**Software Design and Implementation**

**for**

**Uproar**

**Version: 1.0.3**

Prepared by:

Andrew Honigman, William Baar, Keila Cressman,

Isaiah Jones

**Abstract:**

The software we have designed is a for 2-player platforming fighting game called Uproar, which is usable on many different gaming consoles and computers. This game features multiple maps and characters to choose from, who shoot at each other in order to drain each other’s health bars and claim victory. The game also features menus such as a start menu, a character select menu, and a win screen which allows users to return to their previous match or the character select screen. This project was developed over the course of three months and was implemented entirely using the Unity engine and assets bought from the Unity Asset Store.

**Table of Contents**

1. **Introduction 4**
2. **Current Solutions 4**

**2.1 - Smash Bros. 4**

**2.2 - Samurai Gunn 4**

**2.3 - Differences 4**

1. **Design Goals 5**

**3.1 - Functional Requirements 5**

**3.2 - Non-Functional Requirements 5**

1. **Logical View 5**

**4.1 - High-Level Design 5**

**4.2 - Detailed Class Design 7**

1. **Use Case View 8**

**5.1 - Loading Game 8**

**5.2 - Multiple Maps 9**

**5.3 - Multiple Characters 9**

**5.4 - Go to Match Screen 9**

**5.5 - End of Match Screen 10**

1. **Design and Implementation 10**
2. **Results and Future Work 21**

**7.1 - Gantt Chart 21**

**7.2 - Final Performance and Results 22**

**7.3 - Future Goals 23**

**1 Introduction:**

Our creation is a new 2D multiplayer game titled Uproar. Developed with the Unity gaming engine, Uproar is a 1v1 fighting game that utilizes strictly shooting based combat, unique to other games of the genre. We decided upon Unity because of how user-friendly it is. Unity had a plethora of tutorials for us to watch and as well as go through step by step. Unity also has numerous amounts of assets, a type of plug-in, that we used to help ease the game creation process. Through the use of these gifts from Unity we eventually brought our game to fruition.

**2 Current Solutions:**

**2.1 Smash Bros:**

Super Smash Bros. is a 2-8 player fighting game with platforming elements and stages. Characters have a wide variety of attacks which have different effects on the recipient. When characters are damaged by attacks, their percentage value increases, and their launch distance from attacks increases with it. The goal of the game is to build percentage high enough so that you can knock another character outside of the screen boundary. Victory is determined either by points scored at the end of a time limit or via every other character losing all their lives.

**2.2 Samurai Gunn:**

Samurai Gunn is a 2-4 player fighting game with platforming elements and stages. Every character has two attacks, a melee attack and a ranged attack with limited uses, and they all play exactly the same. Whenever characters are damaged by attacks, they are instantly knocked off the screen and forced to respawn, granting a point to the attacking player. The game ends once one of the players achieves a certain amount of points.

**2.3 Differences:**

Our game, as of now, only has a single attack in the form of a projectile, and only allows for two players. Characters have set health bars, and once those health bars are depleted via attacks or collision with a hazard, the surviving player is deemed the victor. Our game is simpler and slower than the other two and features a small tutorial box to help players learn the game quickly. The game also gives users the ability to instantly replay a match with the same settings without any loading in between, and overall the menus and UI are much more quick and snappy.

**3 Design Goals:**

**3.1 Functional requirements:**

* The output of the system must be visual so that users can keep track of their actions and receive feedback.
* The output of the system must also have audio so that background music can be heard and allow users to become more engaged.
* The system must be able to accommodate for multiple controller devices.
* The system must come with a manual or digital guide in order to instruct users how to navigate and learn the game.
* The system must be able to transition to different scenes on player command.
* The characters in the system must be able to jump and shoot.

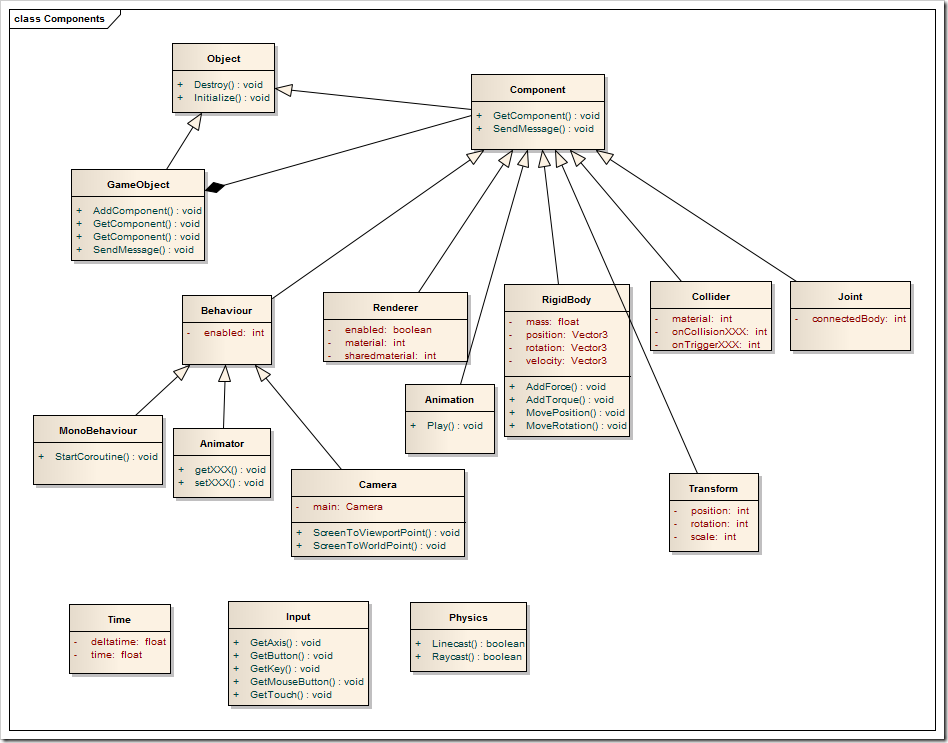
**3.2 Nonfunctional requirements:**

* The starting screen must load in less than 8 seconds.
* The game must perform actions based on input within 3 frames of the command being pressed.
* The system must not take unneeded memory from outside its boundaries.
* The characters must not be able to attack multiple times in less than half a second.
* The projectiles spawned by characters should not stay on screen for more than 2 seconds.

**4 Logical View:**

**4.1 High-level design:**

The Unity Game Engine system works as an independent program that can be installed and run on a computer. The diagram below details how its different components interact. Everything within a scene is a GameObject, which inherits from the Object class. Within these GameObjects, there is the option to add components with different functions. For example, you can add a rigidbody2D component to a character GameObject, so physics and C# scripts can be applied. In addition, the environment around the Unity system involves a game console of the user’s choosing. Each console has its own unique controller to provide input, a monitor to visually display output, and a speaker system to display sound effects/background music. Unity supports many different gaming consoles and allows the user to create a build with the console of his/her choosing. The list of consoles includes: PC, Mac and Linux Standalone, Universal Windows Platform, tvOS, PS4, iOS, Xbox One, Android, WebGL, and Facebook.



**4.2 Detailed class design:**

|  |
| --- |
| Character |
| -charSprite: object  -hurtbox: object  -colBox: object  -meleeAnim: animation  -shootAnim: animation  -runAnim: animation  -jumpAnim: animation  -movSpd: int  -movVector: Vector  -damage: int  -comeback: int |
| +run(): void  +jump(): void  +melee(): void  +shoot(): void  +getHurt(): int  +knockOut(): void  +projDisable(): void  +comebackCheck(): int  +respawn(): void |

|  |
| --- |
| Level1 |
| -colGround: object  -colHazard: object  -levDesign: object |
| +hazardPlayer(int): void |

|  |
| --- |
| ProjectileWeapon |
| +projectileHitbox: object  +projectileSprite: object |
| +projActive(): int  +projTravel(): void  +collidePlayer(int): int |

|  |
| --- |
| HealthSystem |
| -healthBar: object  -healthSlider: object  -healthValue: int |
| +decreaseHealth(): void |



**5 Use case view:**

**5.1 Tabular Use Case : Loading Game**

|  |  |
| --- | --- |
| Actors | User, Gaming Engine |
| Description | A user will be able to turn on the game by accessing game’s link in Unity. |
| Data | Game link |
| Stimulus | User entering and access the link |
| Response | Game loads and runs successfully |
| Comments | Game must be in console that can run Unity |

**5.2 Tabular Use Case : Multiple Maps**

|  |  |
| --- | --- |
| Actors | User, Keyboard, Game Engine |
| Description | User may navigate through the Map selection screen and select a Map |
| Data | Key input, Map selection screen |
| Stimulus | Command issued by user to keyboard to game engine |
| Response | System is given a Map to load for combat screen |
| Comments | Must have a keyboard for selection options |

**5.3 Tabular Use Case : Multiple Characters**

|  |  |
| --- | --- |
| Actors | User, Keyboard, Game Engine |
| Description | User may navigate through the Character selection screen and select a Character |
| Data | Key input, Map selection screen |
| Stimulus | Command issued by user to keyboard to game engine |
| Response | System is given a Character to load for combat screen |
| Comments | Must have a keyboard for selection options |

**5.4 Tabular Use Case : Go to Match Screen**

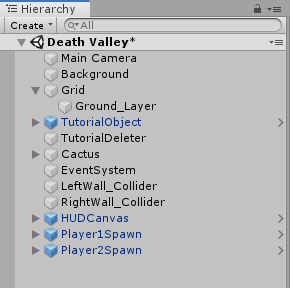
|  |  |
| --- | --- |
| Actors | User, Keyboard, Game Engine |
| Description | User may navigate through screen selections to reach the match selection screen to play with a friend. |
| Data | Key input, Match selection board |
| Stimulus | Command issued by user to keyboard to game engine |
| Response | Match scene will appear to begin fighting |
| Comments | To get to a match quicker it is necessary |

**5.5 Tabular Use Case : End of Match Screen**

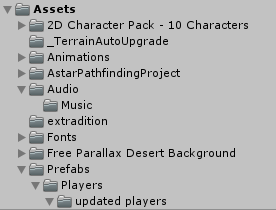
|  |  |
| --- | --- |
| Actors | User, Keyboard, Game Engine |
| Description | Once one of the Players is destroyed User has two options. User can instantly start same game or go back to menu with character and map options. |
| Data | Key input, End of Match Board |
| Stimulus | Command issued by user to keyboard to game engine |
| Response | Match scene will reappear to begin fighting or Character/Map selection screen is displayed for new Match scene |
| Comments | A selection must be made in order to continue |

**6 Design and Implementation:**

To give an overview of the general tools the Unity engine granted us, the building blocks of Unity that were used for this project are Assets, Objects, Scenes and Scripts. Assets are anything that the Unity project uses to modify or manipulate objects. Assets used for our project range from animations, images, fonts, character sprites, scripts, and scenes. Objects are elements that are directly placed in a scene, such as character spawners, tilemaps, and canvases to place text and imagery. Scenes are the different “episodes” of the game; they are the areas in which objects are placed and divided, and each scene has its own set of objects that usually only exist within that scene. Finally, Scripts are used in order to code the game using C#, and we attached those scripts to game objects in order to determine how they interact with their environment.



*Heirarchy View for Objects*

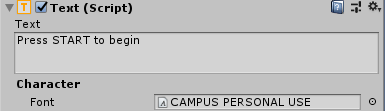
**

*Asset Folders*

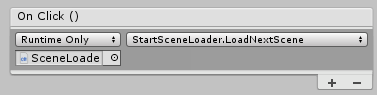
Beginning with the design of the menus, we used text, image and button components that were available as part of the “Canvas” type of Unity objects. Looking through the Scene view of the Unity project, we resized and rearranged the text box for the on-screen message, the background color, the image used for the map select screen and the buttons in order to make a neat screen layout. To modify the images and text, we typed in the text we wanted displayed into the text component along with a font we stored as an asset, and we placed the image we wanted to use in the Image component field. We used the buttons to navigate between different Scenes, or menus/stages in the Unity project, but assigning them methods within scripts that are designed to change the scene to a certain scene index. The scene index would depend on either the button pressed or the settings of other objects in the system, specifically the Global object that will be discussed later.



*Picture of the Start button*

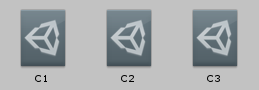
**

*Text Component of Button*

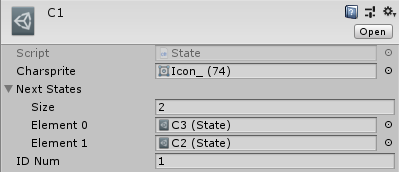
**

*Clicking Button triggers LoadNextScene in script*

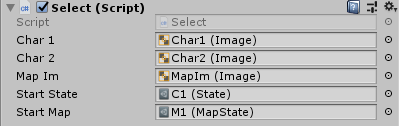
In order to allow the user to choose what characters and maps they wanted to play, we had to create functionality for the character select screen that would carry its data over to future scenes. The character icons and map icon that the users can choose on the character select screen are contained in special objects called “States”. States are so-called ScriptableObjects, which are essentially instances of a script that you can choose to make an option to create in the Asset menu. ScriptableObjects are useful for storing data, and in this case we are using them to store the images on the character select screen. Within these State files, they also store an input field in the Unity inspector called nextStates[], which is an array of States that each State can swap to on the character select screen. Finally, they store an integer that is used to look at them in later scenes. Using a separate script called “Select”, we display the images in the currently active states on the screen, and we allow the player to switch which states are active with different key presses. When the button on the UI is clicked, the system checks the ID integer of the currently active MapState, and loads the Scene that is indicated by that ID and contains the correct map.



*Character State objects*

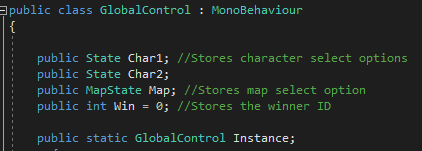
**

*Elements of State objects*

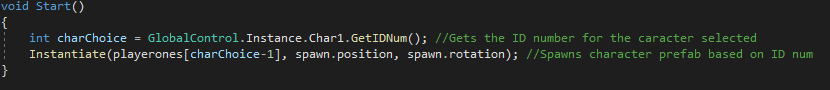
**

*Select script changes these images based on active states*

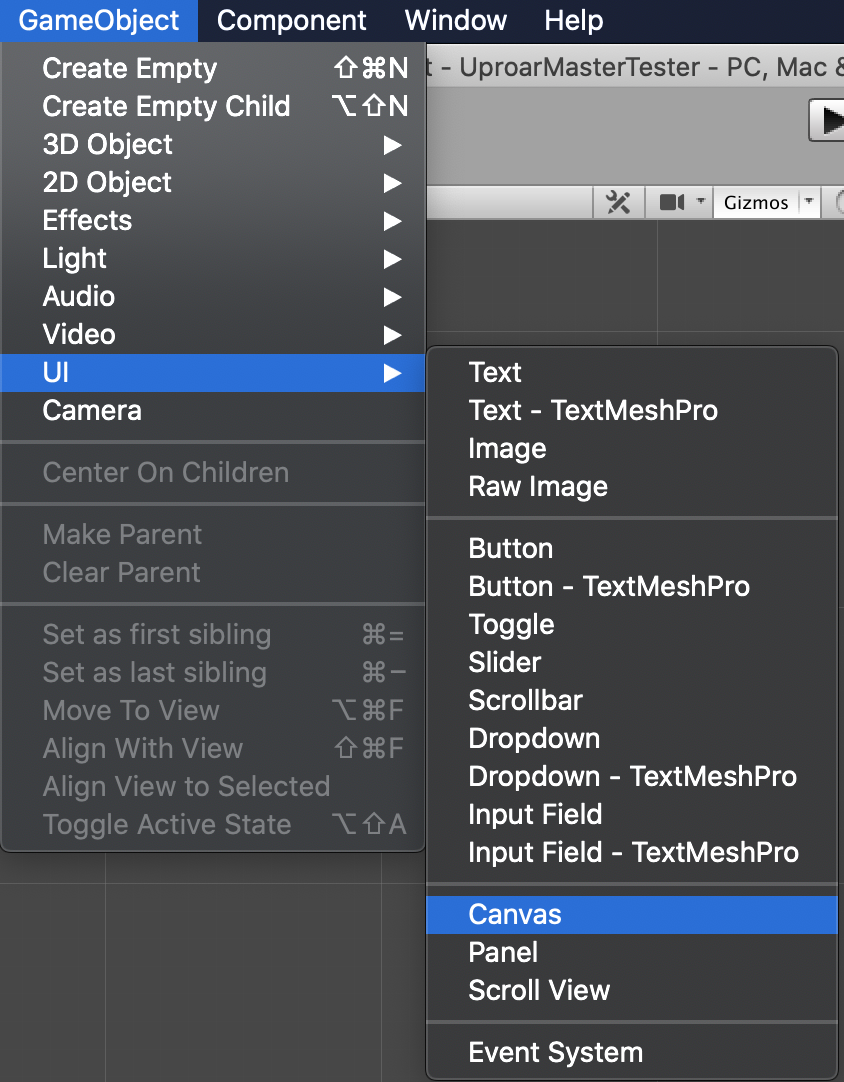
Each Scene in Unity contains its own objects and settings, so whenever we transition to another Scene all Objects and effects of Scripts are destroyed. In order to save the character select screen options, the player who wins each battle, and what map to return to on instant rematch, we needed to make an Object with a Script attached that has the function dontDestroyOnLoad(). This Object is called the “Global” Object, and the public Script is called “GlobalControl”. The GlobalControl Script makes the Global object an instance of it, and the instance stores four different variables. It records the 3 different States used in the character select Scene and an integer that is used for determining who won in a match. When another scene loads, it is able to look at the GlobalControl instance and decide what characters to spawn on the combat screen, as well as what victory message to display on the win screen and what map Scene to return to if someone selects the Instant Rematch button.



*Variables stored on Global object*

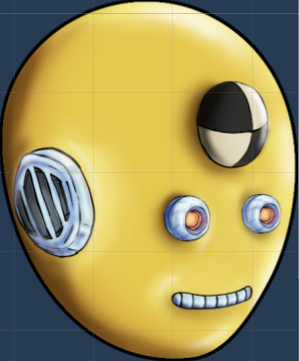
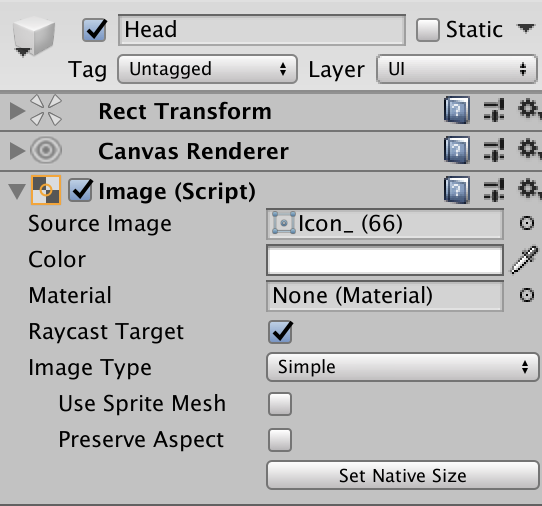
**

*Character spawner looks at Global object to decide who to spawn into the scene*

For the design of the Heads up display, or HUD as it will be referred to from now on, we used the “Canvas” GameObject which was supplied by Unity(See Figure A).

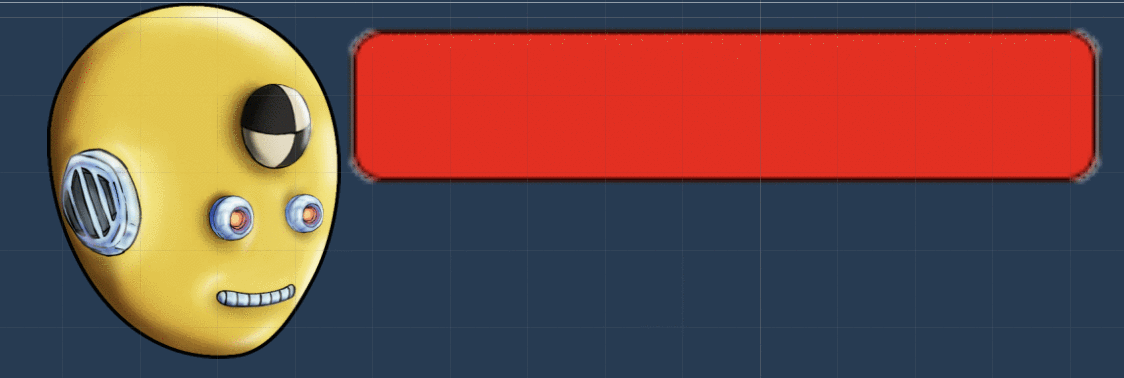
*[Figure A] Location of Canvas object in Unity*

This is where we have placed all the health elements for our players. First, we created an empty child of the HUD object to place our UI elements for an individual character, this element we called “Player 1”. From there we created another child from the Player 1 object this time, however, it’s an image UI element. This image object will be for displaying the head of the player’s chosen character(See Figures B/C).

*[Figure B] Head Icon [Figure C] Inspector view of Image UI element*

The next child added to Player 1 was the slider UI element for the health bar, with children of its own, a background, fill area, and fill. The background element is just to provide the empty space for the fill area if the player takes damage and the fill area decreases(See Figure D).



*[Figure D] Showing how slider value decreases to reveal background (viewable on drive)*

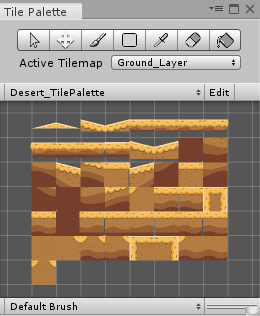
The fill area element controls the size of the area to be filled, so we adjusted the size to be just a bit smaller than the background. When the fill area decreases, it fits just inside to the background. The fill element simply controls the color and type of fill; for example, the fill could not fill the center of the area or the fill could increase instead of decrease. For a fighting game, however, we decided that the fill area should be filled completely and decrease. Once these elements have been placed in the canvas where we wanted them at the size we wanted, we duplicated them and moved them to the opposite side of the canvas and made sure we like the orientation of the UI elements(See Figure E).



*[Figure E] Final Prefab with two seperate players*

Finally, we made the HUD into a Prefab to make it easily adjustable, duplicable, and portable to our other maps.

For the map design, we used downloadable tilesets and background images to customize three unique map themes: winter, desert and beach. Unity provides two user-friendly features called “Tilemap” and “Tile Palette”, to assist in the creation of maps. The Tile Palette window allows for the user to drag and drop tilesets, sprites or sprite texture assets, which can then be painted directly onto the Scene (See Figure 1).



*[Figure 1] Tile Palette Window*

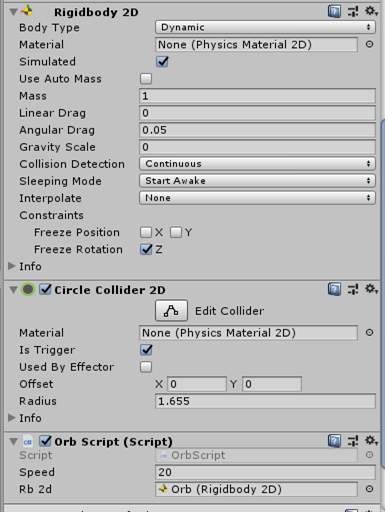
Before painting onto the scene, you must create a Tilemap GameObject. This will show up as “Grid” on the hierarchy with the child Tilemap. Once Grid is selected, click on any tile from the Tile Palette window and begin painting onto the scene. In addition, adding in a background layer is simple. You can download and import any images from Google, the Unity Asset Store, etc., and simply drag the chosen image into the hierarchy. All backgrounds and tilesets used in Uproar were downloaded from the Asset Store.

Furthermore, our desert and winter themed maps, “Death Valley” and “Antarctica”, both have a hazard that deals one-hundred points of damage to the players on collision. On Death Valley, the hazard is a pit of cacti, and on Antarctica, the hazard is the snow at the bottom of the screen. In order to create these hazards, we needed to implement a damage/health system, which involved a C# script. In our PlayerController scripts, we added one global variable: public int currentHealth = 100, and three methods: OnTriggerEnter2D(Collision other), TakeDamage(int amount) and Death(). The global variable serves as the player’s starting health amount. The method OnTriggerEnter2D(Collision other) waits for an event to occur. This event would be any form of collision, whether it be from a bullet, hazard, collectible, etc. The Unity Game Engine provides a “Tag” feature, which allows all GameObjects to be labeled. This label is then used within scripts to access specific GameObjects. Some examples of tag names used in Uproar are: Hazard, Weapon, Player1, and Player2. Once we assign the “Hazard” tag to the cacti and snow GameObjects, we can easily access them in the player’s OnTriggerEnter2D(Collision other) method. We can identify which GameObject the player came in contact with a simple else-if statement.

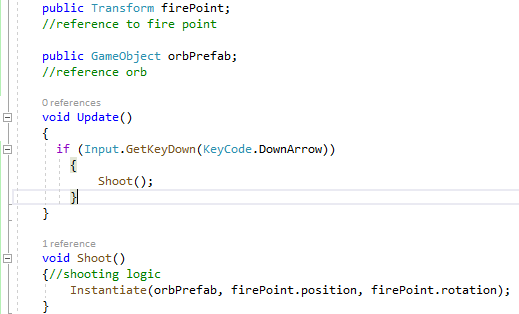
Next, once the the OnTriggerEnter2D(Collision other) method determines that the player has collided with a hazard, it calls the TakeDamage(int amount) method, and passes the argument 100. Within the TakeDamage(int amount) method, the parameter “amount” is subtracted from the player’s global variable, currentHealth. Since hazards are set to deal one-hundred damage points, the player is instantly killed, and the method Death() is called. Death() destroys the player game object and signals the GlobalController to transition to the end scene. The same process takes place when one player shoots another, however, bullets only deal ten damage points, so the Death() method is not called (unless the player has ten or less health points).

For movement, the character buttons are connected to the left and right arrow keys or the A and D buttons. Making a character move left/A gives a movement input of -1 and moving right/D gives an input of 1. These inputs are given via an “Axis” in Unity, which is set up through the Build Settings. These movements are performed by applying a left or right force to the character’s RigidBody2D component. We needed two separate “Axis” setups to accommodate different inputs from two players. When making the characters jump, we apply an upward force to their RigidBody2D component, and use a variable to determine how many times they can jump. We also use the collider components of the characters and tilemaps to determine if the character has touched the ground. If the boolean for isGrouded is true, then the amount of times the character can jump is returned to an initial value, in our case 2 times. The characters also have a set value for the downward force of gravity which is naturally given to us by Unity.

Now we shall dive into the technical part of creating a projectile. First every projectile is given the following components (See Figure 2) : (1) “Rigidbody 2D” to help with the physic aspects. Here we have the “Body Type” set to “Dynamic” because we would like this game object to move when it needs to. We freeze the “Z Rotation” so when the game object does collide with another game object it does not fly in a misguided direction. (2) “Circle Collider 2D” to detect a collision. The “Is Trigger” is checked so we may use it to create an event when the collider touches another game object. (3) weapon script is attached to help control the speed and manipulate the “Rigidbody 2D” of the projectile. In the script we have it set that if the projectiles collider is to touch another game object it would destroy itself.

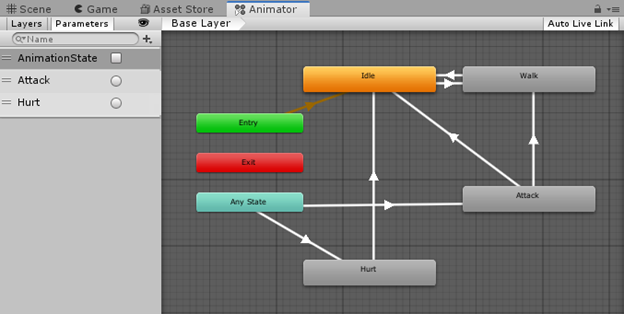


*[ Figure 2 ] Projectile Components*



*[ Figure 3 ] Weapon Script*

In the “PlayerController1/2” script (See Figure 3) a “Transform” component is added to the place where the projectile will spawn from. There is also a method for the down arrow key to call another method Shoot() that instantiates the weapons prefab at that point.

.

*[ Figure 4] Animator*



*[ Figure 5 ] Animations*

Now we’ll proceed to the animation tab and the animations. We have each set of snip animations that each character can do (See Figure 5). The picture above (See Figure 4) helps set up the event in which a certain button is pressed or when a Trigger is activated. The parameter “AnimationState” is a boolean between the Idle and Walk animations in the Animator window. If no buttons are pressed the “AnimationState” is set to false. Once a button is pressed it sets the animation state to true and activates the walk animation. The parameter “Attack” is a Trigger. Much like the “Is Trigger” for the components when the down button is pressed it switches the animation arrows to activate the “Attack” animation and once done goes back to “Idle”. The parameter “Hurt” is also a trigger that happens in the event of a projectile's collider touching the player.

**7 Results and Future Work:**

**7.1 Gantt Chart**

Below is a copy of our Gantt Chart schedule. We completed all tasks necessary to have a fully functioning video game, but struggled to complete some tasks of lower precedence. For example, due to the time constraint, we never implemented respawn points or character warping.

|  |  |  |  |
| --- | --- | --- | --- |
| ID | Task Name | Duration | Predecee |
| 1 | **Software Project** | 16 wks |  |
| 2 | Design | 3 wks |  |
| 3 | **Specification** |  |  |
| 4 | Proposal Document | 1.5 wks | 2 |
| 5 | Requirements Document | 1.5 wks | 4 |
| 6 | Elevator Pitch Presentation | 1 wk | 4 |
| 7 | Read Book Chapters 1-7 | 3 wks | 5 |
| 8 | Unity Technical Presentation with live demo | 1 wk | 7 |
| 9 | **Programming** |  |  |
| 10 | Unit Tests for first map (background, tilesets, hazards) | .5 wk | 8 |
| 11 | Create map designs | 1 wk | 10 |
| 12 | Create/Download Characters | .5 wk | 7 |
| 13 | Character movement/Animations | 2 wk | 12 |
| 14 | Health/Point/Pickup system | 1 wk | 7 |
| 15 | UI implementation (menus) | 1 wk | 8 |
| 16 | UI implementation (incombat) | 1 wk | 8 |
| 17 | Menu tree/transitions | 1 wk | 15 |
| 18 | Program hazards | .5 wk | 11, 14 |
| 19 | Program respawn point | .5 wk | 11, 14 |
| 20 | Implement character warping | 1 wk | 11, 13 |



**7.2 Final Product Performance and Results**

Once Uproar is launched, a UI start menu is displayed to the screen with a “Start” button. Once the button is selected, the game transitions to a UI character and map select scene. Uproar has a 2-player game mode without a CPU mode, so two users are needed to play. Both players have the option between three different characters and three different maps. The maps provide three different themes: desert, winter and beach, and are titled “Death Valley”, “Antarctica”, and “Sunnyland”. Once both players have chosen a character and agreed on a map, there is another “Start” button for them to press. This transitions to the scene containing the chosen map, while spawning the chosen characters on opposite sides of the map. All maps have a tutorial UI with instructions on how to play, which can be removed by pressing the spacebar. “Death Valley” and “Antarctica” are unique because they have a hazard that deals one hundred points of damage to the characters on collision. In “Death Valley”, the hazard is a pit of cacti, while in “Antarctica”, snow is the hazard.

Furthermore, each character starts with one hundred health points, so falling into a hazard instantly ends the game. In addition, the characters have the ability to run, jump, idle, and shoot. When shooting, each bullet does ten damage points on contact, so a character must shoot his/her opponent ten times successfully to win. Once a character’s health drops below zero points, the game ends and transitions to an end scene. This end scene displays the winning player, and two different options: a button that allows for an instant rematch with the same map and characters, and a button that transitions back to the character select scene.

**7.3 Future Goals**

We have more than twenty future goal ideas, however, only tasks of highest priority will be listed. First, Uproar currently has one game mode: two-player without an option to play against the CPU. This is an issue because some users may not have a friend to play with. Also, if Uproar were to be played at a party, only two people could play at once, which would result in others waiting to play. Therefore, our first goal is to create CPU, four-player and eight-player game modes.

Next, we began implementing collectibles on each map, but did not have enough time to fully debug the feature. The collectibles were developed to add a score system into Uproar. For every coin collected by a player, he/she would receive twenty-five points, and once his/her score were to equal three hundred points, that player would win the game. The main problem was, we could not figure out how to respawn the collectibles after they were collected. Also, the player’s bullets would collide with the coins and disappear. This all being said, our future goal includes fully debugging our collectible implementation and then adding in power-ups, like speed and/or damage increase.

Lastly, now that we have created a fully functioning video game, we must receive feedback from potential customers or friends. So far, we are the only people that have played and tested Uproar. We need to make sure we are satisfying the interests of our targeted audience. Therefore, we plan to advertise our current product as a free-to-download beta version. This will serve as free unit-testing, which will save us time and resources.