University of the Pacific

No Land Beyond

Alex Almanza: a_almanza@u.pacific.edu

Claryse Adams: c_adams18@u.pacific.edu

COMP 195 Senior Project

Professor Michael Canniff

7 February 2021

https://github.com/comp195/spring-2021-final-project-no-land-beyond

System Architecture

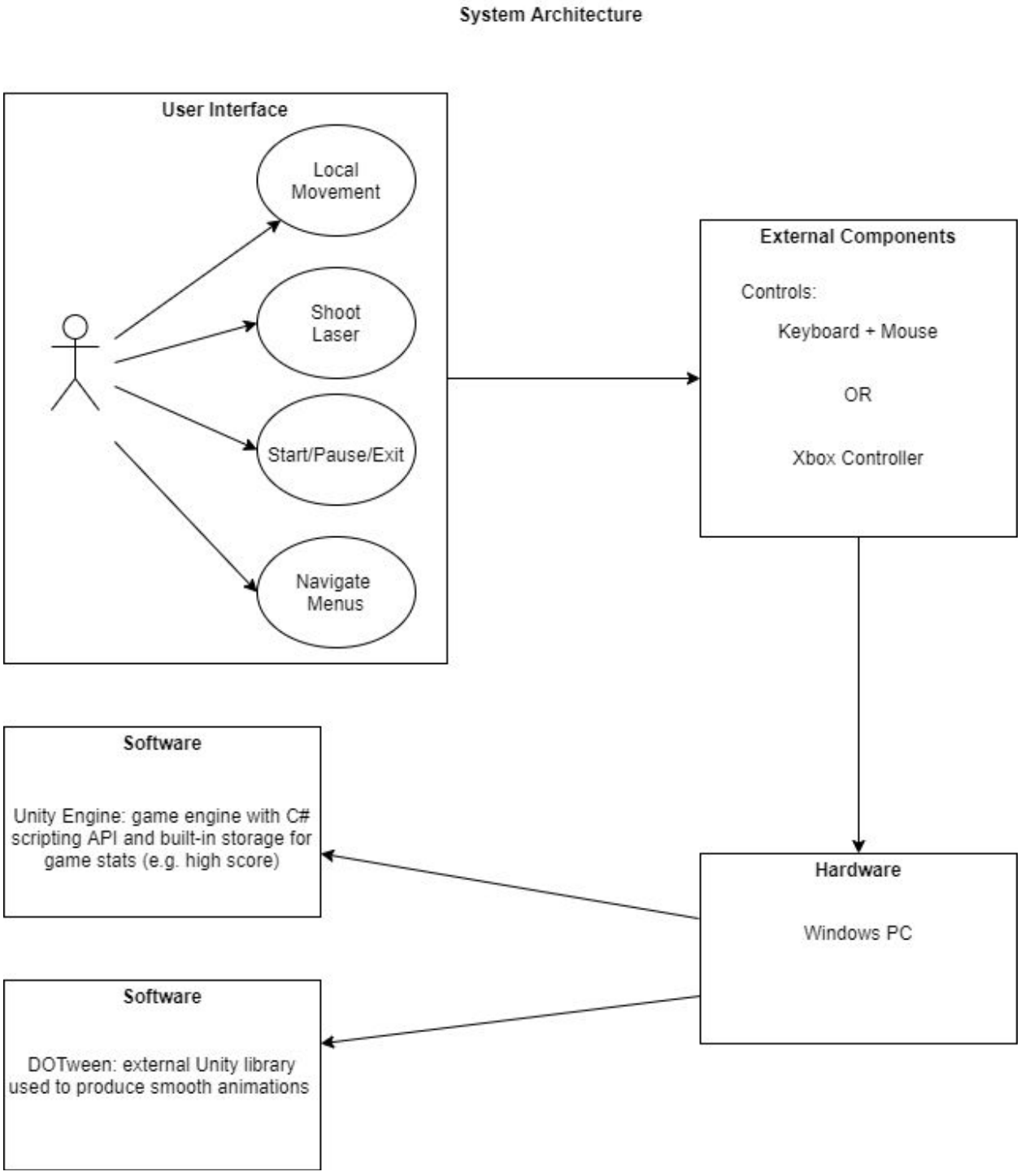

Figure 1.1: System Architecture

Hardware, Software, and System Requirements

Minimum System Requirements:

Requires a 64-bit processor and operating system

- OS: Windows 7 (64-bit and with latest updates)
- Processor: Dual Core 2.2 GHz or equivalent
- Memory: 4 GB RAM
- Graphics: 1 GB VRAM or better / Nvidia GTX 650 or AMD Radeon HD 6790
- DirectX: Version 11
- Storage: 2 GB available space

Recommended System Requirements:

Requires a 64-bit processor and operating system

- OS: Windows 10 (64-bit)
- Processor: Dual Core 2.2 GHz or equivalent
- Memory: 8 GB RAM
- Graphics: 2 GB VRAM or better / Nvidia GTX 670 or AMD Radeon HD 7970
- DirectX: Version 11
- Storage: 2 GB available space

External Interfaces

Our system is meant to run on one Windows-powered computer. The finished product will include an executable file. As such, a standard mouse, keyboard, and monitor are all required in order for the user to properly interface with the application. Additionally, we will include the option for the user to use a standard gamepad — such as an Xbox One or Xbox Series X controller — instead of a keyboard and mouse.

Software Design

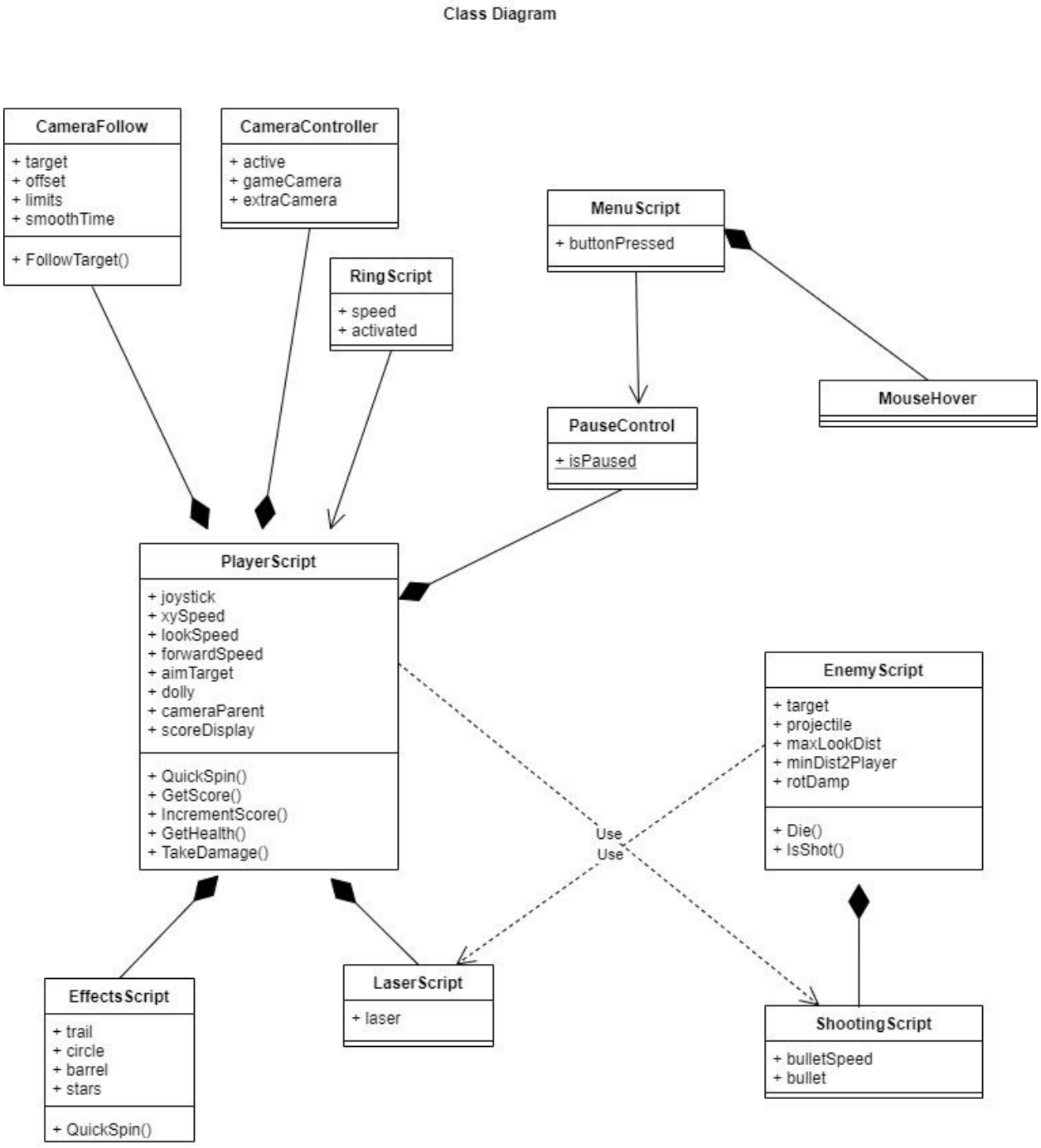

Figure 2.1: Class Diagram

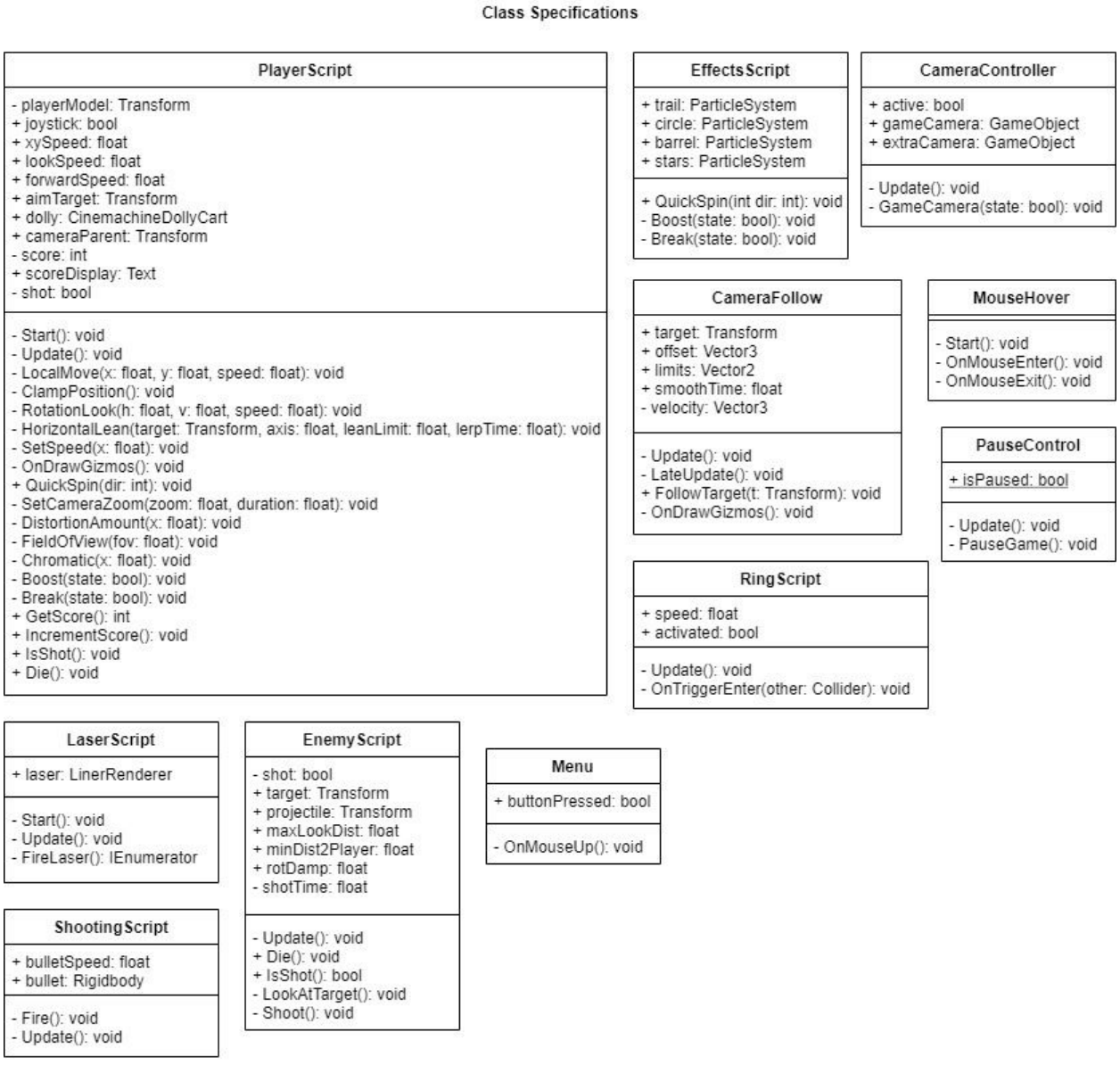

Figure 2.2: Class Specifications

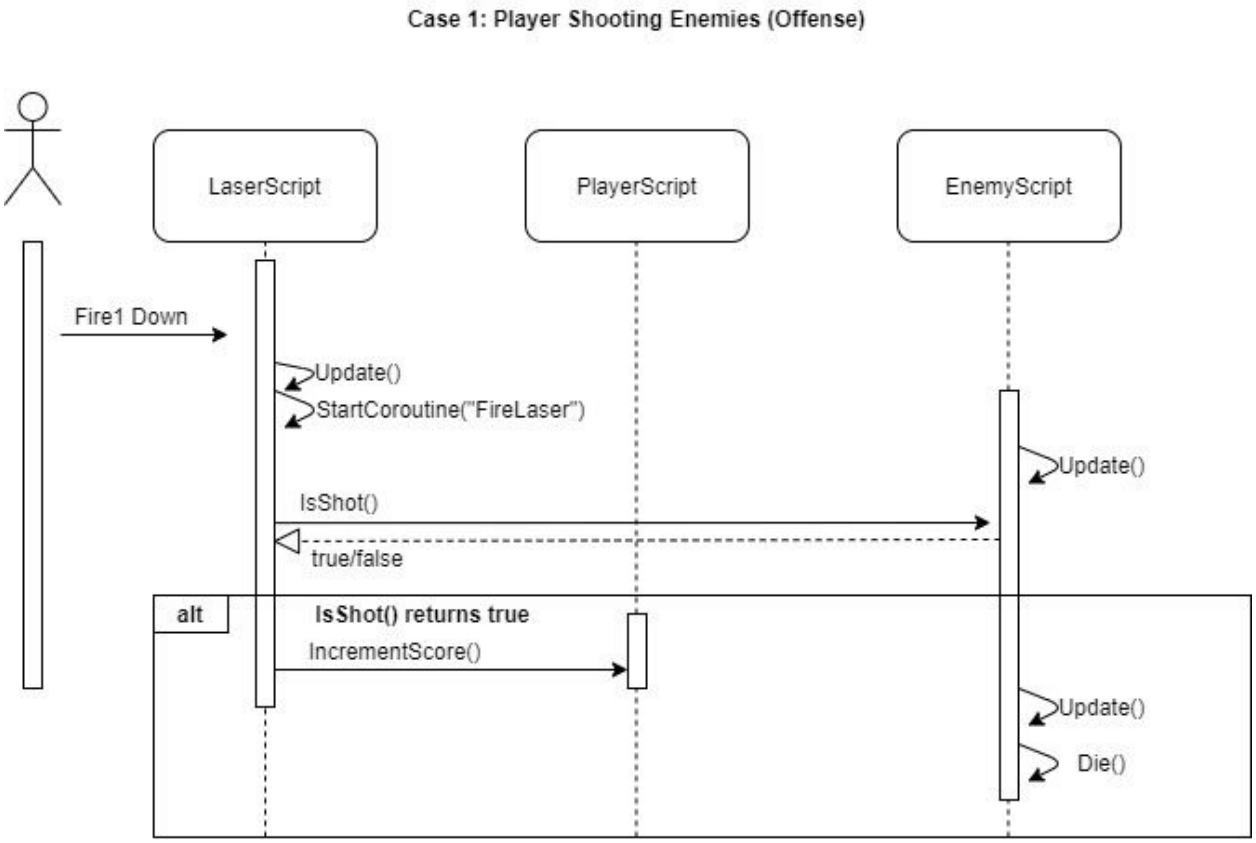

Figure 2.3: Sequence Diagram: Player shooting

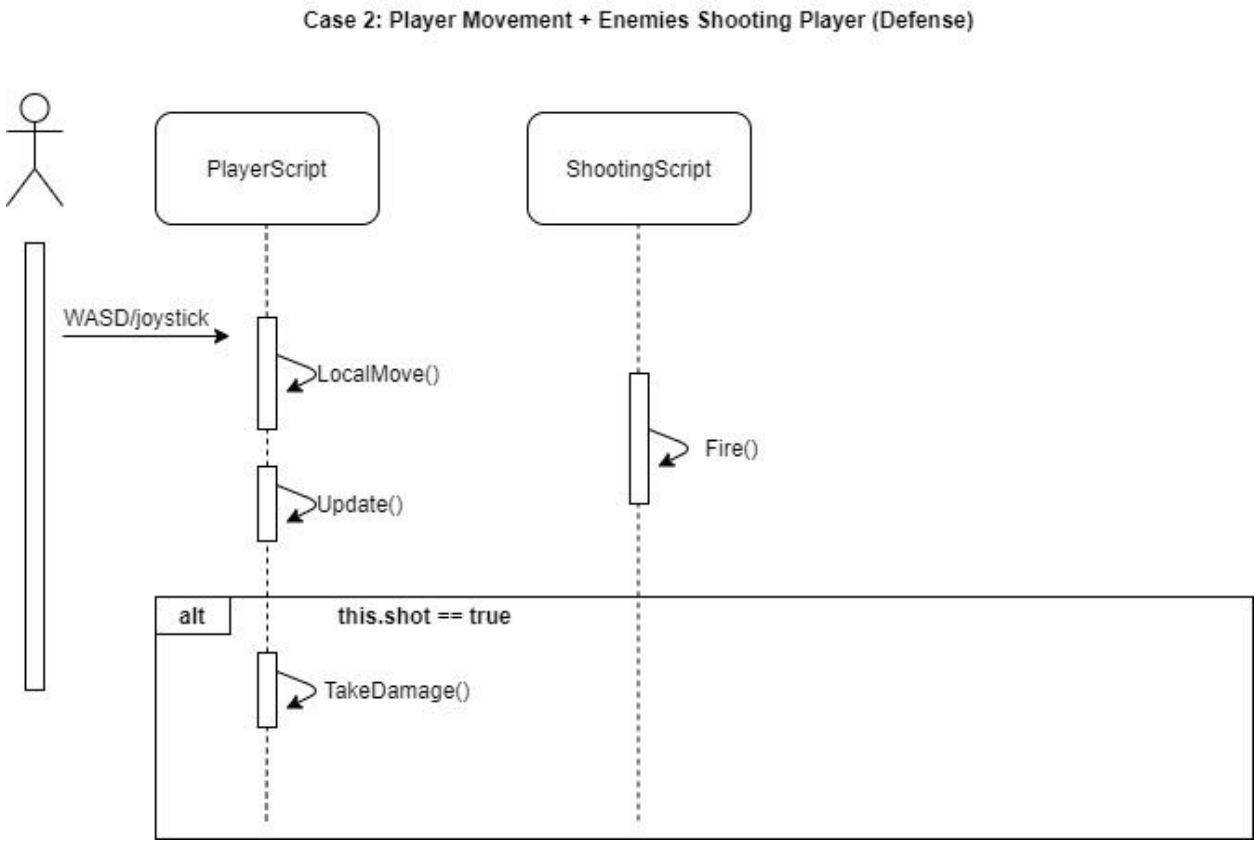

Figure 2.4: Sequence Diagram: Enemy AI and Player defensive movements

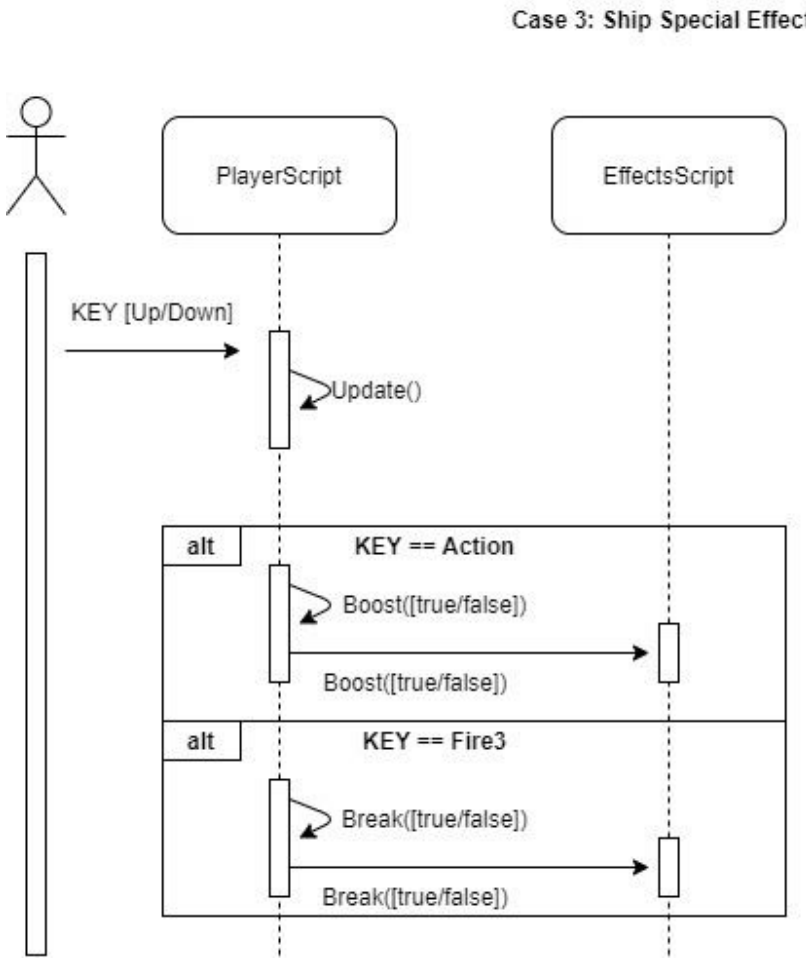

Figure 2.5: Sequence Diagram: Player model special effects

One of the central considerations taken when outlining the structure of this project's code is the fact that in game engines like Unity, a great deal of behavior of objects in the game world, known as GameObjects, and their relationships to each other are already pre-programmed by the game engine itself. C# scripts each containing a single class come into play only when behaviors more complex than what the engine already has pre-programmed are required of a specific GameObject or prefab. Developers may attach a variety of components to a GameObject to customize their behavior, including scripts, sound files, and physics assets like collision boxes and gravity. GameObjects may also become parents to additional GameObjects, causing the child object to travel with the parent and also allowing developers to leverage Unity's scripting API for the purpose of communication between classes defined in the scripts attached to the two objects (Unity Technologies, 2021).

In the class diagram shown above, each class will be attached to a single GameObject and the UML arrows connecting them actually show the relationships between the GameObjects to which these scripts will be attached. For example, the PlayerScipt class is shown as a composition of the LaserScript class because the LaserScript is to be attached to an object that

will be made a child to the player model object to which the PlayerScript will be attached (Philipp, 2018).

Leveraging the game engine and its related scripting API aids in the application of two of Ivan Marsic's (2012) software design principles, which are the "expert doer principle" and the "high cohesion principle". The expert doer principle dictates that responsibilities should be assigned to classes that would store the majority of information required for the responsibilities so that the classes do not have to communicate with each other too much. For example, all functionality related to the health and score of the player's character will be assigned to the PlayerScript class. The high cohesion principle states that a single class should not have too many computational responsibilities. For example, even though the laser shooting functionality is something that the user controls, we have elected to place it in a separate script (LaserScript) rather than place it in PlayerScript which is already responsible for knowing and altering the player's health, score, and movement. Ultimately the design process for this project consisted of finding the ideal balance between limiting the responsibilities of a single class and limiting the amount of communication needed between different classes.

User Interface Design

UI design will be inspired by the original Star Fox 64's design, with some modifications in order to achieve a more modern feeling game. The primary gameplay UI elements will include, at the very least, a player health bar, a score counter, and a reticle for player shooting.

We intend on having a main menu which will be the first thing a user will see when the application is launched, and it will have a few options:

NEW GAME
- Will start the First Level! Each level is automatically queued after the first level is completed.

OPTIONS
- Will take the Player to the Options submenu. Options will include things like Music Volume, FX Volume, Master Volume, and possibly a Controls menu.

HOW TO PLAY
- Will take the Player to the Tutorial Level, where the player will be able to run through a quick level that will help the player get used to the game's mechanics.

LEVELS
- Allows the player to select from a list of all levels which level to play.

QUIT
- Will exit the application.

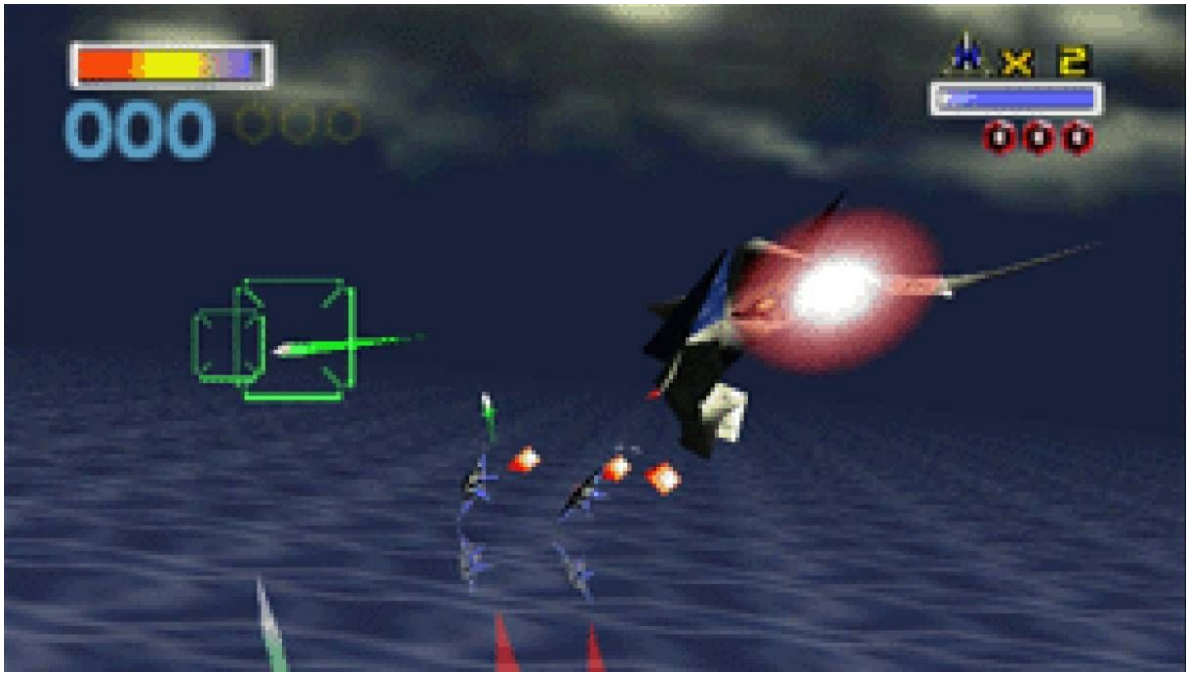

Figure 3.1: Star Fox 64 UI Design

Additionally, we plan on having a basic pause screen for the player's convenience. The pause screen will only be accessible when a level is being played, and will have similar features to the main menu, including a RESUME button, an OPTIONS button, and a QUIT button.

Figure 3.2: Main Menu Concept

Glossary of Terms

- *Star Fox 64*: the primary inspiration for our project. The project's level structure, gameplay mechanics, and maybe even some features will be derived from this classic game. Our project of course will eventually be completely different than Nintendo's work.
- 3D scrolling shooter: also known as a "Rail Shooter". Scrolling shooters are classic shooters — first or third person — where the player has no control over the camera. Think of a classic arcade game, like *The House of the Dead,* where the camera is scripted to move the player through a level.

There are not many terms that are specific to our project — yet!

References

Adams, C. (2021) [GitHub repository]. *Firstpersonlasergun.* Retrieved February 03, 2021, from https://github.com/cadams99/FirstPersonLaserGun

Clunk47. (2013, November 23). *Simple bullet script*. Retrieved February 02, 2021, from https://answers.unity.com/questions/581576/simple-bullet-script.html

John. (2020, February 13). *The right way to pause a game in Unity*. Retrieved February 02, 2021, from https://gamedevbeginner.com/the-right-way-to-pause-the-game-in-unity/#pause_time_scale

Jschap1. (n.d.). *How to make a main menu in Unity*. Retrieved Febrary 02, 2021, from https://www.instructables.com/How-to-make-a-main-menu-in-Unity/

Marsic, I. (2012). *Software engineering*. New Brunswick, New Jersey: Rutgers University

Mix and Jam. (2019, August 16) [GitHub repository]. *Starfox-railmovement.* Retrieved February 04, 2021, from https://github.com/mixandjam/StarFox-RailMovement

Philipp. (2018, June 25). *How to make a class diagram and data model of a Unity3D game.* Retrieved February 01, 2021, from https://gamedev.stackexchange.com/questions 160112/how-to-make-a-class-diagram-and-data-model-of-a-unity3d-game

Star Fox 64 (Nintendo 64 version) [Video game]. (1997). Kyoto, Japan: Nintendo Entertainment Analysis & Development.

Unity Technologies. (2021, January 19). *Introduction to components*. Retrieved February 05, 2021, from https://docs.unity3d.com/Manual/Components.html

ZefanS. (2016, March 31). *Enemy AI for shooting game*. Retrieved February 01, 2021, from https://answers.unity.com/questions/1162762/enemy-ai-for-shooting-game.html