Project Title: *Soul Keeper*

Team Members: Vince Curran, Kevin Vo, Logan Park

Work Done
Characters: All characters have health/damage/combat implemented. The player's controls are all done (walk, run, attack, jump, etc.). The enemies are controlled by a state machine which is mostly finished.

Interactables: Certain objects can be interacted with. This includes checkpoints, which update the player's spawn position. Doors are also implemented, with many configurables.

UI: All major UI components have been implemented for functionality, this includes menus, health bars, and more.

FX: Scripts to control effects have been finished. This really just consists of the timed destruction of whatever objects are FX. They are instantiated by whatever object is 'making' the effect. IE a sword makes a 'sword hit' effect on contact, which consists of an audio clip and small particle effect.

Save and Load: Persistent data is implemented through serialization of data into JSON files. There are both 'UserSettings' which control things between any save game such as volume and quality, as well as 'PlayerSettings' which are individual save games.

Changes/Issues Encountered
Player Rolling: Originally the 'roll' action was implemented using the same movement as the player, but changing the animation. This led to a really bad experience, where rolling would not actually make you cover more ground than normal and you could change direction during it. This was fixed by making the roll take control from the player during the duration of it, so when you hit the button, it sends you rolling a certain speed for an amount of time, and you cannot change direction or use other controls for that time.

Enemies Seeing Through Walls: Originally, enemies would just check for the direction and range of the player, to see if they should start chasing or attacking. This meant they could see through walls and would aggro if the player was on the other side of one. This was fixed by using raycasts to check for environment objects from the enemies eyes to the player.

Patterns

MVC:
Model - Player, PlayerController
View - GameObject in Scene
Controller - PlayerController
User interacts with PlayerController (controller) through PlayerInput, which updates Player/PlayerController/Animator etc. values (model), which then get reflected in the game scene (view).

State:
A state machine (EnemyStateMachine, EnemyState) is being used to control the behavior of enemies.

Interface:
Implemented by several classes, including Interactable, Attack, and Character. They each provide a range of methods & data designed to be final, extended, or overridden.

Singleton:
Used by several different classes which we want to ensure only one of throughout the lifetime of the application. Most notably these are the GameManagers (PauseManager, MainMenuManager, FadeToBlack, etc.).

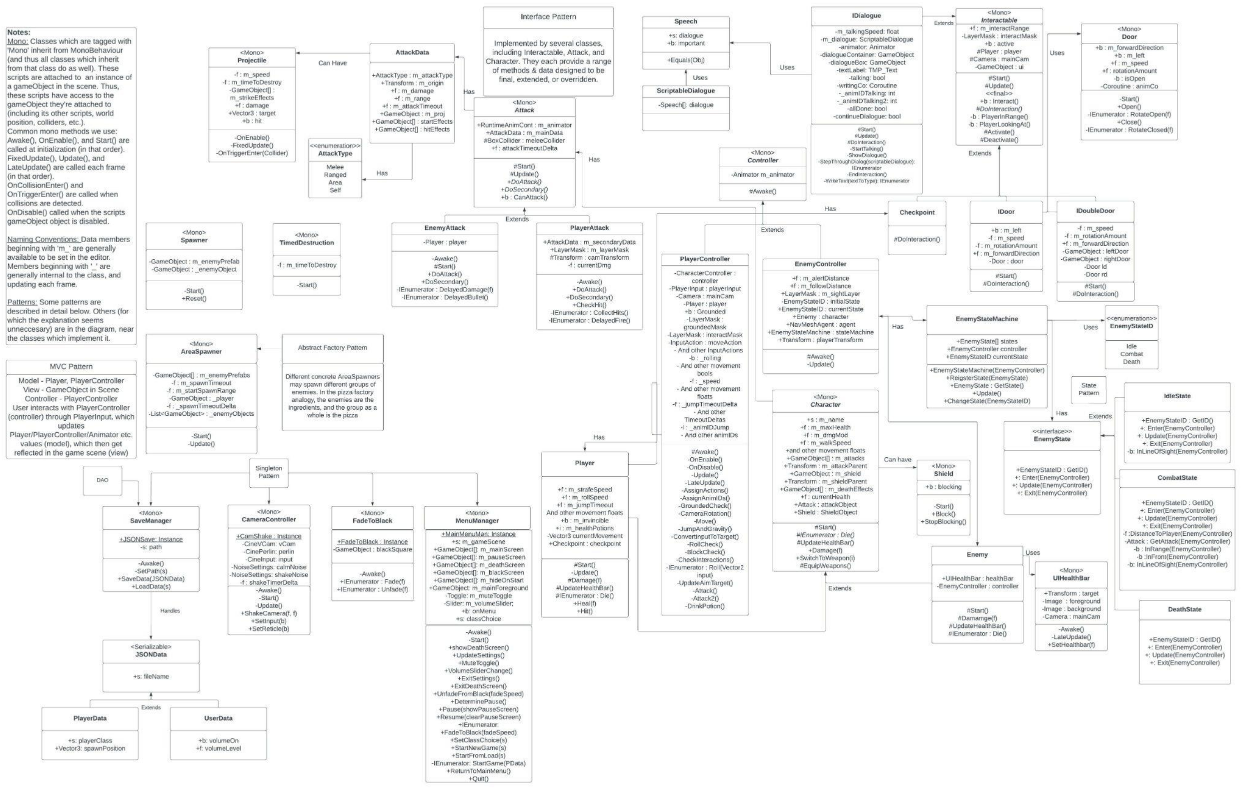
Abstract Factory:
Different concrete 'AreaSpawners' may spawn different groups of enemies. In the pizza factory analogy, the enemies are the ingredients, and the group as a whole is the pizza.

DAO:
The 'JSONSave' class provides an interface for interacting with the game's persistent data.It provides simple 'save' and 'load' operations so other classes do not need to know the details.

Plan for Project 7

By the Project 7 deliverable we plan to have a working demo of a game. We will have all the assets and the level built. The interface will be standard for a game. Over the next week we plan to apply the scripts to our objects, build the level, and fix any bugs that come up. For the last week, we plan to do playtesting and apply any finishing touches and bug fixes that come up.

Class Diagram (also included as a standalone pdf in GH)



Notes:
Mono: Classes which are tagged with 'Mono' inherit from MonoBehaviour (and thus all classes which inherit from that class do as well). These scripts are attached to an instance of a gameObject in the scene. Thus, these scripts have access to the gameObject they're attached to (including its other scripts, world position, colliders, etc.).
Common mono methods we use: Awake(), OnEnable(), and Start() are called at initialization (in that order). FixedUpdate(), Update(), and LateUpdate() are called each frame (in that order).
OnCollisionEnter() and OnTriggerEnter() are called when collisions are detected.
OnDisable() called when the scripts gameObject object is disabled.

Naming Conventions: Data members beginning with 'm_' are generally available to be set in the editor. Members beginning with '_' are generally internal to the class, and updating each frame.

Patterns: Some patterns are described in detail below. Others (for which the explanation seems unneccesary) are in the diagram, near the classes which implement it.

MVC Pattern
Model - Player, PlayerController
View - GameObject in Scene
Controller - PlayerController
User interacts with PlayerController (controller) through PlayerInput, which updates Player/PlayerController/Animator etc. values (model), which then get reflected in the game scene (view)

Interface Pattern
Implemented by several classes, including Interactable, Attack, and Character. They each provide a range of methods & data designed to be final, extended, or overridden.

Abstract Factory Pattern
Different concrete AreaSpawners may spawn different groups of enemies. In the pizza factory analogy, the enemies are the ingredients, and the group as a whole is the pizza