**Software Design Specification**

Video Game “Delta Dungeon”

**Version 1.0 approved**

**Prepared by Team Delta**

**Steven DeMartini  
Eric Pacheco  
Joshua Peckham  
Abhishek Rajbhandari  
Jackson Wise**

**Revision History:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Revsion** | **Description** | **Autor** |
| 11/14/17 | 1.0.0 | intro, initial draft | Eric Pacheco,  Abhishek Rajbhandari,  Jackson Wise,  Joshua Peckham  Steven Demartini |
| 11/16/17 | 1.0.1 | Outline,System Overview | Eric Pacheco,  Abhishek Rajbhandari,  Jackson Wise,  Joshua Peckham  Steven Demartini |
| 11/21/17 | 1.0.2 | System Overview | Eric Pacheco,  Abhishek Rajbhandari,  Jackson Wise,  Joshua Peckham  Steven Demartini |
| 11/24/17 | 1.0.3 | Class scripts, variables and methods | Eric Pacheco,  Abhishek Rajbhandari,  Jackson Wise,  Joshua Peckham  Steven Demartini |
| 11/25/17 | 1.0.4 | Scope, Glossary, System Architecture | Eric Pacheco,  Abhishek Rajbhandari,  Jackson Wise,  Joshua Peckham  Steven Demartini |
| 11/25/17 | 1.0.5 | System Architecture, Detailed System Design | Eric Pacheco,  Abhishek Rajbhandari,  Jackson Wise,  Joshua Peckham  Steven Demartini |
| 11/26/17 | 1.0.6 | System Architecture, Detailed System Design | Eric Pacheco,  Abhishek Rajbhandari,  Jackson Wise,  Joshua Peckham  Steven Demartini |

**Table of Contents**

[**1.**](#_3as4poj) **INTRODUCTION** 4

[1.1.](#_1pxezwc) Document Outline 4

[1.2.](#_49x2ik5) Document Description 7

[*1.2.1.*](#_3znysh7) *Introduction* 7

[*1.2.2.*](#_2p2csry) *System Overview* 7

[**2.**](#_147n2zr) **DESIGN CONSIDERATIONS** 11

[2.1.](#_3o7alnk) Assumptions and Dependencies 11

[2.2.](#_23ckvvd) General Constraints 11

[2.3.](#_ihv636) Goals and Guidelines 11

[2.4.](#_32hioqz) DeveloReods 11

[**3.**](#_1hmsyys) **ARCHITECTURAL STRATEGIES** 12

[**4.**](#_41mghml) **SYSTEM ARCHITECTURE** 15

[**5.**](#_vx1227) **POLICIES AND TACTICS** 26

[**6.**](#_3fwokq0) **DETAILED SYSTEM DESIGN** 26

[6.1.](#_1v1yuxt) Classification 9

[6.2.](#_4f1mdlm) Definition 9

[6.3.](#_2u6wntf) Responsibilities 10

[6.4.](#_19c6y18) Constraints 10

[6.5.](#_3tbugp1) Composition 10

[6.6.](#_28h4qwu) Uses/Interactions 10

[6.7.](#_nmf14n) Resources 10

[6.8.](#_37m2jsg) Processing 10

[6.9.](#_1mrcu09) Interface/Exports 11

[6.10.](#_46r0co2) Detailed Subsystem Design 11

[**7.**](#_2lwamvv) **GLOSSARY** 68

[**8.**](#_111kx3o) **BIBLIOGRAPHY** 69

# Introduction

* Purpose: The Following Software Design Specification gives a general explanation of the “Delta Dungeon” video game Software. It will go over the outline of the software. It will also run through the process of developing the game. The “Delta Dungeon” video game is a third person shooter game. It provides a top down view of the character as they explore several rooms of a dungeon.

## 1.1 *Document Outline*

The following is an outline of the Software Design Specifications. All of the components discussed are listed below.

* Introduction
* System Overview
* Design Considerations
  + Assumptions and Dependencies
  + General Constraints
  + Goals and Guidelines
  + Development Methods
* Architectural Strategies
  + Overview
  + Key Components
* System Architecture
  + UML of Software
  + Description of level Generation manager
  + Description of Character Generation
  + Description of special effects
* Policies and Tactics
  + Description of Policies and Tactics
* Detailed System Design
  + Game Manager
  + Boss Manager
  + Boss\_HS\_RoomTeleporter
  + Boss
  + ItemCreator
  + Pickups\_GunEffects
  + RoomGeneration
  + PauseMenu
  + CameraPerlinShake
  + GenericShake
  + EnemyManager
  + Fight
  + Hazard
  + LevelGenManualManager
  + LevelGenMaunal\_Abhi
  + LevelGenManual\_Eric
  + LevelGenManual\_Jackson
  + LevelGenManual\_Joshua
  + LevelGenManual\_Steven
  + LevelRoomLayout
  + PlayerManager
  + Projectile
  + MovementWave
  + Proj\_Grenade
  + Proj\_Mine
  + Proj\_PlasmaOrb
  + Respawn
  + TrackingMine
  + WeaponsManager
  + DeltaMusicManager
* Glossary
* Bibliography

**Scope:**

This Software Design Description document provides a detailed explanation and structure of the Delta Dungeon video game software. It also provides implementation details necessary to fit the need of the Software Requirements Specification. The following documentation shall be used in conjunction with other documents provided throughout the software development process of The Delta Dungeon video game software.

**References:**

Unity Official Documentation:

<https://docs.unity3d.com/Manual/index.html>

Unity Dev Assets:

<http://devassets.com/>

Our music box:

Royalty free music  
  
Unity Asset Store:  
https://www.assetstore.unity3d.com

### Methodology, Tools and Techniques

Unity (2017.2 .0F3 Personal) is the main tool used for the creation of this software, Google Drive is used to write documentation, Visual Studio 2107 Community edition and Mondo Develop 6.1 are used to write code for all of our scripts. Adobe Photoshop and GIMP are used to make sprite assets for the special effects.

## 1.2 *Document Description*

This document provides a general explanation of the development process for the “Delta Dungeon” Software. It identifies the required resources to use the software. This document also provides graphical layouts and explanation of key components that belong to the software. By providing all of the previously described information, this document will aid in producing test cases, changes and maintenance for the Delta Dungen software.

### 1.2.1 Introduction

The first Section of this document introduces the project. The next section explains the design considerations. These considerations include: Assumptions, General Constraints, Goals and Guidelines it also provides the Development Methods. In section 3.0 the Architectural Strategies are explained which provide design strategies that provide an overall organization of the software. Section 4.0 has to deal with the System Architecture. It gives a high level overview of the functionality of the system. Section 5.0 explains the policies and tactics of the Software. Section 6.0 is dedicated to Defining System Design which provides a more detailed explanation of the system architecture section.

### 1.2.2 System Overview

|  |  |
| --- | --- |
| Module | Description |
| Initiation | System powers on and loads opening screen |
| Opening Screen | Black screen transitions to the Team Delta logo.    From there to the Main Menu game prompt of the “TV” with options. This prompt will have the following buttons:   1. “Play” 2. Change the floor between floor 1 through 10 and boss level. 3. Modify difficulty level:    1. Easy    2. Moderate    3. Difficult 4. “Game Information” |
| Transition to Play | The game will fade through the TV screen and hover above the character. |
| Playing Initial level Generation | The map will be generated for the Player(5) and loaded on the screen.  For this generation, there will be 10 randomly selected floors for the regular game that are generated with one boss room teleporter. |
| Player Generation | The game will respond to the playable character on the screen that the user will be able to move in the directions mentioned in the SRS in section 3 part A User Interface.  Spawn:    Right Punch:    Left Punch:    Right Kick:    Left Kick:    Fire Weapon: |
| Enemy generation | The enemies will be deployed throughout the stage upon the activation of the level. |
| Level Completion | When the Player(5) defeats the boss (after activating the boss teleporter) and completes the level, the next floor will be generated and the process will start over. |

# 2. Design Considerations

## *Assumptions and Dependencies*

* Related software or hardware - Delta Dungeon shall not require any software for the game to run. The hardware that shall be advised to have are at least four gigabytes of ram and a modern desktop or laptop. Refer to SRS section 3(b) for further inquiries. If the user does not meet the hardware requirements the game shall not run at full potential
* Operating systems - Delta dungeon shall at a minimum run on Mac OS X 10.12 Sierra and Windows 10 Home version 1703. For more details please refer to SRS document section 3(c).
  + If software/hardware is below suggested recommendations game may not operate at its full capacity and may cause overheating and unexpected damage to your computer.
* Game will rely heavily on the software provided by Unity, meaning that all the functionality is expected to perform and as is it is illustrated by Unity.
* End-user characteristics

## *General Constraints*

* Hardware or software environment - The constraints on hardware and software can be referred back to the SRS documents on section 3.b and 3.c
* End-user environment - The constraints the user would need a computer which meets the hardware and software requirements of the game. Enough knowledge of the English language to operate the game (able to operate the Main Menu). Internet connection to download the game.
* Interoperability requirements - Delta Dungeon only requires an internet connection to be downloaded to a computer system.
* Security requirements (or other such regulations)- There are no security requirements for delta dungeon.
* Memory and other capacity limitations- Constraints regarding Memory for delta dungeon can be referred to in the SRS document in section 3.b.
* Time Limitation- as the course of design is approximately 10 weeks, there is little available time to become masters at C# and the ins and outs of Unity.
  + 4 of the people on this project are new to C# leaving a learning curve
    - Makes the majority of time becoming familiar with the basics
  + This will cause the animation and quality of the game to be less than hoped for. If there was more time to develop Team Delta could improve further upon all aspects of the game.

## *Goals and Guidelines*

All Delta dungeon members shall be able to read and process the diagram in order to create a similar or better solution.

Every detail of the system architecture shall be included in the document so that there are no ambiguities while designing the Delta dungeon software.

Use case diagrams shall be simple and easy to follow for the reader to track how the game works in the given use cases.

## *Development Methods*

The Delta dungeon team has incorporated a mixture of the Agile and Spiral development methods. The team has been building on a the basic idea of a dungeon crawler game. These development methods has given the team ability to add iterations and new ideas to the game as it is still in the developing phase. Multiple iterations of the game went through Spiral approach. Developing one piece then going back and redesigning the same piece occurred a lot.

# 3. Architectural Strategies

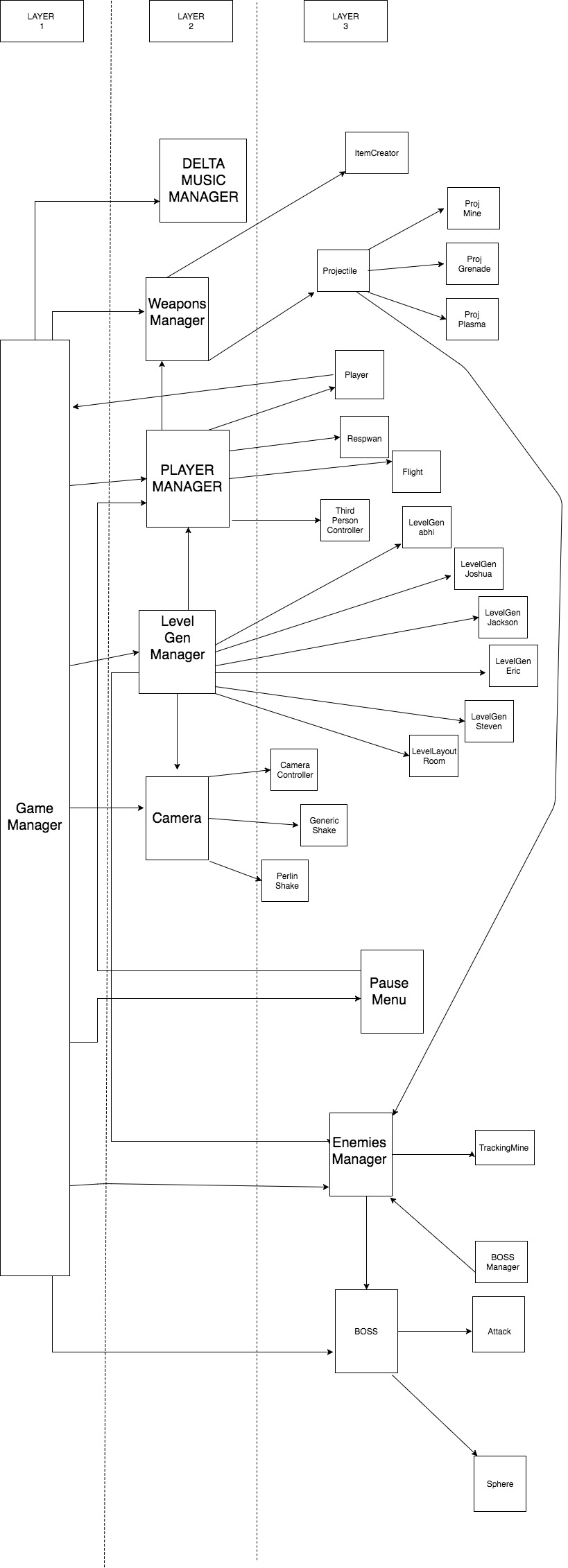
This game will be written in C# utilizing UnityEngine library and the basic architecture of that Unity Engine outlines, such as: built in libraries. Since C# is an Object-Oriented Programming language, it will be controlling the front, middle and back end of the game. This allows the game's loading screen script(1) to be a stand-alone document that will open the Player(5) options to start the game. Delta Dungeon will be broken down into multi-leveled scripts(1) of the game. Each of the different scripts will be interacting with one another in order to make the features(2) act in their respective ways.

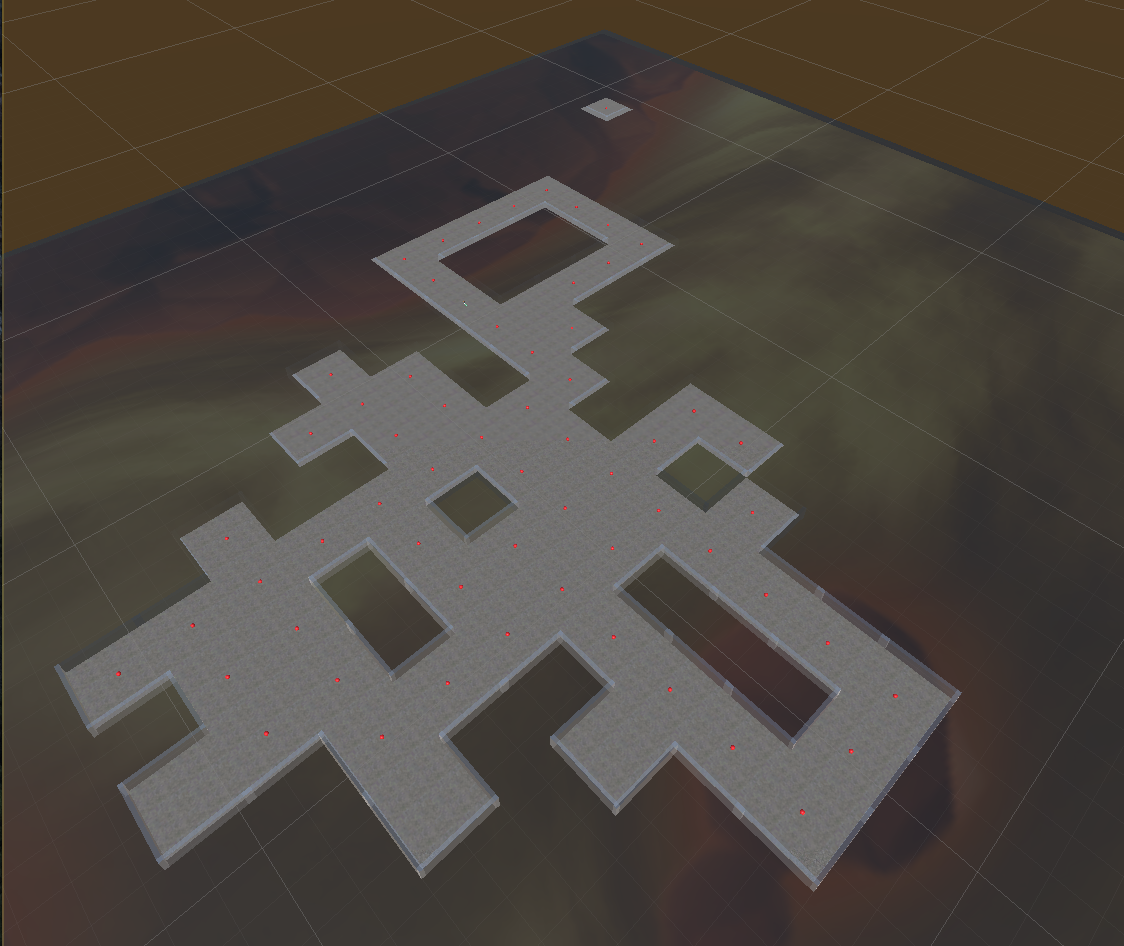
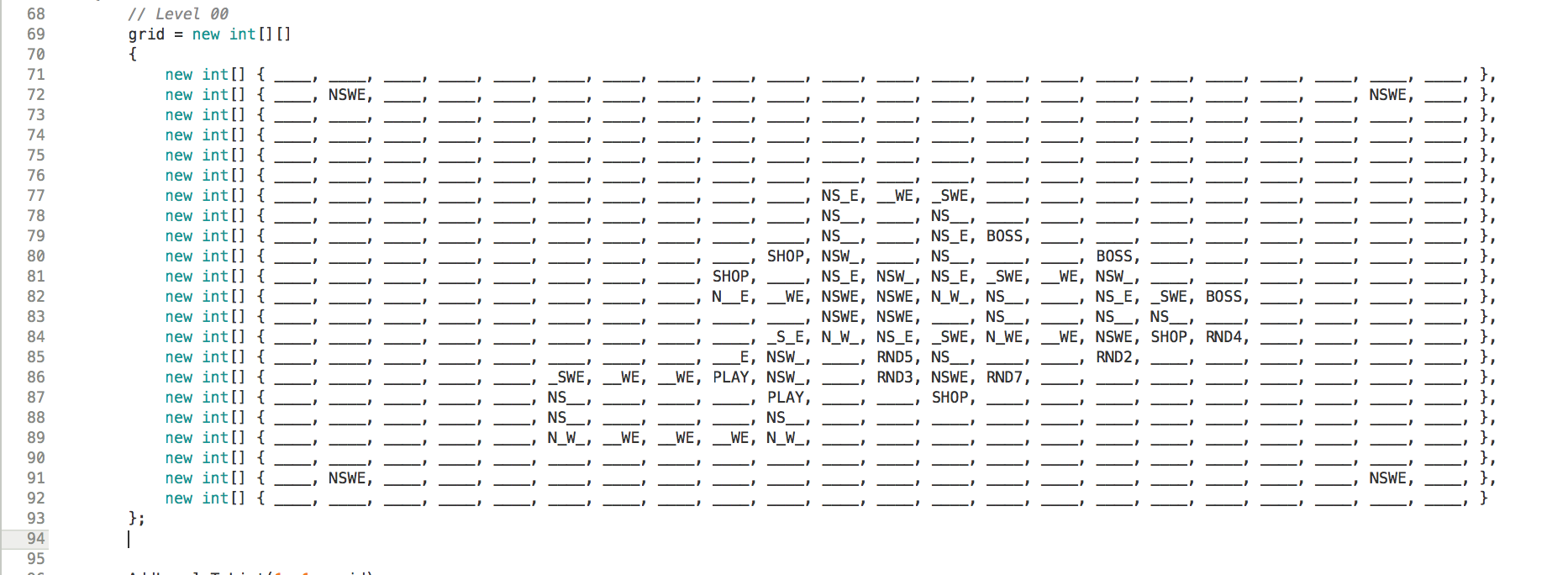
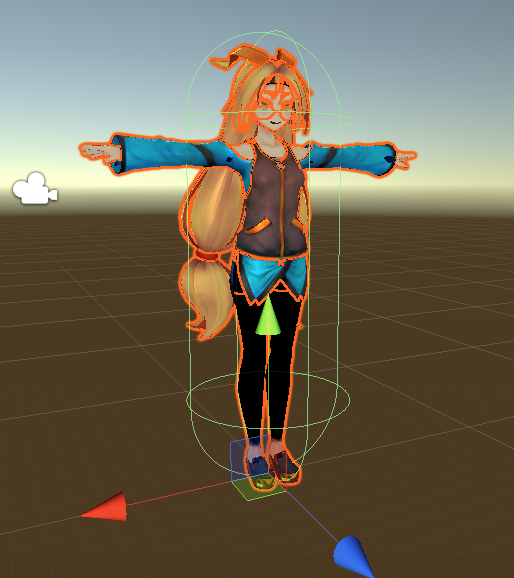
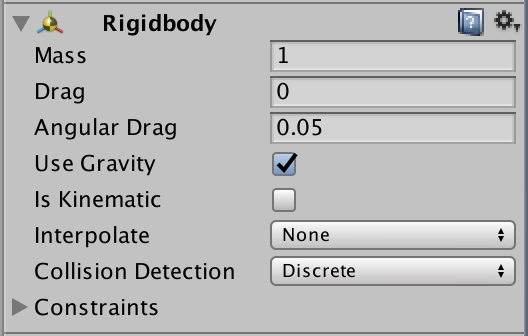
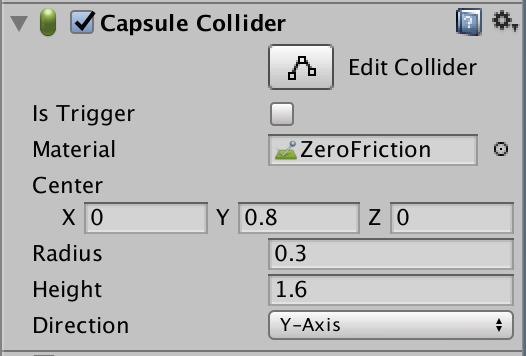
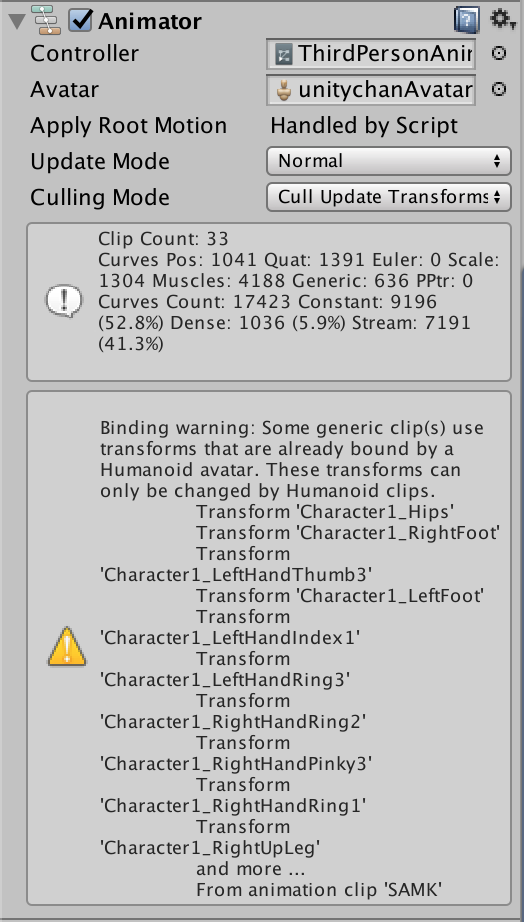
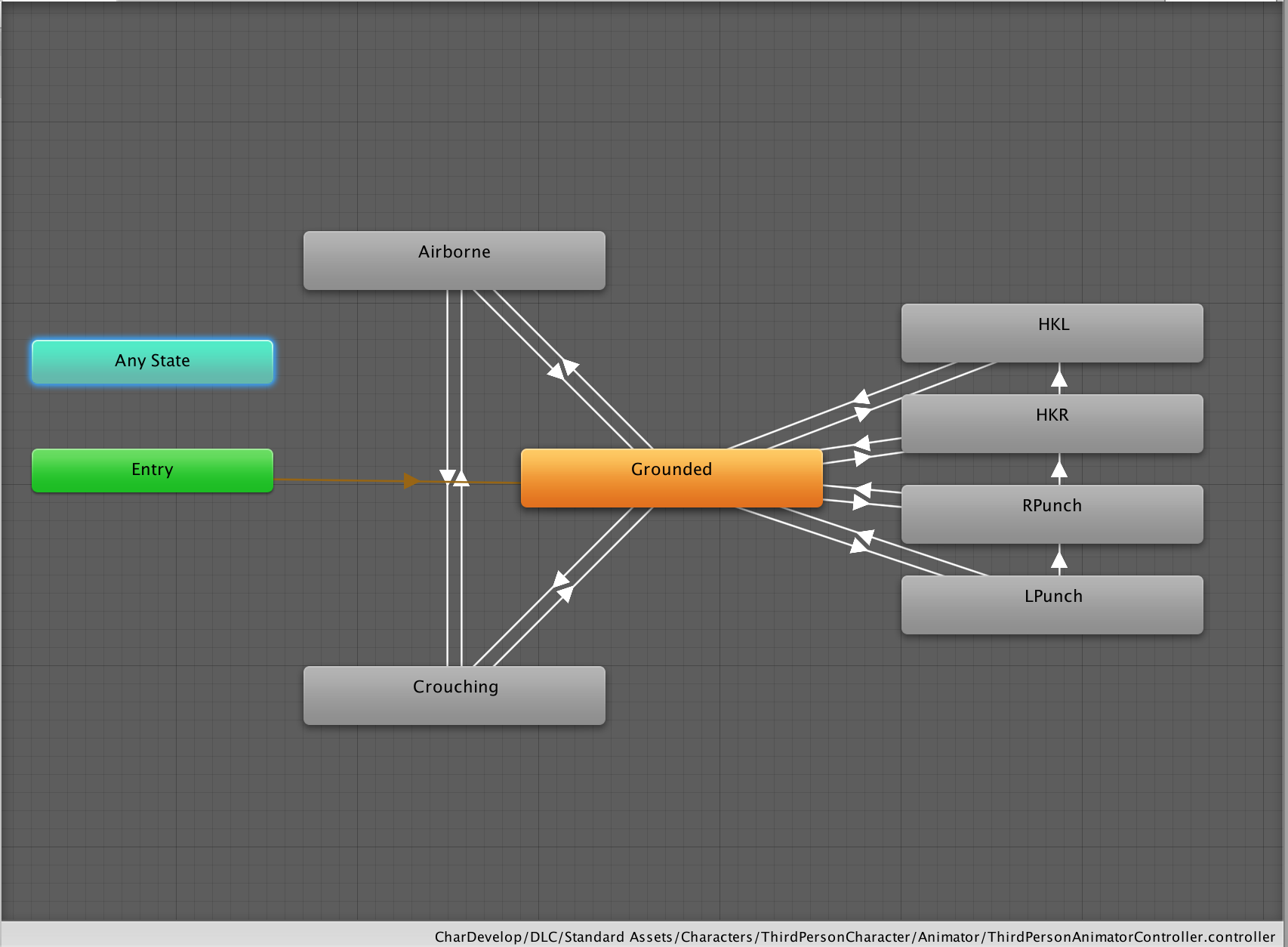
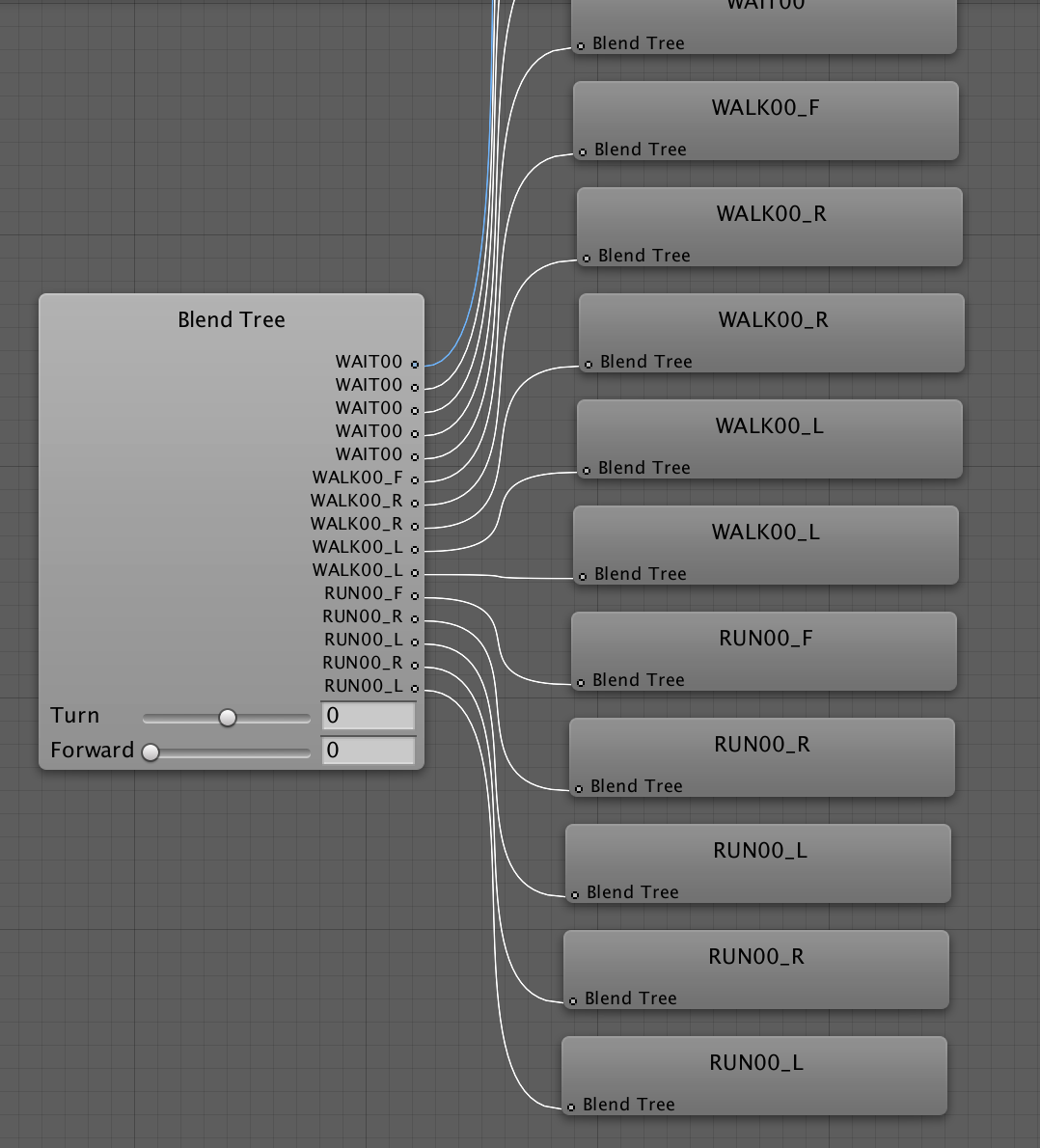
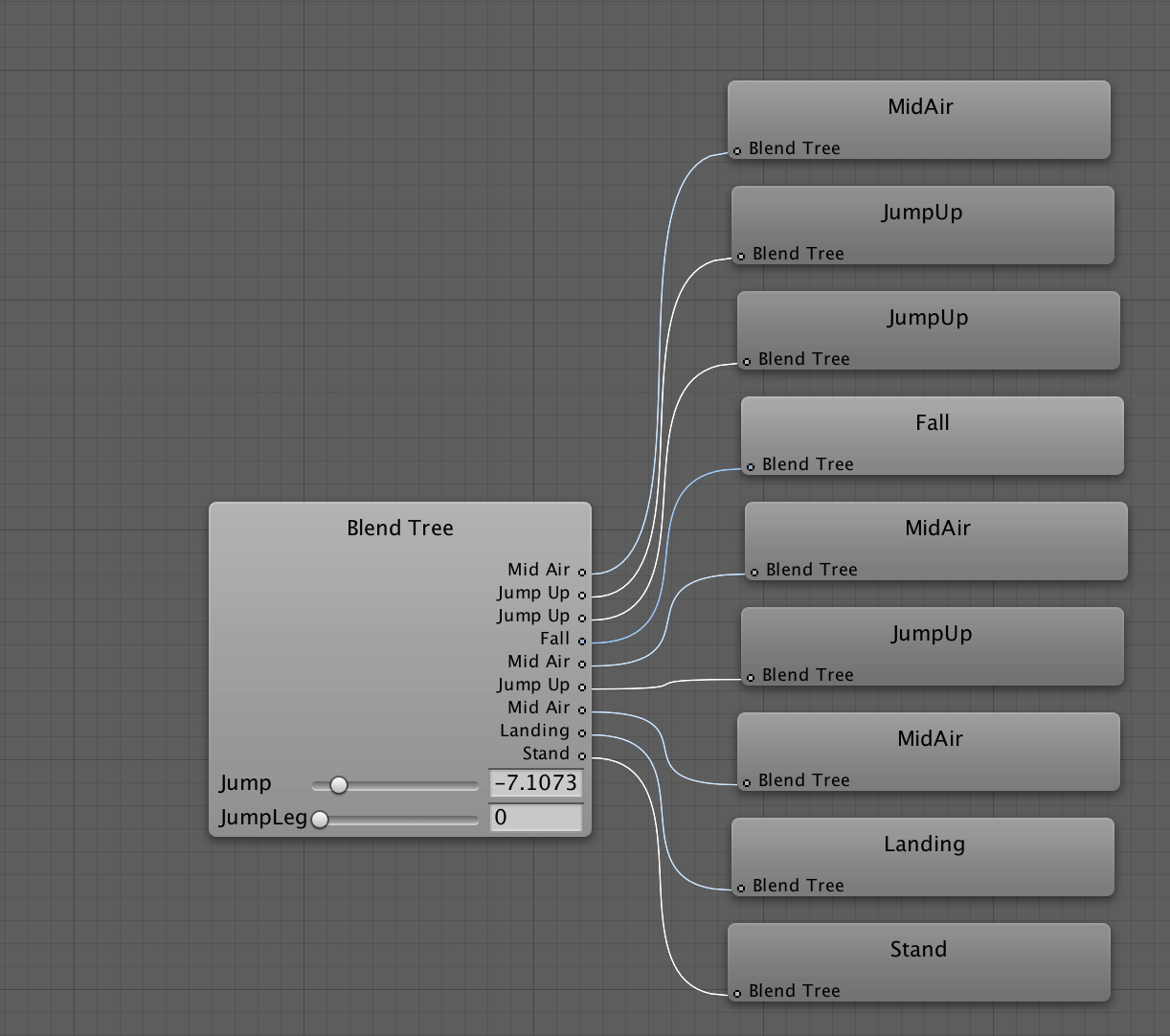
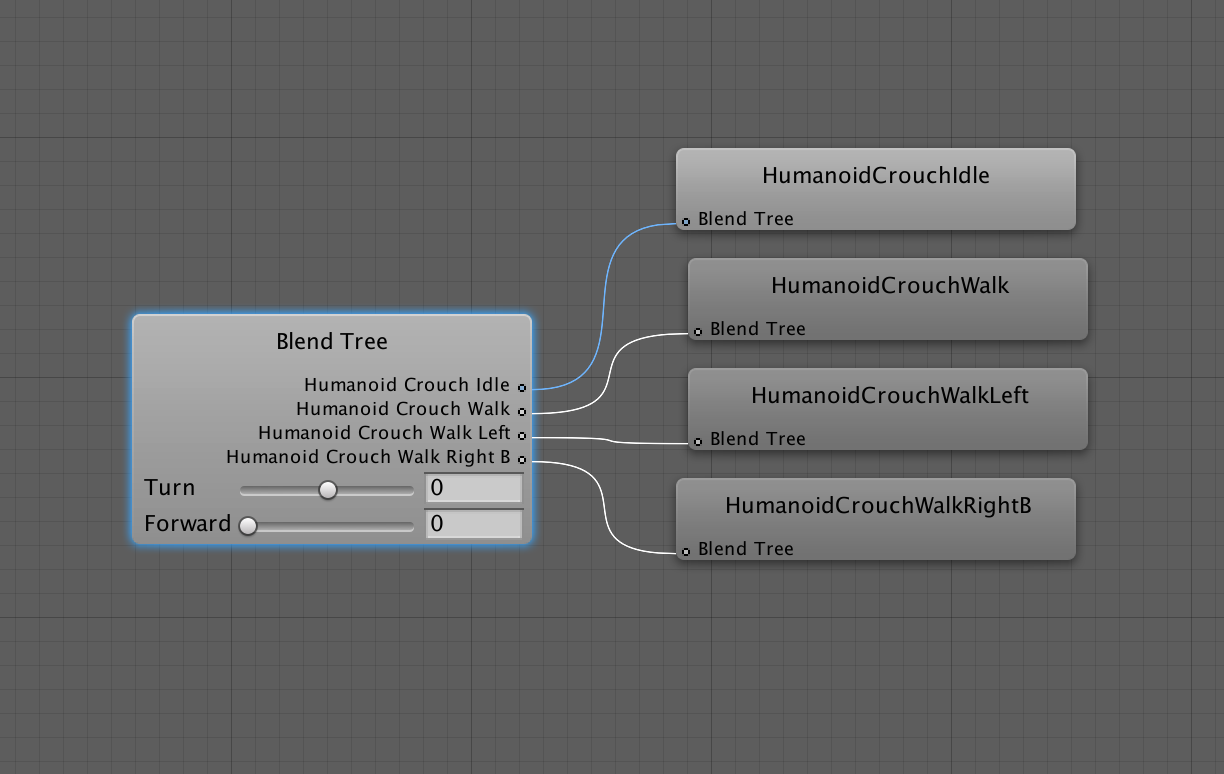
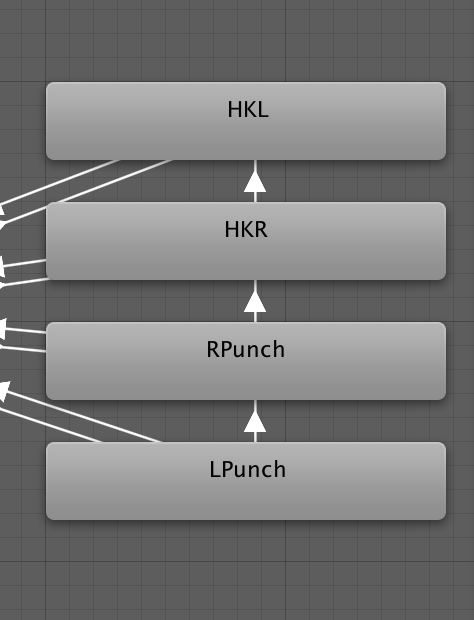
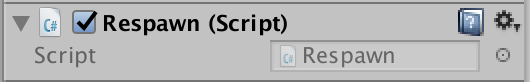
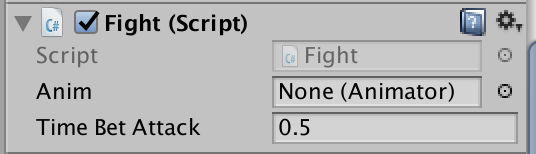
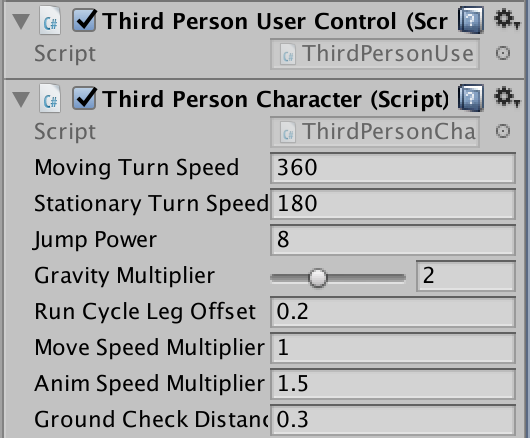
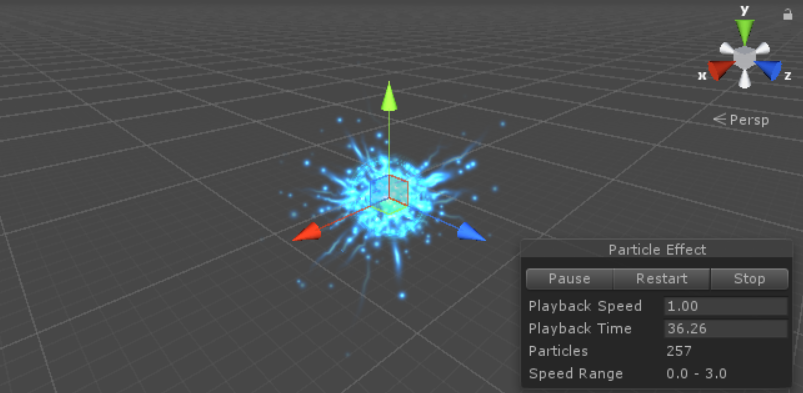
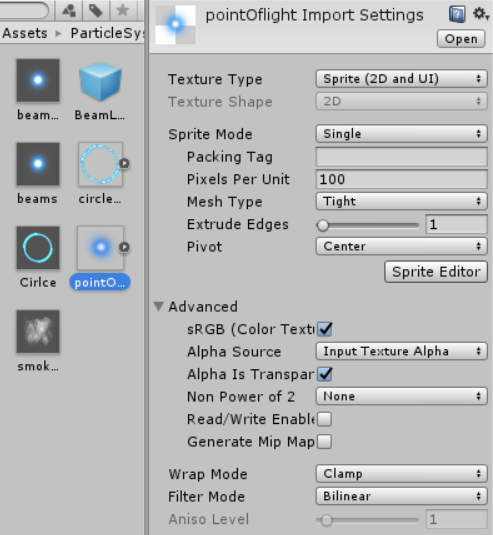
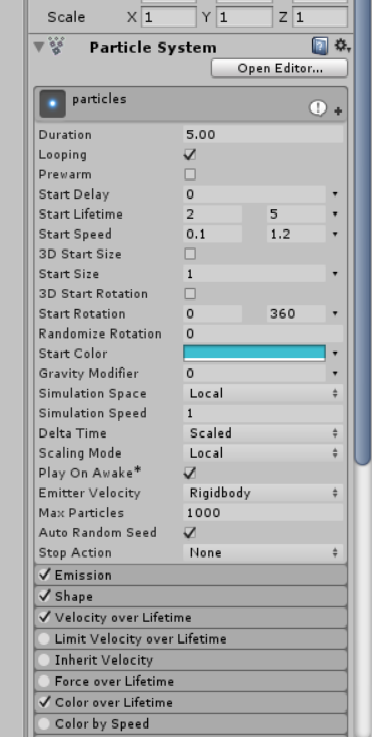
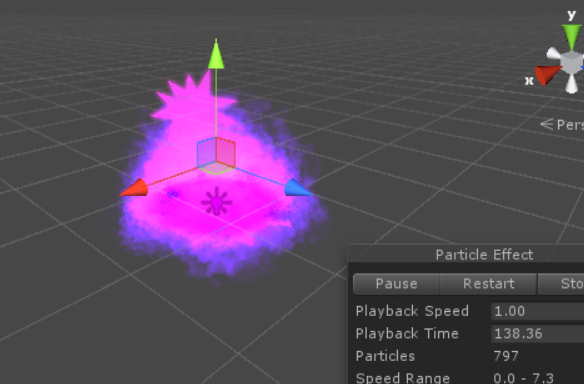
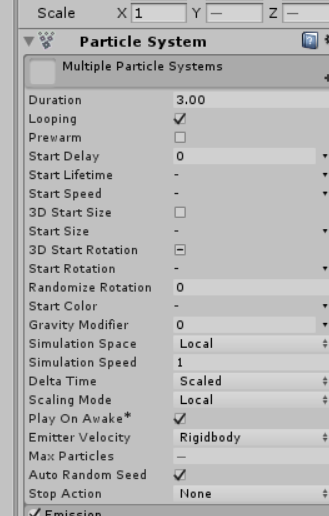
The scripts(1) of the game are:

* GameManager
  + Controls main functionality of the game, the timing that objects are created and instantiated, as well as the user interfaces such as pause menu, game menu etc.
* PauseMenu
  + Pauses the game by freezing the Player(5) and enemies in the game. It opens up a menu that gives access to restarting the level and quitting the game.
* BoardManager.cs
  + This will set up, breaks down and initializes the dungeon layout. This includes creating/clearing the ground, walls and background.
* CameraPerlinShake.cs
  + This allows the camera to shake smoothly on different types of impacts and actions taken between the different objects(4). Implements GenericShake but removes rapid movements between positions.
* DestroyByTime.cs
  + Controls the amount of time that objects(4) stay active during the playtime of the game. This can be attached to the prefab like particle effects that are only meant to be ran once.
* DeltaMusicManager.cs
  + This allows the music to be swapped by anything that calls the change song method. Passes in an audio clip and the song manager will change the sound in the game object.
* GenericShake.cs
  + This allows the camera to move rapidly from left to right and right to left on different impacts made between objects(4).
* EnemyManager.cs
  + Generates enemies of the playable character, while instantiating the amount of damage that will be done, setting up the audio clips from the enemy, locating the enemy, and performing the different actions(3) possessed by the enemy.
* Fight.cs
  + Controls the animations that the Player(5) uses to make fighting actions, such as: low kick, high kick, right punch, left punch, and firing a weapon.
* Hazard.cs
  + This is the parent class for all the objects(4) that can damage the Player(5), such as traps and explosives from the enemy.
* LevelGenManual\_Manager.cs
  + This creates a level map from a 2d int array and sets the vertical and horizontal length while adding grids to the level lists. This will contain the 10 main levels.
* LevelGenManual\_Abhi.cs/ LevelGenManual\_Eric.cs/ LevelGenManual\_Jackson.cs/ LevelGenManual\_Joshua.cs/ LevelGenManual\_Steven.cs
  + This is an extension of LevelGenManual\_Manager.cs. This creates a level map from a 2d int array and sets the vertical and horizontal length while adding grids to the level lists.These are used for each of the game developer to test and modify their own scripts and see how they interface on the standard construction of the game.
* LevelRoomLayout.cs
  + This is the code that allows for the internal room layout design for the 10 different levels. It is called by the LevelGenManual scripts randomly chosen at runtime.
* PlayerManager.cs
  + This controls the damage taken by enemies, the health received by food items, damage the Player(5) delivers to different objects(4) in the game, the audio of the Player(5) (dying, jumping… ect)
* Projectile.cs
  + Projectile is used for shooting projectiles and it is the parent script for grenade/mine/plasma orb projectiles.
* MovementWave.cs
  + This sets the movements for projectiles(6) in order to move it in a sinusoidal manner while traveling forward.
* Proj\_Grenade.cs
  + Proj\_Grenade is a child of Projectile. Code that controls the firing direction (forward and slightly upward), and applies the downward force upon them, as well as the detonation upon contacting enemies, walls or the ground.
* Proj\_Mine.cs
  + Proj\_Mine is a child of Projectile. The code controls the mines firing direction (forward and slightly upward), and the effects of downward force. Sets it so that mines will detonate upon contacting enemies and walls Mines stay on the ground upon touching the floor. Will explode after a set amount of time or if an enemy touches them first.
* Proj\_PlasmaOrb.cs
  + Proj\_PlasmaOrb is a child of Projectile. This code controls the plasma orbs phasing through and damaging enemies. Orbs will die upon hitting a set amount of enemies Orbs die upon hitting walls.
* Respawn.cs
  + This code is triggered when the collision collider interacts with a death zone( out of boundaries) or when the Player’s(5) health is empty.
* TrackingMine.cs
  + TrackingMine is a child of EnemyManager. This Code controls the tracking of the mine when a Player(5) passes through an outer sphere collider.(11) When this happens it will begin tracking player and will explode when player passes through inner sphere collider(11)
* WeaponsManager.cs
  + WeaponsManager sets Player’s(5) gun components allows for firing player guns. Code covers Gun implementation for individual gun type stats, firing type, shot type and projectile type.
* CameraController.cs
  + The Camera Controller sets the camera to focus on the player and follows the player around the game. The code checks to see if the player is alive. The code also checks if the camera is in the right position so that is it constantly tracking the player.

# 4. System Architecture

**UML**

**

* The project was divided up into some core concepts of the game, the character creation, level generation, special effects camera, music and sound effects. The level generation is the largest portion of the code. It is highly detail with comments to help the reader have a clear understanding of what the code is doing, making the maintainability much more simple.
* The responsibilities for character generation include getting the three dimensional models for the main character in the game and getting her animations ready. This includes jumping, walking, standing still, and holding different weapons.
* The Level Generation Manager was responsible for designing and implementing how different tiles making up a new room would be created and laid out after the user hits the play button on the main menu screen depicting a television. A spawn point for the character to enter must be created and a spawn point for the boss must be created so the models enter the room in the correct spot.
* The responsibilities for music and sound effects are to gather sound tracks that would play at different points throughout the game and then implement those sounds to be played at the same time. managing them so they transition correctly and the correct sounds come out at the right times. Sounds include Gun sounds, character noise, enemy noises, and sound tracks that are to be played in the game and during the different menus. Sound effects like walking and shooting are also part of this responsibility too.
* The responsibilities for video effects and camera effects are to manage special effects for weapons, attacks, and interactions throughout the game. They consist of prefabs for effects that can be used in various levels.
* Level Generation
  + 
  + The process of map generation comes from 2D int arrays as such:
  + 
* Character Development
  + Base Model
    - 
    - This is base 3D model of UnityChan, initial download provides the formatted bone structure and compiled visual graphics that can be seen.
  + Rigid Body Component
    - 
    - This component is an essential piece of the character development process, as it allows the character to be affected by gravity, different forces, and interactions with other objects(4) in the game.
    - This is heavily impacted, based on how the script is written.
  + Capsule Collider
    - 
    - Similarly to the Rigid body, this is the interaction capsule. It is two semi-spheres connected by a rectangle that surrounds the game object and is used as a trigger for different events, such as if the player goes out of bounds and interacts with another field it would cause it to respawn back into the proper section.
      * The capsule around the player at the top is the capsule collider.
  + Animator
    - 
    - The animator controls the animation states and the transitions between them for the designated avatar(7). It also incorporates the controller script that handles the standard movements and animations of the character, through the utilization of the animation controls.
    - 
    - This is the overall animation flow chart. Entry is the start state that flows directly into the Grounded-State Blend Tree:
      * 
      * Grounded-State Blend Tree controls the basic standing animations, walking animations, and the transitions between them.
    - Grounded state can be transitions into to 6 different States:
      * Airbourne-State Blend Tree:
        + 
        + This blend tree controls the jumping animations from blend by pressing the spacebar, which shows the transitions from bending knees to the jump up, mid air, landing and standing back up.
      * Grounded-State Blend Tree:
        + 
        + This blend tree controls the crouching animations from blend in UnityChan, which include the crouching standard, crouch walking in two directions.
      * Fighting Movement States:
        + 
        + These are controlled by clicking the right mouse button. This allows the transitions to do a combo between the 4 different states.
  + Respawn Script:
    - 
    - This is a script attachment to function with the Collision Collider to activate a trigger to respawn when the player is out of bounds.
  + Fight Script:
    - 
    - This is a script attachment to control the animation of the fight combo animations: Left Punch, Right Punch, Right High Kick, and Left High Kick. This is triggered through the right click of the mouse.
  + Third Person Controller Script:
    - 
    - This character attachment script is used to help control the animation of the Player, as well as adding different gravity/force to the player ability to turn in a direction, jump power.. ect.
  + Particle Systems for effects
    - 
    - All of the Special effects use particle systems. Each effect used multiple particle systems and each system has different speeds, colors, sizes and time durations
  + Development of the Special Effects:
    - 
    - There was no scripting involved in the making of the particle effects.
    - Most of the work was image editing and generation in the Adobe Photoshop Software
    - The rest of the work was done using the Inspector tab in Unity. The Inspector tab allowed for control of time duration, colors and particle speed.
    - 
  + Not all effects are Created Equal
    - Explosion teleport effect is instant and is not meant to last too long.
    - 
    - It will appear and disappear just as quickly.
    - 

# 5. Policies and Tactics

The project was divided up into some core concepts of the game, the character creation, level generation, special effects and cameras, and music and sound effects. The level generation is really complex so any one person in charge of that part could have another member help them with it. All the broken up parts are still responsible for implementing their content into the project. A dedicated engineer was in charge of menus and putting the game together. That job consisted of polishing player interface and managing the state of the game and level progression. Testing the software was left up to each individual that was in charge of that software then the dedicated engineer in charge of putting the individual pieces together was in charge of testing and debugging from there. If some complications were found during testing the original team member would work with the implementation to create a solution to the problem and implement it. The current iteration of room generation is a product of this method. The original implementation had tiles placed randomly. This led to errors and complications that were unnecessary and inconsistent to manage during implementation even though the design worked during independent testing. The build was changed to have a random number decide an index of a predetermined floor layout then have the room built according to that layout instead of placing three dimensional objects(4) randomly. This implementation still has the element of a random level that the game was originally designed to have but does not include random room generation that produce floor plans that aren’t viable for game play.

# 6. Detailed System Design

1. High priority features:

|  |
| --- |
| **GameManager : MonoBehaviour** |
| levelGenerator: public GameObject pauseMenu: public GameObject mainMenu: public GameObject mainMenuButtons: public GameObject teamDeltaLogo: public GameObject player: public GameObject bossRoomTeleport: public GameObject menuForeground: public GameObject tvScreenText: public GameObject tvScreenBackground: public GameObject itemMaker: protected GameObject menuMaterial: protected Material menuIsFading: protected bool menuYOffset: protected float = 30f menuZOffset: protected float = -5f skipTeamLogo: protected bool currentMoney: private int highScore: private int |
| Awake(): void DisplayTeamLogo: protected void DisplayTeamLogoFade(): protected IEnumerator MenuForegroundFadeIn(fadeTime: float): protected IEnumerator MenuForegroundFadeOut(fadeTime: float): protected IEnumerator DisplayMainMenu(): protected void BeginGamePrep(): public void BuildLevels(): protected void MakePlayer(): protected void MakeBossTeleporter(): protected void PlayerIsActive(playerIsActive: bool): protected void PlayerAppearsOnLevel(): protected void PlayerAppearsHelper(): protected IEnumerator MakeItemManager(): protected void MoveMenuToPlayer(): protected void MenuBackgroundIsActive(isActive: bool): protected void SetPauseBackgroundFade(): protected void FadeTVScreen(): protected IEnumerator MoveCameraTowardsPlayer(): protected IEnumerator StartGame(): public void MoneyAdded(money: int): public void MoneyRemoved(money: int): public bool GameOver(): public void GameRestart(): public void |

* + 1. Classification:
       1. Game Manager Class
    2. Definition:
       1. Controls main functionality of the game, the timing that objects(4) are created and instantiated, as well as the user interfaces such as pause menu, game menu, LevelGenManager, etc.
    3. Responsibilities:
       1. This is responsible for setting up the game
    4. Constraints:
       1. MenuForegroundFadeIn must take in a positive non zero value (1.0f)
       2. MenuForegroundFadeOut must take in a positive non zero value (1.0f)
       3. Money(8) added must take in a positive int above zero
       4. MoneyRemoved must take in an a positive int above zero. Money total will not reduce below zero.
    5. Composition:
       1. The transition methods handle what the player sees. These methods are DisplayTeamLogo, DisplayTeamLogoFade, MenuForegroundFadeIn, MenuForegroundFadeOut, DisplayMainMenu(), MenuBackgroundIsActivel, FadeTVScreen, and MoveCameraTowardsPlayer
       2. the game preparation methods handle setting up subsystems of the game. these methods include BeginGamePre, BuildLevels, MakePlayer, MakeBossTeleporter, MakeItemManager, MoveCameraTowardsPlayer, and StartGame.
       3. GameOver and GameRestart shall help change the state back to the main menu and reset the game.
    6. Uses/Interactions:
       1. Shows the team logo, displays the menu until an option is selected by the user, transitions to the level when play is selected.
    7. Resources:
       1. Unity game engine import library.
    8. Processing:
       1. Game manager is the highest process in the game.
    9. Interface/Exports:
       1. Changes player state between menu, inside a level and inside a boss room. Also controls the transition time with FadeIn and FadeOut methods.

|  |
| --- |
| **BoardManager : MonoBehaviour** |
|  |
| InitializeList(): void BoardSetup(): void |

* + 1. Classification:
       1. Level Building Class
    2. Definition:
       1. This manages the development and tear down of the levels.
    3. Responsibilities:
       1. Clears grid in preparation for creation
    4. Constraints:
       1. n/a
    5. Composition:
       1. Made of 2 functions:
          1. InitializeList()

Clear our list gridPositions.Loop through x axis (columns) and at each index add a new Vector3 to our list with the x and y coordinates of that position.

* + - * 1. BoardSetup()

Instantiate Board and set boardHolder to its transform. Loop along x axis, starting from -1 (to fill corner) with floor or outerwall edge tiles and Loop along y axis, starting from -1 to place floor or outerwall tiles.Choose a random tile from our array of floor tile prefabs and prepare to instantiate it. Instantiate the GameObject instance using the prefab chosen for to Instantiate at the Vector3 corresponding to current grid position in loop, cast it to GameObject. Set the parent of our newly instantiated object instance to boardHolder, in order to help avoid cluttering the hierarchy.

* + 1. Uses/Interactions:
       1. Used by LevelGenManager, interacts with all methods and scripts mentioned in section 3 of the SDS.
    2. Resources:
       1. Standard resources need, enough memory to properly load and build level. Race conditions and deadlocks are not a concern in this portion.
    3. Processing:
       1. Please refer to composition.
    4. Interface/Exports:
       1. Please refer to composition

|  |
| --- |
| **LevelGenManualManager : MonoBehaviour** |
| \_\_\_\_: protected const int = 0 NSWE: protected const int = 1 N\_\_\_: protected const int = 2 N\_W\_: protected const int = 3 NSW\_: protected const int = 4 N\_WE: protected const int = 5 NS\_\_: protected const int = 6 NS\_E: protected const int = 7 N\_\_E: protected const int = 8 \_\_W\_: protected const int = 9 \_SWE: protected const int = 10 \_SW\_: protected const int = 11 \_\_WE: protected const int = 12 \_S\_\_: protected const int = 13  \_S\_E: protected const int = 14 \_\_\_E: protected const int = 15 XXXX: protected const int = 16 RND1: protected const int = 21 RND2: protected const int = 22 RND3: protected const int = 23 RND4: protected const int = 24 RND5: protected const int = 25 RND6: protected const int = 26 RND7: protected const int = 27 RND8: protected const int = 28 RND9: protected const int = 29 PLAY: protected const int = 30 SHOP: protected const int = 31 BOSS: protected const int = 32 grid: protected int[22][22] levelsList: protected List<LevelRoomLayout> |
| StartLevelBuilding(): public void SetDebugsOff(): protected void AddLevelsToList(xMult: int, yMult: int, newGrid[][]): protected void SetGrids(): protected void ChooseGrid(): protected void SetRoomDimensionMultipliers(width: int, height: int): protected void SetRoomDimension(): protected void GenerateBigRooms(grid int[][]): protected void GetBossRoomLocation(): public Vector3 MakePlayer(): public oid MakeCamera(): protected void MakeLava(): protected void GenerateDeepPlane(): protected void GenerateGround(grid: int[][]): protected void GenerateRooms(grid: int[][]): protected void CreateTestEnemies(pos: Vector3): protected void CreateWallPiece(wallPosition: Vector3, wallScale: Vector3): protected GameObject CreateLeftWall(row: int, column: int, position: Vector3): protected void CreateRightWall(row: int, column: int, position: Vector3): protected void CreateBottomWall(row: int, column: int, position: Vector3): protected void CreateTopWall(row: int, column: int, position: Vector3): protected void GenerateWalls(grid: int[][]): protected void GenerateGrid(): private void |

* + 1. Classification:
       1. Level builder
    2. Definition:
       1. Parent level building script
    3. Responsibilities:
       1. Builds a level of empty rooms and boundary walls so the player doesn’t fall off the level.
    4. Constraints:
       1. Used by the GameManager, then discarded
    5. Composition:
       1. Several functions
          1. StartLevelBuilding()

Prepares 2D int arrays for choosing

Sets 2D int array grid to be used

* + - * 1. SetDebugsOff()

Used by GameManager, turns off debug bools for testing.

* + - * 1. AddLevelsToList(int xMult, int yMult, int[][] newGrid)

Adds level layout (newGrid) with dimension stretch multipliers to the levelsList.

* + - * 1. SetGrid()

Creates a list of 10 int[][] levels to be randomly chosen from.

* + - * 1. ChooseGrid()

If debugging specific levels, chooses selected level to debug.

Otherwise, chooses a random int[][] grid from LevelsList.

Calls GenerateGrid() upon completion.

* + - * 1. SetRoomDimensionMultipliers(int width, int height)

Sets int amount to stretch room width/height.

* + - * 1. SetRoomDimensions()

Sets to modified room width and height with multipliers.

* + - * 1. GenerateBigRooms(int[][] grid)

Calls MakeLava()

Calls GenerateDeepPlane()

Calls GenerateGround(grid)

Calls GenerateWalls(grid)

Calls GenerateRooms(grid)

If debugging the camera, call MakeCamera()

* + - * 1. GetBossRoomLocation()

Returns a Vector3 of the bossRoomLocation

* + - * 1. MakePlayer()

Instantiate player

Set player position to position of PLAY room (choose one PLAY room if there are several)

set playerSpawned to true

* + - * 1. MakeCamera()

If there is a camera active in the scene, disable it

Instantiate testCamera

* + - * 1. MakeLava()

Instantiate lava

Set lava localSclae to (900,900,1)

set lava position to (300,lavaDistance,300)

* + - * 1. GenerateDeepPlane()

Create a primitive Plane

Set Plane localScale to ((roomWidth/10f) \* 30, 1, (roomWidth / 10f) \* 30))

Set position to ((roomWidth) \* 11, distanceBelowStage, (roomHeight) \* 11)

Set LayerMask name to “Floor”

Set deepPlane material color to deepGroundColor.

* + - * 1. GenerateGround(int[][] grid)

For each position on the int[][] grid, do the following:

Create a primitive Plane

Set Plane Material to groundMaterial

Set Plane localScale to ((roomWidth / 10.0f), 1, (roomHeight / 10.0f))

Set position to (column \* roomWidth, 0, rowOffset \* roomHeight) (rowOffset is used to build upward from bottom of array)

Set plane tag to “Ground”

If debugging enemies and the room != BOSS and != PLAY, create a testEnemy in the center of the room.

* + - * 1. GenerateRooms(int grid[][])

Calls RoomGeneration’s BuildRoomsAtPosition method for each room in the int[][] array. Uses same spacing as GenerateGround().

* + - * 1. CreateTestEnemies(Vector3 pos)

Instantiate testEnemy at pos, increase pos.y + 0.5

* + - * 1. CreateWallPiece(Vector3 wallPosition, Vector3 wallScale)

Instantiate wall piece at specified position and scale

Add tag “Walls”

* + - * 1. CreateLeftWall(int row, int column, Vector3 position)

Calls CreateWallPiece

Makes a column of wall pieces 1 unit to the left of the room

* + - * 1. CreateRightWall(int row, int column, Vector3 position)

Calls CreateWallPiece

Makes a column of wall pieces 1 unit to the right of the room

* + - * 1. CreateTopWall(int row, int column, Vector3 position)

Calls CreateWallPiece

Makes a column of wall pieces 1 unit above the room

* + - * 1. CreateBottomWall(int row, int column, Vector3 position)

Calls CreateWallPiece

Makes a column of wall pieces 1 unit below the room

* + - * 1. GenerateWalls(int[][] grid)

For each room int in the array, creates walls above/below/right/left of room if there’s an empty space (int == \_\_\_\_) next to the room

Stores playerStartPosition at room PLAY

Stores bossRoomLocation at room BOSS

* + - * 1. Generate Grid()

If debugging room dimension multipliers, set these

Otherwise, set room Width and Height to 1 or to randomized amount if this is chosen

Calls SetRoomDimension()

Calls GenerateBigRooms(grid)

* + 1. Uses/Interactions:
       1. Called by GameManager to build levels
    2. Resources:
       1. Uses memory to store and process the int[][] arrays. This memory is freed after the level is generated and the script is destroyed
    3. Processing:
       1. Refer to composition
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **LevelGenManual\_Abhi : LevelGenManualManager** |
| (variables exist within parent class) |
| SetGrids(): protected override void |

* + 1. An extension of LevelGenManualmanager
    2. Stores data for 10 levels
    3. See accompanying Level Layouts document for array layouts

|  |
| --- |
| **LevelGenManual\_Eric : LevelGenManualManager** |
| (variables exist within parent class) |
| SetGrids(): protected override void |

* + 1. An extension of LevelGenManualmanager
    2. Stores data for 10 levels
    3. See accompanying Level Layouts document for array layouts

|  |
| --- |
| **LevelGenManual\_Jackson : LevelGenManualManager** |
| (variables exist within parent class) |
| SetGrids(): protected override void |

* + 1. An extension of LevelGenManualmanager
    2. Stores data for 10 levels
    3. See accompanying Level Layouts document for array layouts

|  |
| --- |
| **LevelGenManual\_Joshua : LevelGenManualManager** |
| (variables exist within parent class) |
| SetGrids(): protected override void |

* + 1. An extension of LevelGenManualmanager
    2. Stores data for 10 levels
    3. See accompanying Level Layouts document for array layouts

|  |
| --- |
| **LevelGenManual\_Steven : LevelGenManualManager** |
| (variables exist within parent class) |
| SetGrids(): protected override void |

* + 1. An extension of LevelGenManualmanager
    2. Stores data for 10 levels
    3. See accompanying Level Layouts document for array layouts

|  |
| --- |
| **LevelRoomLayout : MonoBehaviour** |
| \_\_\_\_: protected const int = 0 NSWE: protected const int = 1 N\_\_\_: protected const int = 2 N\_W\_: protected const int = 3 NSW\_: protected const int = 4 N\_WE: protected const int = 5 NS\_\_: protected const int = 6 NS\_E: protected const int = 7 N\_\_E: protected const int = 8 \_\_W\_: protected const int = 9 \_SWE: protected const int = 10 \_SW\_: protected const int = 11 \_\_WE: protected const int = 12 \_S\_\_: protected const int = 13  \_S\_E: protected const int = 14 \_\_\_E: protected const int = 15 XXXX: protected const int = 16 RND1: protected const int = 21 RND2: protected const int = 22 RND3: protected const int = 23 RND4: protected const int = 24 RND5: protected const int = 25 RND6: protected const int = 26 RND7: protected const int = 27 RND8: protected const int = 28 RND9: protected const int = 29 PLAY: protected const int = 30 SHOP: protected const int = 31 BOSS: protected const int = 32 grid: protected int[22][22] |
| LevelRoomLayout(xStretch: int, yStretch: int, setGrid: int[][]) SetMultipliers(xMult: int, yMult: int): public void SetGrid(newGrid: int[][]): public void GetGrid(): public int[][] |

* + 1. Classification:
       1. Level Generation
    2. Definition:
       1. A Script for defining constants that are used in the Level Generation classes
    3. Responsibilities:
       1. Defines constants and creates grids. The grids are used to make a layout of a level.
    4. Constraints:
       1. Only works with level generation classes
    5. Composition:
       1. Five functions:
          1. LevelRoomLayout(int xStretch, int yStretch, int[][] setGrid)

makes the grid layout for the level

* + - * 1. SetMultipliers(int xMult, int yMult)

Sets the values for the X and Y multipliers for the grid

* + - * 1. SetGrid(int [][] newGrid)

Copies the grid layout

* + - * 1. GetGrid()

Returns the Grid.

* + 1. Uses/Interactions:
       1. Interacts with level generation classes
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **RoomGeneration : MonoBehaviour** |
| \_\_\_\_: protected const int = 0 NSWE: protected const int = 1 N\_\_\_: protected const int = 2 N\_W\_: protected const int = 3 NSW\_: protected const int = 4 N\_WE: protected const int = 5 NS\_\_: protected const int = 6 NS\_E: protected const int = 7 N\_\_E: protected const int = 8 \_\_W\_: protected const int = 9 \_SWE: protected const int = 10 \_SW\_: protected const int = 11 \_\_WE: protected const int = 12 \_S\_\_: protected const int = 13  \_S\_E: protected const int = 14 \_\_\_E: protected const int = 15 groundMaterial: public Material wallPiece: public GameObject wallYPos: protected float = 1f xPosOffset: protected int = 0 zPosOffset: protected int = 0 xPosOffsetHelper: protected float = 0.5f zPosOffsetHelper: protected float = 0.5f xShift: protected int = 0 zShift: protected int = 0 cubeShift: protected float = 0.5f roomWidth: protected int = 18 roomHeight: protected int = 18 roomWidthMultiplier: protected int = 1 roomHeightMultiplier: protected int = 1 |
| Start(): void SetMultipliers(height: int, width: int): public void SetInitialPoints(xInit: int, yInit: int): public void BuildRoomAtPosition(startPos, Vector3, roomType: int): public void BuildRooms(roomType: int): protected void BuildInnerRoom(): protected void BuildAllCorners(): protected void BuildTopCorners(): protected void BuildBottomCorners(): protected void BuildLeftCorners(): protected void BuildRightCorners(): protected void BuildULCorner(): protected void BuildURCorner(): protected void BuildLLCorner(): protected void BuildLRCorner(): protected void BuildCornerWall(xPos: int, zPos: int): protected void BuildBottomWall(): protected void BuildLeftWall: protected void BuildRightWall: protected void BuildTopWall: protected void WallsHorizontal(startXPos: int, startZPos: int): protected void WallsVertical(startXPos: int, startZPos: int): protected void |

* + 1. Classification:
       1. level generation
    2. Definition:
       1. a class that randomize the creation of the level.
    3. Responsibilities:
       1. Builds each section of the room in random portions except for the preset height of the walls and to create 3-6 enemies and 2-4 obstacles per room.
    4. Constraints:
       1. The code is still in process and will be used to replace the LevelGenManualManager’s room builder to create randomized rooms.
    5. Composition: -- Same as LevelGenManualManager, with minor differences: one difference is instead of preset blank floors it creates randomized floors with differing tiles, enemies and obstacles(9). In addition, the outer walls are destructible and are set to the last position of the room dimensions (rather than one unit past the room edge as with the LevelGenManual).
       1. Start()
          1. test to make sure the floors and walls were generated and connected correctly.
       2. SetMultipliers()
          1. Allows for a manually entered number to multiply the room length and the wall height by the basic setting.
       3. SetInitialPoints()
          1. sets the bottom left starting point of the room.
       4. BuildRoomAtPosition()
          1. Calls BuildRoom at set points.
          2. give it to me baby. I’ll take all your function
          3. lolz to the max
       5. BuildRooms()
          1. Builds walls based on int values at room position
       6. BuildInnerRoom()
          1. (see LevelGenManualManager for details)
       7. BuildAllCorners()
          1. Makes destructible wall pieces in all four corners.
       8. BuildTopCorners()
          1. Makes destructible wall pieces in the topcorners.
       9. BuildBottomCorners()
          1. Makes destructible wall pieces in the bottom corners.
       10. BuildLeftCorners()
           1. Makes destructible wall pieces in the leftcorners.
       11. BuildRightCorners()
           1. Makes destructible wall pieces in the right corners.
       12. BuildULCorner()
           1. Makes a single destructible wall piece in the upper-left corner of the room
       13. BuildURCorner()
           1. Makes a single destructible wall piece in the upper-rightcorner of the room
       14. BuildLLCorner()
           1. Makes a single destructible wall piece in the lower-left corner of the room
       15. BuildLRCorner()
           1. Makes a single destructible wall piece in the lower-right corner of the room
       16. BuildCornerWall(xPos: int, zPos: int)
           1. Makes a single destructible wall piece in the specified location
       17. BuildBottomWall()
           1. Makes a row of destructible wall pieces up to but not including the corner points of the room.
       18. BuildLeftWall()
           1. Builds the left-side room wall as a set of destructible wall pieces in all but the corner pieces.
       19. BuildRightWall()
           1. Builds the right-side room wall as a set of destructible wall pieces in all but the corner pieces.
       20. BuildTopWall()
           1. Builds the top of a wall using X and Y offsets;
       21. WallsHorizontal()
           1. Maks a Horizontal column of the walls
       22. WallsVertical()
           1. Makes a vertical column of the walls
    6. Uses/Interactions:
       1. No user interface, auto creates on the activation of the game and or completion of the level.
    7. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    8. Processing:
       1. Refer to LevelGenManual Composition
    9. Interface/Exports:
       1. n/a

|  |
| --- |
| **PlayerManager : MonoBehaviour** |
| life: protected float = 50f harmOffset: protected float = 100f shieldAmount: protected float = 0f canAttack: public bool canMove: public bool stunTimeLeft: private float |
| LateUpdate(): void SetAnims(): void DamagePlayer(damage: float): public void StunPlayer(stunTime: float): public void StunPlayerHelper(stunTime: float): protected IEnumerator AllowControl(setting: bool): public void AllowAttack(setting: bool): public void AllowMovement(setting: bool): public void Die(): protected void |

* + 1. Classification:
       1. A player class
    2. Definition:
       1. Manages public variables that are used for Player conditional Checking
    3. Responsibilities:
       1. Keeps track of Movement, attacking ability, hit points and life of Player
    4. Constraints:
       1. Only works with the player
    5. Composition:
       1. Nine Functions
          1. Awake()

Disables the player until the player is fully faded in

* + - * 1. LateUpdate()

Updates Anims variable to fix delay errors with player animations

* + - * 1. SetAnims()

Sets a value for the anims variable

* + 1. Uses/Interactions:
       1. Interacts with player classes
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **Respawn : MonoBehaviour** |
| startPos: private Vector3 startRot: private Quaternion |
| Start(): void OnTriggerEnter(col Collider): void |

* + 1. Classification:
       1. Collision Collider Class
    2. Definition:
       1. Upon interaction with a collider tagged “death” the player will be respawned to last checkpoint(10)
    3. Responsibilities:
       1. This component, prevents the player from leaving the boundaries of the game board.
    4. Constraints:
       1. The only that this engaged is upon the collision of the two independent colliders tagged as “Player” and “Death”.
    5. Composition:
       1. Two Functions:
          1. Start()

Saves the original location of the Player to load the Player back into its original position.

* + - * 1. OnTriggerEnter()

When the collision collider interacts with a tag called “Death” the trigger activates and respawns the player to starting position.

* + 1. Uses/Interactions:
       1. The component utilizes the LevelGenManager to create the “Death” collider.
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition
    4. Interface/Exports:
       1. n/a

|  |
| --- |
| **Fight : MonoBehaviour** |
| anim: public Animator numAttack: private int clickTime: private float timeBetAttack: public float lastAttack: private float combo1: private bool combo2: private bool combo3: private bool |
| Update(): void |

* + 1. Classification:
       1. A script for the player
    2. Definition:
       1. Motions for controls of the player’s fight movements
    3. Responsibilities:
       1. Controls the players fight movements and attaches controls to certain keyboard keys
    4. Constraints:
       1. Works only with the player
    5. Composition:
       1. Two Functions
          1. Start()

initializes class variables

* + - * 1. Update()

attaches different fighting motions to the ‘F’ key

* + 1. Uses/Interactions:
       1. A stand alone class for the players fighting movements
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **EnemyManager : MonoBehaviour** |
| speed: protected float hitPoints: protected float canAttack: protected bool canMove: protected bool canBeDamaged: protected bool isAttacking: protected bool isTracking: protected bool player: protected GameObject |
| SetPlayer(target: GameObject): public void SetCanBeDamaged(canTakeDamage: bool): protected void Damage(damage: float): public virtual void DiePrep(): protected virtual void Die(): protected void |

* + 1. Classification:
       1. Enemy Class
    2. Definition:
       1. A parent script for shared content among enemies
    3. Responsibilities:
       1. Handles movement, speed, hit points, attacking and moving abilities.
    4. Constraints:
       1. Used only for enemies.
    5. Composition:
       1. Five functions
          1. SetPlayer(GameObject target)

sets the player as the target for enemy to track

* + - * 1. SetCanBeDamaged(bool canTakeDamage)

Determines whether or not the enemy can be damaged

* + - * 1. Damage(float damage)

Harms the enemy

* + - * 1. DiePrep()

Sends a message to the console that the DiePrep() function from the parent class is being called and the kills enemy

Die()

destroys the enemy

* + 1. Uses/Interactions:
       1. Parent class for enemies.
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **TrackingMine : EnemyManager** |
| proximityExplode: public float explosion: protected GameObject |
| OnTriggerEnter(other: Collider): protected void FollowPlayer(target Transform): private IEnumerator DiePrep(): protected override void Explode(): private void |

* + 1. Classification:
       1. Collider Class
    2. Definition:
       1. Upon player entering mine bounds enemies start tracking player
    3. Responsibilities:
       1. When Player passes through an outer sphere collider it begins tracking player.
    4. Constraints:
       1. The only constraint is that it is engaged when the tag player is in bounds of the mine.
    5. Composition:
       1. OnTriggerEnter(other: Collider):
          1. Checks if player has entered mine bound
       2. FollowPlayer(target Transform):
          1. This shall follow the player after player has entered mine bound.
       3. DiePrep():
          1. Initializes Explode()
       4. Explode():
          1. Creates an explosion after death occurs.
    6. Uses/Interactions:
       1. SphereCollider interacts with gameObjects tagged as “Player.” Mine will track said player and explode upon contact.
    7. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    8. Processing:
       1. Refer to Composition
    9. Interface/Exports:
       1. N/A

|  |
| --- |
| **BossManager : EnemyManager** |
| fireBrows: protected GameObject deathExplosion: protected GameObject explosionPoint: protected GameObject bossMain: protected Boss maxHitPoints: protected float |
| Start(): protected void Damage(damage: float): public override void BeginSecondPhase(): protected IEnumerator DiePrep(): protected override void DiePrepBoss(): protected IEnumerator DeathExplosion(): protected void |

* + 1. Classification:
       1. Sub class of Enemy manager
    2. Definition:
       1. This class controls all the characteristics of the boss level.
    3. Responsibilities:
       1. This class is responsible for effects created by the boss and how much damage the boss can inflict and take.
    4. Constraints:
       1. Heavily relies on the LevelGenManager. If the LevelGenManager does not compile then this script will not be executed.
    5. Composition:
       1. Start():
          1. Activates boss’s game object to instantiate itself, as well as setting the starting health of the enemy.
       2. Damage():
          1. This is the reaction function that sets the different phases to activate. Depending on the damage taken by the Player it will shift to the Second phase and the dying phase.
       3. BeginSecondPhase():
          1. Starts the second phase with different sets of attacks and animation functions of the boss.
       4. DiePrep():
          1. activates animations and death of the boss.
       5. DiePrepBoss():
          1. activates the sounds, explosions animation, and the length of the action of the boss dying.
       6. DeathExplosion():
          1. controls the animations of the explosions of the boss
    6. Uses/Interactions:
       1. This based on the LevelGenManager activated and executed properly.
    7. Resources:
       1. Unity game engine import library and standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    8. Processing:
       1. Refer to composition
    9. Interface/Exports:
       1. n/a

|  |
| --- |
| **Boss : MonoBehaviour** |
| mainSphere: protected GameObject bossManager: protected Boss\_Manager sphereAgile: protected const int = 0 sphereBigBang: protected const int = 1 sphereSpiral: protected const int = 2 shereWaterfall: protected const int = 3 bossSpheres: protected GameObject[] spheresRotationAmount: protected float = 1f sphereRaiseSpeed: protected float = 2.0f sphereDepthFromCenter: protected float = -20f sphereHeightAfterRaising: protected float = 0f sphereDistanceFromCenter: protected float = 18f debugBoss: protected bool spheresRotation: protected Vector3 rotateSpheres: protected bool spheresAreRising: protected bool |
| Awake(): protected void Start(): protected void FixedUpdate(): private void CreateMainSphere(): protected void CreateSecondarySphere(): protected void ResetRotation(): protected void RaiseMainSphere(): protected void RaiseMainSphereHelper(): protected IEnumerator RaiseSecondarySpheres(): protected void RaiseSphere(sphere: GameObject): protected IEnumerator LowerSecondarySpheres(): public void LowerSphere(sphere: GameObject): protected IEnumerator |

* + 1. Classification:
       1. Boss Class
    2. Definition:
       1. Sets up the boss assets and working components. Details in Responsibilities section.
    3. Responsibilities:
       1. Creates five boss spheres that rise up from the ground and rotate around player. Main sphere calls out attacks for the secondary spheres. Four secondary spheres have their own unique attacks - agile, big bang, spiral, waterfall. Spheres begin inactive and are made active as needed.
    4. Constraints:
       1. This component will be created even if the others fail, however it will not be able to fulfill its purpose to assist the Boss if the boss and Player does not generate.
    5. Composition:
       1. Several functions:
          1. Attack(int attackType, Vector3 from)

Sets an attack to fire from the specified point to the center of the boss stage (Vector3 to = new Vector3(0,0,0)).

Calls Attack(attackType, from, to);

* + - * 1. Attack(int attackType, Vector3 from, Vector3 to)

Instantiates specific attack type from the array.

Alters angle so that object is fired towards the “to” position.

Calls method which corresponds to the chosen attack type.

* + - * 1. Fireball(GameObject attack)

Shoots a fireball from the main sphere towards the player

Originates from Fireball Sphere

* + - * 1. Agile(GameObject attack)

Shoots a rapid succession (10 shots at 0.2f second delays) of agile projectiles towards the center of the stage.

Originates from Agile Sphere

* + - * 1. BigBang(GameObject attack)

Creates an explosion at the player position.

Originates from BigBang Sphere

* + - * 1. Spiral(GameObject attack)

Fires a spiral projectile towards player.

Originates from Spiral Sphere

* + - * 1. Waterfall(GameObject attack)

Fires a succession (5 shots at 0.1f second delay) of water projectiles towards player.

Originates from Waterfall Sphere

* + 1. Uses/Interactions:
       1. Uses player Transform to fire at player where applicable.
       2. Uses sphere Transform positions to fire from corresponding sphere.
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to composition.
    4. Interface/Exports:
       1. N/A.

|  |
| --- |
| **Boss\_HS\_RoomTeleporter : MonoBehaviour** |
| (no variables needed) |
| OnTriggerEnter(other: Collider): private void RaiseStage(): public void |

* + 1. Classification:
       1. Trigger Class
    2. Definition:
       1. Activations on trigger to teleport character to boss room
    3. Responsibilities:
       1. Upon the “Player” collider contacting the GameObject Collider it will activate the teleporter and send the player into the boss room.
    4. Constraints:
       1. If the LevelGenManager does not construct the original room or the boss room, this class will operate or function within the expected realm.
    5. Composition:
       1. OnTriggerEnter()
          1. Compares the collision collider tag “Player” and if it is true it will start the translation of the room into the boss room.
       2. RaiseStage()
          1. Will cause the room to be modified by raising the ground (lava, deepBackground, player) and instantiating the final boss room.
    6. Uses/Interactions:
       1. Same as Constraints
    7. Resources:
       1. Unity game engine import library and standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    8. Processing:
       1. See composition
    9. Interface/Exports:
       1. n/a

|  |
| --- |
| **WeaponsManager : MonoBehaviour** |
| bulletLocation: public Transform currentProjectile: protected GameObject allProjectiles: public GameObject[] allBurstEffects: public GameObject[] WeaponStats(shotNumber: int, shotName: string, delayBetweenShots: public float,   angleError: public float): protected struct gunPistol: protected const int = 0  gunRifle: protected const int = 1  gunShotgun: protected const int = 2  gunRocket: protected const int = 3  gunGrenade: protected const int = 4  gunMine: protected const int = 5  gunPlasma: protected const int = 6  firingNormal: protected const int = 0  firingSpread: protected const int = 1 firingHeavy: protected const int = 2 firingAutomatic: protected const int = 3  shotsNormal: protected const int = 0  shotsRapid: protected const int = 1  shotsWave: protected const int = 2  shotsRusty: protected const int = 3  shotsBroken: protected const int = 4  shotsDoubleBarrel: protected const int = 5 currentFiringType: protected int currentShotType: protected int currentWeapon: protected int currentProjectileType: protected int currentSmokeType: protected int allWeapons: new protected weaponStats[7]  {  new weaponStats(gunPistol, "bulletNormal", .50f, 0f),  new weaponStats(gunRifle, "bulletRifle", 0.2f, 2f),  new weaponStats(gunShotgun, "bulletShotgun", 1.0f, 5f),  new weaponStats(gunRocket, "bulletRocket", 2.2f, 0f),  new weaponStats(gunGrenade, "bulletGrenade", 1.50f, 0f),  new weaponStats(gunMine, "bulletMine", 1.5f, 0f),  new weaponStats(gunPlasma, "bulletPlasma", 3.0f, 0f)  } shotsDelay: protected float shotsDelayModifier: protected float shotAngleError: protected float shotAngleErrorModifier: protected float shotDudFireChance: protected float lastShotTime: protected float |
| Start(): private void Update(): private void SwitchGuns(weaponNumber: int, newFiringType: int, newShotType: int): public void SwitchGunType(weaponNumber: int): protected void SwitchFiringType(newFiringType: int): protected void SwitchShotType(newShotType: int): protected void SwitchProjectileType(newProjectileType: int): protected void SwitchSmokeBurstType(): protected void ResetModifiers(): protected void ChangeModifiers(): protected void CheckFiringType(): protected void CheckShotsType(): protected void ShootPrep(): protected void SmokeBurst(): protected void Shoot(): protected void ShootHelper(): protected void ShootHeavy(): protected void ShootHeavyHelper(): protected void DoubleShot(): protected IEnumerator DoubleShotHeavy(): protected IEnumerator DoubleShotSpread(): protected IEnumerator ShootSpread(): protected void ShootSpreadHelper(): protected void |

* + 1. Classification:
       1. Weapons management and activation
    2. Definition:
       1. The Weapons Manager class defines the types of weapons the types of ammunition and effects of the shots are available in the game.
    3. Responsibilities:
       1. This Component shall provide weapons variations and different firing styles for the weapon that has been selected
    4. Constraints:
       1. Relevant only to the player. Only player can interact with this component
    5. Composition:
       1. SwitchGuns():
          1. This allows the player to select a weapon type
       2. SwitchGunType():
          1. This function changes to new gun stats and projectile type.
       3. SwitchFiringType():
          1. This Function allows the change in firing patterns
       4. SwitchShotType():
          1. This Function allows the change in shot effectors in the weapons
       5. SwitchProjectileType():
          1. This Function allows the change in projectile type for the weapon picked by the player.The weapon stats shall remain the same.
       6. SwitchSmokeBurstType():
          1. This shall change the smoke burst effect to coincide with the type of ammo selected.
       7. ResetModifiers():
          1. This Function shall reset all modifiers.
       8. ChangeModifiers():

a. This Function shall reset and then re-calculate all modifiers.

* + - 1. CheckFiringType():

a. Sets current firing type modifiers.

* + - 1. CheckShotsType():

a. Sets current shots type modifiers.

* + - 1. ShootPrep():

a. Checks and initiates shot based on current firing type.

* + - 1. SmokeBurst():

a. Releases a puff of smoke from the barrel which corresponds to the equipped projectile type.

* + - 1. Shoot():
         1. This is the basic shot that is given at the start of the game. Adds a double shot if said firing type is enabled.
      2. ShootHelper():
         1. Instantiates a bullet at the gun barrel position and rotation to match the on-screen pointer.
         2. Adds angle error or movement wave if equipped.
      3. ShootHeavy():
         1. This allows 3 shots with minimal (2.5f between rounds, starting from center) spread when fired
      4. ShootHeavyHelper():
         1. Instantiates a bullet at the gun barrel position and rotation to match the on-screen pointer.
         2. Adds angle error or movement wave if equipped.
      5. DoubleShot():
         1. This function allows in multiple shots fired with a spread between them.
      6. DoubleShotHeavy():
         1. Waits 0.05 seconds, then calls ShootHeavyHelper()
      7. DoubleShotSpread():
         1. Waits 0.05 seconds, then calls ShootSpreadHelper()
      8. ShootSpread():
         1. Calls either ShootSpreadHelper() or DoubleShotSpread() if double shot is equipped
      9. ShootSpreadHelper():
         1. This allows 4 shots with moderate (7.0f between rounds, starting from center) spread from the center direction.
    1. Uses/Interactions:
       1. If player shooting ability is not disabled and the time since last shot is greater than the shot delay, user pressing the “fire” button shall begin firing sequence for equipped gun type.
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to composition
    4. Interface/Exports:
       1. This is applied every time a power up, weapon or weapon variant is picked up.

|  |
| --- |
| **ItemCreator : MonoBehaviour** |
| randomGunType: public GameObject randomFiringType: public GameObject randomShootingType: public GameObject randomProjectileType: public GameObject randomGunWithModifiersType: public GameObject |
| MakeRandomGunPickup(pos: Vector3, int type): public void PlayerStartPickups(pos: Vector3): public void |

* + 1. Classification:
       1. A weapon class
    2. Definition:
       1. Makes weapons and projectiles
    3. Responsibilities:
       1. Determines the type of weapon pickup that is being generated
    4. Constraints:
       1. only used for weapons and projectiles
    5. Composition:
       1. MakeRandomGunPickup(pos: Vector3, int type)
          1. Sets the position for a random gun.
       2. PlayerStartPickups(pos: Vector3)
          1. Calls MakeRandomGunPickup() function several times. A new position is used each time.
    6. Uses/Interactions:
       1. Interacts with different gun game objects
    7. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    8. Processing:
       1. Refer to composition
    9. Interface/Exports:
       1. N/A

|  |
| --- |
| **Projectile : MonoBehaviour** |
| damage: public float = 1f speed: public float = 20f gunLocation: public Vector3 explosion: public GameObject |
| Start(): void Shoot(): protected virtual void OnTriggerEnter(other: Collider): private void DamageEnemy(other: Collider): protected void DiePrep(): protected void Die(): private void |

* + 1. Classification:
       1. A projectile class
    2. Definition:
       1. A parent script for grenade, mine and plasma orb projectiles
    3. Responsibilities:
       1. Projectile.cs is used for shooting projectiles
    4. Constraints:
       1. Works only with certain weapons
    5. Composition:
       1. Six Functions
          1. Start()

calls the Shoot() function

* + - * 1. Shoot()

Shoots projectile and normalizes direction and speed

* + - * 1. OnTriggerEnter(Collider other)

Checks if the the bullet hits an enemy or boundary

* + - * 1. DamageEnemy(Collider other)

Damages the enemy

* + - * 1. DiePrep()

Checks if projectile has an explosion. If it does than an explosion occurs

calls the Die method

* + - * 1. Die()

Destroys the bullet

* + 1. Uses/Interactions:
       1. interacts with weapons classes such as grenade.cs
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. See Composition
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **Proj\_Grenade : Projectile** |
| upwardForce: protected float |
| Shoot(): protected override void OnTriggerEnter(other: Collider): private void |

* + 1. Classification:
       1. Child Class of Projectile
    2. Definition:
       1. Flies in an arch first going up and then effected by gravity bring it back down. They detonate on impact on the floor, wall or enemy.
    3. Responsibilities:
       1. It is meant to be used as a damaging force sent forth from a weapon at an opponent that will cause damage by subtracting health points away from the enemy.
    4. Constraints:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    5. Composition:
       1. Shoot()
          1. Fires the projectile while incorporating the upward direction and gravity.
       2. OnTriggerEnter()
          1. Compares to see if the projectile collides with “Walls”, “Ground”, or “Enemy” if it is true it detonates and causes damage to enemies if enemies are hit.
    6. Uses/Interactions:
       1. This is a child class of Projectile. This is used by Player and WeaponManager.
    7. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    8. Processing:
       1. See Composition
    9. Interface/Exports:
       1. n/a

|  |
| --- |
| **Proj\_Mine : Projectile** |
| upwardForce: protected float = 20f gravityResist: protected float = 3f rb: protected Rigidbody inMotion: protected bool |
| Awake(): void Shoot(): protected override void ResistGravity(): protected IEnumerator OnTriggerEnter(other: Collider): private void DieCountdown(): protected void |

* + 1. Classification:
       1. Child Class of Projectile
    2. Definition:
       1. Flies in an arch first going up and then effected by gravity bring it back down. They attach to the floor where they land and will explode on touching an enemy or after a set amount of time.
    3. Responsibilities:
       1. It is meant to be used as a damaging force sent forth from a weapon at an opponent that will cause damage by subtracting health points away from the enemy, as well as sticking to the ground where it lands and exploding on contact or after set amount of time. Will be destroyed on impact of a wall or damaging an enemy four times.
    4. Constraints:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    5. Composition:
       1. Awake()
          1. Attaches rigidbody component and Calls ResistGravity()
       2. Shoot()
          1. Fires the projectile while incorporating the upward direction and gravity.
       3. ResistGravity()
          1. manages the gravity effects on the projectile
       4. OnTriggerEnter()
          1. Compares to see if the projectile collides with “Walls” and “Ground”, this will cause the projectile to stick to the surface and set the timer for destination.
          2. Also compares the collision tag with “Enemy” and if it is true it detonates and causes damage.
       5. DieCountdown()
          1. Starts the countdown for the explosion of the projectile.
    6. Uses/Interactions:
       1. This is a child class of Projectile. This is used by Player and WeaponManager.
    7. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    8. Processing:
       1. See Composition
    9. Interface/Exports:
       1. n/a

|  |
| --- |
| **Proj\_PlasmaOrb : Projectile** |
| numberHits: protected int |
| OnTriggerEnter(other: Collider): private void |

* + 1. Classification:
       1. Child Class of Projectile
    2. Definition:
       1. This is a type of projectile that has a health count on how many enemies it can hit before it disappears. Flies in a straight line.
    3. Responsibilities:
       1. It is meant to be used as a damaging force sent forth from a weapon at an opponent that will cause damage by subtracting health points away from the enemy, as well as phase through them and continue moving forward. Will be destroyed on impact of a wall or damaging an enemy four times.
    4. Constraints:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    5. Composition:
       1. OnTriggerEnter()
          1. Compares the collision tag of the bullet with “Wall” to destroy the object.
          2. Compares the collision tag with “Enemy” and decreases the health of the enemy by 1 and if the impact has hit 4, it destroys the projectile.
    6. Uses/Interactions:
       1. This is a child class of Projectile. This is used by Player and WeaponManager.
    7. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    8. Processing:
       1. See Composition
    9. Interface/Exports:
       1. n/a

|  |
| --- |
| **MovementWave : MonoBehaviour** |
| amplitude: public float = .2f angleDistance: protected float = Mathf.PI rb: Rigidbody |
| Awake(): void FixedUpdate(): void |

On FixedUpdate, MovementWave adjusts the Vector3 by transform.right \* moveDist \* Mathf.Cos(angleDistance) \* amplitude.

-angleDistance is then decreased .2f.

* + 1. Classification:
       1. Visual Effects Class for Projectiles
    2. Definition:
       1. To give visual effects to the movement of projectiles.
    3. Responsibilities:
       1. Make the projectile move in a sinusoidal manner while travelling forward.
    4. Constraints:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    5. Composition:
       1. Two Functions:
          1. Awake()

This is gets the Rigidbody information to alter the position of the projectile.

* + - * 1. FixedUpdate()

This is what causes the projectile to actually sway by modifying the move distance, amplitude, and angle.

* + 1. Uses/Interactions:
       1. This affects all Projectiles and childs of the class.
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. See Composition
    4. Interface/Exports:
       1. n/a

|  |
| --- |
| **Pickups\_GunEffects : MonoBehaviour** |
| gunPistol: protected const int = 0  gunRifle: protected const int = 1  gunShotgun: protected const int = 2  gunRocket: protected const int = 3  gunGrenade: protected const int = 4  gunMine: protected const int = 5  gunPlasma: protected const int = 6  firingNormal: protected const int = 0  firingSpread: protected const int = 1 firingHeavy: protected const int = 2 firingAutomatic: protected const int = 3  shotsNormal: protected const int = 0  shotsRapid: protected const int = 1  shotsWave: protected const int = 2  shotsRusty: protected const int = 3  shotsBroken: protected const int = 4  shotsDoubleBarrel: protected const int = 5 randomizePickup: public bool randomGunTypeChance: protected float = .3f randomFiringTypeChance: protected float = .3f randomShotTypeChance: protected float = .3f randomProjectileTypeChance: protected float = .3f gunType: private int firingType: private int shotType: private int projectileType: private int |
| Awake(): void RandomizeItem(): private void DisplayName(): private void NameGun(): private void NameFiring(): private void nameShotType(): private void NameProjectileType(): private void StartGunParticles(): private void StartFiringParticles(): private void StartShotParticles(): private void StartProjectileParticles(): private void OnTriggerEnter(other: Collider): private void |

* + 1. Classification:
       1. Weapon Modifier Pickup
    2. Definition:
       1. A player weapons altering pickup.
    3. Responsibilities:
       1. Alters Player weapon and modification settings depending on randomly chosen pickup types.
       2. Can change one or all gun, firing, shot and projectile types based on a randomized value check.
    4. Constraints:
       1. Only changes player weapon and modifiers
    5. Composition:
       1. Made of several functions:
          1. Awake():

If the pickup is randomized, call RandomizeItem()

Otherwise, start particle effects based on pickup type (gun, firing, shot or projectile)

* + - * 1. RandomizeItem()

Run random.value once each for each of the variables (gun, firing, shot, and projectile). If the value is less than the specified chance of each type, set the pickup to have one of those types from their respective values at random.

If no items pass the Random.value checks, set all four types to be attached to the pickup. For each type, randomly choose one of the types from their respective values at random.

* + - * 1. DisplayName()

Create a text display to show the name. firing, shot and projectile types (if applicable) of the pickup. Calls each of the respective methods listed below.

* + - * 1. NameGun()

Attaches the name of the gun type (if it is chosen) to the pickup text name.

* + - * 1. NameFiring()

Attaches the name of the firing type (if it is chosen) to the pickup text name.

* + - * 1. NameShotType()

Attaches the name of the shot type (if it is chosen) to the pickup text name.

* + - * 1. NameProjectileType()

Attaches the name of the projectile type (if it is chosen) to the pickup text name.

* + - * 1. StartGunParticles()

If a gun type is chosen, instantiate gun particles.

* + - * 1. StartFiringParticles()

If a firing type is chosen, instantiate gun particles.

* + - * 1. StartShotParticles()

If a shot type is chosen, instantiate gun particles.

* + - * 1. StartProjectileParticles()

If a projectile type is chosen, instantiate gun particles.

* + - * 1. OnTriggerEnter(Collider other)

If other has a “Player” tag, find the attached WeaponsManager and change the player’s weapon settings to the pickup types specified.

If the pickup type is gun type only, change the players gun type and reset the remaining modifiers.

* + 1. Uses/Interactions:
       1. Interacts with the player character.
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to composition
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **DestroyByTime : MonoBehaviour** |
| timeToDie: protected float = 2.0f |
| Start(): void |

* + 1. Classification:
       1. Stand Alone Class
    2. Definition:
       1. Sets a set life on an object in the game, making sure that it doesn’t stay active for an extended period of time.
    3. Responsibilities:
       1. This is an effect during game play, it causes the bullets that have been shot from a gun or from an enemy to disappear. This allows for projectiles to not over live their intended lifespan.
    4. Constraints:
       1. Relevant to only Projectiles
    5. Composition:
       1. One Function:
          1. Start()

Utilizes built in function Destroy(), that takes the targeted gameObject and the designated float that holds the amount of time it will survive.

* + 1. Uses/Interactions:
       1. Projectiles.cs and its child classes, as well as TrackingMine.cs
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. This is applied every time a projectile is created and used.

|  |
| --- |
| **Hazard : MonoBehaviour** |
| name: public new string damageDone: protected float canHarmPlayer: protected bool canHarmEnemies: protected bool aliveAfterContact: protected bool |
| Hazard(): public DamageEnemy(other: Collider): protected void DamageHazard(): protected void DamagePlayer(other: Collider): protected void OnTriggerEnter(other: Collider): void |

* + 1. Classification:
       1. used for enemies and traps
    2. Definition:
       1. A parent Class for Objects that can harm players.
    3. Responsibilities:
       1. Applies damage to the player and enemy
    4. Constraints:
       1. Relevant mostly to the player and enemy
    5. Composition:
       1. Four Functions
          1. Hazard()

A Constructor that gives a defult value to the name variable

* + - * 1. DamageEnemy(Collider other)

Controls the enemy damage

* + - * 1. damgeHazard()

Checks to see if player or enemy is dead

* + - * 1. DamagePlayer(Collider other)

Controls player damage

* + - * 1. OnTriggerEnter(Collider other)

checks if the player or the enemy is hit with damage

* + 1. Uses/Interactions:
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **PauseMenu : MonoBehaviour** |
| PauseUI: public GameObject paused: private bool player: private GameObject pm: private PlayerManager |
| Awake(): void Start(): void Update(): void PausePressed(): public void Restart(): public void MainMenu(): public void Quit(): public void |

* + 1. Classification:
       1. Stand alone class
    2. Definition:
       1. A class responsible for handling how the pause menu is executed
    3. Responsibilities:
       1. The Pause menu class allows to pause, pull up menu ,quit and restart.
    4. Constraints:
       1. must be called by a manager class.
    5. Composition:
       1. Made of 7 Functions
          1. Awake():

Sets the Pause Menu as inactive.

* + - * 1. Start():

This method sets up the access to the player movement script in order to disable movement while paused.

* + - * 1. Update():

Checks to see if the pause button (“escape”) is pressed. Call PausePressed() if so.

* + - * 1. PausePressed():

i. This method allows the player to pause/unpause and also set player controls to active and inactive.

e. Restart():

i. Restarts level by loading it again.  
 f. MainMenu():

i. Reloads the Main scene.  
 g. Quit():

i. Quits the game entirely

* + 1. Uses/Interactions:
       1. changes the game state to the pause menu. from there the menu acts as a catalyst to change the state depending on the player choice. the choices are based on mainMenu() quit() restart() and pausePressed()
    2. Resources:
       1. unity game engine
    3. Processing:
       1. when the paused bool is changed to true.
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **DeltaMusicManager : MonoBehavior** |
| song: public AudioSource |
| Start(): void  Update(): void ChangeSong(AudioClip song2): public Void{  song.Stop() : struct  song.clip = song2 : struct  song.Play() : struct  } |

* + 1. Classification:
       1. Music Class
    2. Definition:
       1. Manages the playing of music and transitioning between songs.
    3. Responsibilities:
       1. This is the background class of operating functions that allows different songs to be played.
    4. Constraints:
       1. The menu must be created inside of the GameManager.cs in order to activate this aspect of the game.
    5. Composition:
       1. Two Functions:
          1. Update()

Checks for updates each frame.

* + - * 1. ChangeSong()

uses built in struct AudioSource to access the different functions to change the song.

.Stop()

stops current song

.clip

sets new song

.Play()

plays loaded song

* + 1. Uses/Interactions:
       1. this will be collaborating with
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **CameraController : MonoBehaviour** |
| player: public GameObject nextCamPosition: protected Vector3 offset: privateVector3 cameraHeight: public float = 30f cameraZOffset: public float = -2f followSpeed: protected float = 40f followPlayer: protected bool approachingPlayer: protected bool |
| Start() LateUpdate() AttachPlayer(): protected void FollowPlayer(shouldFollow: bool): public void SetNextPosition(pos: Vector3): public void ApproachNextCamPosition(tempFollowSpeed: float): public IENumerator MoveTowardsPlayer(): protected void IsApproachingPlayer(): public bool |

* + 1. Classification:
       1. Camera class
    2. Definition:
       1. This class allows the camera to follow the character around the level
    3. Responsibilities:
       1. Attaches and follows the player. It can also disable a camera and set the speed of the camera
    4. Constraints:
       1. It is only used for the player camera
    5. Composition:
       1. Several functions:

1. Start():
   1. Initializes a three dimensional Vector with the height of the camera and the position along the Z-axis
2. LateUpdate():
   1. Checks to see if the camera is following the player. If it is then the MoveTowardsPlayer() function is called

c. AttachPlayer():

i. Checks to see if the player exists if the player doesn’t then a camera is attached to the player.

d. FollowPlayer():

i.Attache the camera to the player and follows

follows the player.

e. SetNextPosition():

i. Sets the camera to a three dimensional vector

position that is passed in as a parameter

f. ApproachNextCamPosition():

i.Attaches the camera to the player and moves it

in position to follow the player

g. MoveTowardsPlayer():

i.Sets the position of the camera close the the

players position

h. IsApproachingPlayer():

i.Checks to see if the camera is following the

player

* + 1. Uses/Interactions:
       1. Moves the position of the camera with relation to the player
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **GenericShake : MonoBehaviour** |
| isShaking: [NonSerialized] public bool |
| ApplyShake(noise: Vector2): public void Enable(): public void Disable(): public void |

* + 1. Classification:
       1. Effects Class
    2. Definition:
       1. Superclass that applies a rapid shaking movement to the camera.
    3. Responsibilities:
       1. when an event is triggered then the camera will shake. The Camera tends to not be a smooth shake and can be unpleasant.
    4. Constraints:
       1. Relevant mostly to the player. Will not work with other elements
    5. Composition:
       1. Three functions
          1. ApplyShake(Vector2 noise)

Uses a two dimensional vector to get the X and Y coordinates and moves them in different positions

* + - * 1. Enable()

enables shaking

sets the isShanking varible to true

* + - * 1. Disable()

disables the shake

resets the position of the camera

* + 1. Uses/Interactions:
       1. CamerPerlinShake.cs
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. N/A

|  |
| --- |
| **CameraPerlinShake : GenericShake** |
| magnitude: public float = 2f frequency: public float = 10f |
| Update(): void PerlinShake(): public Vector2 |

* + 1. Classification:
       1. Child Class of GenericShake
    2. Definition:
       1. This an effect that will be applied to the Player to shake the camera.
    3. Responsibilities:
       1. This allows the camera to shake smoothly on different kinds of impacts and actions taken between the different objects(4). Implements GenericShake but removes rapid movements between positions.
    4. Constraints:
       1. There must be a Player attached to the LevelGenManager in order for this effect to be used.
    5. Composition:
       1. Two Functions
          1. Update()

updates every frame to verify if it the action should be shaking. By checking inherited class ApplyShake().

* + - * 1. PerlinShake()

sets the power of the shake to the X and Y components of the camera to make the vibration effect the camera appropriately.

* + 1. Uses/Interactions:
       1. This component takes effect when the Player is damaged by an enemy or obstacle. It is applied through the LevelGenManager and the attached Player and Camera that are needed for the game script to work.
    2. Resources:
       1. Standard software expectations defined in the SRS. There should be no race or deadlock conditions that would affect this.
    3. Processing:
       1. Refer to Composition.
    4. Interface/Exports:
       1. n/a

# 7. Glossary

An ordered list of defined terms and concepts used throughout the document.

* **Viable Level Layout:** Is an organization of 3d tiles such that the tiles come together to form a room suitable to be played on. An example to this is that tiles will not show up alone as to trap the user to make areas of the map inaccessible. A viable level layout consists of a continuous series of tiles that are all connected. A viable level layout also consists of not to many and not too little amount of tiles placed as to not make the level too big or too small.
* **Particle Systems:** A representation of effects that are not solid such as liquids, smoke, clouds, etc. These systems are mostly made up of small simple images that are displayed and moved around in numerous quantities. These effects have predetermined lifetimes and usually only last for a few seconds.
* **Perlin Noise**: A random pattern of floating numbers that is made in a two dimensional plane. The pattern is done in waves that increase and decrease.
* **Script**(1) : a document containing code that is needed for the game to operate and function as necessary.
* **Features**(2) : pertaining to any ideal situation in that is meant to be executed. Such as: taking damage, running into a wall, fighting, jumping, running.
* **Objects**(4) : player, enemies, floor, walls and features of the game
* **Actions**(3)  : walk, run, attack
* **Player**(5) : character controlled by the user playing the game.
* **Projectiles**(6) : attacks coming from enemies and player from any weapon that shoots a bullet or an attack.

# 8. Bibliography

A list of referenced and/or related publications.

Brad Appleton <[brad@bradapp.net](mailto:brad@bradapp.net)>

<http://www.bradapp.net>

Levels Layout  
Accompanying document by Team Delta

Unity Official Documentation about Particle Systems

<https://docs.unity3d.com/Manual/ParticleSystems.html>

Unity Official Documentation about Perlin Noise

https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html