed of two main parts, the joystick movement and the singular finger tap responsible for shooting.
    - Joystick: I found a third party joystick asset that I integrated into my game.
    - Finger Tap: Although the joystick handled the user movement, I still had to determine which finger was holding the joystick and which finger was tapping the screen. This led to many difficulties but I was able to effectively isolate the joystick finger from the tapping finger and provide mutual exclusion to the system as a whole using Unity's unique finger id implementation.

NavMesh2D

- Rather than creating my own enemy AI scripts, I utilized a third party asset called a NavMesh2D. The NavMesh essentially bakes a map of the 2D tileset and provides A* pathing algorithms to NavMesh agents on the map. In this case, all of my spiders acted as agents on the NavMesh and used the Player as their target. Below is a depiction of how the NavMesh creates a mapping of where enemies are able to walk.

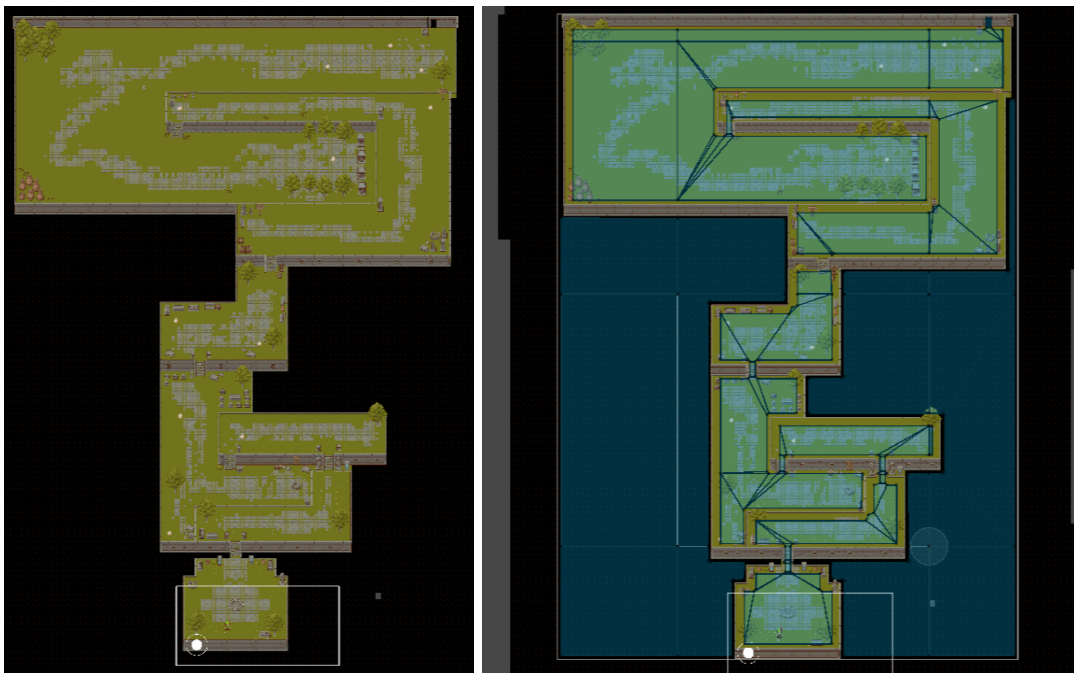


Figure 10: Example of a level with and without the created NavMesh2D

Player Controller

- The player controller is responsible for reading and translating the information from the joystick into player movement.
- This script additionally provides the correct values to the player's animator to display the correct animations
- This component is tightly coupled to the Shooting component that works with the PlayerController to manage the shooting system.

```
if (joystick.Horizontal != 0 && joystick.Vertical != 0)
{
    // limit movement speed diagonally, so you move at 70% speed
    horizontal = moveLimiter * joystick.Horizontal;
    vertical = moveLimiter * joystick.Vertical;

    m_animator.SetFloat("Horizontal", horizontal);
    m_animator.SetFloat("Vertical", vertical);
}
```

Figure 11: Code snippet depicting joystick usage and setting values to the Animator

Enemy System

- Enemy is a base class that has a few simple functions responsible for creating the NavMesh agent, taking damage, and moving the enemy towards the player
- FastSpider: This is the most simple extension of Enemy, the spider simply moves towards the player at a set speed.
- SpiderBoss: The most complex of the enemies, the spider boss has two stages. At first, the boss lays eggs that spawn multiple fast spiders. Once the boss's health is less than half, it begins to fire webs towards the player that slow the user down.

Figure 12: Screenshot from the final level, showing the SpiderBoss as well as the created FastSpiders

- ShootingSpider: This enemy extension is similar to that of the fast spider except that the speed is set much slower and the spider periodically shoots back at the player
- EggSpawner: This object was used in coordination with the SpiderBoss but additionally throughout other parts of the game. This item spawns any given spider with a given periodicity.

UI Setup

- The UI consists of five different panels, each with their own function
- Start Menu: Holds the Start Game button, Options Menu Button and the Title
- Options Menu: This menu allows the user to choose the difficulty level as well as adjust the volume using the slider
- Pause Menu: This menu is brought up in game, and can access the main menu, options menu, or return to the game
- Death Menu: This menu displays the "YOU DIED" message with an option to return to the menu
- Win Menu: This menu displays the "YOU WON" message with an option to return to the menu
- Health Bar and Joystick: The health bar and joystick are both additionally UI elements and can be seen in the bottom right picture of Figure 13.

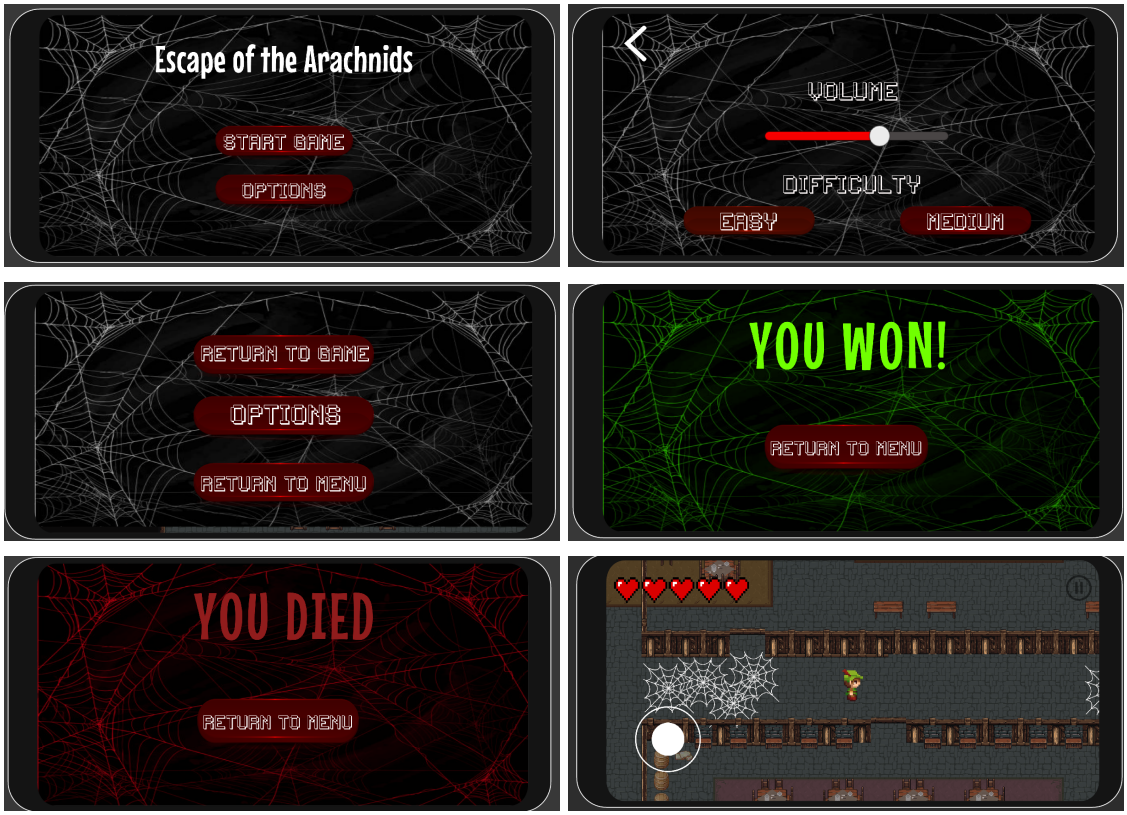


Figure 13: Screenshots of the different UI pages

Sound

- There are two different types of audio sources in the game, one deals with the background audio and the others are individual to the object they are attached to. In other words, each bullet has its own "shooting" noise that is attached to its object and played on awake.

Animation

- Animator: Unity provides a tool called the "Animator" that works as a visual state machine, which helps in determining which animations to play at any given time through code. Below is an example of the Animator state machine for the Player. As you can see, there are many different animation states, and the Animator's map view layout is convenient for understanding transitions between animations. The transitions are triggered within the code of the PlayerController by setting values and triggers to play the correct animation.

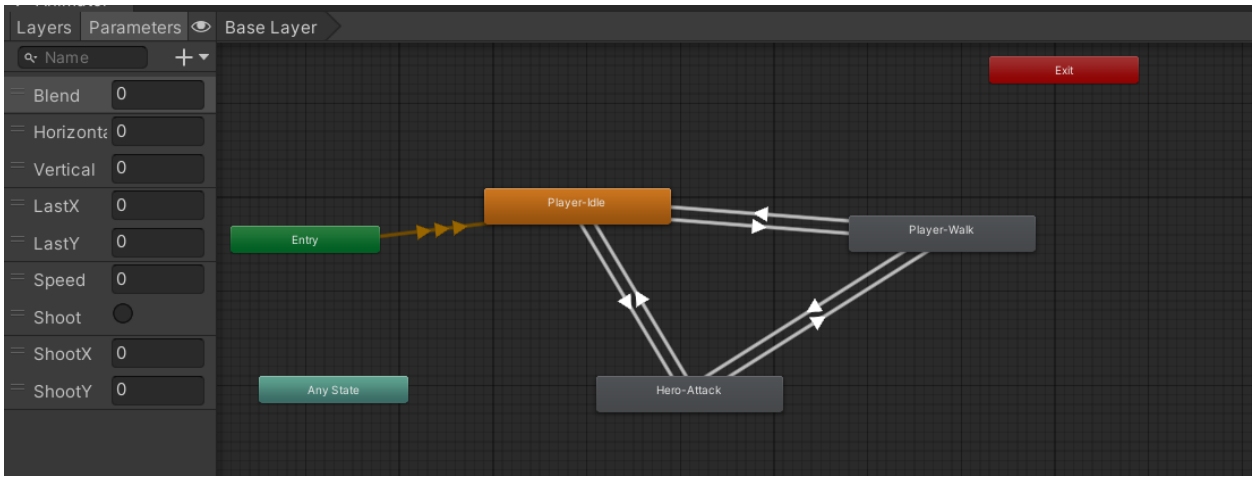


Figure 14: Animator state machine for the Player's animation states

Camera

- The camera setup was rather simple, by fixing the camera's (x,y) location to be that of the player's, the camera smoothly follows the movement of the player. As seen in the code below, the SmoothDamp function is used to provide a small lag to the camera giving the game a more natural feel.

```
public class CameraController : MonoBehaviour
{
    // distance from camera to player
    private Vector3 offset = new Vector3(0, 0, -10);
    private float smoothSpeed = 0.125f;
    private Vector3 velocity = Vector3.zero;
    public Transform playerTransform;

    Unity Message | 0 references
    private void Update()
    {
        Vector3 targetPosition = playerTransform.position + offset;
        transform.position = Vector3.SmoothDamp(transform.position, targetPosition, ref velocity, smoothSpeed);
    }
```

Figure 15: Code snippet highlighting the simplicity of the CameraController

Conclusions

Challenges and Difficulties

As with any sophisticated software, the Unity game engine presented an initial learning curve. Its complex interface and extensive features, while powerful, initially proved daunting. Navigating the Unity Editor and understanding its many panels and settings was a challenge, particularly when dealing with advanced features and plugins. This included deciphering the Inspector, Hierarchy, and Project panels, and understanding how to effectively use the Scene View and Game View.

One specific challenge I faced was implementing a touch interface for the game. With the game being intended for mobile devices, touch input was an essential part of the user interaction. Ensuring that touch inputs were handled correctly, particularly in the context of multiple concurrent touches, required a clear understanding of the Unity Input System and handling touch events effectively.

I also struggled to understand the UI system in Unity. The interactions between Canvas, UI elements, and the Event System were not immediately clear to me. Determining the best way to structure my UI, manage screen scaling on different devices, and handle user input was a significant challenge. In coordination with my UI/UX design course, I was able to apply some of these skills into the different menus within the game.

Optimizations

As with any game, there are still many optimizations I could make to improve the performance. Although my game is not CPU intensive enough to need these efficiencies, I think it is still important to point out some of the potential issues with my design.

Singletons: At the moment, I am using singleton's within my code base for objects such as the player and other game controllers. A singleton is created with the following code below to ensure that there is one and only one instance of that given class at a time. This becomes convenient for quickly accessing these objects, however it doesn't allow for much flexibility for growth. At the moment, my PlayerController class is a singleton, meaning that if I wanted to make the game two-player, I would likely have to change not only the PlayerController script, but many other scripts as well.

```
43
44          // establish the singleton pattern
45          void Awake()
46          {
47              if (Instance == null)
48              {
49                  Instance = this;
50              }
51              else
52              {
53                  Destroy(gameObject);
54              }
```

Figure 16: Example of how to make a class follow the singleton pattern

Object Pooling: The process of creating and destroying GameObjects is a costly operation, and some aspects of my game could benefit from object pooling. This works by creating a set, or "pool", of objects at the start of a scene and then reusing these objects as necessary, instead of creating new ones. For instance, pooling the enemy spiders of each level would likely improve performance, especially in the event of hundreds of spiders alive at a time. Seen below is an example of a situation that could occur in my game that would result in an excessive amount of spiders spawning, which could potentially lead to performance deficits. With object pooling, there would be a set number of spiders created upon initialization, removing this issue.

Figure 17: Example of a situation that would benefit from object pooling

Future Work

On top of the optimizations above, there are a few other features below that I simply didn't have the time to complete.

IOS testing: With Unity's built in cross platform compiler, I would have enjoyed testing and implementing the game onto an IOS device as well.

Profiling: More advanced developers often use the Unity Profiler to analyze different performance aspects of the game. This tool is rather complex and luckily I didn't have any unknown bottlenecks within my game, but I think I would benefit from learning how to use this tool.

Create More Levels: At the moment, the game is rather short and is really only exciting to play once or twice. I think a survival based game mode that increases in difficulty over time would be another great addition to the game.

Difficulty: At the moment, the options menu allows the user to select a difficulty, but I intend to include a Difficulty Manager script that would increase/decrease the difficulty of the game respectively.

Final Takeaways

Design Importance: Creating a game from scratch requires extensive design and planning prior to beginning the implementation. By creating flow charts, a UML diagram, and a storyboard, I set myself up with strong visual representations of both gameplay and code organization. However, looking back on the process, I think I would have benefitted from creating this documentation earlier in the first quarter.

Importance of Version Control: In the first few weeks of development, I slacked on setting up version control as I had never worked on such a large project. After nearly losing my entire project when my laptop randomly died on me, I made sure to research and set up GitHub LFS to avoid any potential risks like this in the future.

User Experience: By seeking advice from roommates and friends throughout the process, I was able to determine which features users enjoyed and find areas of improvement that I had not considered originally. Although informal, these questions and prototype testing results shaped many aspects of the game.

Game Engine Knowledge: After spending many weeks utilizing the Unity Editor and Unity Game Engine, I've gained a fundamental understanding of the tools required to create a game. With this solid understanding, I feel well-equipped to continue expanding my knowledge and learning new aspects of the Unity Engine.

Asset Credits

Player Art
Alias: Ansimuz
https://assetstore.unity.com/packages/2d/characters/tiny-rpg-forest-114685#description
https://ansimuz.com/site/

Level 1 Artwork
Alias: Szadi Art
https://www.artstation.com/szadiart

Level 2 Artwork
Krishna Palacio
https://www.minifantasy.net/

Level 3 and 5 Artwork
Zhivko Minchev
https://zmgamedesign456577029.wordpress.com/

Level 4 Artwork
Alias: Cainos
https://cainos.itch.io/

Joystick Asset
Fenerax Studios
https://assetstore.unity.com/publishers/32730

Spider Art
Graphic Artst: Stephen “Redshrike” Challenger
Contributor: William. Thompsonj
https://opengameart.org/content/lpc-spider

NavMesh2D
Github: h8man
https://github.com/h8man/NavMeshPlus