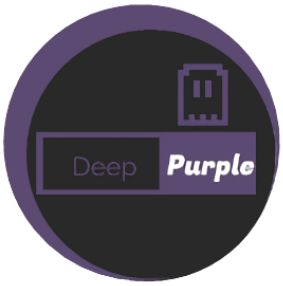


Deep Purple: Galactic Gladiator

Developers Manual

Volume 1:
Overview

April 2019

1.0 Table of Contents

1.1 Team and Roles

Team Member	Role
Michael	IT Manager
Zach	Software Architect
Robert	QA Manager
Kyle	Project Manager
Oshan	Coding Standards
Gabriel	Documentation Specialist

1.3 Notice

This document contains information about the source code for the game Galactic Gladiators. This information will change, but is current as of the date April 4, 2019. Galactic Gladiators is programmed in **Unity version: 2017.4.21f1** and we cannot ensure that it will run properly on any other version of Unity. To download Unity visit: https://docs.unity3d.com/Manual/GettingStartedInstallingHub.html

The information about the source code, does not include all information about all of the scripts. Only the most important scripts, and the most important information. For more detail the source code is well commented and provides more complex information for how the systems works, and or, talk to another system of the game. This document, however, will explain very thoroughly how these scripts work. The reader should come to an understanding of how the game is set up after reading through this document in its entirety.

1.2 Context Diagram

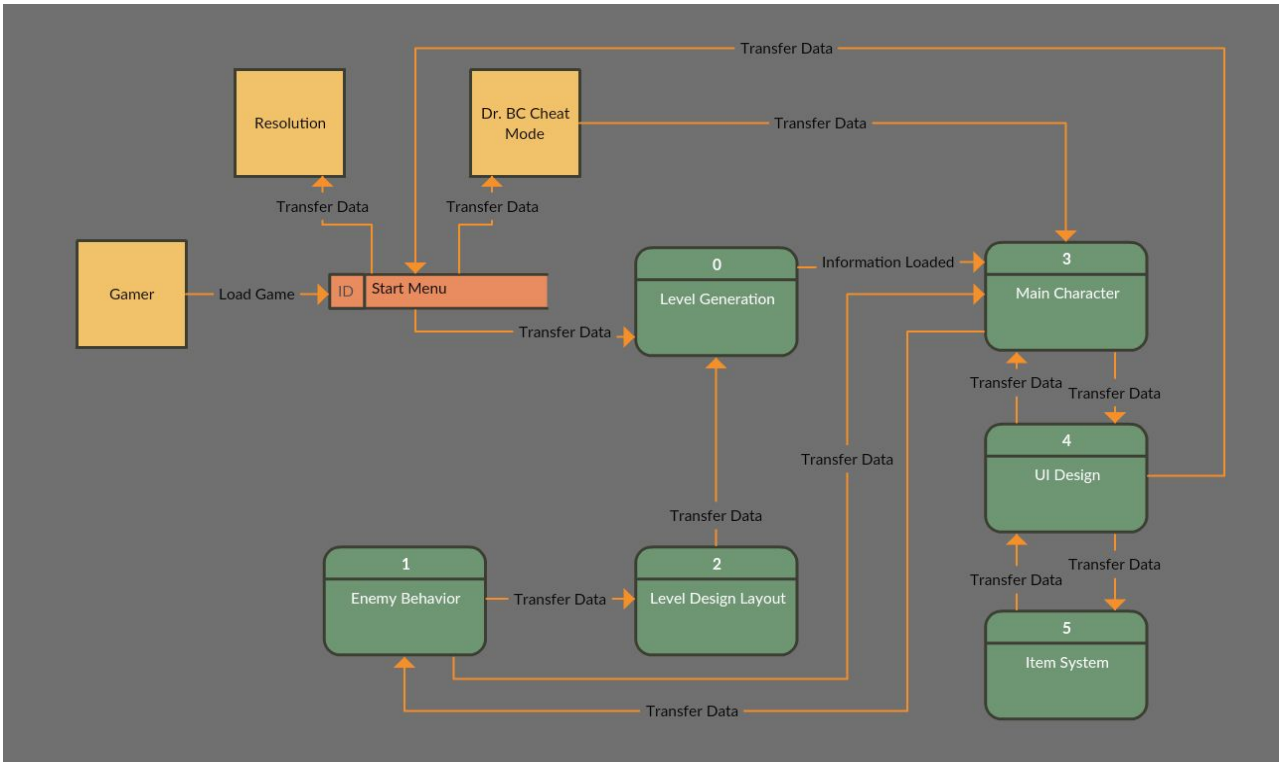
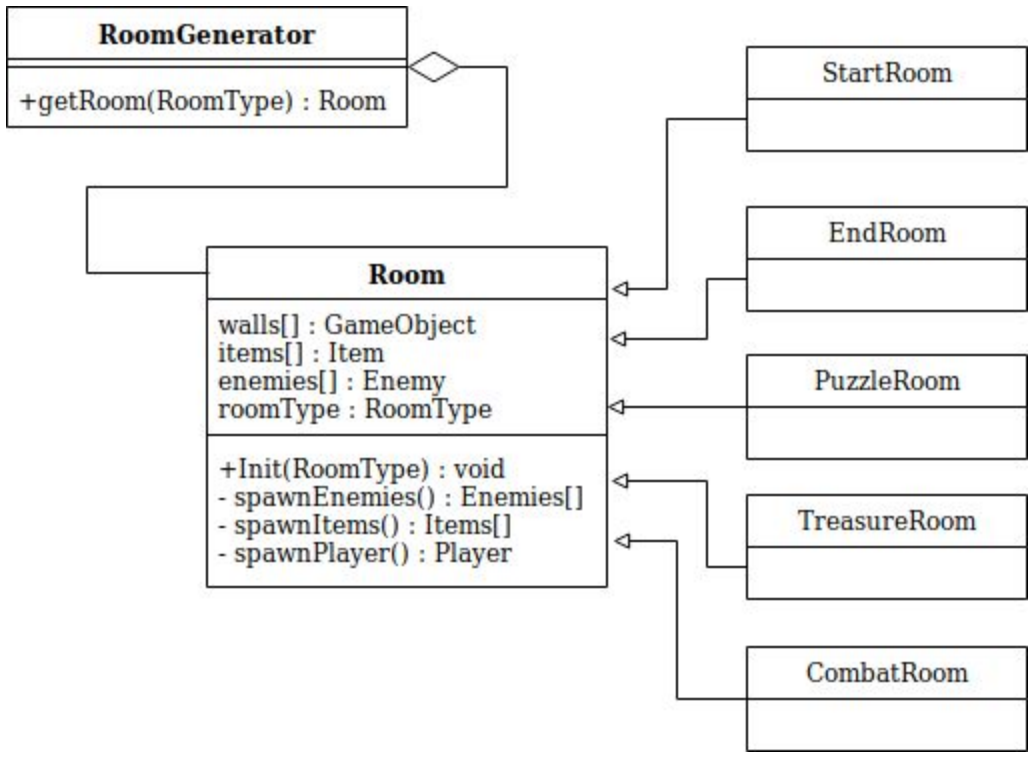


Diagram	Topic	Programmer
0:	Level Generation	(Zack)
1:	Enemy Behavior	(Robert)
2:	Level Design Layout	(Michael)
3:	Main Character	(Oshan)
4:	UI Design	(Gabriel)
5:	Item System	(Kyle)

1.3 Michael's Code

Class Diagram:



Description:
Every Room in Galactic Gladiators is randomly generated. The layout for a basic room is a square, in which it can be stretched to fit the desired room size. Inside of this room walls will be randomly generated, which the complexity in which these walls are generated can be modified to make more interesting rooms. However, the more complex, the higher the chance there will be places in the room which cannot be reached.

Going on to the different types of rooms that can be generated there is, StartRoom, EndRoom, PuzzleRoom, TreasureRoom, and CombatRoom.

StartRoom

- The start room will be the room in which the player will initially spawn into.

EndRoom

- The end room is where the boss will spawn, which the player needs to kill in order to beat the game.

PuzzleRoom

- Puzzle rooms, are rooms where the player needs to solve a puzzle, which there are currently two. The doors are locked once you enter into these rooms, and open once you solve the puzzle.
- The first puzzle is there is a box in which the player must push it to a marked location to beat the room.
- The second puzzle is there are rabbits that spawn, and the player must catch the pink rabbit among the white rabbits in order to beat the room.

TreasureRoom

- Treasure rooms will spawn a chest inside, and once the player comes close enough to the front of this chest, the chest will open and spawn a random item for the player.

CombatRoom

- Combat rooms are rooms in which, spawn points will be set randomly and a specified enemy will spawn in this room. Once the player enters the room and gets close enough the enemy will then chase after, and try to kill the player.

RoomGenerator.cs

RoomGenerator is more like RoomManager; Keeps a reference to every object I have to instantiate (floor, walls, doors, objects, lights, etc), a List of every Room object, defines Room Types,
contains a Get function to create a Room object, and a BuildDoors function which connects bordering rooms then builds walls around each room. Also has a get/set function for room size and room coordinates, but those also exist in Room.cs and should be used instead to get/set for individual rooms, not from here.

```
public static GameObject Get(Vector3 Zero, RoomType rt = RoomType.None)
```

Declares an empty gameobject as a room, and attaches specific room type as component.

```
public static void BuildDoors()
```

checks each Room's box collider for nearby box colliders attached to other rooms, finds overlapping wall sections and puts Door objects there. Finds where the room edges overlap, puts a Doorway there. Finally it calls the BuildWall function, which puts up walls between all the doors. I'm adding doors to two different rooms, so i have to do DoorList.Add on the correct room.

After all the doorways between rooms are found, build the walls and turn on the lights.

Room.cs

Base class for Room. The Init function creates the floor and ceiling and adds a collider trigger to each room (for detecting player entering/exiting). Contains a SetLighting function, for setting color and intensity of room lights; OnTriggerExit/onTriggerEnter which toggles lights when player enters, and toggles boolean variable PlayerInRoom; GetWalls/BuildWall, builds walls between each corner, GetInnerWalls which builds the inner walls, and Decorate which adds objects (including teleporters). and get/set for room size and zero coordinate.

```
public void Init()
```

Builds Floor and Ceiling.
Box Collider tells me when player enters or exits a room; also gets bordering rooms.

```
public void SetLighting
private void OnTriggerEnter
private void OnTriggerExit
```

turn lights on or off when player enters or leaves.

```
public void GetWalls()
```

call BuildWall for each outer wall.
Finds a random point along outer wall, and finds point on wall directly opposite. Enter recursive function GetInnerWalls

```
public void Decorate()
```

this function just makes it look prettier (adds random objects like computer screen). and adds teleporters.

```
public void BuildWall
```

Build a wall between start and end, if doors == true then search for doors and build walls between them. Start at wall's starting location; increase length of wall segment(segmentEnd -> segmentStart) until hits a door, or hits wallEnd. build wall there, start again.

```
public void GetInnerWalls(Vector3 start, RaycastHit endHit, int depth)
```

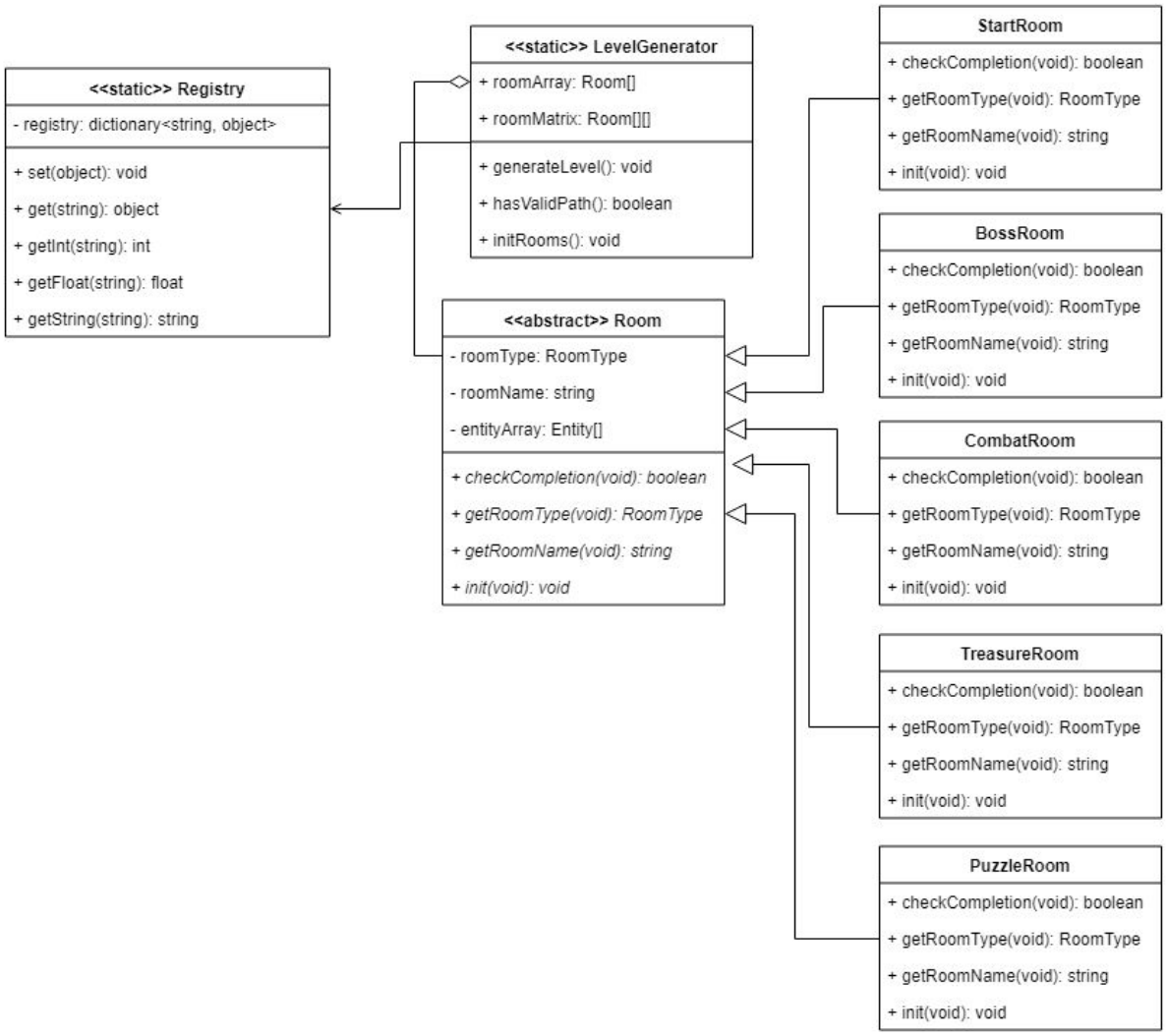
Build wall between start and endHit, then find a random point between the two, increase recursion depth variable, and start again.

All C# Files

./StartRoom.cs
./PuzzleThree.cs
./Room.cs
./PuzzleTwo.cs
./PuzzleZero.cs
./OpenDoor.cs
./PuzzleRoom.cs
./Robot.cs
./CombatRoom.cs
./TreasureRoom.cs
./WallTransparency.cs
./PuzzleFour.cs
./BossRoom.cs
./RoomGenerator.cs
./Teleporter.cs
./PuzzleOne.cs
./TestCase3.cs

1.4 Zach's Code

Class Diagram:



Description:
Galactic Gladiators main component that makes it interesting is the random generation. This random generation for generating how the rooms are placed next to each other uses fractals in order to achieve a different experience each playthrough. This generation uses the rooms that Michael programmed and created for the level design. As the generation does not work, unless Michaels code is working.

LevelGenerator.cs

This level generation scheme uses an (x, y) integer vector space, denoted "V", where each coordinate is linearly transformed into the (x, y, z) float vector space, denote "W". We are doing this from a top down perspective, so the actual transformation, denoted "T", T : V -> W is written in the form (roomSize * x, 0, roomSize * y). You will notice that the y coordinate in vector space W is always 0, because that y coordinate represents vertical height and we are only generating a flat grid for each room position.

```
Dictionary<Vector2Int, RoomGenerator.RoomType> RoomGeneratorRandom()
```

The random room generator generates the level without any type of design in mind. We begin at the start room coordinates (0, 0) and then begin the generation loop from there. Each loop, we select a random coordinate from the dictionary and move in a random direction (North, East, South, West). We then check if there is already a room at that location. If there is no room at that location, then add that room to the dictionary. If there is a room at that location, skip over that and try again. This system has the potential to generate very odd looking room layouts, but has a
tendency to cluster all of the rooms in a reasonably square like structure.

```
Dictionary<Vector2Int, RoomGenerator.RoomType> RoomGeneratorTFractal()
```

The TFractal room generator creates the level based on a recursive fractal algorithm. The fractal pattern is loosely based off of the T-branching fractal which is sometimes referred to as an H tree. The most basic form of the algorithm starts with a line segment of some length. Then draw two shorter line segments at right angles to the first line segment through the endpoint. Recursively apply this algorithm to create higher and higher levels. This algorithm has been modified and uses random number generation to create a variety of room patterns.

Identify where boss room should be located. This is done by attaching the boss room to a random room of degree one for the purposes of this particular function, degree refers to the number of rooms surrounding the randomly selected room. We include rooms that are one diagonal move away as well. The reason we do all of this is to ensure that the boss room always ends up at the end of a corridor. We want the player to be able to reach every possible room in the level without fighting the boss until they're ready. bossRoomLocs stores every coordinate where a boss room could be located.

```
public void TFractalRecursive
```

This function implements the recursive fractal algorithm. The pattern is loosely based off of the T-branching fractal which is sometimes referred to as an H tree. Our implementation is far more random. The most basic form of the algorithm starts with a line segment of some length. Then draw two shorter line segments at right angles to the first line segment through the endpoint. Recursively apply this algorithm to create higher and higher levels. This algorithm has been modified and uses random number generation to create a variety of room patterns. What follows is an explanation of our algorithm: Creating a line segment with the length parameter in the direction of direction parameter. Calculate the direction of west and east relative to the direction that the line segment was originally moving. For example, if the original line segment was moving west, turning right would leave you facing south and turning left would leave you facing north. A simple modulus calculation is done to determine new direction. Once the new directions have been calulated, recursively repeat these steps by moving left, forward, and right. The magnitude of each new line segment is calculated using direction multiplied by a random number between 0.6 and 0.9. This helps keep the branches long. This differs from the original T-branching fractal in that:
1) We also move forward in addition to left and right
2) We calculate each new length using a random number generator
Ultimately, this creates very interesting room layouts that tend to contain tunnels loops, corridors, and blocks all at the same time.

```
public RoomGenerator.RoomType RandomRoomType()
```

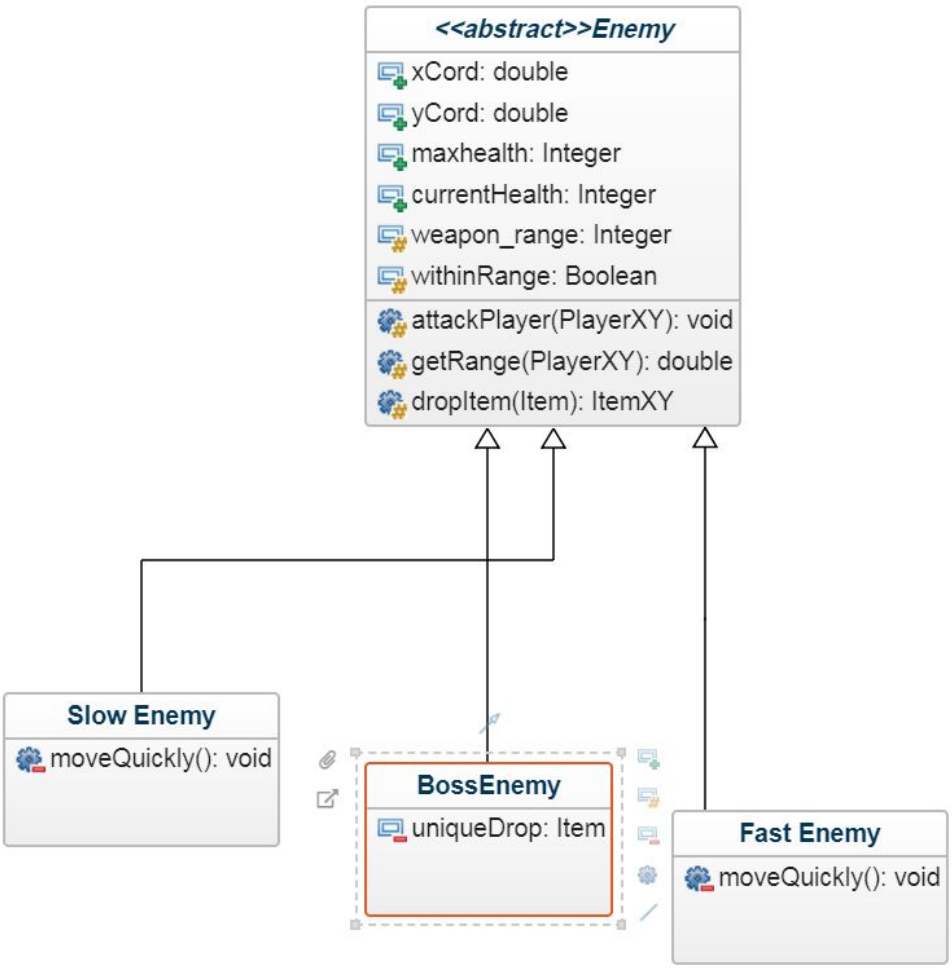
Use random number generator to return a random room type. The Random.Range function utilizes a uniformly distributed number generator so we can set the probabilities for each room type just by generating a random float between 0 and 1 and checking which range it falls into.

All C# Files

./LevelGenerator.cs
./Registry.cs

1.5 Robert's Code

Class Diagram:



Description:
The enemies in Galactic Gladiators are an important part of the game, as they add the challenge in which the player needs to overcome in order to advance. The enemies in the game chase after the player once the player gets into a certain range of the enemy. These enemies are used by the room generation in order to spawn enemies in specified points in the rooms.

BasicEnemy

Max distance an enemy will begin moving torward player
Min distance to player and enemy will get
Movement speed of enemy
Stores actions class for animations

```
void Update()
```

Gets the location of the player and moves the enemy towards the placer once the player is in the specified range.

```
public void takeDamage(DamageSource damageSource)
```

Whenever the enemy is hit the player will be dealt that much damage.

EnemyManager

Gets the spawn locations from the rooms in order to properly find where the enemy should be instantiated.

```
void Spawn()
```

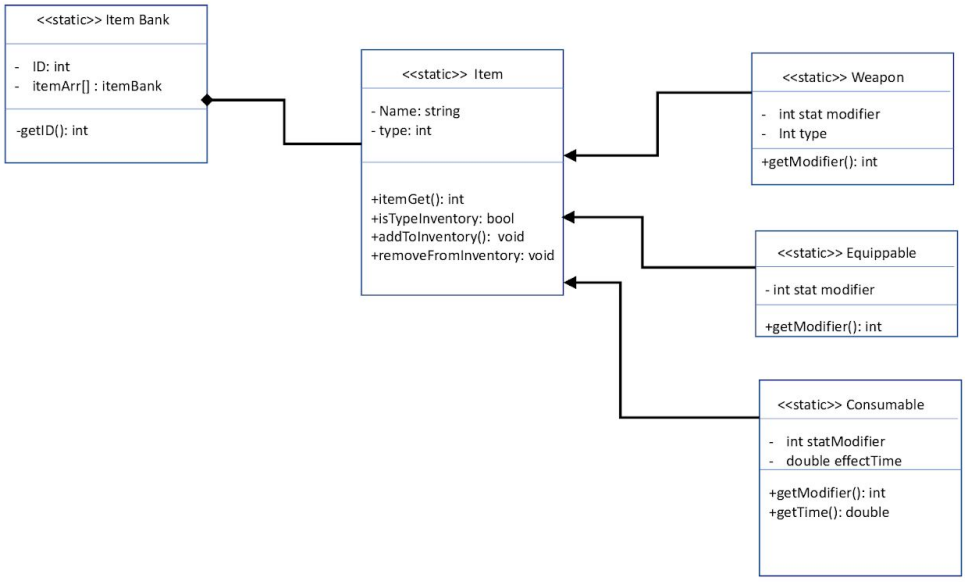
Handles the spawning of enemies. Sequentially goes through the list of spawn points to spawn the enemies.

All C# Files

./BasicEnemy.cs
./EnemyManager.cs

1.6 Kyle's Code

Class Diagram:



Description:
The item system for Galactic Gladiators adds a lot of needed depth to the game. There are three different type of items that will be added to the game. These are "Weapons", "Equippable", and "Consumables".

Weapons

- Are used by the character script of Oshan. These weapons change how the character attacks.

Equippable

- These are stat boosts for the character. They increase integer values attached the the character that affect damage, speed, and health.

Consumable

- Are used to increase health of the player, up to the players maximum set health.

BasicBullet.cs

Is the base in which all projectiles for weapons use.

```
private void OnTriggerEnter(Collider collider)
```

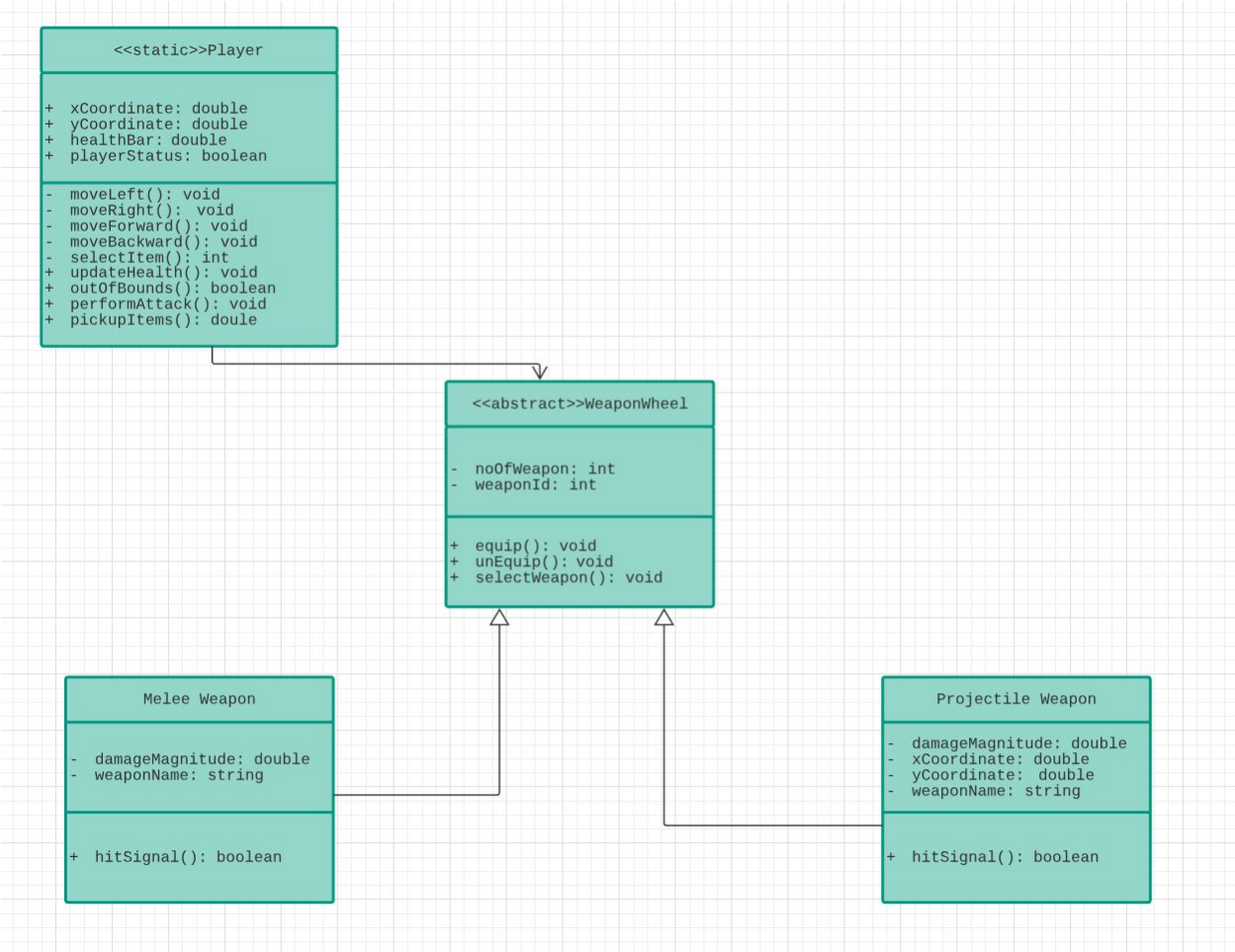
Instantiates a projectile that has information attached to it. This is the damage value and the damage type.

All C# Files

./DamageType.cs
./BasicBullet.cs
./IAttackable.cs
./DamageSource.cs

1.7 Oshan's Code

Class Diagram



Description

The character is the most important part of Galactic Gladiators, as it is how the player interacts with the game. The player has multiple values attached to it such as, health, speed, and damage. All of these are integer values and can be modified through items. The character also has movement attached to it that can be controlled through the "w","a","s","d", and "space" keys. Lastly the character can also attack. This is accomplished by the left mouse button. The attack changes based off of the weapon you currently have equipped. These weapons are from Kyles item system.

PlayerController.cs

The ClayerController is how the player directly interacts with the player character in the game.

```
void attack()
```

This fires a projectile which will change based on the weapon you currently have equipped.

```
public void takeDamage(DamageSource damageSource)
```

This function changes the health value of the player.

```
public int GetHealth()
```

This function returns the health value of the player, so that other players can interact with this value.

CameraController.cs

The CameraController is how the player gets to see what is happening in the game. Giving necessary information to the player.

```
void Update()
```

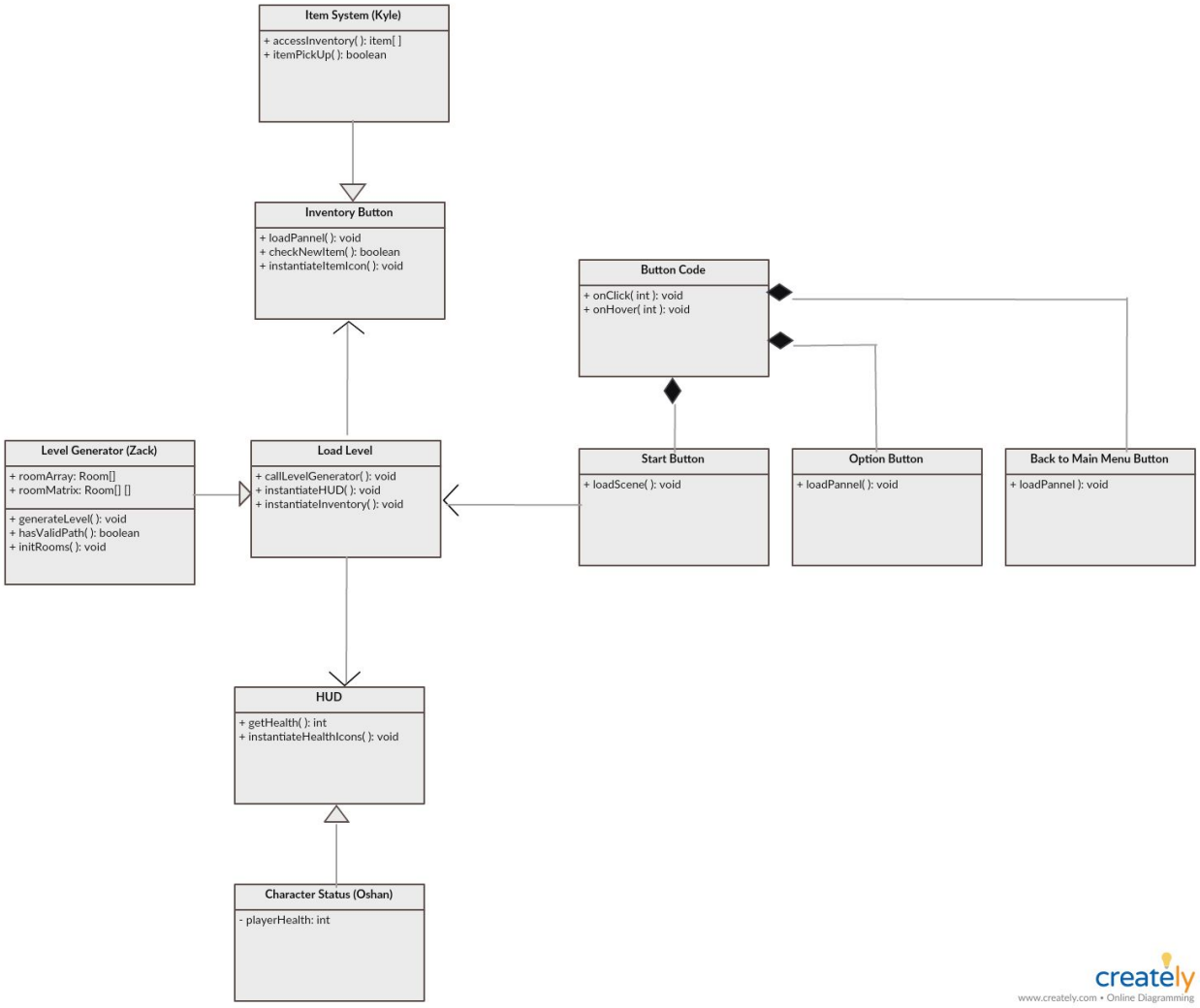
Smoothly adjusts the camera in order to follow the player and keep the player centered on screen.

All C# Files

./PlayerSpeed.cs
./PlayerHealth.cs
./CameraController.cs
./PlayerController.cs

1.8 Gabriel's Code

Class Diagram:



Description

The user interface is a very important part of the game as it displays critical information to the player, as well as is how the player can access the inventory of the player. The user interface also is responsible for how the player accesses menus. This includes accessing the option menu, quitting the game, starting the game, and choosing between first and third person.

As there has been a lot of communication between the item system and the user interface, in order to properly get the inventory working, there are scripts written here that do not directly pertain to the user interface, however, help with the item system. These scripts are the Item.cs script, ItemPickup.cs scripts, and Interactable.cs script.

Item.cs

Is a very basic script that is a data structure for items.
Contains

- Name
- Icon Sprite
- Type of Item
- Model for Item

```
public virtual void Use()
```

Is meant to be overwritten by the item system. This is how every item works when called in the inventory.

Interactable.cs

Interactable can be attached to a gameobject, and then will display a meshed sphere where player must enter in order to interact with that object. Contains adjustable values to change the size, and where the sphere is placed.

```
public virtual void Interact()
```

Is meant to be overwritten to change what happens once the player is in the range of the item to interact with it.

```
void OnDrawGizmosSelected()
```

Draws the sphere to visually see where the interact radius is.

ItemPickup.cs

Picks up an item from the item system.

```
public override void Interact()
```

Overrides the Interact function from the Interact.cs script to call the pickUp function.

```
void PickUp()
```

The item is picked up and added to the inventory. After the game object is then deleted.

Inventory.cs

Is the base for the Inventory to be built from. Creates a list of items for the inventory to store.

```
public bool Add(Item item)
```

Checks the type of item, and then applies the proper logic to process that item.

```
public void Remove (Item item)
```

Removes an item from your inventory, which removes it from the list.

InventoryUI.cs

Shows the visuals for the entirety of the UI.

```
void Start ()
```

Initializes the whole inventory

```
void UpdateUI ()
```

Updates when an item has been added/removed to the inventory.

InventorySlot.cs

The individual slots for the inventory

```
public void UseItem()
```

Calls the use function, where the item system handles the logic for what the item does.

```
public void AddItem(Item newItem)
```

Adds the item to the visuals of the inventory.

```
public void ClearSlot()
```

Removes the item from the visuals of the inventory.

```
public void OnRemoveButton()
```

Drops the gameobject from the inventory.

All C# Files

./ItemPickup.cs
./Interactable.cs
./InventoryUI.cs
./MainMenu.cs
./TESTING_SPAWNER.cs
./Item.cs
./ChestOpening.cs
./TESTING_PASSFAIL.cs
./InventorySlot.cs
./Main_Menu_Script.cs
./Inventory.cs
./NameGenerator.cs