Domino Squad

Project Domino

Software Design Document

Name (s):
Avik Shenoy
Thanh Tran
Amir Roochi
Brendan Beagin
Daniel Sarkisian

COMP 490/L

Date: 11/13/2022

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Purpose

The purpose of this SDD is to describe the architecture and structure of Project Domino, our third-person action horror game. The intended audience is our development team and professor Edmund Dantes.

1.2 Scope

Project Domino is a video game being created in Unreal Engine 5. The game will be a third-person horror-centric action adventure game with a focus on combat and exploration. The game's art style will be one that is inspired by the PlayStation 1 era from the late 1990s. The game's theme will be inspired by dark medieval/dark fantasy, with horror elements. The gameplay will feature multiple levels, enemies, bosses, non-player characters, and role-playing elements (upgrades, inventory). The game will also feature a narrative that guides the player along the journey.

1.3 Overview

This SDD explores the overall construction of Project Domino. The SDD explains Project Domino's class structure, such as the parent/child C++ class structure. The SDD also covers our user interface design. In addition, the SDD explains how our requirements will be satisfied.

1.4 Definitions and Acronyms

HUD - heads-up display

UI - user-interface

AI - artificial-intelligence

mesh - A piece of geometry that consists of a set of polygons that can be cached in video memory and rendered by the graphics card.

TBD - to be determined

WIP - work in progress

2.0 SYSTEM OVERVIEW

Project Domino is a third person video game which is developed using Unreal Engine 5. The game gives the ability to the user that allows them to explore, fight with enemies, pick up some items and follow the story of the game using a narrative which is designed to help the player through different levels of the game. The game also provides an inventory system that allows the user to keep track of items that they have picked up during the game. The game also provides a combat system which allows the player to fight with enemies with different levels of difficulty.

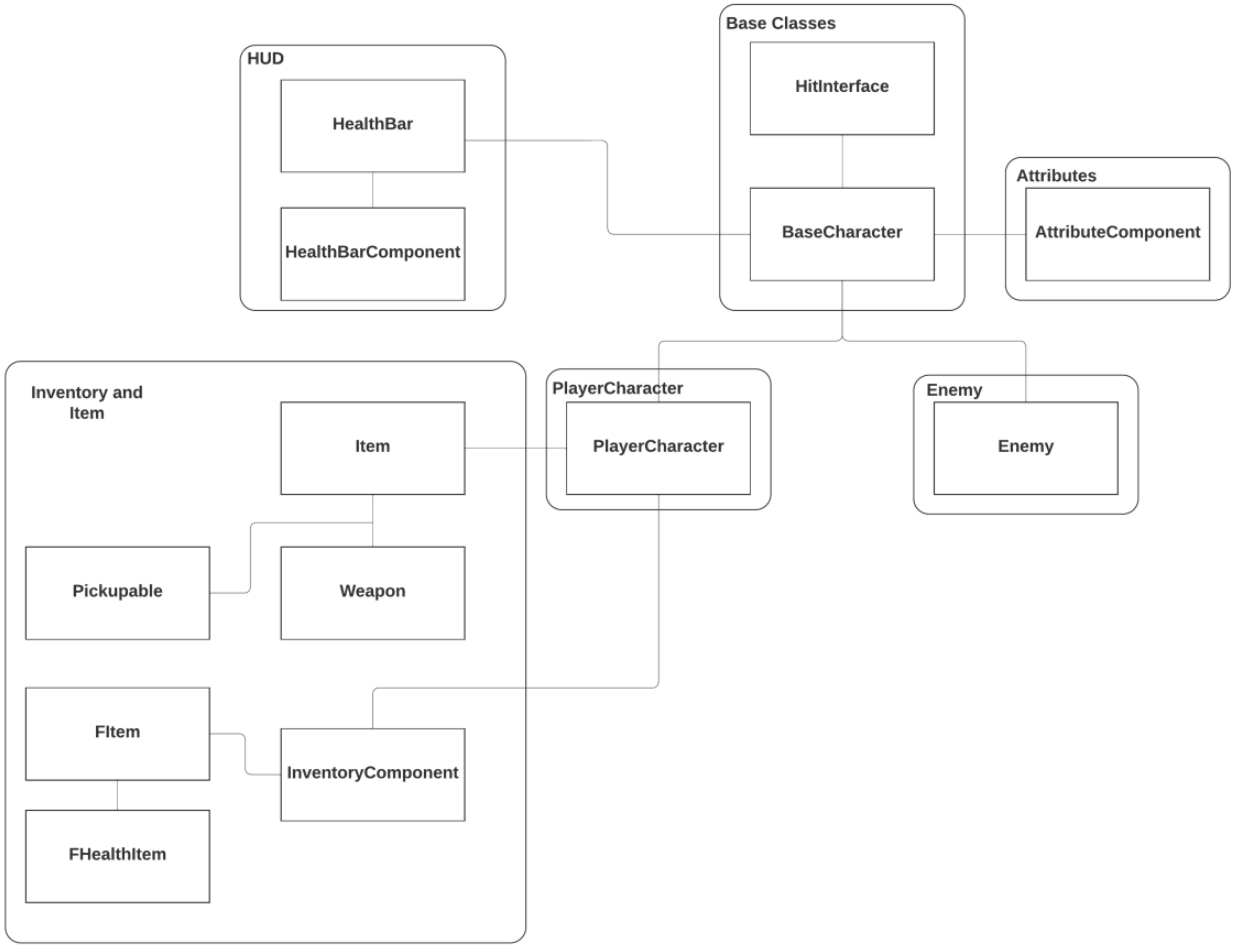
As mentioned the general design of the game would be a third person horror centric adventure game which is inspired by the design and graphical look of the Playstation 1 era. The game is designed to have multiple levels, and multiple characters along with providing the ability of interacting with some elements for the player. To achieve this goal our team designed and developed different C++ parent and child classes which the general description of them is as follows:

- character class which is the base class for different characters of the game.
- components class which is the base class for different components of the game
- Enemy class which is the base class for different enemy characters that the player encounters during the game.
- HUD class which is responsible to keep track of health and stamina of the player and enemy characters.
- Item and inventory system class which is responsible for keeping track of available items along with the items that the player picks up during the game.

For in-game UI design the game is designed to have a main menu that consists of different elements such as start, continue, save, load options and exit. The UI design also consists of a player inventory UI, that helps the users to keep track of items already in their inventory. This game is designed to run on low to medium spec machines. The machine should have access to the internet and runs on Windows 10/11.

3.0 SYSTEM ARCHITECTURE

3.1 Architectural Design



3.2 Decomposition Description

The decomposition of relationships of each software C++ class is discussed in this section. The BaseCharacter class is the base class that both the PlayerCharacter and Enemy classes inherit from. PlayerCharacter and Enemy borrow functionality from BaseCharacter, such as the attack functionality, hit reaction functionality, weapon functionality, animation montage functionality, and much more. BaseCharacter inherits some functionality from HitInterface, which has a function called GetHit(), which both PlayerCharacter and Enemy use their own implementation of. The AttributeComponent class controls functionality of statistical data, such as health, and BaseCharacter uses that data. The HealthBarComponent and HealthBar classes are a part of the HUD (heads up display), and BaseCharacter also uses that data. The Item class is the base class that the Weapon class inherits from, and the Weapon borrows functionality like sphere collision and mesh functionality.

The Pickupable class inherits from the Item base class on top of new functionality

such as implementation of OnPickUp() or storing a reference to the FItem that it's supposed to hold. It’s the Actor representing the FItem in the gameworld.
FItem is the item object itself within the player’s inventory.
FHealthItem inherits from the FItem base class and carries new functionality pertaining to healing such as healAmount or an overridden implementation of the Use() function.
Inventory Component manages the inventory itself. This includes adding, removing, or defaulting an item list.

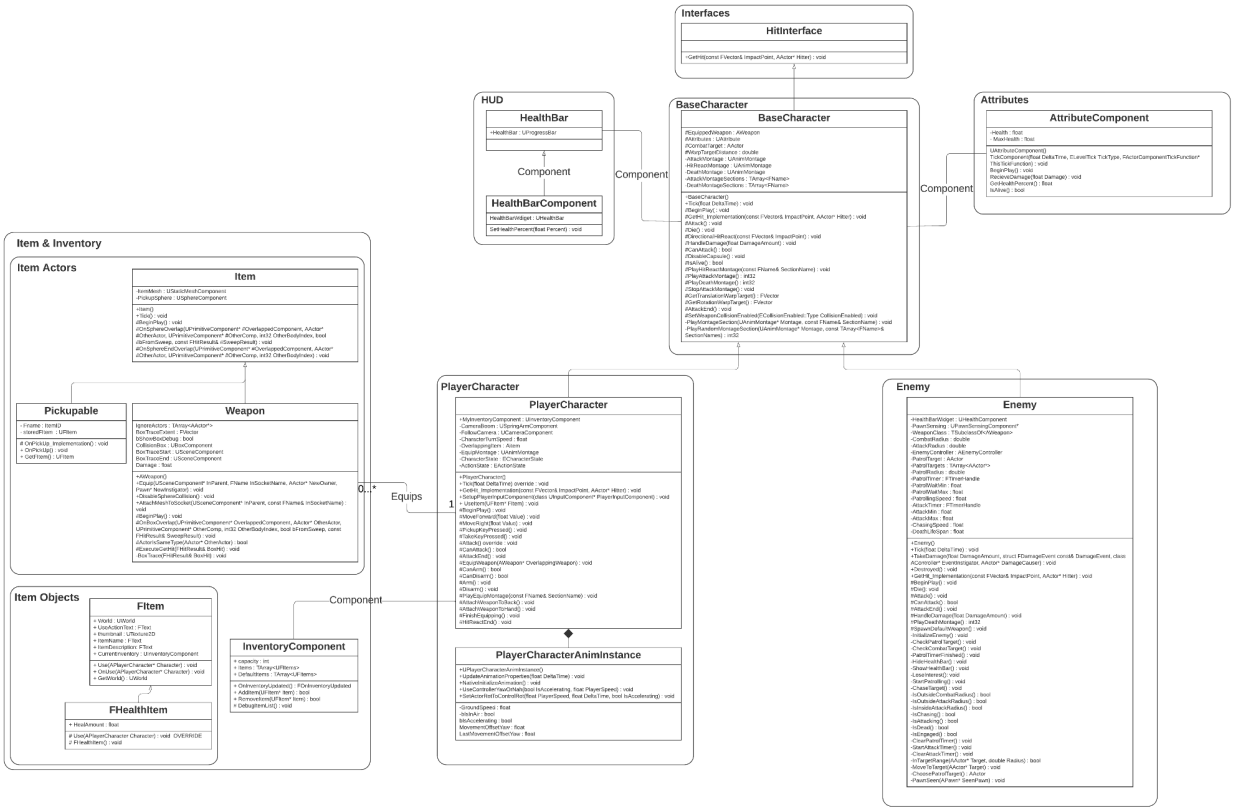
3.3 Design Rationale

Our rationale for how we architected our software is discussed in this section. The BaseCharacter class is the core class for both PlayerCharacter and Enemy. Both Enemy and PlayerCharacter inherit from BaseCharacter, and therefore both classes use functionality from BaseCharacter. We decided to architect our character structure this way because this structure leverages the power and advantages of C++. If two classes use the same functionality, it makes sense to create a base class from which both can inherit from to make code cleaner and more efficient. There are no trade-offs to this approach either, since PlayerCharacter or Enemy can override functionality from BaseCharacter and use their own child version of the class. If this is done, the child class can still call the parent version of the functionality if desired. Therefore, this architecture allows for more modular, unrepeated, and easier to read code that allows for maximum productivity. We used this structure for the Item class as well for the same reasons.

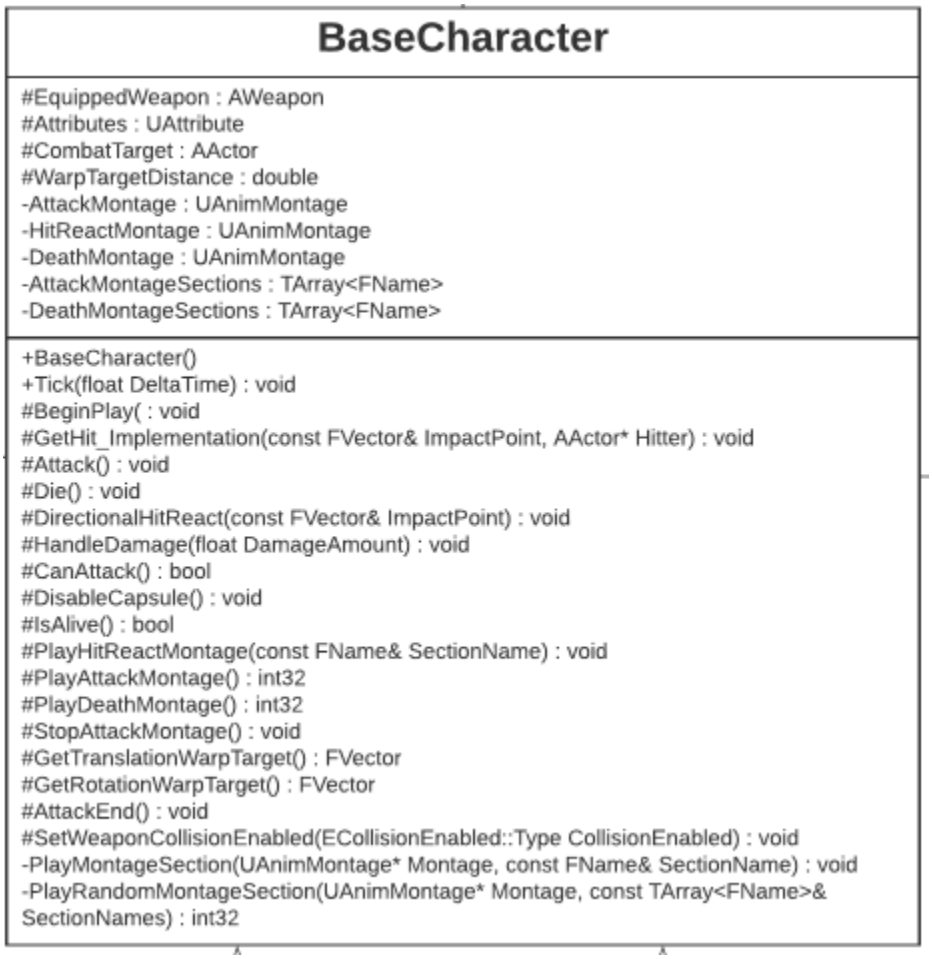
In term of structuring the Inventory system, the separation between the two Item Actors (Item, Pickupable) and Item Objects (Fitem, FHealthItem, etc) is optimal because as a pure object class that only deals with data, FItem has a minimal overhead in term of memory and performance cost. This is in comparison to the Item Actor class, which holds a mesh, pickup spheres, and other things that are unnecessary. In this system, the Item Actor acts as a representation of the FItem within the game’s world, and once interacted / picked up, the FItem is added to the inventory and the item Actor itself is destroyed, maintaining a low performance upkeep.

4.0 COMPONENT DESIGN/DETAILED DESIGN

4.1 UML DIAGRAM (C++)



4.1.1 BaseCharacter

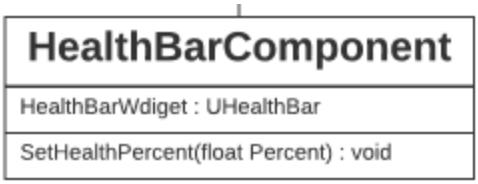


BaseCharacter is a base class that both PlayerCharacter and Enemy inherit from. BaseCharacter inherits from ACharacter and IHitInterface. BaseCharacter handles functionality that is shared between PlayerCharacter and Enemy. For example, BaseCharacter has attack functionality, meaning both PlayerCharacter and Enemy can use that functionality since both the player and enemy can attack. Another example is hit react functionality, both the player and enemy will recoil back when hit by a sword, so both require hit react functionality. Both attacking and hit reactions require animations so BaseCharacter also has animation montage play and selection functionality, which both PlayerCharacter and Enemy can use. If PlayerCharacter or Enemy want to use their own versions of these functions, they can override that functionality and still call the parent versions of the functions if desired.

4.1.2 HealthBar

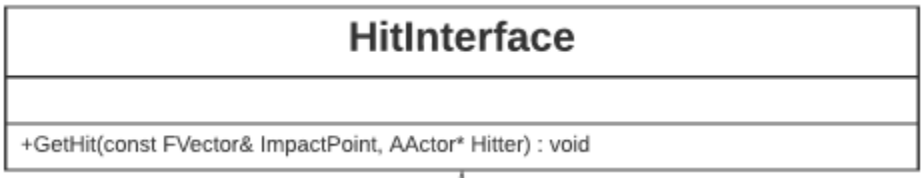
HealthBar is a class that serves as basically an object for HealthBarComponent to use.

4.1.3 HealthBarComponent



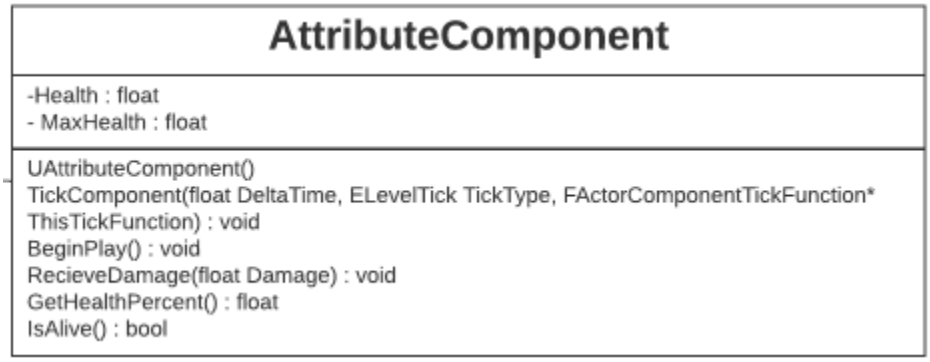
HealthBarComponent is a class that is responsible for setting the HealthBar Percent, which is basically figuring out how much the health bar gauge is filled.

4.1.4 HitInterface



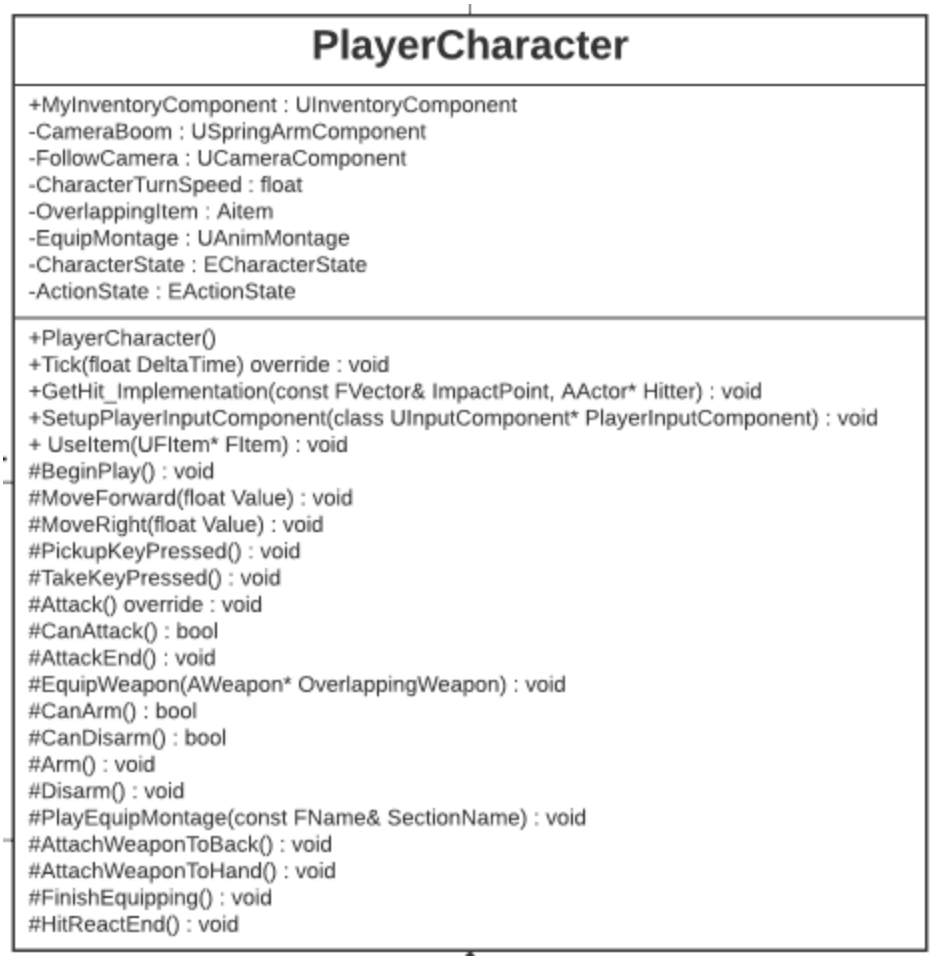
HitInterface is an interface class responsible for handling the hit implementation for all classes that require it. For example, both the enemy and player can “get hit”, but both classes require different functionality when they are hit. Therefore, both Enemy and PlayerCharacter can use their own implementation of GetHit from the interface.

4.1.5 AttributeComponent



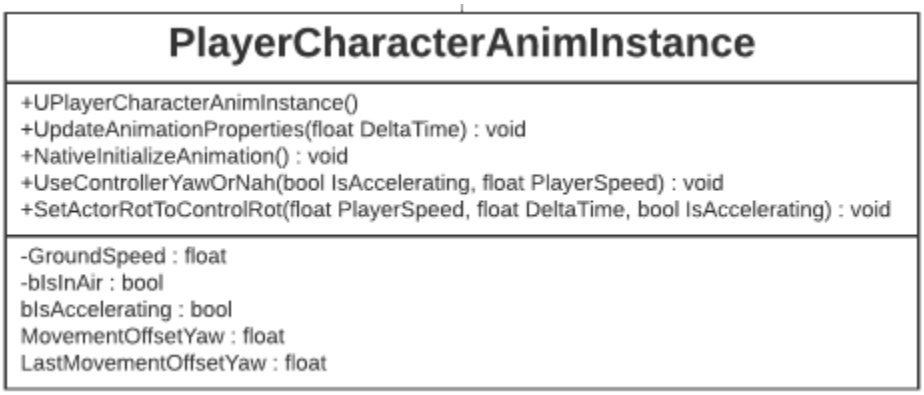
AttributeComponent is a class that is responsible for handling statistical data for the characters, such as health and stamina. For example, information for how much health and stamina is contained in the AttributeComponent, so if Health and Stamina are 100.f, that information can be found in a character’s attributes.

4.1.6 PlayerCharacter



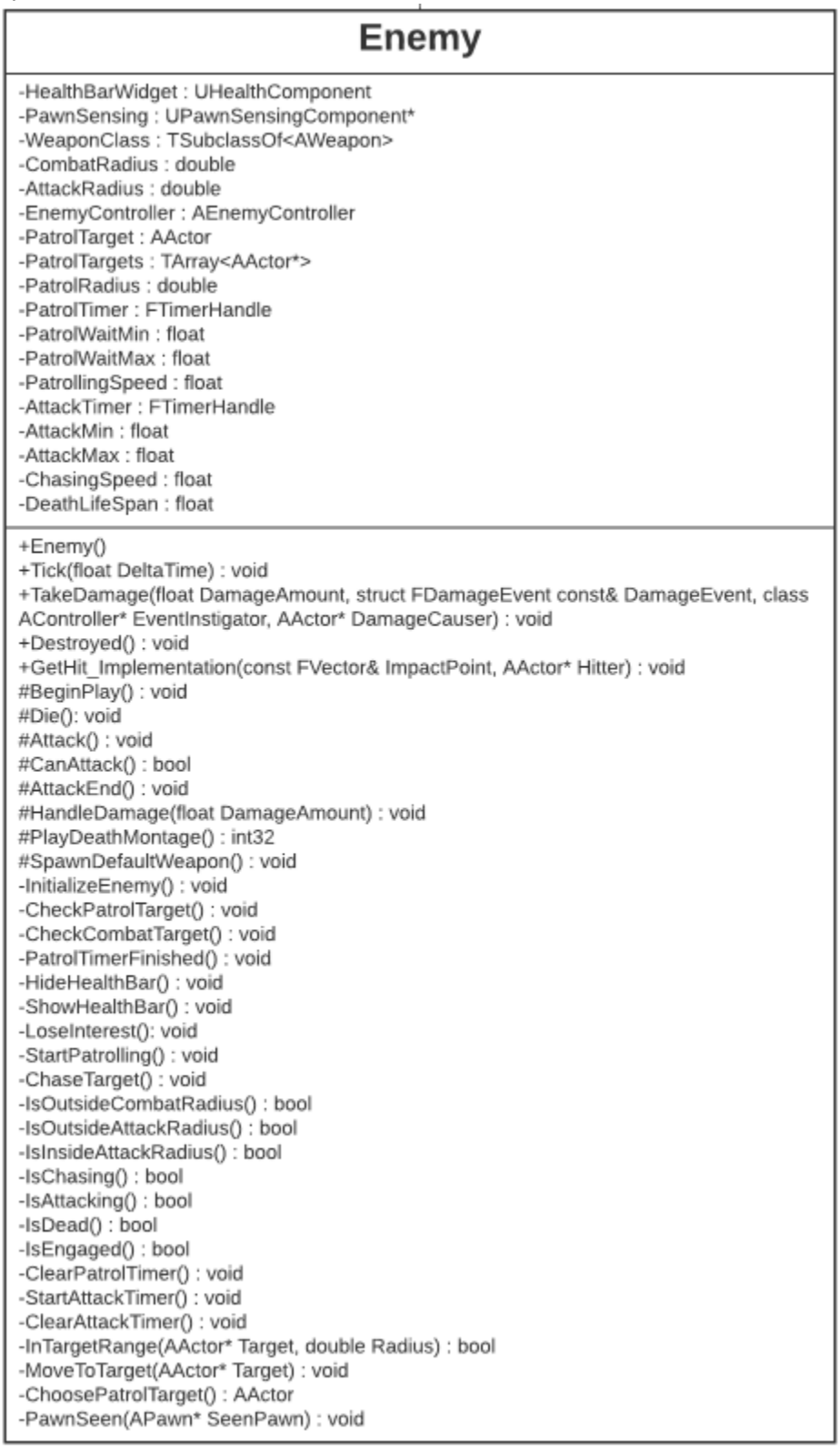
PlayerCharacter is the class that the player inhabits when they play the game. PlayerCharacter handles all sorts of functionality, such as walking, looking, attacking, picking up items, and more. PlayerCharacter handles input from the player, so if the player presses a specific button on their keyboard or gamepad, PlayerCharacter registers that input and calls a function in response.

4.1.7 PlayerCharacterAnimInstance



PlayerCharacterAnimInstance handles some animation properties for the player character, such as what happens when the player is in the air, when they are accelerating, etc.. AnimInstance also handles calculations that are important for determining which animations to play, specifically for strafing and walking backwards.

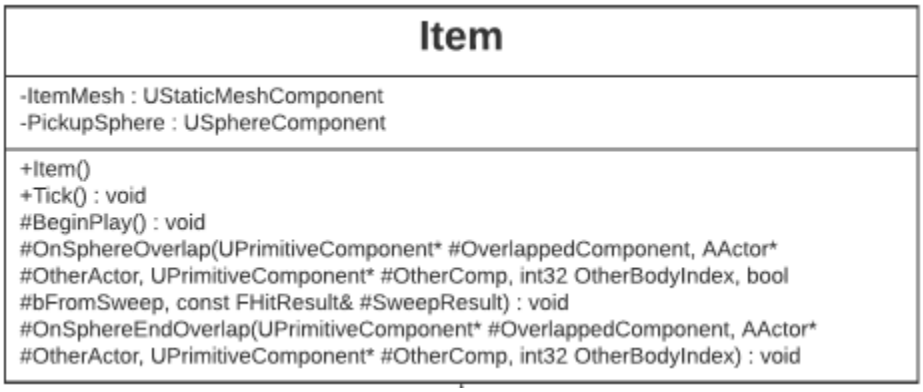
4.1.8 Enemy



The Enemy class is a class that is responsible for handling enemy AI behavior and functionality.

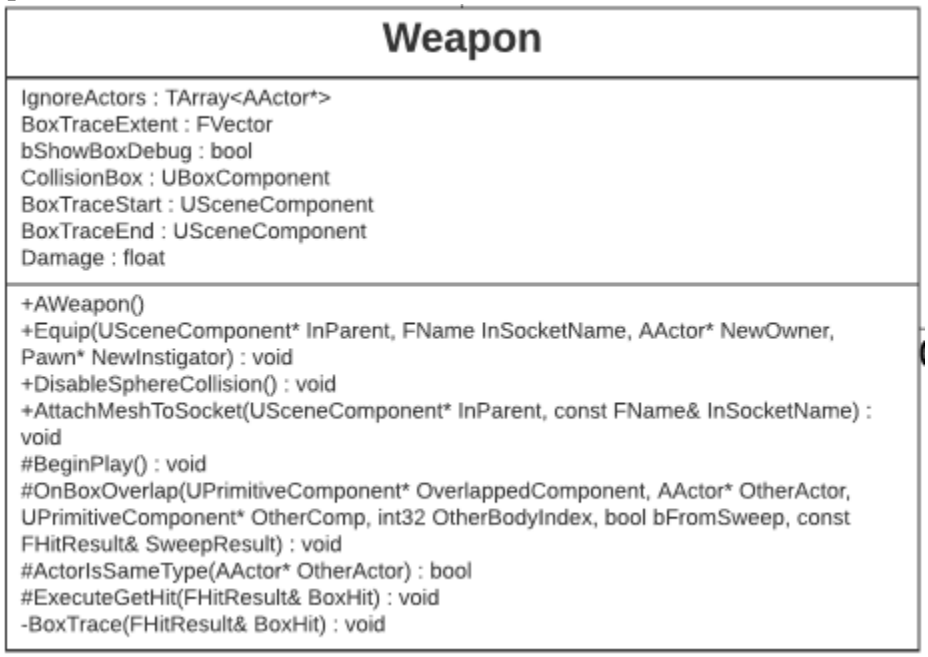
Enemy is by far the most extensive class, as the enemy can be in a number of different states and behaviors, such as patrolling, chasing, attacking, engaging, dying, and more. Enemy is constantly performing calculations to figure out what the enemy should be doing at any given moment, it's an advanced system.

4.1.9 Item



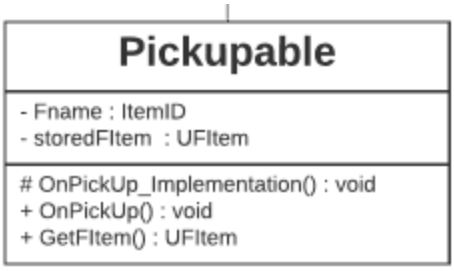
Item is a base class responsible for handling any item that the player can pick up and use. This includes weapons, health items like food and potions, throwable items like bombs, ammunition like arrows, and more. The Item class has an important relationship with the inventory, as an Item is initially in the world as a pickup and stored in the inventory as a usable item.

4.1.10 Weapon



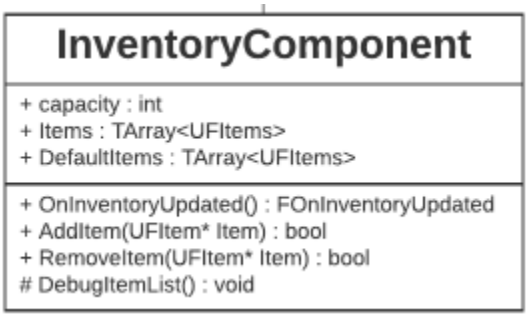
Weapon is a class derived from Item, and serves as a weapon for the player and enemy to use to attack. The Weapon handles when collision is active, and can calculate that collision's impact point and send that data to the character classes.

4.1.11 Pickupable



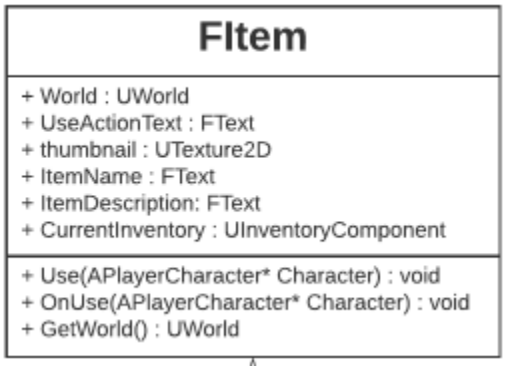
This is an Actor class derived from the Item base class. It holds a reference to the FItem that it's supposed to represent as well as a function for OnPickUp().

4.1.12 InventoryComponent



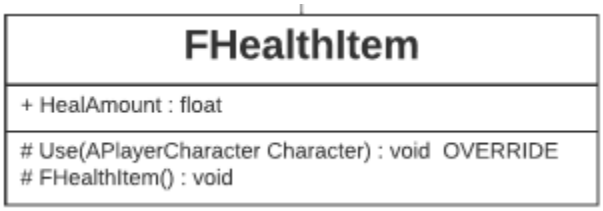
This class manages the inventory. It handles adding or removing items. For now, it also houses the inventory item itself in the form of a TArray

4.1.13 FItem



This is an Object class. It holds data as well as functions defining its usage and what happens when it's used.

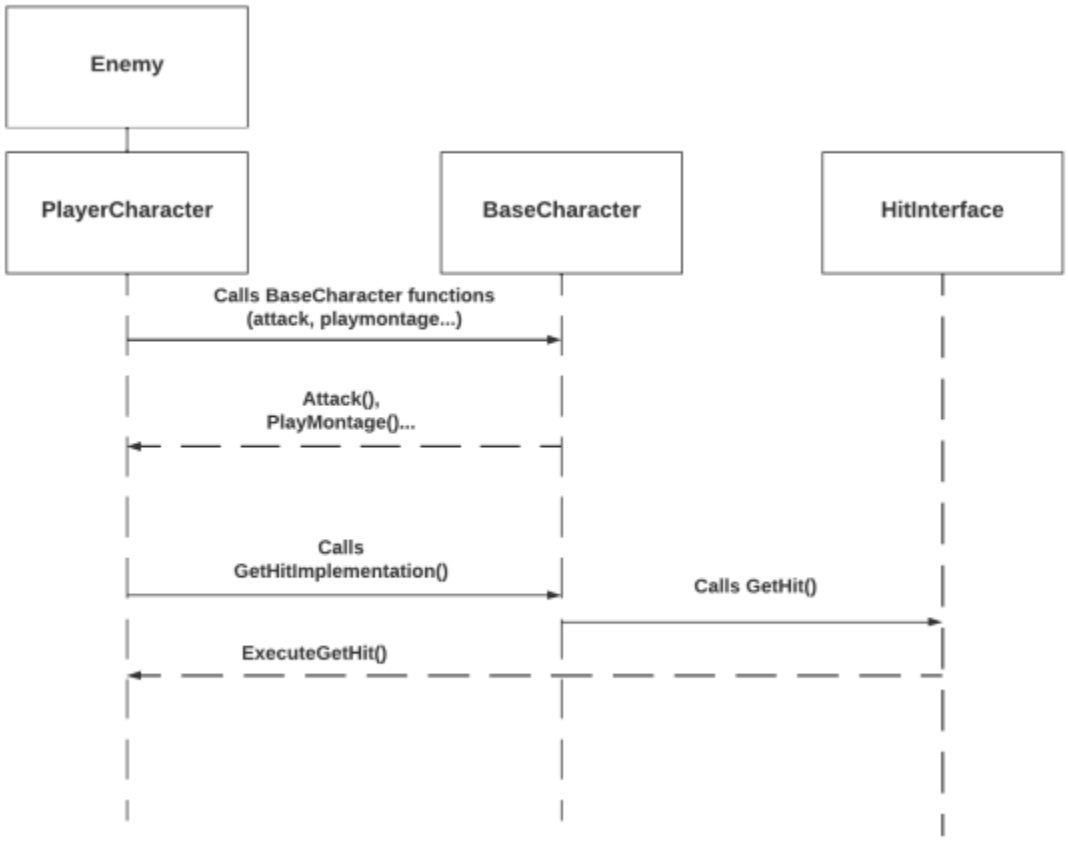
4.1.14 FHealthItem



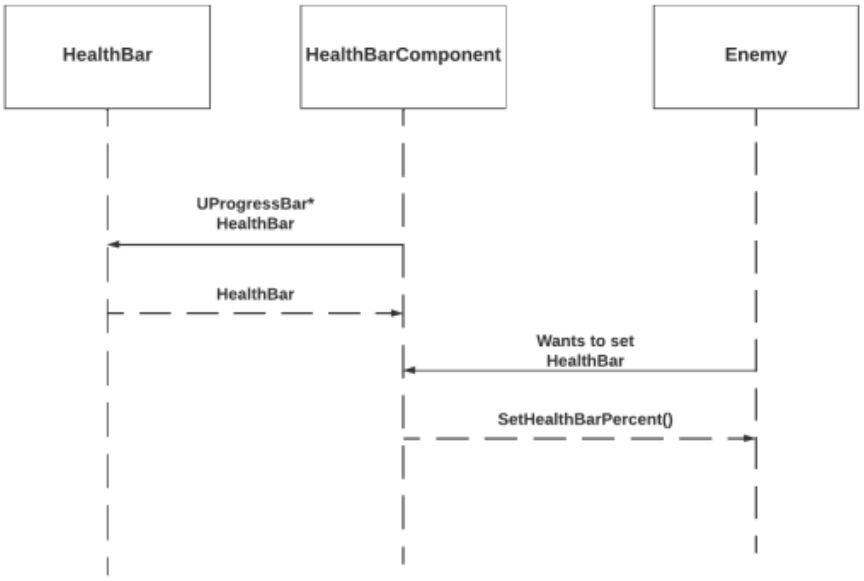
This is derived from FItem. It holds data and function pertaining to healing.

4.2 SEQUENCE DIAGRAMS (C++)

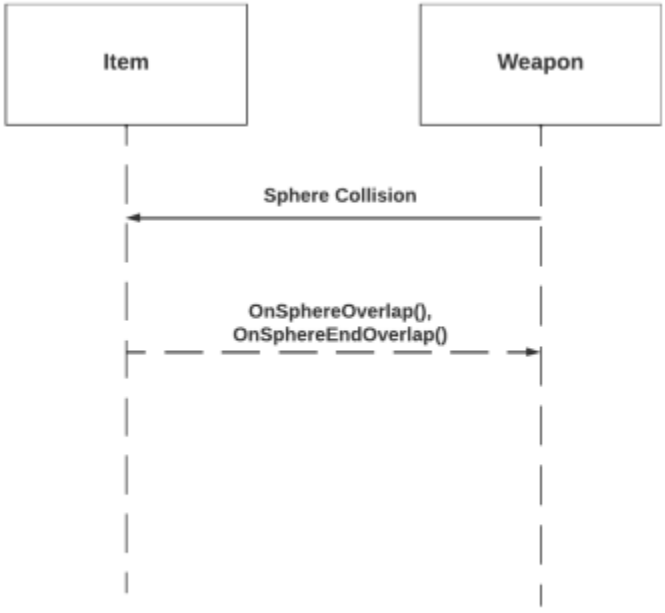
4.2.1 BaseCharacter/PlayerCharacter/Enemy/HitInterface Sequence Diagram



4.2.2 Enemy/HealthBar/HealthBarComponent Sequence Diagram

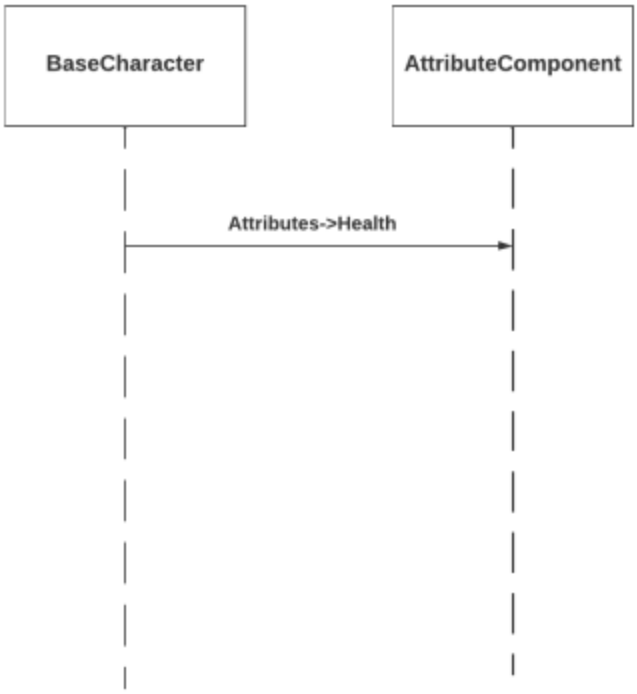


4.2.3 Item/Weapon Sequence Diagram

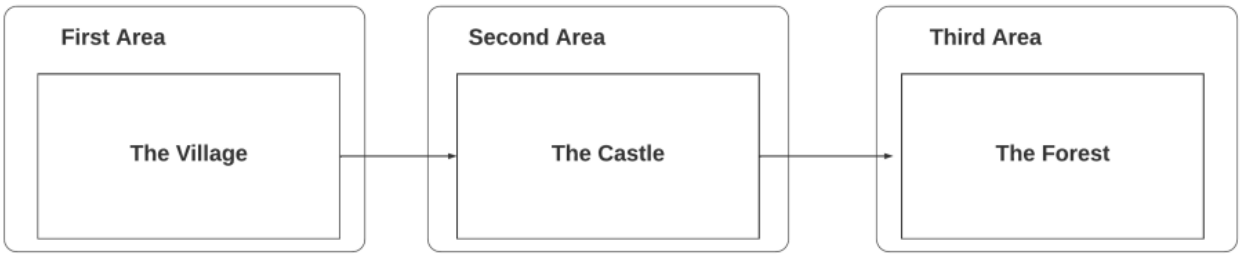


4.2.4 BaseCharacter/Attributes Sequence Diagram

Software Design Document



4.3 LEVEL DESIGN

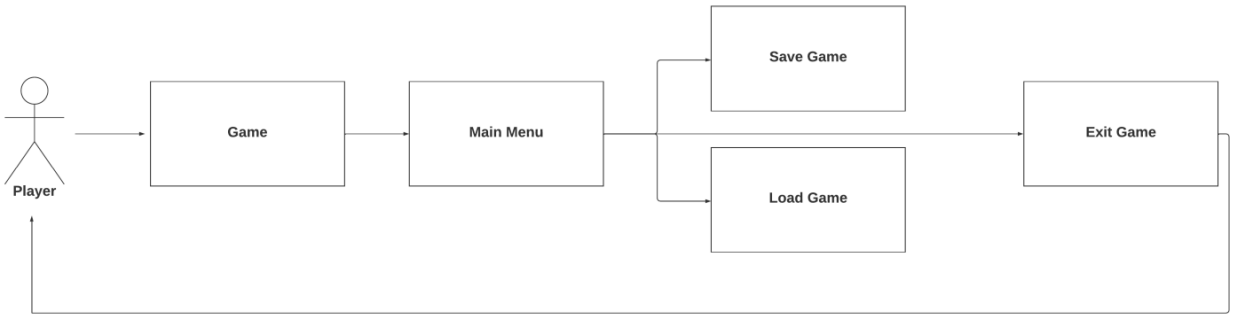


4.3.1 The Village
The Village is the first area that the player will explore in the game. During this section, the player will play as the knight character, and will be in search of the merchant's missing daughter.

4.3.2 The Castle
The Castle is the second area that the player will explore in the game. During this section, the player will play as both the daughter and the knight characters. When playing as the daughter, the player will escape captivity and search for a way out of the castle. When playing as the knight, the player will be searching for the daughter.

4.3.2 The Forest
The Forest is the third and last area that the player will explore in the game. During this section, the player will once again play as the daughter and the knight. They both will be escaping the village together.

4.4 SAVING AND LOADING



4.4.1 Saving and loading

First the player will start up the game. The game will begin and the player can play for however long they desire. Let's say the player plays for one hour. After one hour, the player can then pull up the main menu. In the main menu, there are three options. The player can save their game which will save their progress, the player can load their game which will load their last saved game, or the player can exit the game, which will close the game. The player can first save and load their game, or only save their game, before exiting the game. After exiting the game, the player can start up the game again and will be returned to their previous saved game.

5.0 HUMAN INTERFACE DESIGN

5.1 Overview of User Interface

From the user's perspective, the following UI subsystems facilitates the game experience

1. Main menu
2. Pause menu
3. Ingame HUD
4. Inventory menu
5. Settings menu

1. Main menu (figure 1):
Upon opening the game, the user is presented with a main menu. The options in this menu includes:

1. starting a new game: Starts a new game. If the user has an existing save, prompt if the user wants to override.
2. continue: Continue the game through savestate.
3. settings: opens the settings menu
4. exit to desktop: exist the game and return to user's desktop

2. Pause menu (figure 2):
During gameplay, the user can hit the esc key to bring up the pause menu. This pauses the game and the following options are shown.

1. Unpause: closes the menu and unpause the game
2. settings: opens the settings menu
3. Exit to main menu: Exits to main menu
4. Exit to desktop: Closes the game and exits to desktop

3. Ingame HUD (figure 3):
During gameplay, several dynamic and static elements are shown on screen.

1. Player's health bar: displays the player's current health.
2. Player's action bar: displays the player's current action points.
3. Enemy's health: displays the enemy's current health.

4. Inventory menu (figure 4):
During gameplay, the player can hit the I key to bring up the inventory menu. This pauses the game and brings up the inventory panel.

1. Inventory list: the items currently in the player's inventory
2. Item name: The name of the current item
3. Item description: the description of the current item
4. UItemWidget: houses the item icon as well as the use button
5. Use item: the player can use an item by clicking on it.
    a. by hovering over an item, the player can see the action that the item performs.

5.Settings menu (figure 5):
In Main Menu or during gameplay, the player can navigate to the settings menu to manage the game settings:

1. Audio slider: The player can adjust the audio volume to their liking.
2. Music slider: The player can adjust the music volume to their liking.

5.2 Screen Images

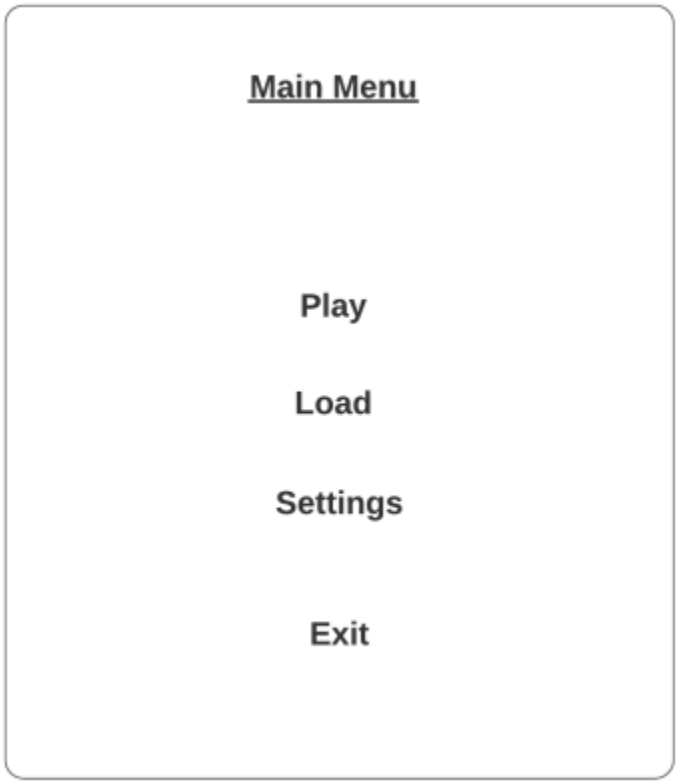


figure 1. Main Menu (WIP)

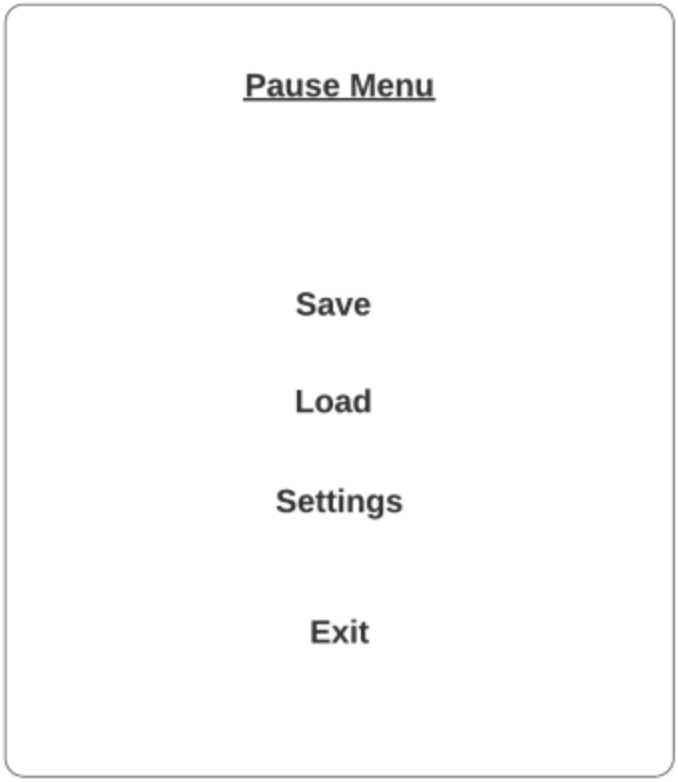


figure 2. Pause Menu (WIP)

figure 3. In-game HUD WIP

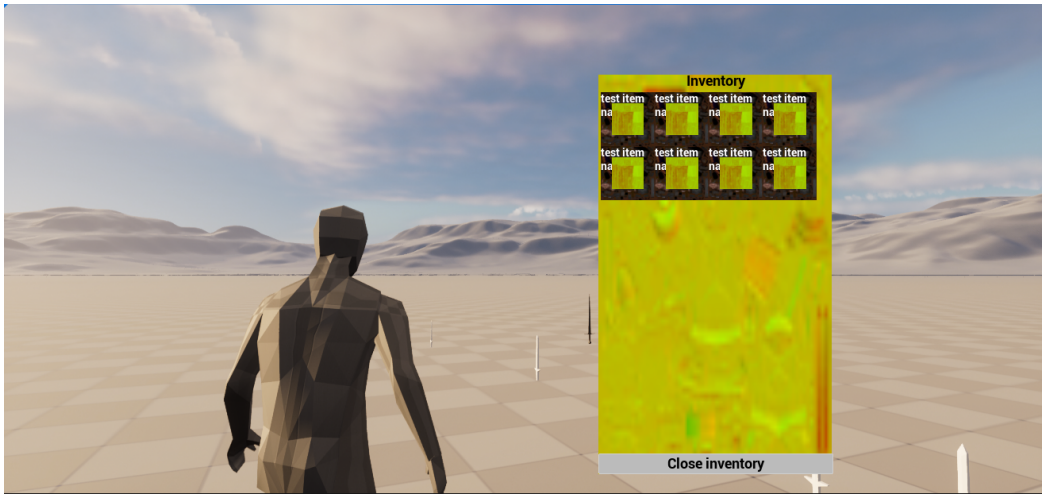


figure 4. Inventory System UI (WIP)

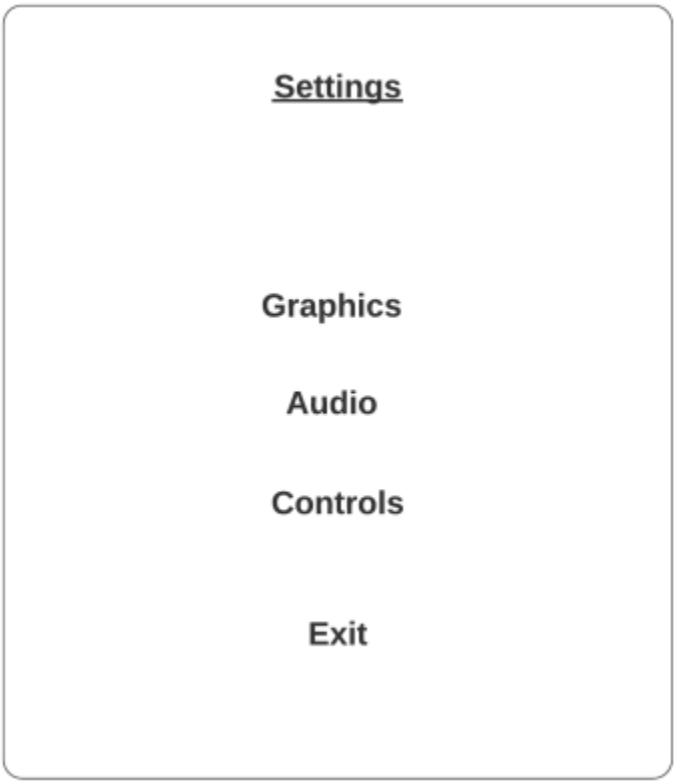


figure 5. Settings Menu (WIP)

5.3 Screen Objects and Actions

A discussion of screen objects and actions associated with those objects.

1. Menu Button: uwidget button tied into any of the menu functionality
2. Panels: UI elements that purposefully take up a section of the screen, sometimes as a backdrop / textured menu
3. sliders: slider used to adjust something granularly
4. UItemWidget: houses the item icon and the use button for said icon
5. Inventory list: generate and display current list of items upon player opening the Inventory
6. HUD health bar: displays the player's current health in a bar.
7. HUD action bar: displays the player's current action points
8. enemy's health bar: displays the enemy health on top of their heads

6.0 REQUIREMENTS MATRIX

SRS Req. ID	Paragraph Title	Component Num.	Component name
Func-001.1	The game shall provide a controller that allows the player to move forward, backward, right and left, as well as fully control their camera.	4.1.6, 4.1.7	"PlayerCharacter, PlayerCharacterAnimInstance "
Func-001.2	The game shall provide the ability of interacting with some specific objects during the game.	4.1.6, 4.1.12, 4.1.13, 4.1.14	" Player Character, InventoryComponent, FItem, FHealthItem"
Func-001.3	The game shall provide a combat system that allows users to fight enemies.	4.1.4, 4.1.5, 4.1.6, 4.1.7, 4.1.8, 4.1.10	" HitInterface, AttributeComponent, PlayerCharacter, PlayerCharacterAnimInstance, Enemy, BaseCharacter0 "
Func-001.4	The game shall provide the user the ability to keep track of their own health and stamina.	4.1.2, 4.1.3, 4.1.5	" HealthBar, HealthBarComponent, AttributeComponent "
Func-001.5	The game shall provide different kinds of enemies.	4.1.8	" Enemy "
Func-001.6	The game shall provide an inventory system that keeps track of the items the player found during their adventure.	4.1.9, 4.1.11, 4.1.12, 4.1.13, 4.1.14	" Item, BaseCharacter1, BaseCharacter2, BaseCharacter3, BaseCharacter4 "
Func-001.7	The game shall provide different locations that give the player the ability to explore different areas of the level.	4.3.1, 4.3.2, 4.3.3	" The Village, The Castle, The Forest “
Func-001.8	The game shall provide two multiple protagonists including the knight and the daughter.	4.1.6	" PlayerCharacter "
Func-003.2	The game shall support playing multiple sessions.	4.4.1	" Saving and Loading "