Zombie Island

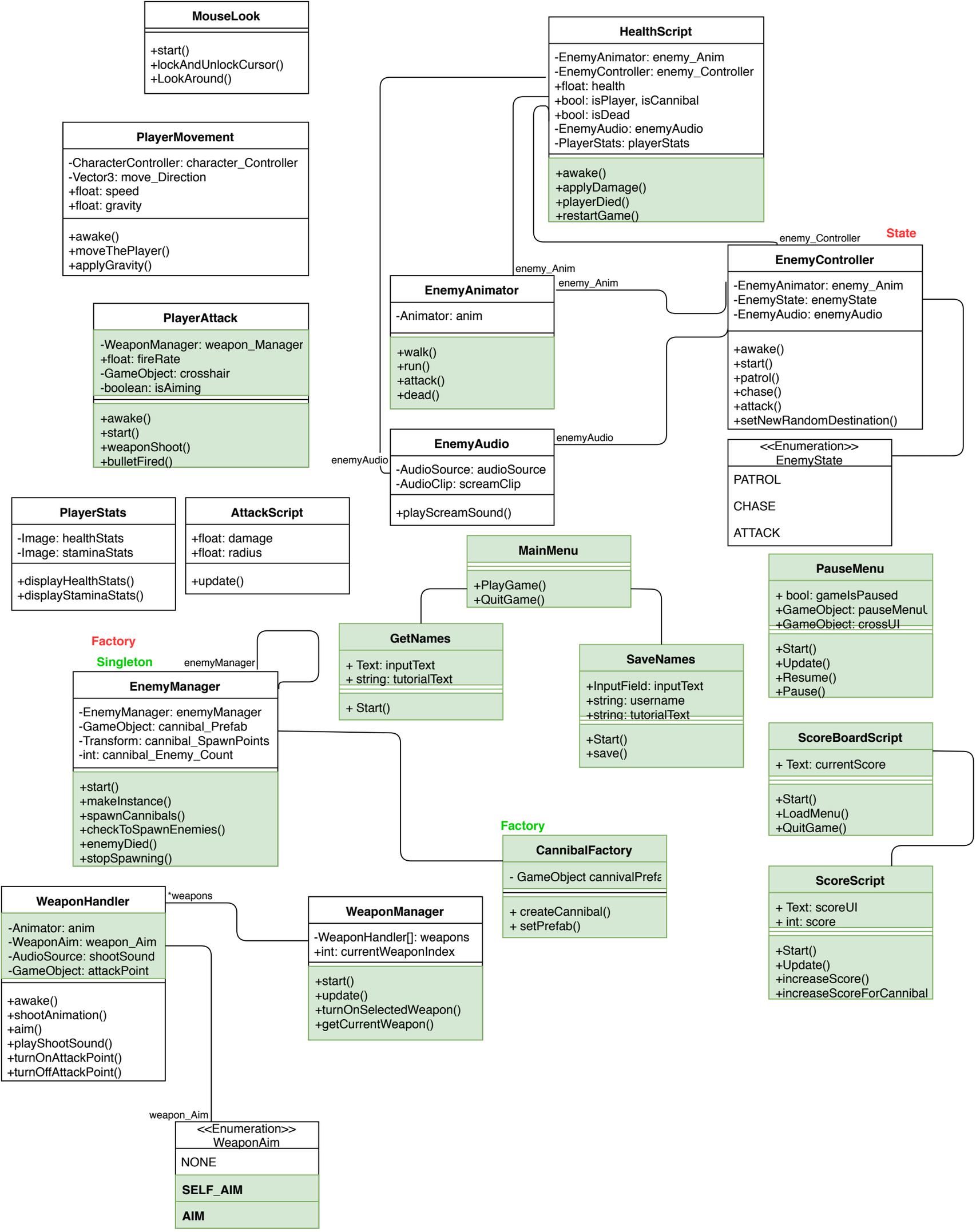
Peng Jiang, Ben Zaeske, Mikayla Pickett

Final State of System

- Features Implemented
    - Generate Island Map
    - Player can use 3 weapons:
        - Melee axe
        - Pistol
        - Rifle
    - Player can move around, sprint, jump and shoot enemies
    - Enemies have 3 different states:
        - Patrol: Walk around randomly
        - Chase: The player is within sight range and they run towards the player
        - Attack: They enemy has reached the player and is attacking them
    - A game timer which is in charge of several things:
        - Player score increases as timer goes on
        - Spawn rates of enemies increases as time goes on
    - The game supports a high score system for a single player
    - Main menu, pause menu, and high score menu.
    - The game is playable on all operating systems with Unity support.
- Features Not Implemented:
    - Leaderboard style score saving - username and score displays upon game over. It does not display any previous scores of previous users. We decided to narrow our scope and not include any high score saving across profiles so that we could focus on other aspects of the game.

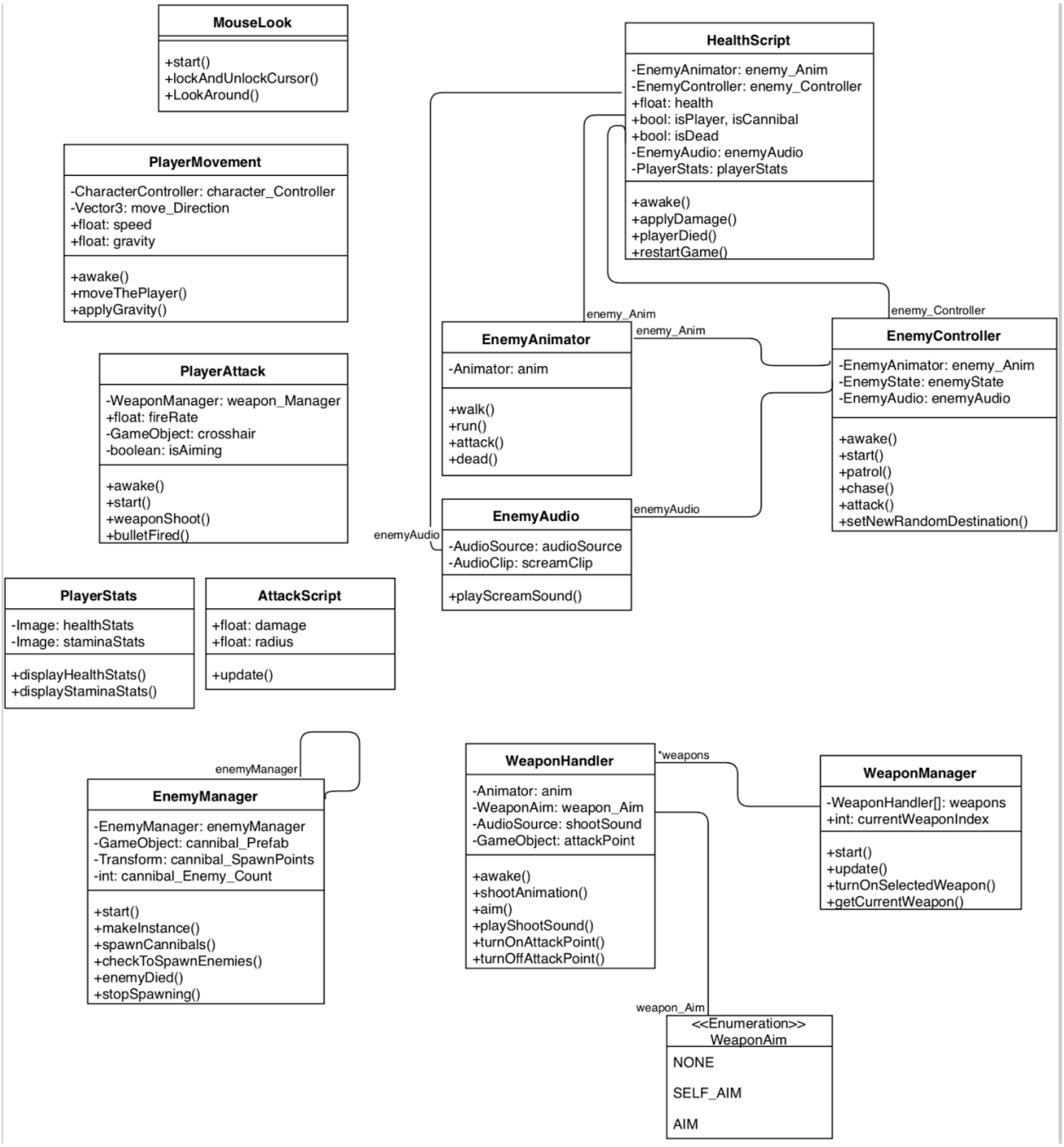
Final Class Diagram and Comparison Statement

- **Final UML Class Diagram is Below:**
    - An explanation of the color scheme is on the page after.


MouseLook
+start()
+lockAndUnlockCursor()
+LookAround()
HealthScript
-EnemyAnimator: enemy_Anim
-EnemyController: enemy_Controller
+float: health
+bool: isPlayer, isCannibal
+bool: isDead
-EnemyAudio: enemyAudio
-PlayerStats: playerStats
+awake()
+applyDamage()
+playerDied()
+restartGame()
PlayerMovement
-CharacterController: character_Controller
-Vector3: move_Direction
+float: speed
+float: gravity
+awake()
+moveThePlayer()
+applyGravity()
enemy_Controller
State
EnemyController
-EnemyAnimator: enemy_Anim
-EnemyState: enemyState
-EnemyAudio: enemyAudio
+awake()
+start()
+patrol()
+chase()
+attack()
+setNewRandomDestination()
enemy_Anim
enemy_Anim
EnemyAnimator
-Animator: anim
+walk()
+run()
+attack()
+dead()
PlayerAttack
-WeaponManager: weapon_Manager
+float: fireRate
-GameObject: crosshair
-boolean: isAiming
+awake()
+start()
+weaponShoot()
+bulletFired()
enemyAudio
enemyAudio
EnemyAudio
-AudioSource: audioSource
-AudioClip: screamClip
+playScreamSound()
<<Enumeration>>
EnemyState
PATROL
CHASE
ATTACK
PlayerStats
-Image: healthStats
-Image: staminaStats
+displayHealthStats()
+displayStaminaStats()
AttackScript
+float: damage
+float: radius
+update()
MainMenu
+PlayGame()
+QuitGame()
PauseMenu
+ bool: gameIsPaused
+GameObject: pauseMenuUI
+GameObject: crossUI
+Start()
+Update()
+Resume()
+Pause()
Factory
Singleton
enemyManager
GetNames
+ Text: inputText
+ string: tutorialText
+ Start()
SaveNames
+InputField: inputText
+string: username
+string: tutorialText
+Start()
+save()
EnemyManager
-EnemyManager: enemyManager
-GameObject: cannibal_Prefab
-Transform: cannibal_SpawnPoints
-int: cannibal_Enemy_Count
+start()
+makeInstance()
+spawnCannibals()
+checkToSpawnEnemies()
+enemyDied()
+stopSpawning()
ScoreBoardScript
+ Text: currentScore
+Start()
+LoadMenu()
+QuitGame()
Factory
CannibalFactory
- GameObject cannivalPrefa
+ createCannibal()
+ setPrefab()
ScoreScript
+ Text: scoreUI
+ int: score
+Start()
+Update()
+increaseScore()
+increaseScoreForCannibal
WeaponHandler
-Animator: anim
-WeaponAim: weapon_Aim
-AudioSource: shootSound
-GameObject: attackPoint
+awake()
+shootAnimation()
+aim()
+playShootSound()
+turnOnAttackPoint()
+turnOffAttackPoint()
*weapons
WeaponManager
-WeaponHandler[]: weapons
+int: currentWeaponIndex
+start()
+update()
+turnOnSelectedWeapon()
+getCurrentWeapon()
weapon_Aim
<<Enumeration>>
WeaponAim
NONE
SELF_AIM
AIM

- **Color Scheme:**
  - Classes with bold **Green** text next to them list patterns that we tried to implement, and that made it into the final design
  - Classes with bold **Red** text next to them denote places where we tried to implement a pattern but ended up not using it.
  - Classes which are fully highlighted in green denote classes that we have created entirely on our own
  - Classes which are partially highlighted in green denote places were we have adapted, changed, and/or added code from the Unity FPS tutorial we list below.
- **Patterns:**
  - Many patterns that we tried to implement did not make it into our final design. This is discussed further in the OOAD Process Statement further down in the PDF.
  - The **singleton pattern** is used in the creation of the EnemyManager class to allow other scripts to access it safely from outside. This needed to be done in order to make sure that there is only one EnemyManager spawning/tracking enemies and was recommended by the Unity tutorial we followed.
  - We tried to apply the **state pattern** to the EnemyController class. We ended up taking out the sate pattern because it did not work well with the built in Unity MonoBehavior that we needed to inherit from.
  - We tried to apply a **factory pattern** to the EnemyManager class in order to decouple the creation of enemies. For similar reasons as the state pattern, the factory pattern did not play well with what Unity wanted us to do with the instantiation of new enemy objects.
- **Key changes since projects 4 and 5**
  - Implemented a custom island map using assets from the free unity store
  - Simplified the game to include just 3 weapons and 1 type of enemy.
  - Implemented menus and the proper flow/ transitions between them
    - Main menu
    - Pause menu
    - End game/score menu
  - Implemented a score which increases based on time
  - Increased enemy spawn rates to become more frequent as the game progresses.
  - Attempted to add a factory pattern to supplement the creation of enemies in the EnemyManager class. (Even though this was not included in the final design, our attempted pattern design is still present in the code but commented out)
  - Attempted to change the enemyController to use a State pattern.
  - Located the use of a singleton pattern within EnemyManager.

- **The UML class diagram from project is 5 below:**



Third-Party Code vs Original Code Statement

- Tools: All code was done in Unity
- Most of our code is adapted from the following tutorial
    - https://github.com/beaucarnes/unity_fps

    - See the above color code description and UML diagram for specifics on which files are original, modified, or taken from the tutorial.
- We consulted several sites to help us with Unity questions. They are listed below:
    - https://answers.unity.com/questions/964518/passing-arguments-into-a-constructor-via-addcompon.html
    - https://docs.unity3d.com/ScriptReference/Time-deltaTime.html
    - https://docs.unity3d.com/ScriptReference/PlayerPrefs.html
    - https://www.youtube.com/watch?v=iAbaqGYdnyI
    - https://www.youtube.com/watch?v=FGVkio4bnPQ

OOAD Process Statement

- Design: Because Unity utilizes the concepts of scenes and scripts, working around those to implement object oriented principles was difficult because there were some behaviors that couldn't be avoided. In trying to design using patterns, we quickly realized that there were certain structures/ code patterns that Unity was strongly trying to get us to use. For example, classes which inherit from Monobehaviour "should not" have constructors. Monobehaviour is needed however if you wish to connect to certain built-in Unity functionalities related to creating new game objects, which caused us to decide to leave out our use of the factory pattern.
- Research: It was important for us to research Unity beforehand because Unity already utilizes object oriented principles related to their available classes. These classes (the most important being Monobehaviour) all implement "pattern like" structures, but are not quite exactly what we have discussed in class. An example is our implementation of the "healthScript" class which is very close to being considered an observer pattern with how it deals with sending messages to other classes.
- Additionally, it was important for us to draw from tutorials in Unity that implemented the changes we wanted to make so that we didn't reinvent the wheel. These tutorials provided a foundation for understanding to help us understand some of the more complex components and gave us a starting point for us to be able to create our own behaviors and functionality.