ENSE 400/477

University of Regina
Software Systems Engineering
Fall/Winter 2019/2020 Semesters

System/Process Documentation

Capstone Project:
Stride Champions
Platform Fighting Game

Developers:
Jeremy Cross
Taylen Jones

Advisors:
Craig Gelowitz
Tim Maciag

Table of Contents

Abstract

This document will discuss and outline the design processes used as well the code behind all of our functionality with our project. This document is broken down into two major sections. The first section covers the process and design of our project. This includes character designs, Stages design, user interface design, music design, design processes, and Testing. The second section of this document will cover the code and logic of our game. This section will dive deep into the code and discuss how everything will work.

1.0 Project Design and Development

1.1 Characters

For our project we designed and developed two characters to be developed and implemented into the game. We initially started development in Blender, but after weighing our options we decided that it would be too time consuming without proper training in the software to create fully functioning character rigs. We instead migrated over to Adobe Fuse and Mixamo, which integrated nicely into Unreal Engine 4. Within Adobe Fuse, we created two character models, one female and one male, along with their assets such as clothing, hair, etc. These models were then moved over to mixamo to easily apply rigging and animations before ultimately being imported into Unreal to be used as fully functioning character models.

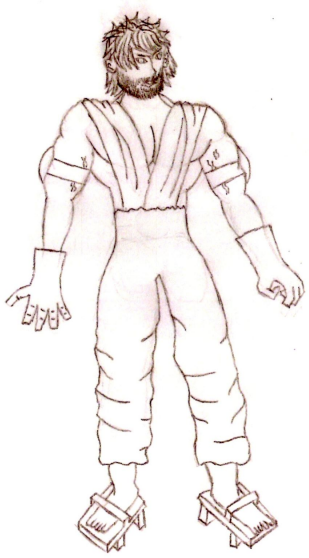

Figure 1.1-1

Figure 1.1-2

Figure 1.1-3

Figure 1.1-4

1.2 Stages

Due to the flexibility we introduced in our story, we were able to have varying themes when developing our stages. The stages were developed directly in the Unreal game engine. We have 2 ready to use stages, The Coliseum (Fig. 1.2-1) and the Cosmic Rift stage (Fig 1.2-2).

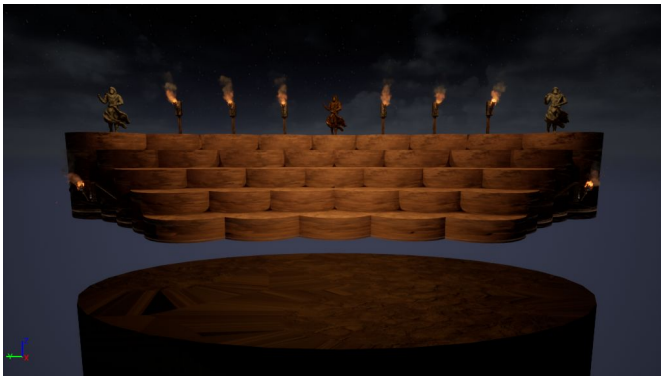

Figure 1.2-1

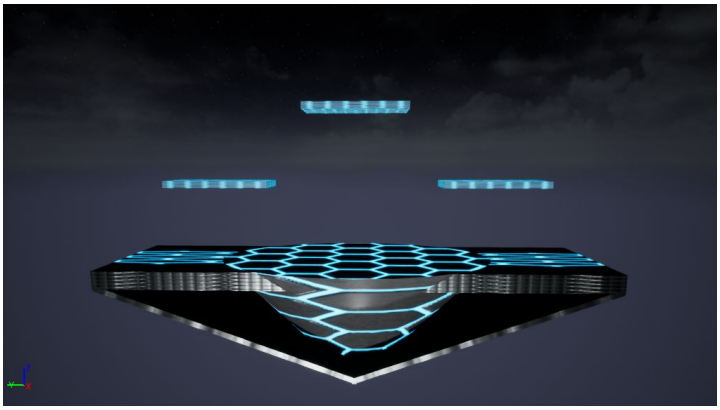

Figure 1.2-2

1.3 User Interface

For our user interface, we tried to make it as seamless and easy to follow as possible. By sticking to design principles taught in previous classes, we kept it clear, concise, consistent, responsive and efficient.

Figure 1.3-1

1.4 Story

The storyline for our project takes place in a futuristic setting, within multiple parallel universes. We chose this type of theme for our project to keep the addition of new characters and stages open, rather than restrict our options. More information can be either found on our github or in game through the extras menu.

1.5 Music

For this project we designed two pieces of original music for our game. These music pieces play during the stages when they are loaded. When designing these songs we tried to build them around the themes of the stages. We used FL studios when designing the songs, however we really only used this program at a surface level. FL Studios is very complex and we really didn't have the time to explore it to any great detail

1.6 Project Roadmap

For our project we cataloged all of our milestones and accomplishments with a user story map. Over the last two semesters we recorded all of the features and functionality as we completed them. Our full USM is available on our github page.

1.7 Design Processes Used

For our project we will use an Agile architecture for our main methodology. We did our work in 2-week sprints with scrum meetings between these with our appointed advisor(s). This methodology allowed for easy adaptation to changes depending on roadblocks faced during development. This method also allowed us to focus on one task at a time, while also getting constant feedback during our meetings, allowing for seamless iterations.

1.8 Technologies Used

Multiple different technologies and software were used in the development of this project. Unreal Engine was used as the main game engine to develop the gameplay mechanics and stage designs. We felt Unreal was the best path to take with this project being our first experience in game development. Along with that, Unreal is developed by Epic Games who have a very large community with a large amount of resources available online.

Adobe Fuse and Mixamo were used to develop the character models and rig them with animations. As discussed in the previous section, they were much easier to work with in regards to importing the models into Unreal Engine. FL studio was used exclusively to create the two original pieces in our project. Github was used for our revision source control, Balsamiq for our prototyping, and Milanote for online character storyboarding.

1.9 Testing

For testing our project we covered many different cases with both black and white box testing. For black box testing we made sure that all of our game mechanics work on a surface level. With white box testing we make sure that the internal code works for all of these mechanics as well. We also left room for comments for each of these tests detailing any issues that there may be. The document for our testing results is available on our github page.

2.0 Code Functionality and Implementation

This section of this document outlines an overview of the design of our systems and explains in detail our code logic for the game.

2.1 System Overview

This section will discuss the System structure of our game. We will discuss all the overall components of our game as well as the structure of our UI here.

Our System consists of three main components. We have the characters and all of the components that come with them. The User interface with all of the menu widgets and levels for the stages. With the main components of the characters, we have the blueprints like the character blueprint that most of our code was implemented in and then the material and textures. The UI consists of all of the game controller and HUD classes as well as the game controller for all of them. The HUD class allows the levels to load the widgets right when the level loads and the game modes adjust certain properties of the level. We have just one game controller and this is responsible for making the mouse cursor visible throughout all of the menus.

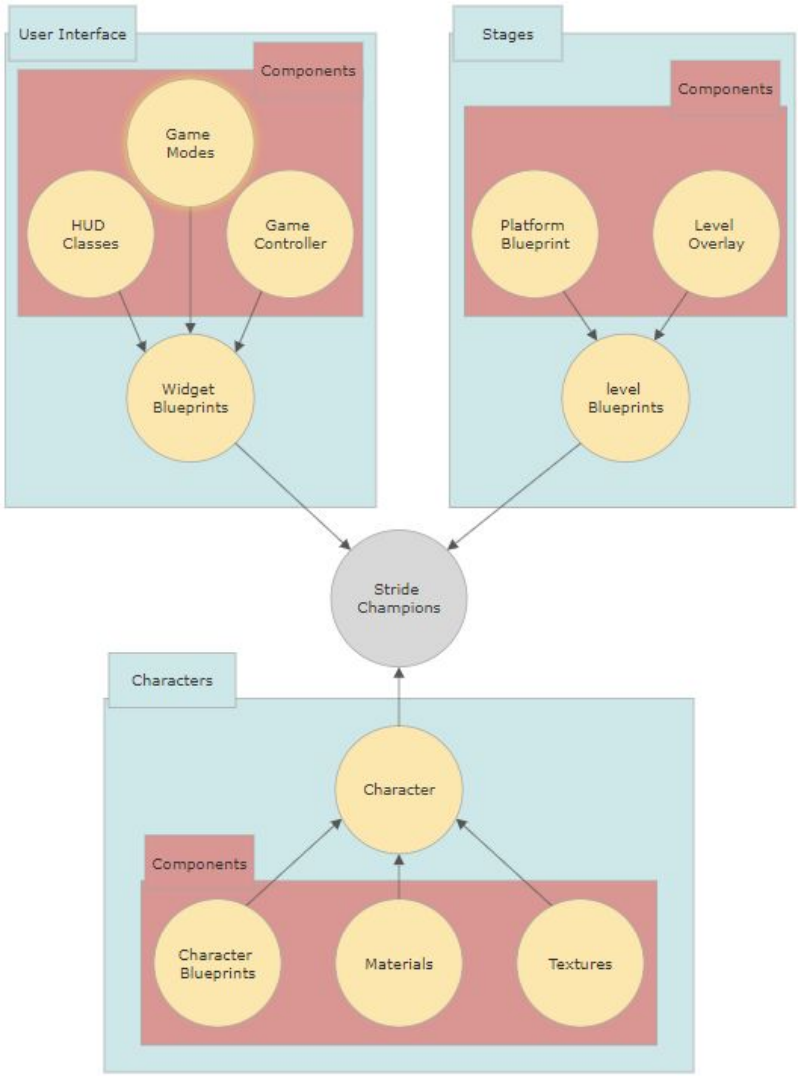

Figure 2.1-1

With our user interface diagram below we have the complete structure of our UI for our game. The colors for the circles represent the following:

- Blue - widgets
- Yellow - levels
- Red - HUD classes
- Orange - game modes
- Green - game controller

The way the menus transition in the diagram is the same way they do in game. The levels for the UI act as a reference point for the options in the pause screen and game over screen to transition to instead of having the stage level in the background. The HUD and game mode circles point there corresponding levels.

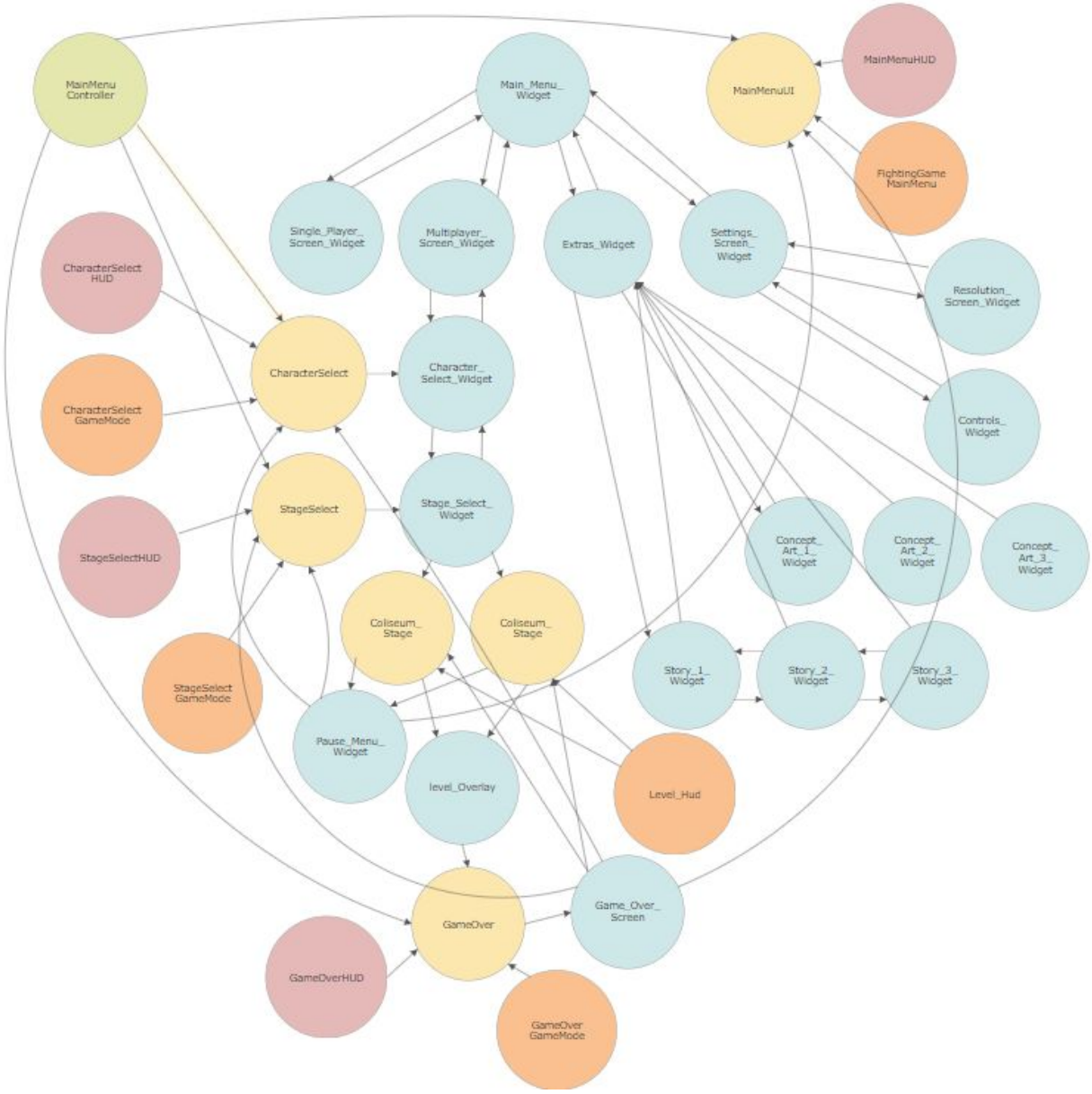
MainMenu Controller
Main_Menu_ Widget
MainMenuUI
MainMenuHUD
FightingGame MainMenu
Single_Player_ Screen_Widget
Multiplayer_ Screen_Widget
Extras_Widget
Settings_ Screen_ Widget
Resolution_ Screen_Widget
CharacterSelect HUD
CharacterSelect
Character_ Select_Widget
CharacterSelect GameMode
Controls_ Widget
StageSelect
Stage_Select_ Widget
StageSelectHUD
Concept_ Art_1_ Widget
Concept_ Art_2_ Widget
Concept_ Art_3_ Widget
Coliseum_ Stage
Coliseum_ Stage
Story_1_ Widget
Story_2_ Widget
Story_3_ Widget
StageSelect GameMode
Pause_Menu_ Widget
level_Overlay
Level_Hud
GameOver
Game_Over_ Screen
GameOverHUD
GameOver GameMode

Figure 2.1-2

2.2 File Structure

When opening the project within unreal the content browser will be located at the bottom of the screen. The image below shows the content folder with all of the folders contained within the project. These folders contain all of the files that are used within the project. In this section we will address all of the folders here and how they are used for this project.

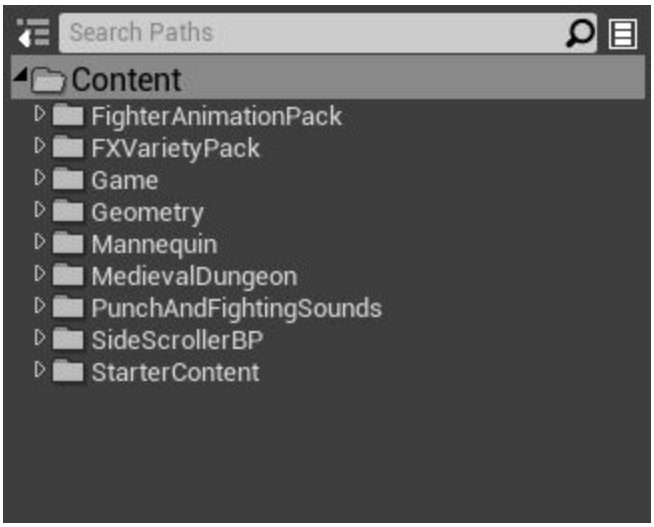

Figure 2.2-1

The folder containing all of our main content is the game folder. The image below shows the content of the game folder.

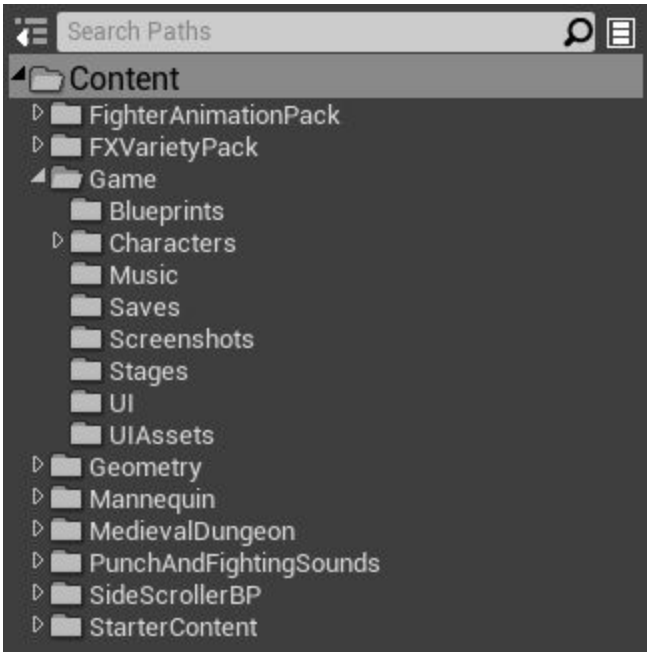

Figure 2.2-2

The blueprints folder contains the game controller, game state stage camera and game mode. Within the character folder we have all of the character information for both characters. The character blueprints, skeleton, skeletal mesh, physics asset, animation blueprint and all of the textures and materials can be found here. The character functionality in sections 2.4.1-2.4.7 is located within each characters respective blueprint class. The music folder contains two original pieces of music that we created for both of the stages. You can listen to wav files here or within the levels when either of the two levels is loaded from the stage select screen. The save folder contains all of the save files that we used to hold the values of variables that needed to be saved for code across different levels and blueprint classes. These will be explained in further detail in the character sections that used them. The screenshots folder contains images that we used for the UI widgets. The information that is used for the concept art, story as well as the images of the characters and stages are located here. The logo used for the main menu screen and the image for the stock icons on the level overlay are also located here. The stage folder contains both the levels for the stages. The functionality for the sections 2.52 and 2.53 are located within the level blueprints for both levels. The UI folder contains all of the widget blueprints for UI. All of the levels, game mode and HUD classes for the UI are located here as well.

We used some asset packs from the unreal marketplace that were used across our project. We used attack animations for our characters from the Fighter animation pack. We then used particle effects from the FX variety pack and sound effects from the punch and fighting sounds pack with the animations. Some of the assets from the medieval dungeon pack were used for the coliseum stage like the torches and statues. The rest of the folders contain the default starter content from when the project was the first started.

2.3 Operating Requirements

The performance of running the coliseum stage on lower level hardware could be challenging. There are instructions provided on the front page of our github within the readme providing the best ways to run our game as well as some bugs that players should be aware of.

2.4 Characters

The functionality for both characters in the following sections is the same asides from the character referencing. The images of the code being referenced in the following sections is taken from the female character blueprint. We named the character blueprints simply female and male characters respectively to make them easier to reference when coding the logic. The code for both players are present in the character blueprints because both cases are covered where either player 1 or player 2 could be the female or male character. There are some differences between player 1 and player 2's code. This is due to an issue we were having where player 2 would run player 1's code. We couldn't understand why this was happening so we had to create additional conditions in the logic to cover this. This will be discussed further in the following sections for where it applies.

2.4.1 Movement

This section will discuss the character movement for both players. The image below shows the game code for player 1's movement. There are some differences between both player 1 and player 2's movement code that will be discussed.

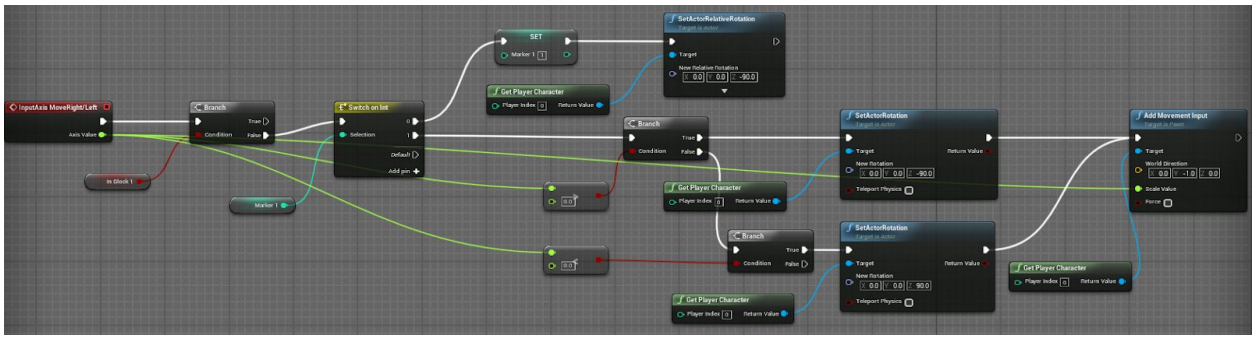

Figure 2.4.1-1

The code first checks to see if the character is in the a block animation. This is to prevent player 1 from moving while they are in the block animation and keeps the player stationary while blocking an attack. If the player is not blocking an attack then then they are free to move. The switch on int has to branches it passes through and uses the marker variable to determine which branch to follow. It first sets the actor of player 1 to be facing inward with an orientation of -90 degrees. This only occurs once when the stage is loaded up and the match begins. Once that branch occurs the marker variable is set to 1 and the remaining movement code is executed. The set actor rotation functions turn the character either one direction or the other. This prevents the character from looking towards the camera when transitioning between directions of left or right. The character in this case will snap left moving in the left direction and then immediately snap right when turning to move in the other direction.

The difference in player 2's code can be seen in the image below where the beginning part of the code checks to see if player 2 is moving on the ground and sets the "Is In Jump" boolean value to either true or false. This check is required for the double jump mechanic that will be explained further on in section 2.4.6.

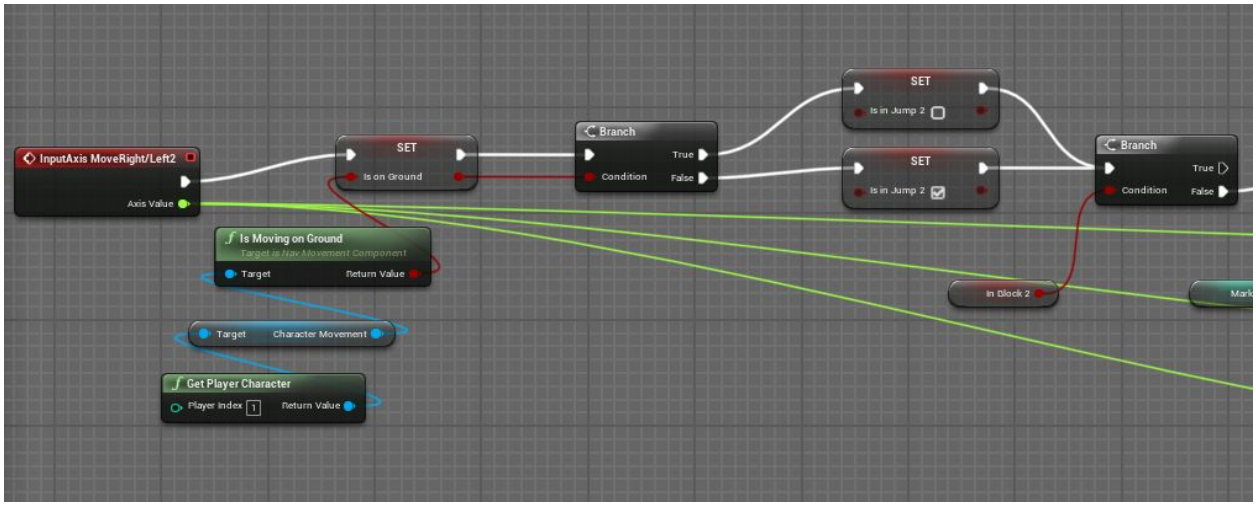

Figure 2.4.1-2

2.4.2 Blocking

For the blocking mechanic, both players have the ability to block the other player's attack. During the block state the player initiating the block doesn't receive damage from the other player's attacks. The image below shows the code for the blocking mechanic for player 1.

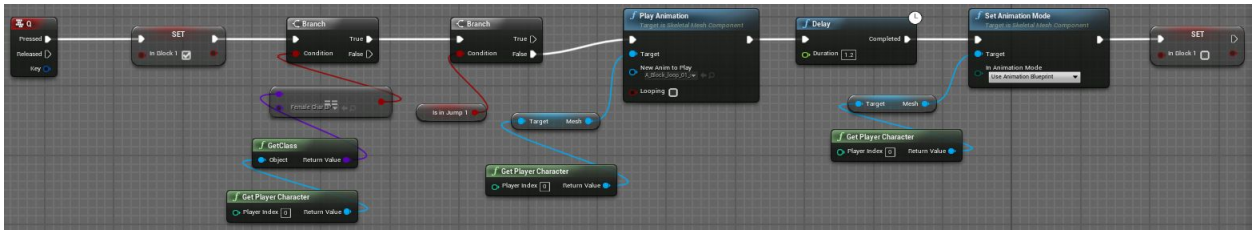

Figure 2.4.2-1

At the beginning of this code the "In Block" boolean variable is set to true. This is useful for checking whether player 1 is blocking for other mechanics like the movement discussed previously. This variable will prove useful in other mechanics in the following sections as well. The branch statement that follows just verifies that player controller 0, in this case player 1 is running the code. The next branch statement checks if player 1 is in a jumping state. This prevents the player from blocking while jumping and is only able to block an attack while being grounded. The block animation plays and once the block is finished the "In Block" variable returns to false signaling to other logic that the blocking state has finished.

The logic for player 2 is similar but an additional branch of logic is required to cover the case of player 2 running player 1's code. The difference is outlined in the image below.

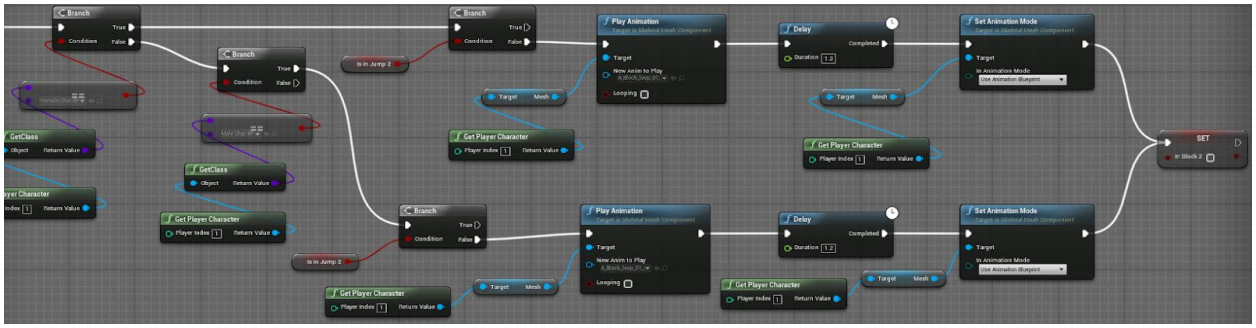

Figure 2.4.2-2

In this case the branch statements check to see which gender of character is using this code. This code will be run by both player 1 and player 2. So if player 1 is the female character and player 2 is the male character then then player 2 will use the second branch of code here and use player 1's "Is In Jump" variable.

2.4.3 Standing Kick

The standing kick attack has a different attack animation for each character. The code in the following image is for player 1.

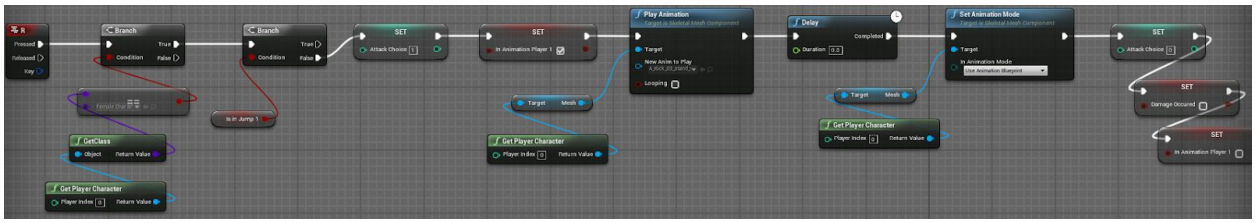

Figure 2.4.3-1

This code first checks to allow player 1 to run the code. The next branch statement checks to see if the player is grounded or not. Player 1 can only use this attack if they are grounded. The variables "Attack Choice" and "In Animation Player 1" are utilized in the attacking collision logic in section 2.4.8. With the "Attack Choice" variable, each of the three attacks are assigned an integer value, in this case standing kick is 1 and this identifies which attack is being executed. The "In Animation Player 1" variable identifies whether player 1 is currently in an attack animation. Once the attack animation is finished these variables are reset along with the "Damage Occured" variable. This variable is used again in section 2.4.8 with the collision logic and checks to see if damage from the attack has occurred against player 1.

The code for player 2 is in the following image. Due to the length of the code it is split into multiple images.

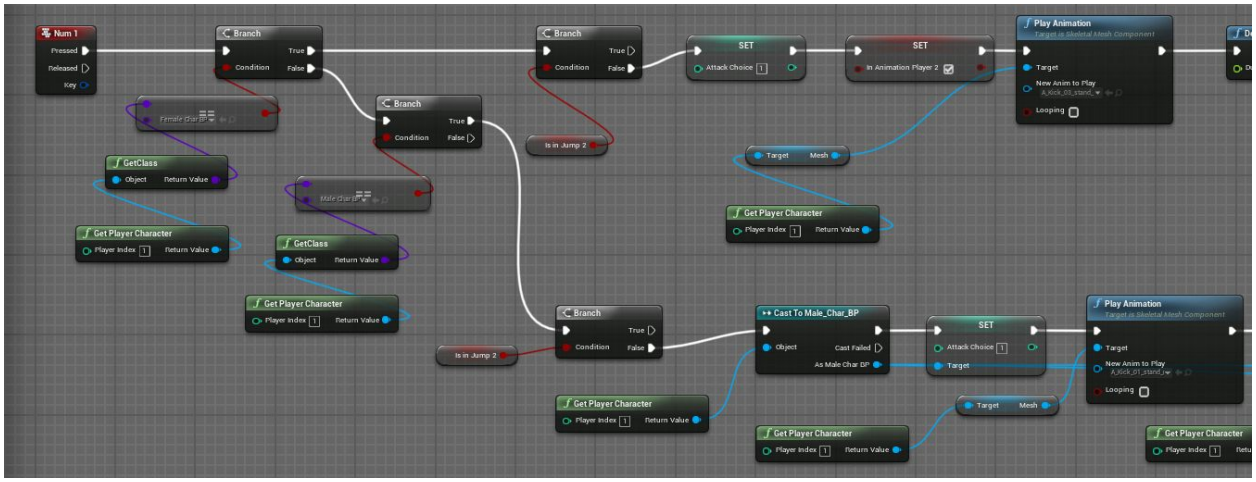

Figure 2.4.3-2

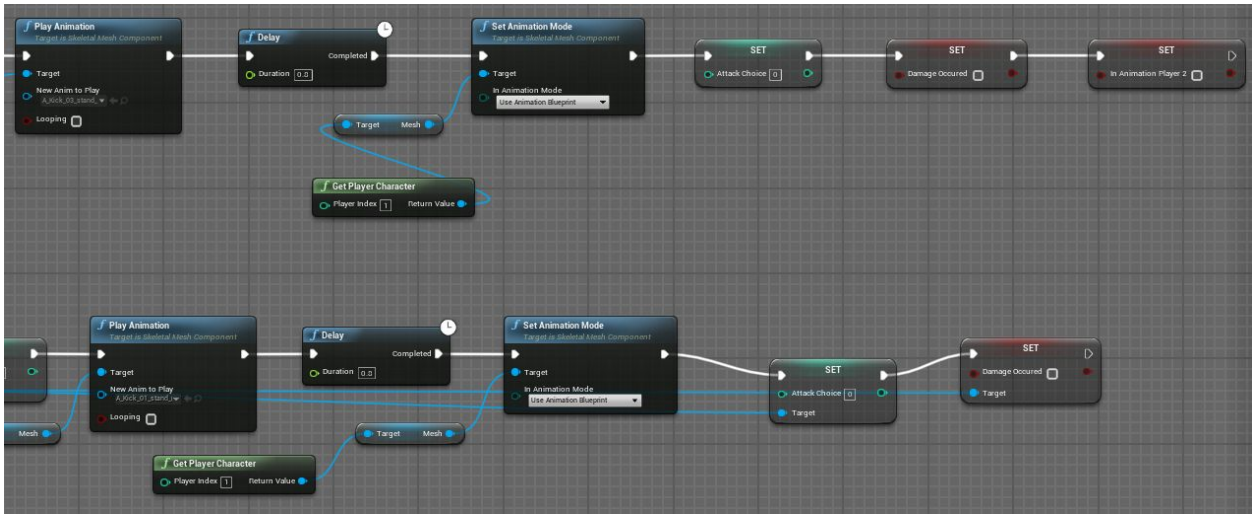

Figure 2.4.3-3

The code here once again accommodates both players by first identifying which gender of character player 2 is controlling. The appropriate path is then taken based on the gender. The rest of the logic occurs the same way as player 1's code. The exception here however that once again player 2 is using the two variables described before within player 1's code. The third variable "Is Animation Player 2" only sets and resets in the case where both players are the female character. The opposite would be the case in the male character blueprint where this variable would only be present in the case where both characters are male. This occurs as well in the other attacks the following sections 2.4.4 and 2.4.5.

2.4.4 Standing Punch

The logic for the standing punch for both players is the same logic as the standing kick. The difference is that the attack animations for both characters are different and the value for the

“Attack Choice” variable is 2 representing a different attack. The images below show the code for both players standing punch attack. Once again the code for player 2 is split into two images.

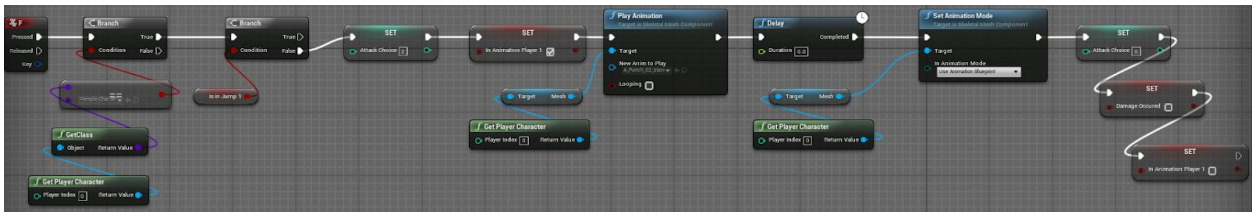

Figure 2.4.4-1

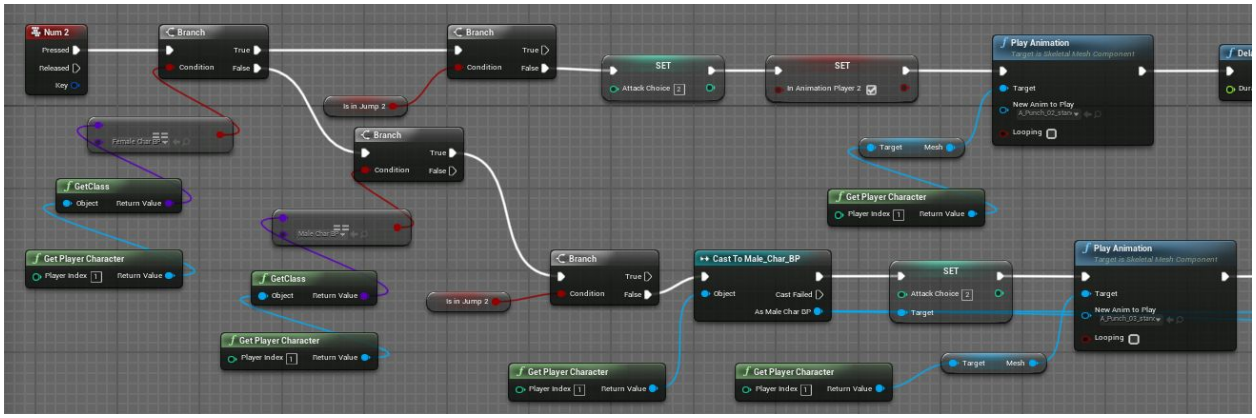

Figure 2.4.4-2

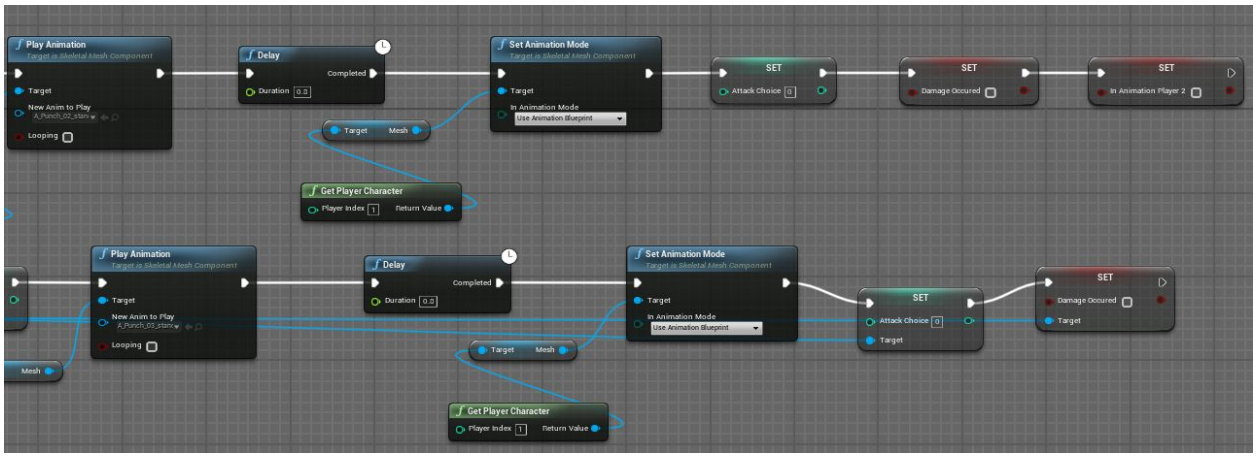

Figure 2.4.4-3

2.4.5 Aerial Kick

Once again the logic remains similar to that of the last two attacks mentioned. The difference here with the characters aerial kicks is that the characters can only start these attacks in the air. So the branching condition when checking the “Is In Jump 1” variable is true instead of false. The “Attack Choice” variable in this case is set to 3 representing the third possible attack choice.

The following images display the code for both player 1 and player 2 respectively with player 2's code split into two images.

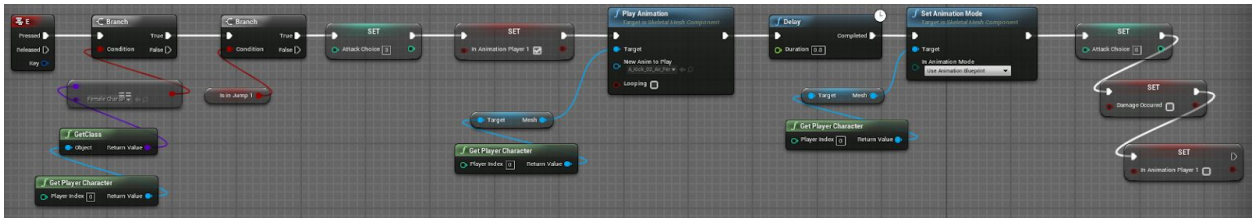

Figure 2.4.5-1

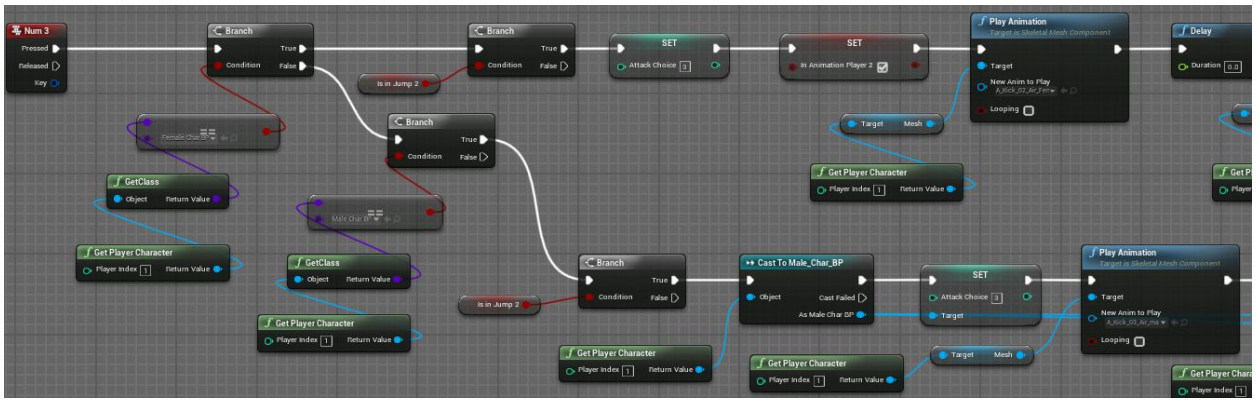

Figure 2.4.5-2

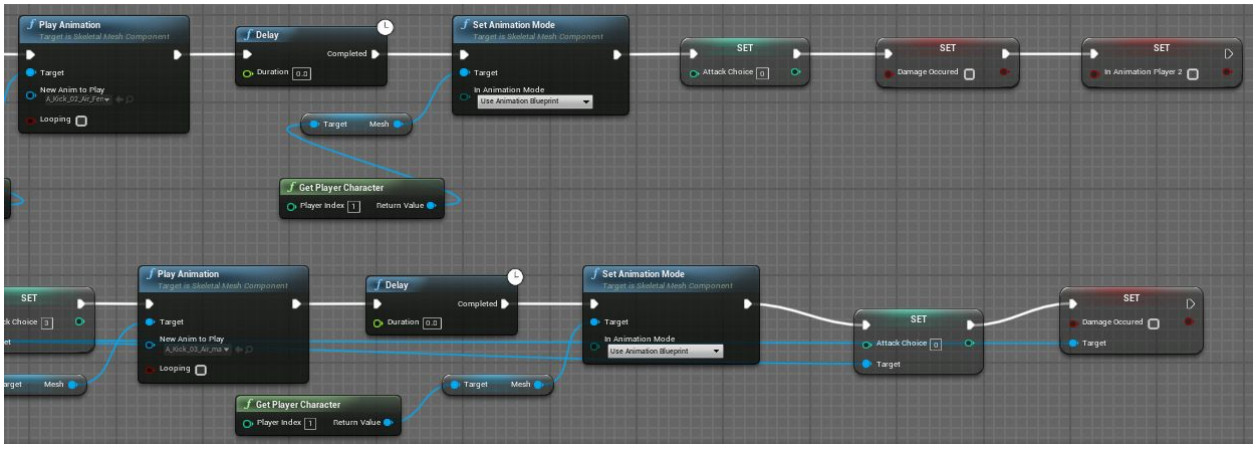

Figure 2.4.5-3

2.4.6 Double Jump

This mechanic allows the players to jump again after they have already jumped. This was done by setting a jump integer variable to determine whether the player has already jumped. If the jump counter is at 1 then the player is still able to jump again. The following code shows the double jump functionality for player 1.

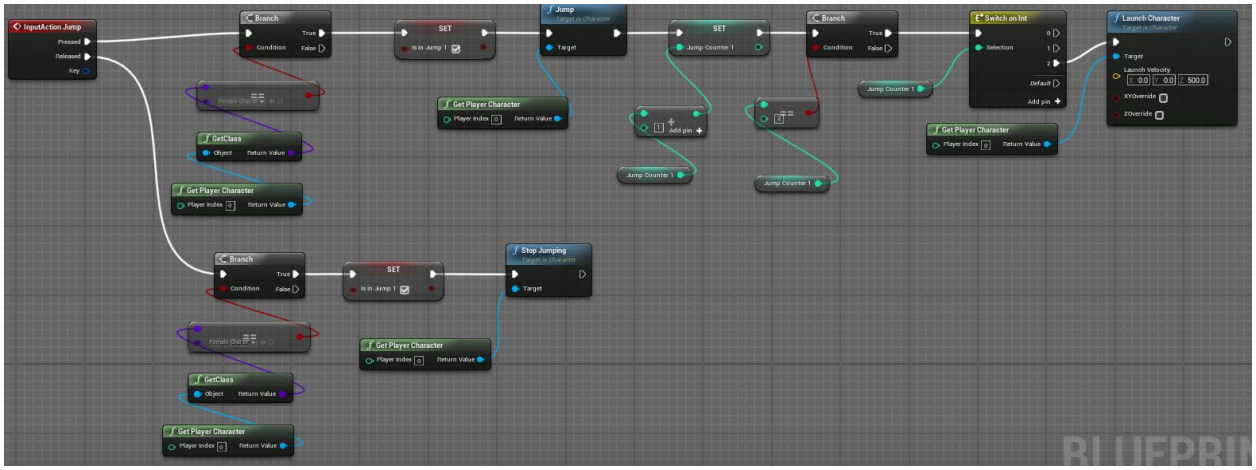

Figure 2.4.6-1

The launch character function is used to launch player 1 once they have jumped once. This acts as a double jump for the player. When player 1 is jumping the “Is In Jump 1” variable is set that was referenced earlier with the three character attacks. While player 1 is in the air and this variable is set then both standing kick and standing punch attacks will remain unusable. However now player 1 will be able to use their aerial attack. The value for “Is In Jump 1” needs to be set to false when the player returns to the ground. This is done in the following code.

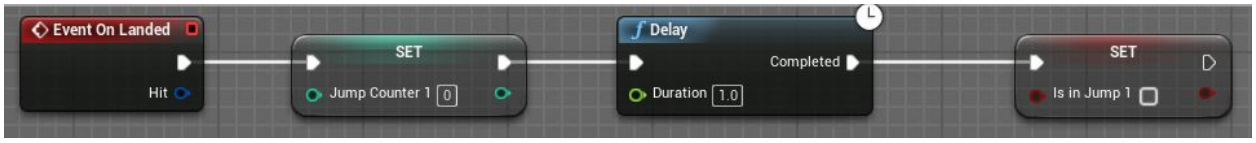

Figure 2.4.6-2

The code resets both the values for “jump counter 1” and “Is In Jump 1” to 0 and false. This code only reset the values of the player 1 version of these variables. This is because the action events within the blueprint only work the object the blueprint is for, in this case the female character. So if player 1 is playing as the female character and player 2 is playing as the male character, the value of “Jump Counter 2” and “Is In Jump 2” would not reset when tied to the female characters “Event On Landed” action event. Even though player 2 playing the male character would run the code and use the variable from player 1 using the female character, we still had to find some way of resetting the “Jump Counter 2” and “Is In Jump 2” without tying them to this action event. If both characters are female, then there's no problem because both characters are female characters using their own action event from their own blueprint class.

That is why back in section 2.4.1 and figure 2.4.1-2 we explained that the value for “Is In Jump 2” is reset when player 2 lands. The next images will display how the value for “Jump Counter 2” is reset within the code for player 2’s double jump functionality.

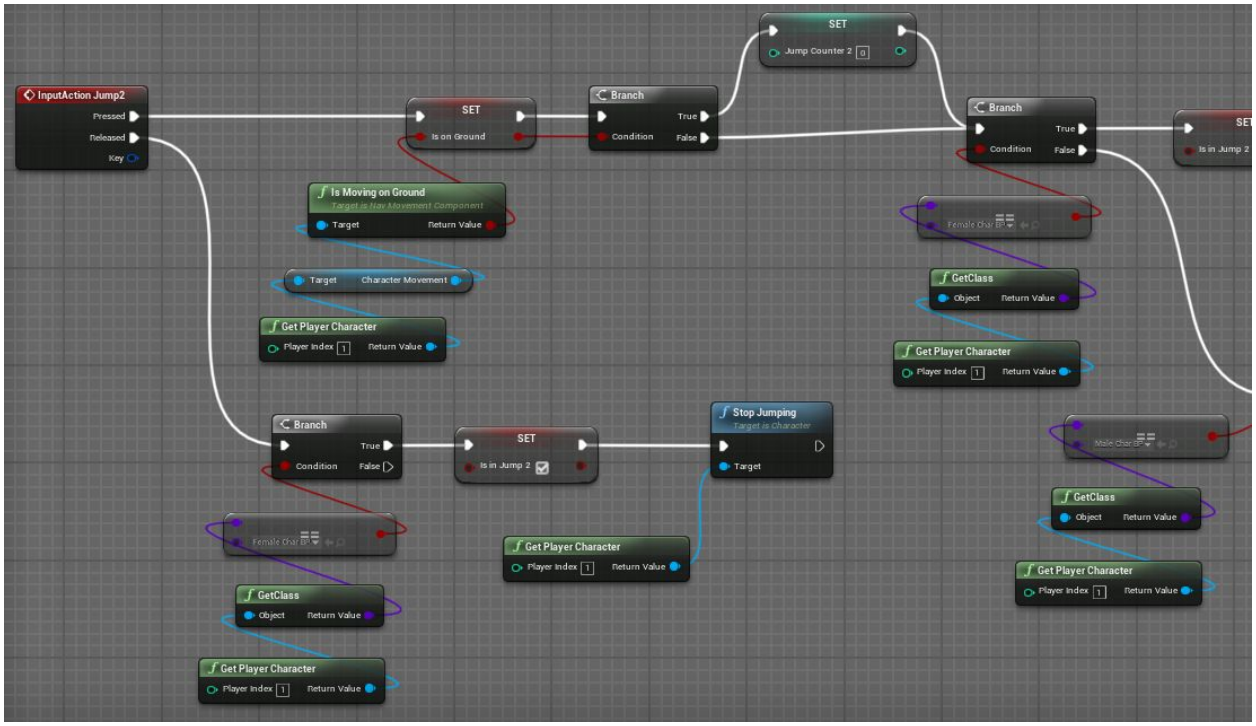

Figure 2.4.6-3

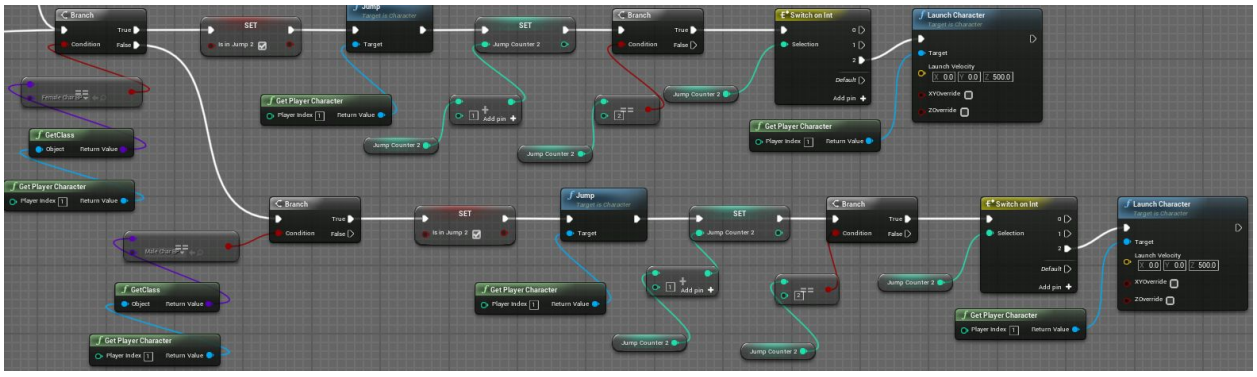

Figure 2.4.6-4

In figure 2.4.6-3 there is a branch condition after the player check reset the value for “Jump Counter 2” once player 2 returns to the ground again. The rest of the code for player 2 follows the same logic stream as the player 1 code.

2.4.7 Pausing

The pausing functionality works the same way with both players where they both share the “P” key on the keyboard to pause the game. The following code displays the logic for pausing the game.

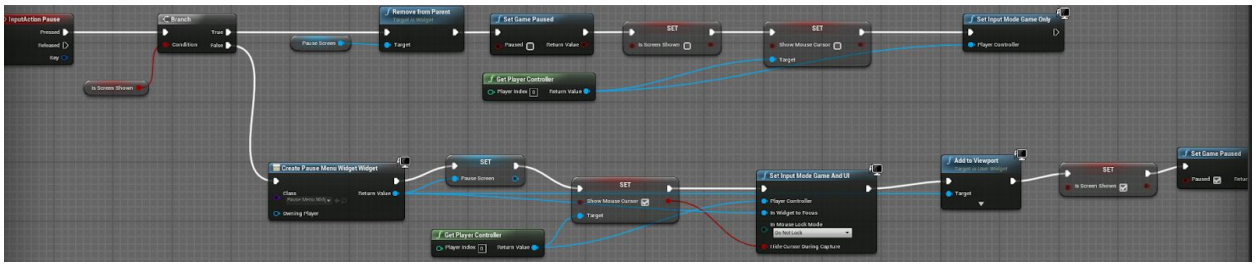

Figure 2.4.7-1

The branch at the start checks to see if the game is paused. If not then then the logic follows the false branch where a pause menu widget is created and a boolean allowing the mouse cursor to be visible while the menu is open is checked true. The widget is then added to the viewport and the “Is Screen Shown” variable is checked to true to identify that the game is paused. If the game is paused then the branch at the beginning travels down the true path where the widget is removed, the variables are reset and the mouse cursor is removed from the screen.

2.4.8 Attacking Collisions

This section will break the logic down for the collisions and explain both players receive damage and take knockback based on the hits from the opposing player. The logic for this section of code was very unpredictable and it was very difficult to understand what was happening with the output it presented in several cases. Even the final result of what we have has a couple bugs that we will also address. For the collisions we used sphere collisions and attached them to the bones of both characters. We attached four sphere collisions to both characters, two for both hand sockets and the other two for both foot sockets. The image below displays the sphere collisions being attached to the character sockets.

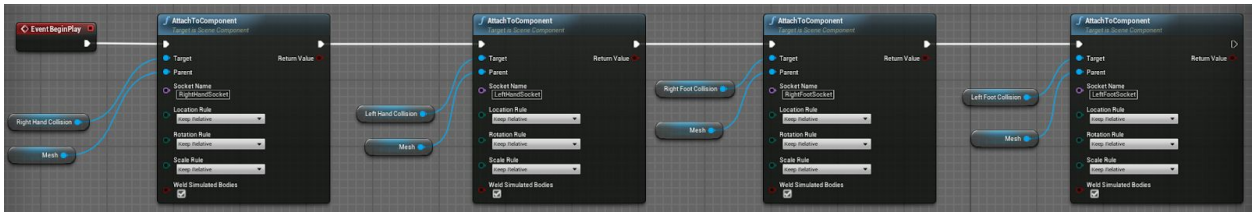

Figure 2.4.8-1

The unpredictable behaviour was due to how the sphere collisions worked. When the “On Component Begin Overlap” action event is used for each of the four sphere collisions the action events begin firing on the object they are attached to as well as other objects they collide with. In this case we had to filter out only the collisions with the opposing character and only during

an attack animation. The image below shows the “On Component Begin Overlap” events comparing a variable to determine whether the other actor, in this case the other player, is playing either the male or female character.

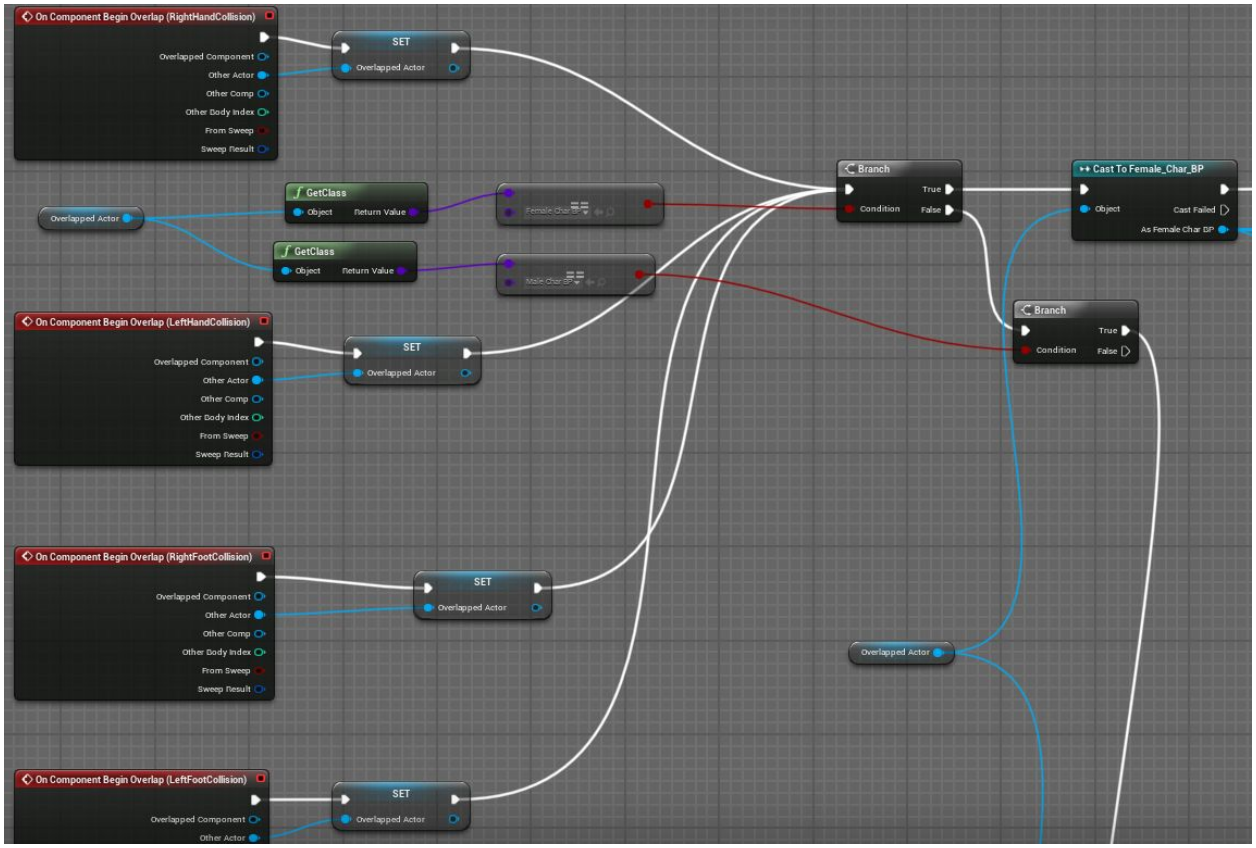

Figure 2.4.8-2

After checking whether the other player is male or female the logic breaks into four different branches, two for the case of the player being female and two for the case of being male. The two cases for both female and male characters then handle the collisions based on whether that player is either player 1 or player 2. The first two branches can be seen in the image below. These two branches will only occur in the instance where both players are playing the female character since this is the female blueprint. The variables that determine which player is attacking are the “In Animation Player 1” and “In Animation Player 2” variables mentioned back in sections 2.4.3-2.4.5 with the character attacks. If the collision overlap is with the female character the values for “In Animation Player 1” and “In Animation Player 2” are false and true then that means that player 2 playing the female character is attacking player 1 playing the female character. The is reversed for the second branch where player 1 is attacking player 2.

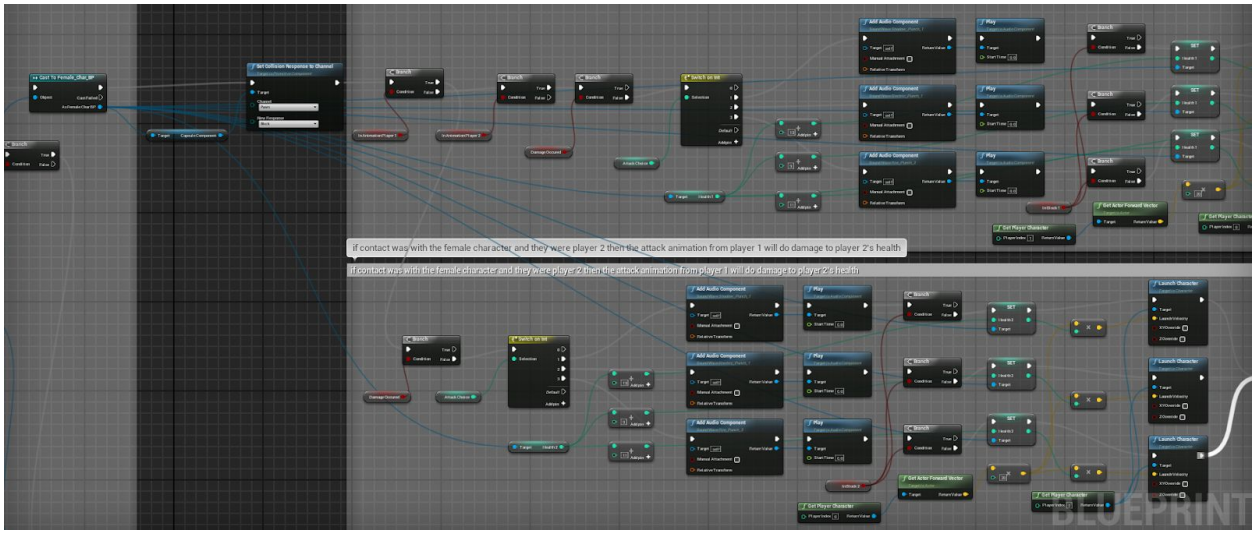

Figure 2.4.8-3

The two cases with the collision of the male character occur when there is either a combination of both the male and female characters between each player. The check to determine which player is playing the male character is easier for these cases. We were simply able to set branch conditions checking if the male character is player 1 or player 2. This logic for some reason did not work with the two female cases from before. That's why we needed the two variables checking the animation status. The “Damaged Occured” variable that was referred back in sections 2.4.3-2.4.5 just ensures the damage the opposing player receives only occurs once during a single attack animation. Below is the code for the other two branches.

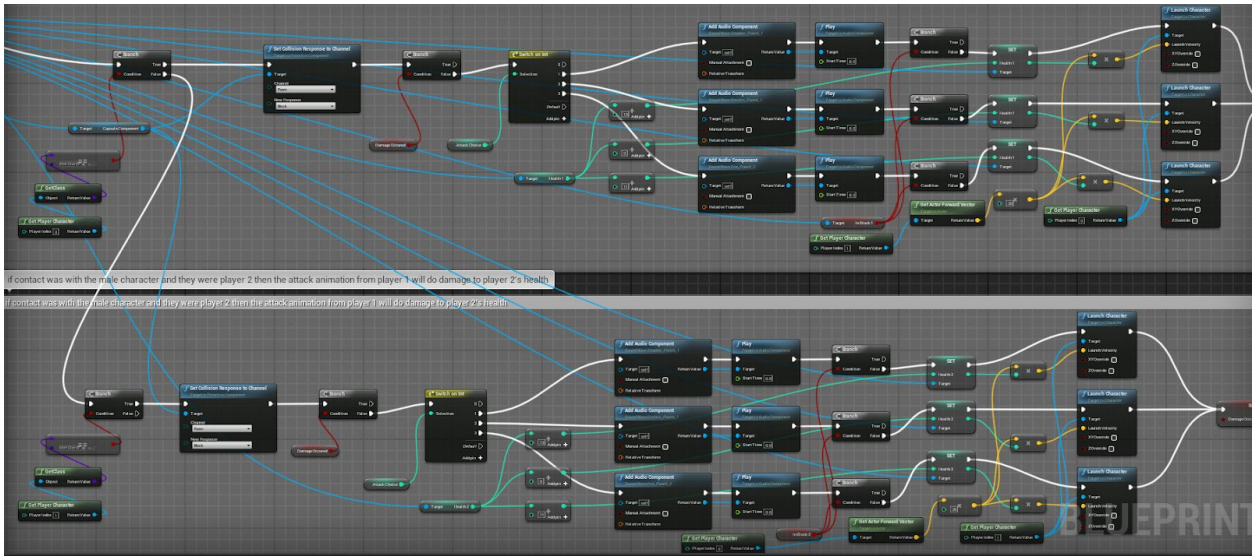

Figure 2.4.8-4

The rest of the code for all four branches sets the sound effects for the attack animations, increases the attacked player's health value and applies knockback based on the current health

value. There is also a branch condition before the damage value is applied to the health to check if the opposing player is in block state. If one of the players is in block state then they will receive no damage from the attack. With the knockback as further damage is added to the player's health the farther they will travel upon being hit.

2.5 Stages

The stages that were designed for the game contain the same code. The code for the following sections can be found within the individual stage blueprints. We will be referencing the code from the Cosmic Rift Stage for the sections below.

2.5.1 Player Spawns

This section covers how the players are spawned into the level. When the stage is first loaded the players character choices are spawned on opposite sides of the level from each other. Player 1's character spawns on the left side of the stage while player 2's character spawns on the right side.

The first part of the code resets the player's stock counts to three when the stage first starts up. Since save variables remember the information stored in them even after the game is closed the stock counts need to be reset upon loading the stage. Both "Player 1 Stock Counter" and "Player 2 Stock Counter" have a default value of three so this resets the stock counts to their original values. The image below shows the stock values being reset in the save file with both player's stock counts being set to three.

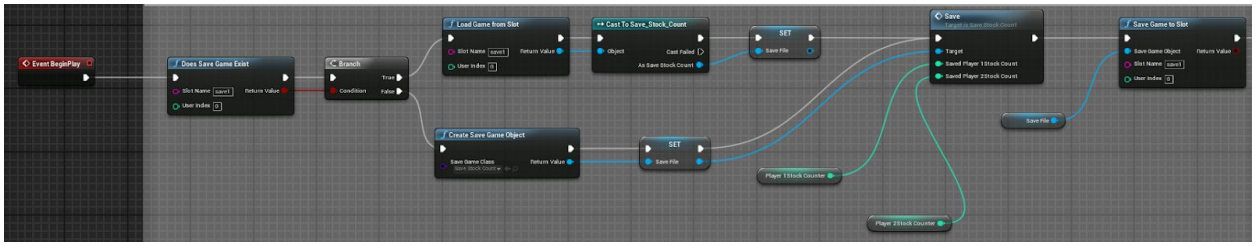

Figure 2.5.1-1

The player's character choices are saved into a save file in the character select screen widget and then loaded into the stage blueprint code. Those choices then determine which character is spawned for each player. The player choice variable either contains the value 0 or 1. The code below shows the players character choices being loaded from the save file.

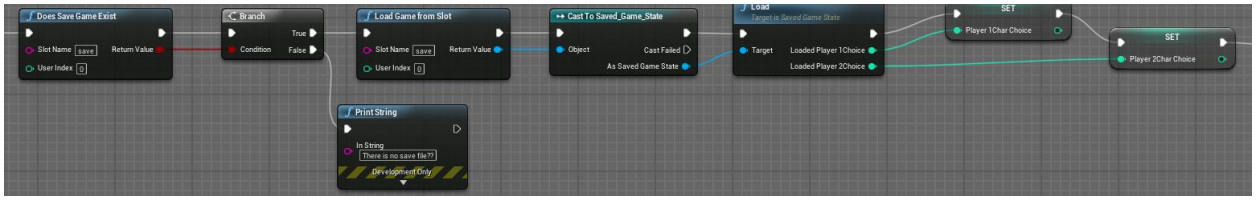

Figure 2.5.1-2

Once the player's character choices are known then we have to spawn player 1's character on the left side of the stage and player 2's on the right. In order to do this we set two different vectors for each player. One of the vectors contains the spawns location for that player and the other vector contains the respawn location for that player. Using this information we could then use the coordinates contained within the vector to position the player's character where they should be on the stage at each player's "NetworkplayerStart" point. For the respawn locations we start the players a small distance off of the ground with a z value of 300. The image below shows the code for this.

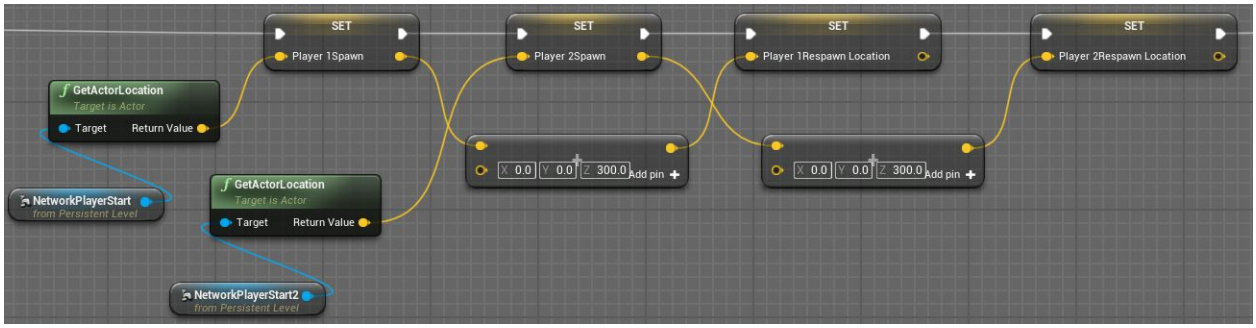

Figure 2.5.1-3

After vectors are loaded with the spawning information the pawns are spawned on to the stage and possessed by the player controller. Player controller 0 is player 1 and player controller 1 is player 2. If the value of the player choice variable is 0 then the female character will be spawned for that player and if the value is 1 then the male character will be spawned. The images below display this functionality for this.

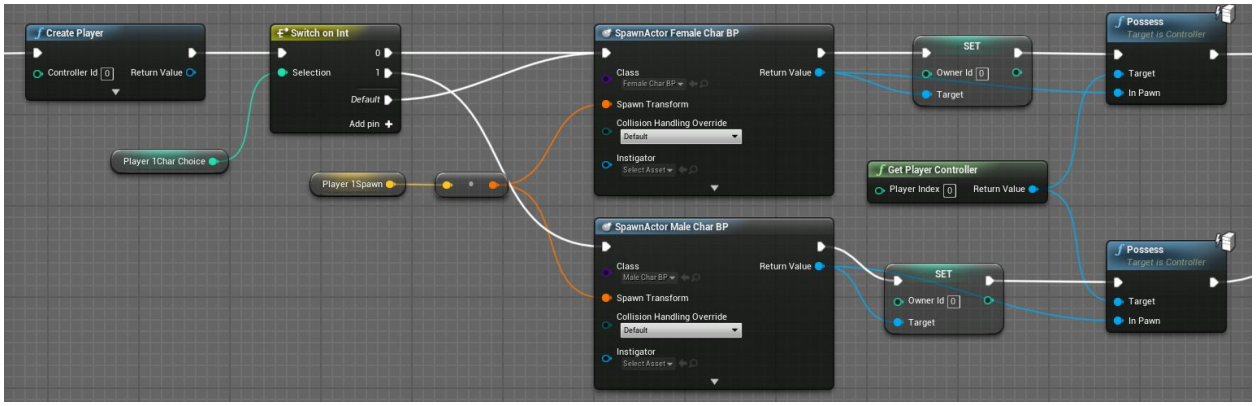

Figure 2.5.1-4

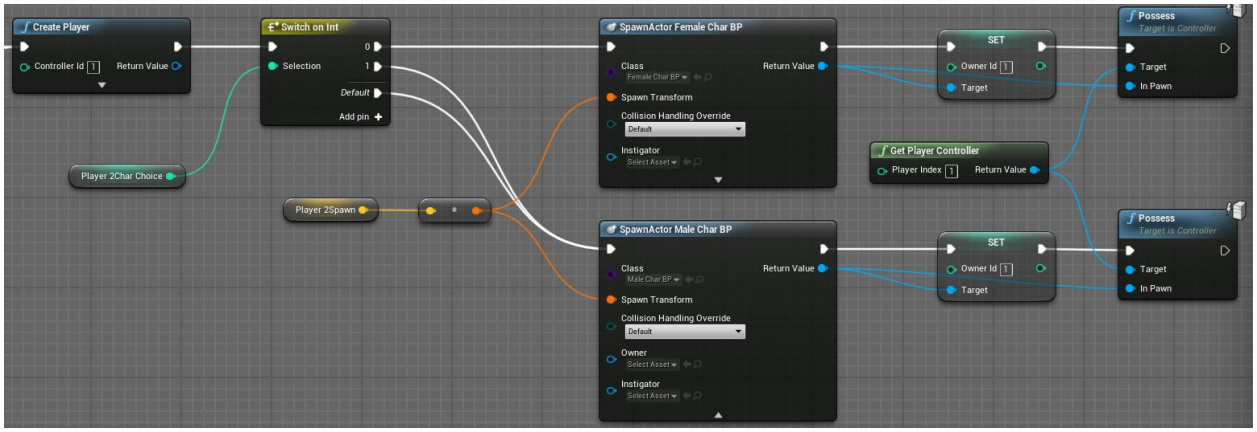

Figure 2.5.1-5

The “OwnerId” variable is set to identify which player is playing the female character and which player is playing the male character. This variable will be used in the next section with respawning the players.

2.5.2 Respawning and Blast Zones

Both of the stages constructed for the game contain four blast zones. These are also known as stage bounds for the players. If a player connects with these blast zones their stock count reduces by one. To implement these in our stages we used box collisions for each of the four blast zones. We then used “OnActorBeginOverlap” action events to activate the code when a player connects with the box collisions. The image below displays the start of the code for this logic.

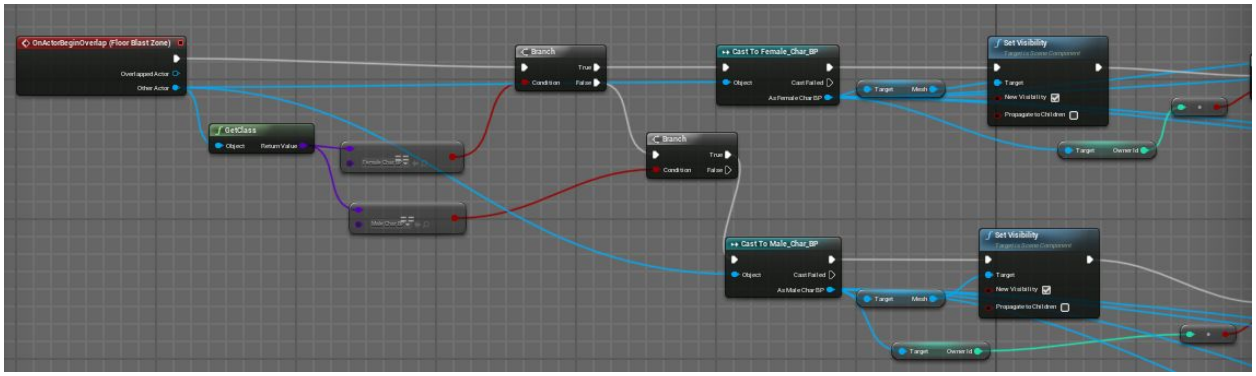

Figure 2.5.2-1

The “OnActorBeginOverlap” action event first checks to see if the collision was with the male or female character. This splits the code into two branches and each branch now checks to see, based on the “OwnerId” variable which players are playing which characters. This way the players can be respawned to the correct location and side of the stage that they started on based on the location information from the player respawn vectors.

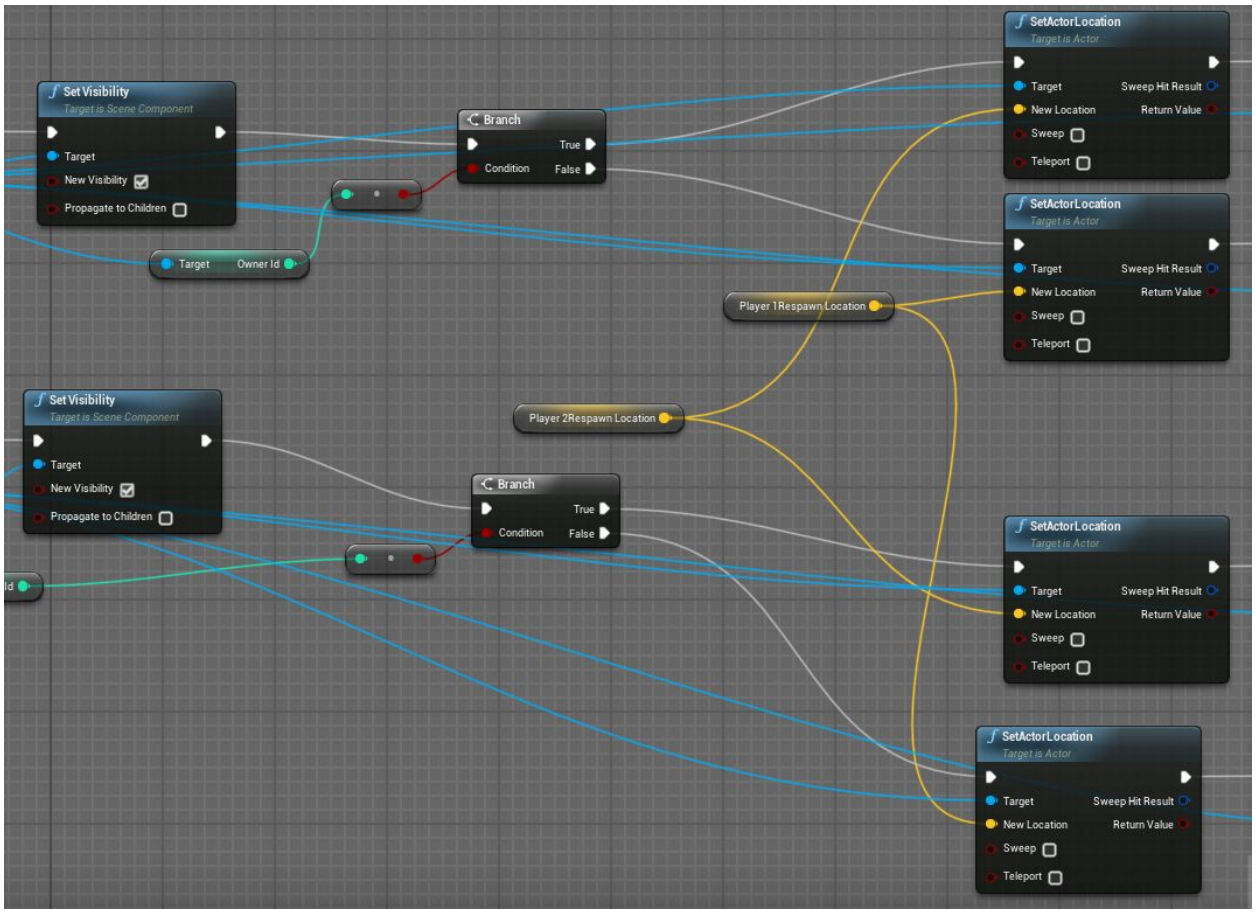

Figure 2.5.2-2

The player that respawned has their health reset back to zero. Their stock count reduces by one as well until they have no stocks remaining.

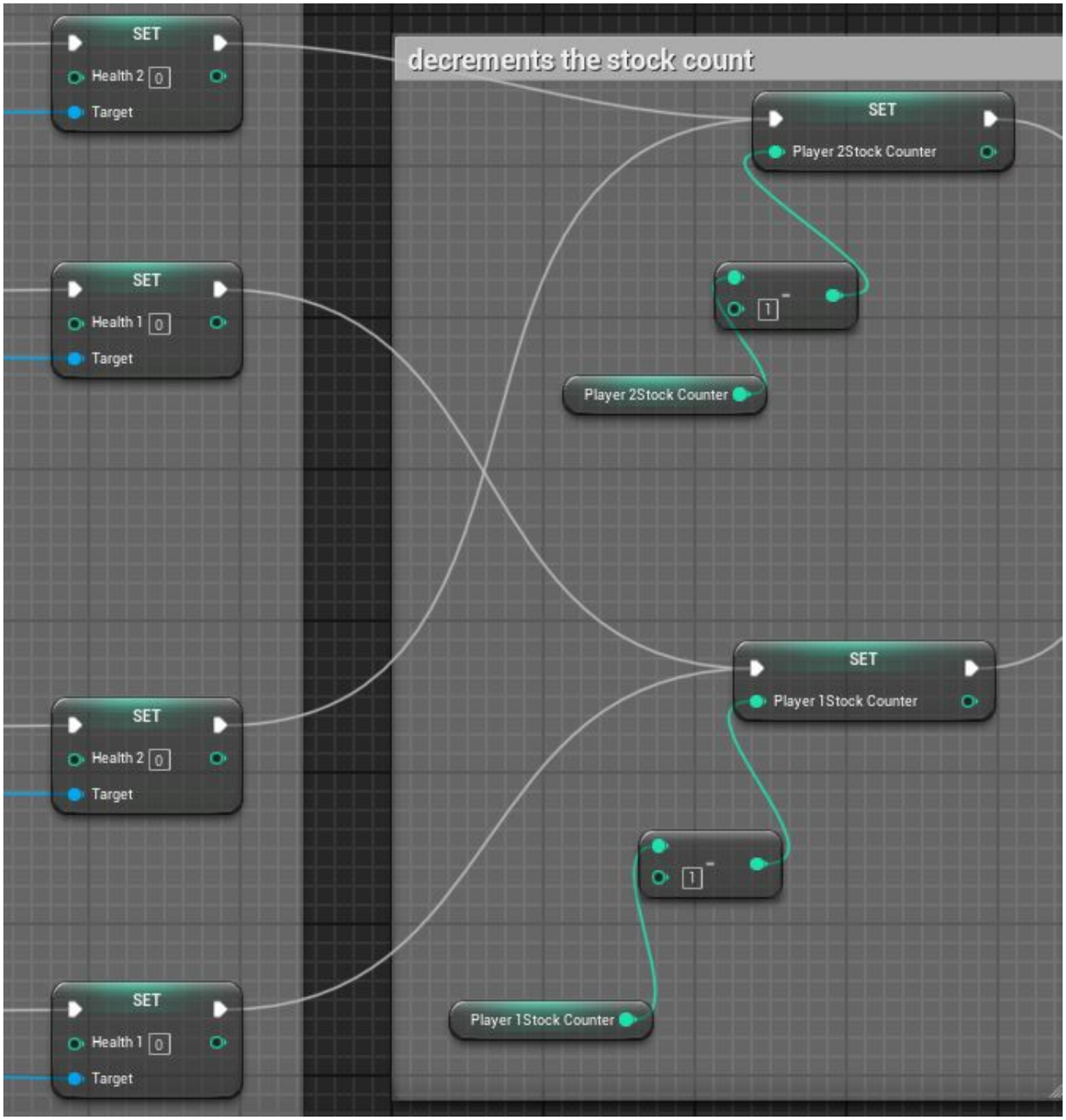

Figure 2.5.2-3

Once the player's stock count is decremented by one, then the new variables are updated in the save file. This is the same logic that was used back in section 2.5.1 with the player spawning logic. However this time we are using this code to simply update the current stock count for each player. This code can be seen again down below.

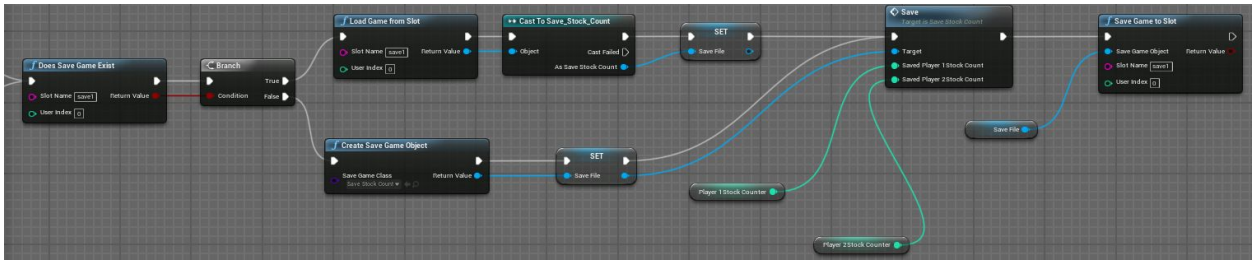

Figure 2.5.2-4

2.5.3 Stage Overlay

The overlay is a widget that can be seen when the stage loads that contains an image of both player's stock count and each player's health percentage. This widget is loaded through the HUD class called "Level_HUD" when the stage is first loaded. The stocks are represented as small circles above the players percentage. Player 1's stocks are red and player 2's are blue. player 1's stocks and percentage are located on the left side of the screen where player 1 originally spawns and player 2's are on the right. This section will discuss the logic behind how the information is changed on the screen.

We built six functions to handle the logic for removing the player stocks. Each function was responsible for removing one of the player's stocks. Since there are six stock icons present on the screen we built six functions to remove them. These functions all contain the same logic with the only difference being the comparison operator at the end.

The code first loads the player stock counter values for each of the players from the save file. This is the same save file that is discussed in the section above that updates the variables when the players respawn. The while loop at the end of the code blocks of the logic flow until the comparison with the player stock count variable becomes true. These functions are continuously firing while the stage is loaded so the logic had to be blocked until the appropriate condition was met. The comparison checks the player stock count variable of the player that the stock is being removed from and checks to see if it is one less than the stock that it is removing. For example if player 1's first stock is being removed then the player stock count value for player 1 would have to be two or less for the logic to proceed past the while loop. Once the condition is met and the logic processes past the while loop then that stock icon's visibility is set to hidden. The images below display the condition of player 1's first stock icon being removed.

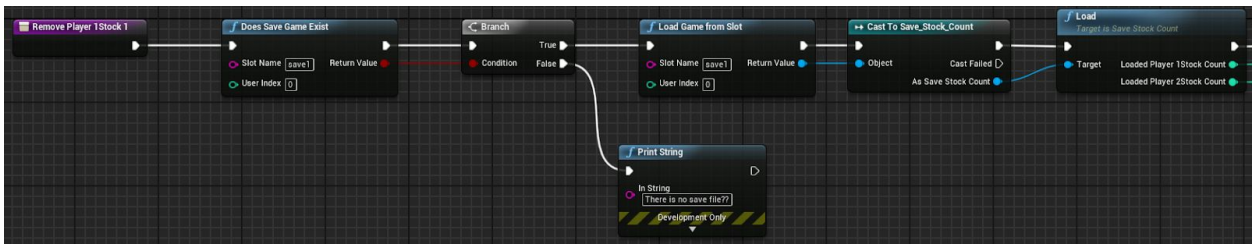

Figure 2.5.3-1

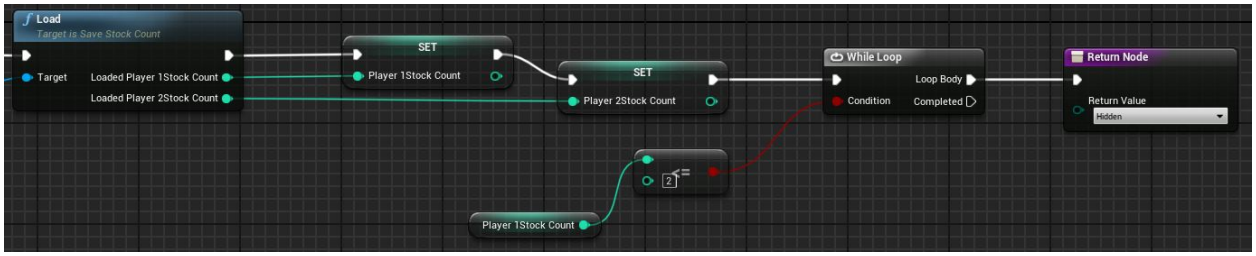

Figure 2.5.3-2

The last two functions control the percentage or health displayed on the screen for both players. These functions again start off by loading from a save file. This time we load the variables for the player choices for the characters. For the function modifying the health value of player 1, the variable for player 1's character choice is compared in a branch condition to zero. If true that means that player 1 chose the female character, if false then player 1 chose the male character. The health value is then updated based on which character was chosen by player 1 from that characters respective blueprint class. The images below show this. The code displayed in this image is from the function changing the percentage of player 1. The function for player 2 is the same code, the only difference is that the branch comparison compares the value for player 2's stock count instead.

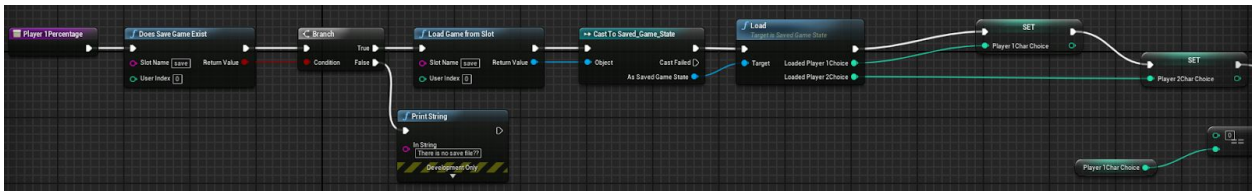

Figure 2.5.3-3

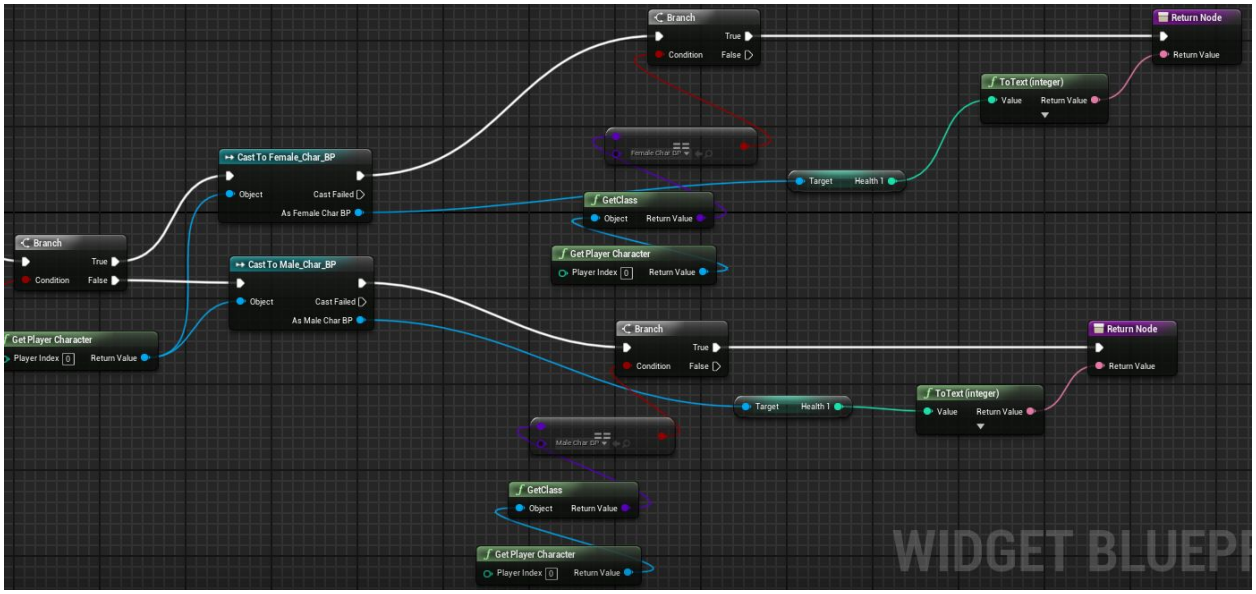

Figure 2.5.3-4

2.5.4 Platforms

The logic for platforms for the Cosmic Rift stage originally worked before sphere collisions were implemented for the attack animations. The Characters would smoothly transition through the platforms and land on top of them. Although now the logic doesn't quite work the same now with the sphere collisions in place. We designed the logic so that when the level starts, by default both players would be able to jump through the platform and branch conditions would compare the height of the character with the box collision above the platform. This would determine whether the character would be pushed through the other side of the platform or not. Once the player would land on the ground again they would once again be able to jump through again. The images below display this.

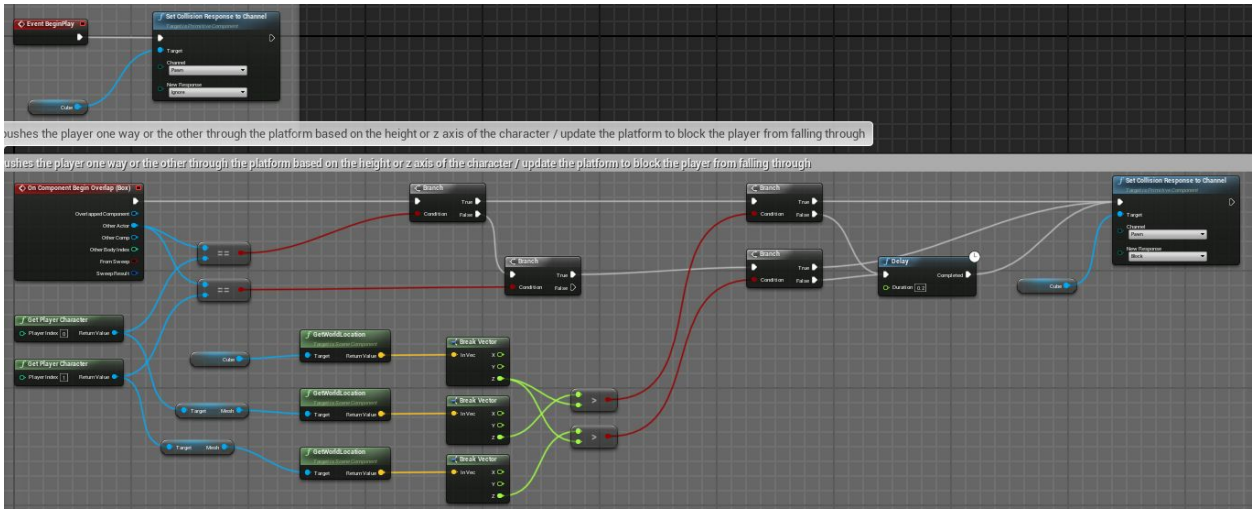

Figure 2.5.4-1

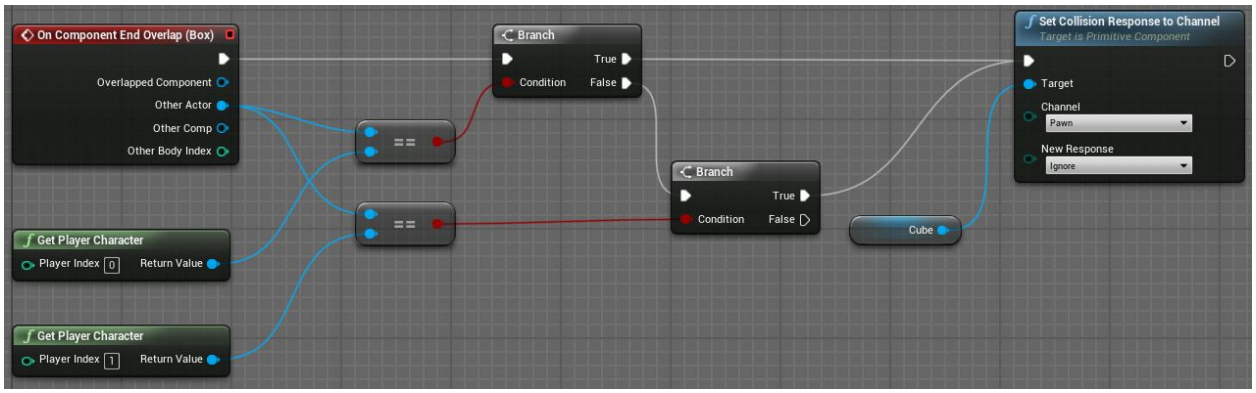

Figure 2.5.4-2

2.6 User Interface

This section discusses all the code for the widgets used to build the menu screens. Most of the functionality for many of the widgets is fairly straightforward with the code for most of them containing similar logic. The code for all of the widgets, HUD classes, and game mode relating to the user interface can be found in the UI folder in our project.

2.6.1 Menu Levels

There are four menu levels that we have in the project. Three of these act as reference points to return to with the options available from the pause screen when in a stage. The other level is to display the game over screen when the game is over and one player has lost all their stocks. Instead of having the stage level in the background, these blank levels will load and the menu widgets will be traversed from them. The functionality of the pause menu will be further discussed later in section 2.6.13.

The four levels consist of:

- MainMenuUI
- CharacterSelect
- StageSelect
- GameOver

Each of these levels uses a HUD class to load their respective widgets. These HUD classes are:

- MainMenuHUD
- CharacterSelectHUD
- StageSelectHUD
- GameOverHUD

The HUD classes are driven by the game modes for each class, these consist of:

- FIghtingGameMainMenu
- CharacterSelectGameMode
- StageSelectGameMode
- GameOverGameMode

2.6.2 Main Menu Screen

The main menu screen will be the first screen that is displayed to the user when the game first starts. From here the user will have the ability to transition between one of five screens. These consist of:

- Single Player
- Multiplayer
- Extras
- Settings
- Exit Game

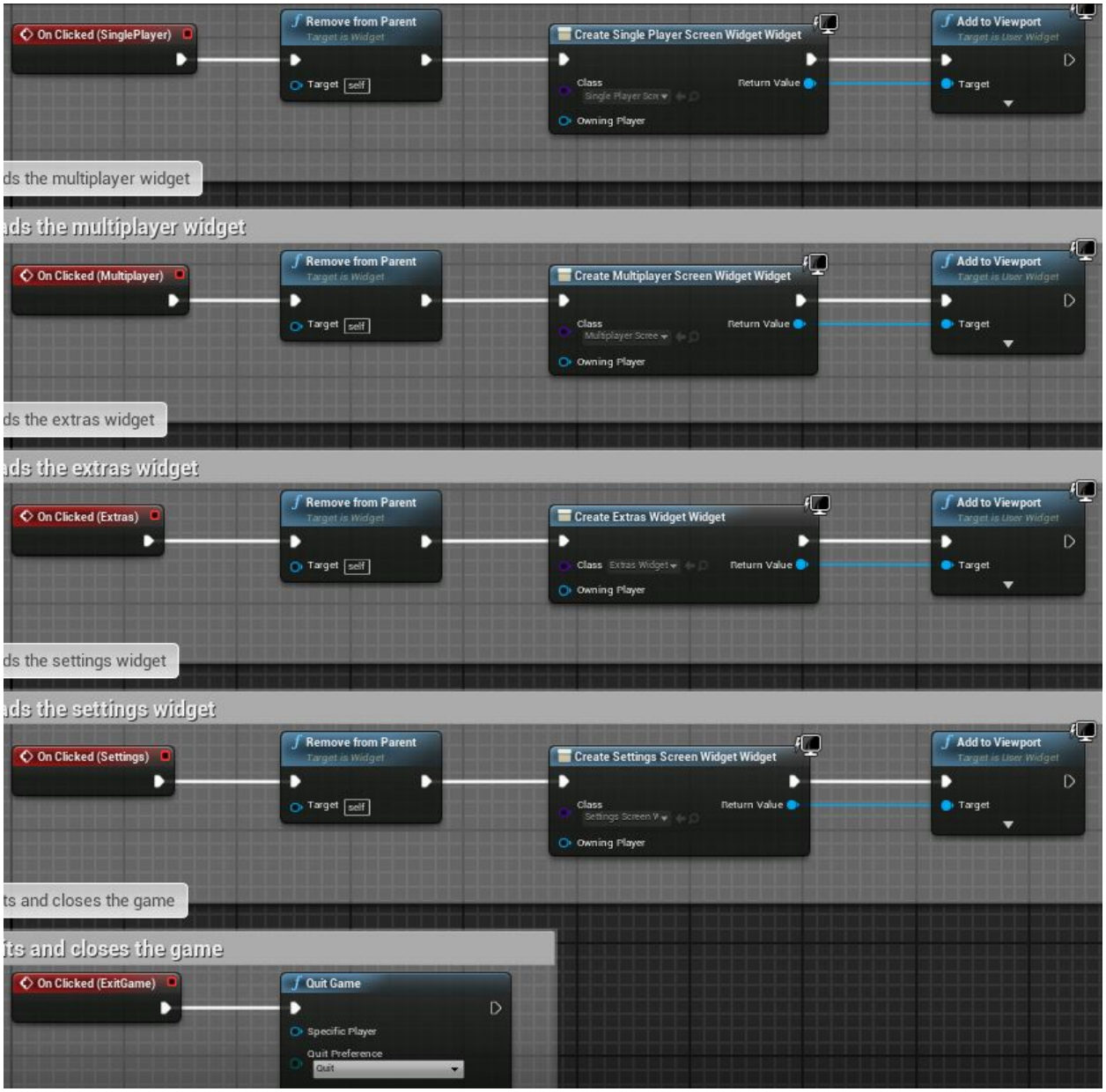

Figure 2.6.2-1

This image shows the code for transitioning between screens. Each “OnClicked” action event first removes the existing widget, creates the new one for the screen that's being opened and then adds it to the viewport. The “OnClicked” action event for quitting simply exits the game in this case.

2.6.3 Single Player Screen

The single player has the buttons, one for story mode and the other for practice mode. However for the scope of this project these features are unavailable. The only operating functionality here is the main menu button which returns the user back to the main menu.

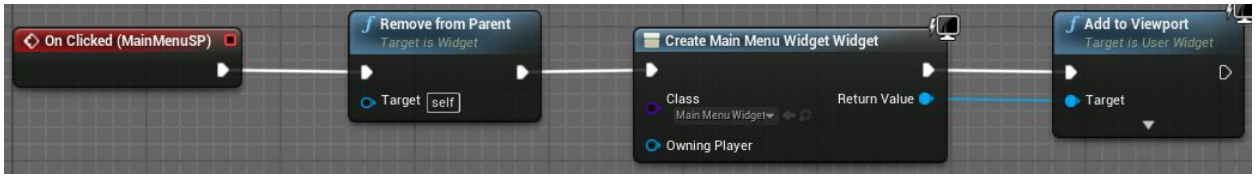

Figure 2.6.3-1

2.6.4 Multiplayer Screen

The multiplayer screen allows the user to access the local multiplayer or return to the main menu. Again for the scope of this project the online component is not operational.

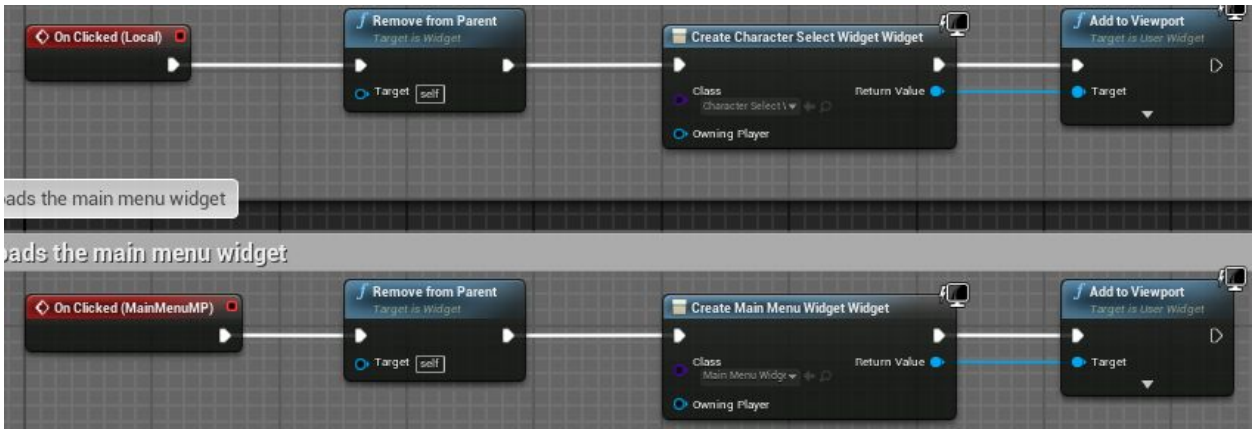

Figure 2.6.4-1

2.6.5 Extras Screen

The extras screen allows the user to access the concept art and story information about our game. The user can also return to the main menu as well.

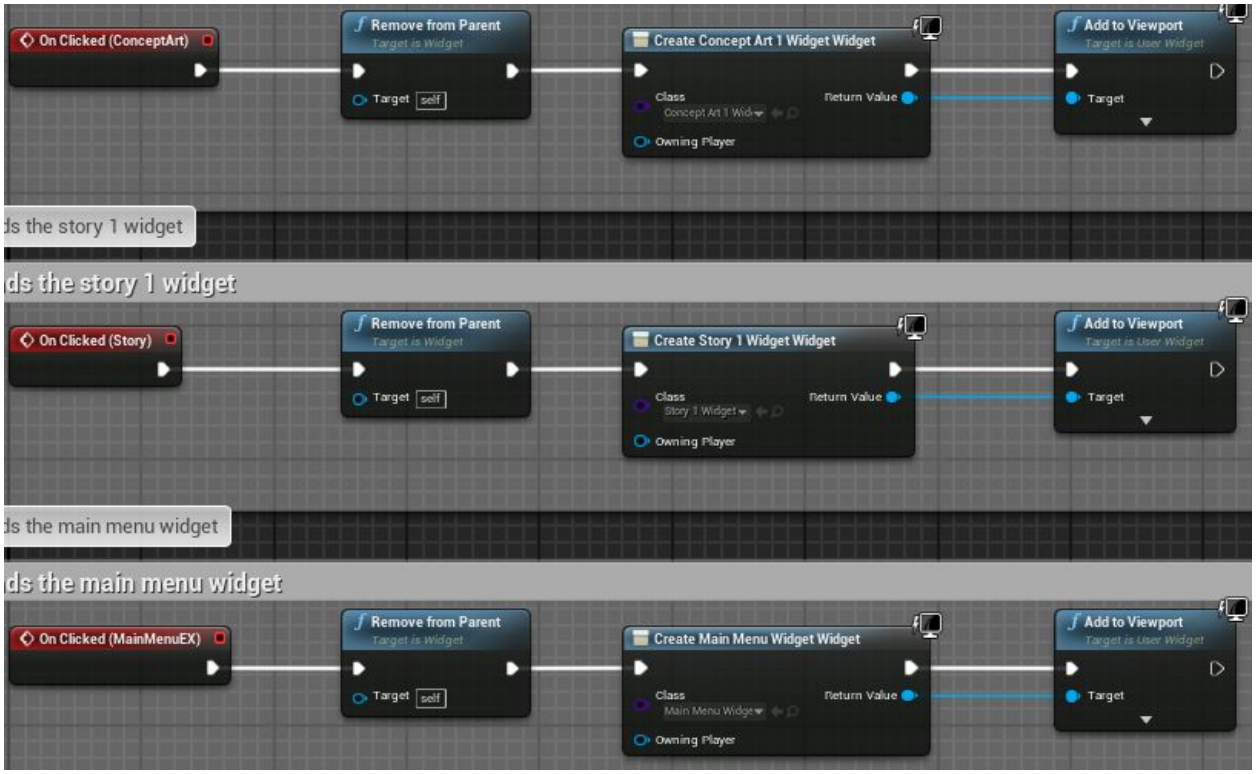
On Clicked (ConceptArt)
Remove from Parent
Target is Widget
Target
self
Create Concept Art 1 Widget Widget
Class
Concept Art 1 Wid
Return Value
Owning Player
Add to Viewport
Target is User Widget
Target
ds the story 1 widget
ds the story 1 widget
On Clicked (Story)
Remove from Parent
Target is Widget
Target
self
Create Story 1 Widget Widget
Class
Story 1 Widget
Return Value
Owning Player
Add to Viewport
Target is User Widget
Target
ds the main menu widget
ds the main menu widget
On Clicked (MainMenuEX)
Remove from Parent
Target is Widget
Target
self
Create Main Menu Widget Widget
Class
Main Menu Widge
Return Value
Owning Player
Add to Viewport
Target is User Widget
Target

Figure 2.6.5-1

2.6.6 Settings Screen

The settings screen allows the user to access game controls information as well as change the resolution settings. The user can also return to the main menu as well.

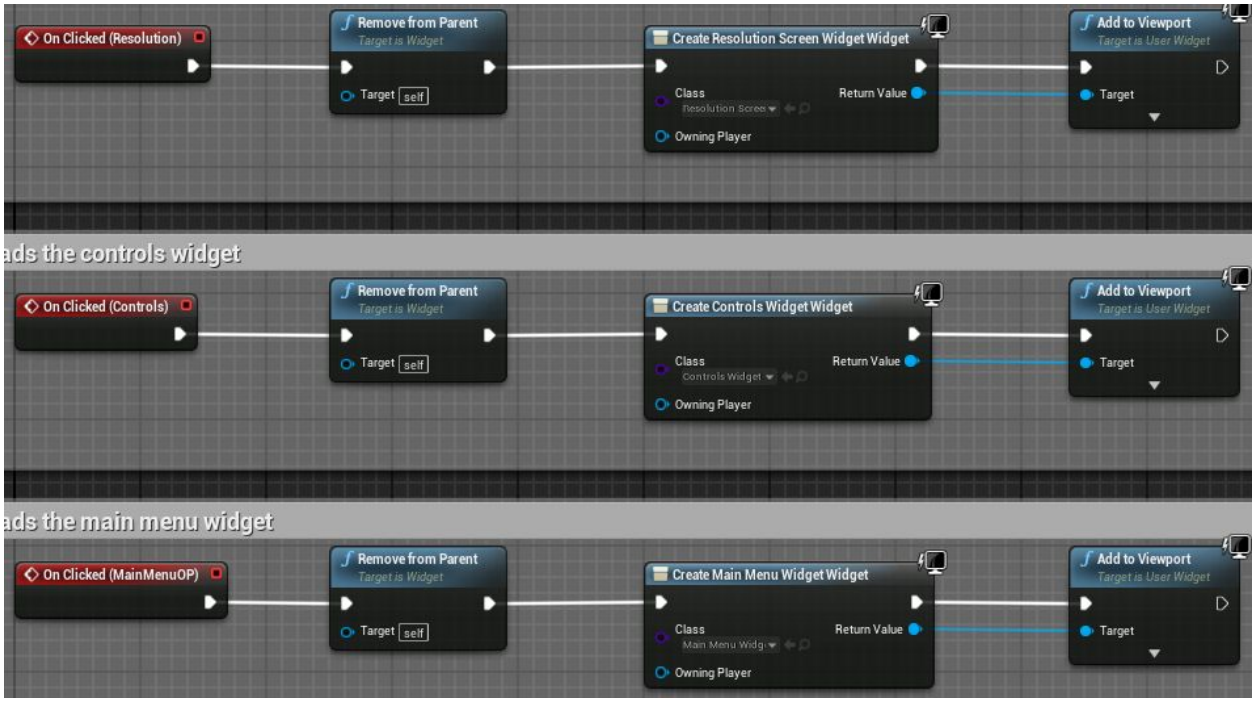

Figure 2.6.6-1

2.6.7 Resolution Screen

The resolution screen allows the user to change the screen resolution with three different options. The user can also return to the main menu as well.

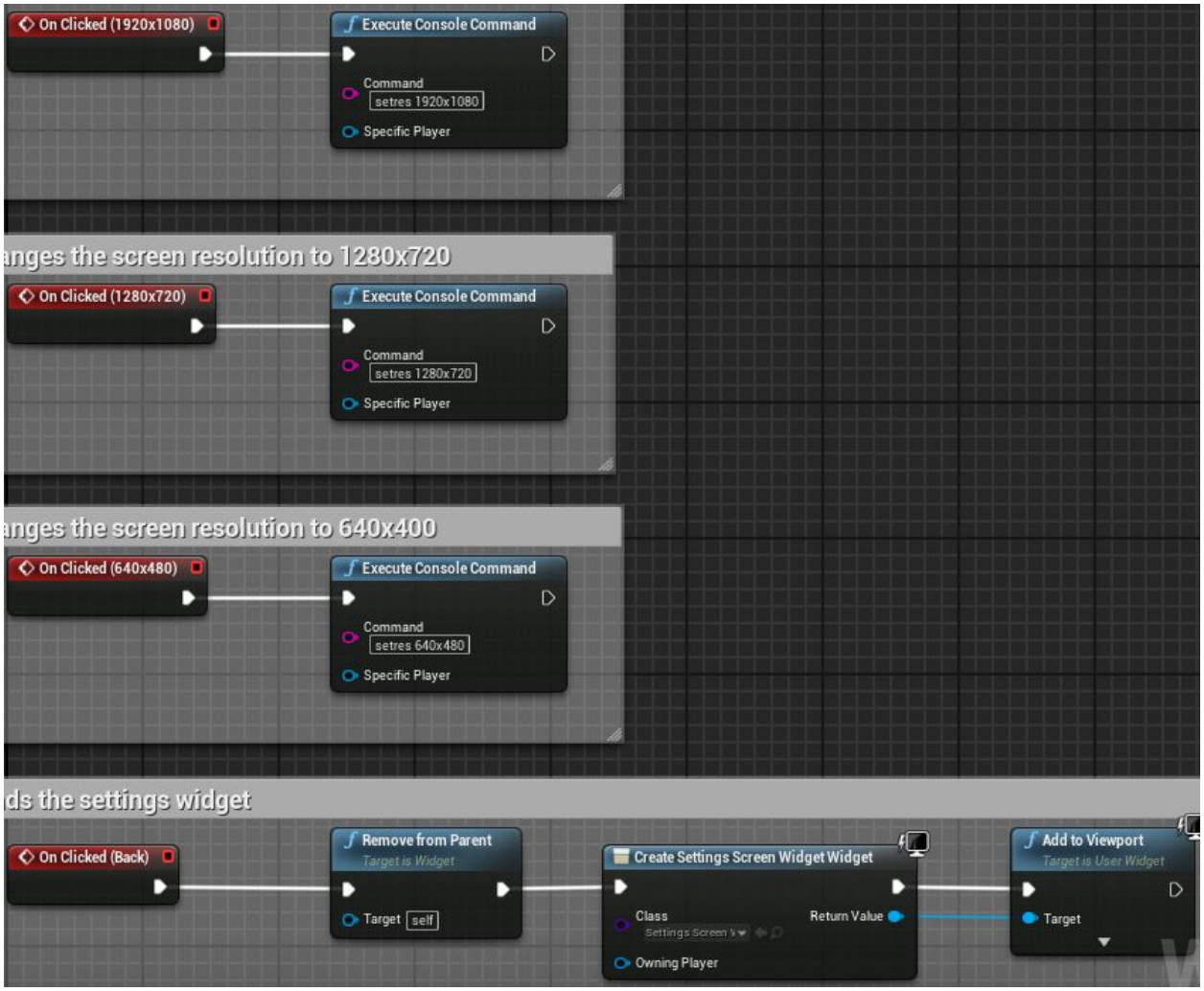

Figure 2.6.7-1

2.6.8 Controls Screen

This screen provides the user with control information for both player 1 and player 2. There is also a back button provided that allows the user to return to the settings menu.

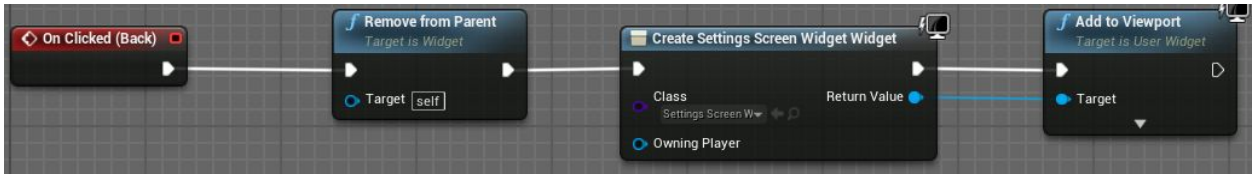

Figure 2.6.8-1

2.6.9 Concept Art Screens 1, 2 and 3

These three concept art screens provide concept for the character design and logos for the game. The back button returns the extras screen in any of the three concept art screens. There are arrows in the lower right and left hand corners that allow the user to traverse between the

three screens. The images below represent the code for concept art 1, concept art 2 and concept art 3 widgets respectively.

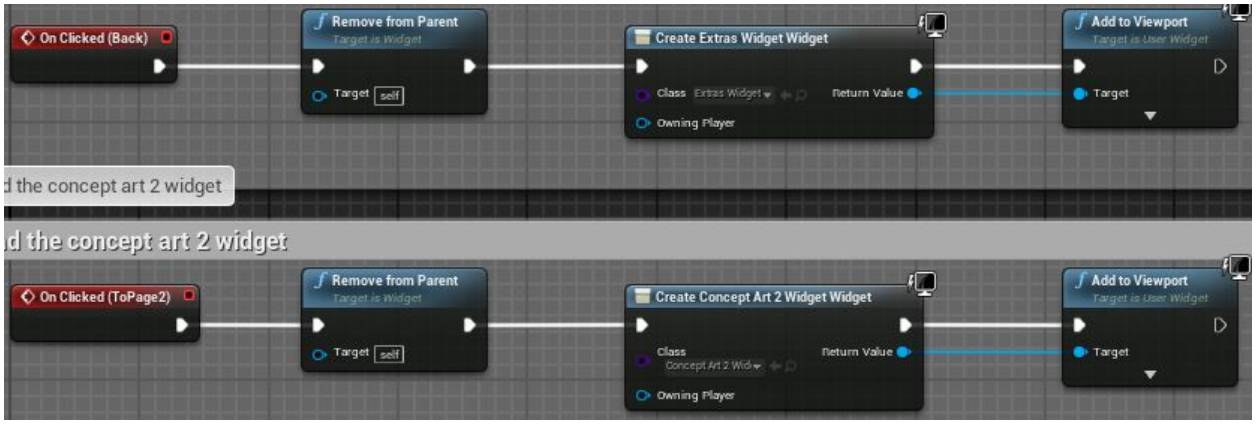

Figure 2.6.9-1

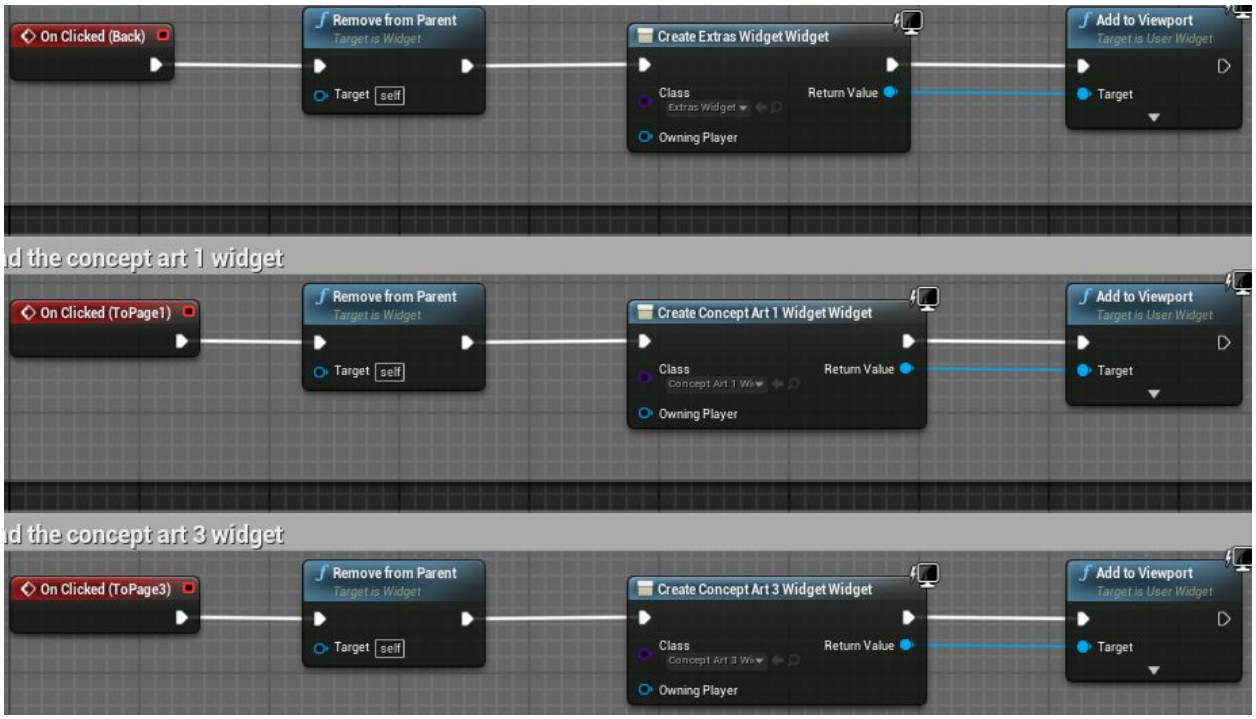

Figure 2.6.9-2

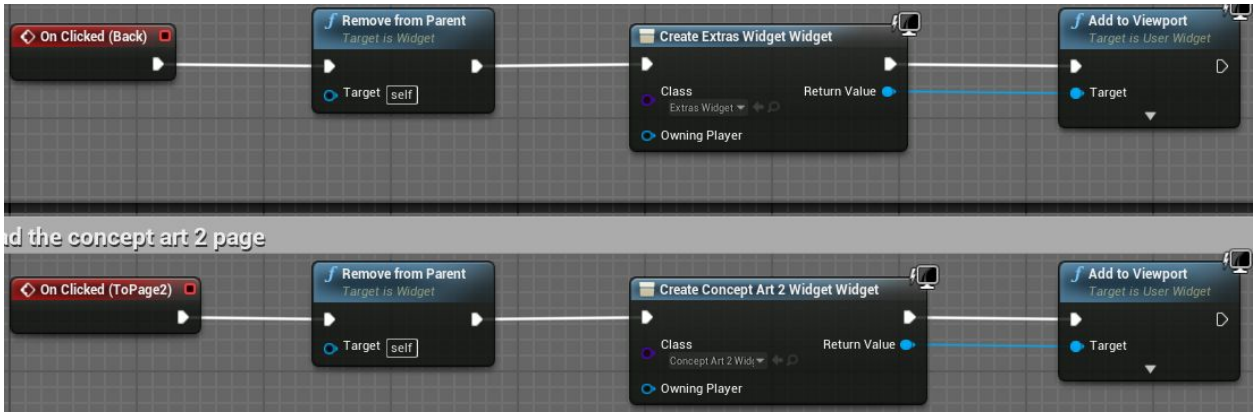

Figure 2.6.9-3

2.6.10 Story Screens 1, 2 and 3

With these three story screens, the user can view the story behind our game. The three screens contain the story and information surrounding our game world and characters. Just like concept art screens from the last section, the user can traverse between all three of the story screens using the arrow buttons in the lower right and left corners of each screen. The back button in the upper left corner of the screen will also return the user to the extras screen in any of the three story screens. The images below represent the code for story 1, story 2 and story 3 widgets respectively.

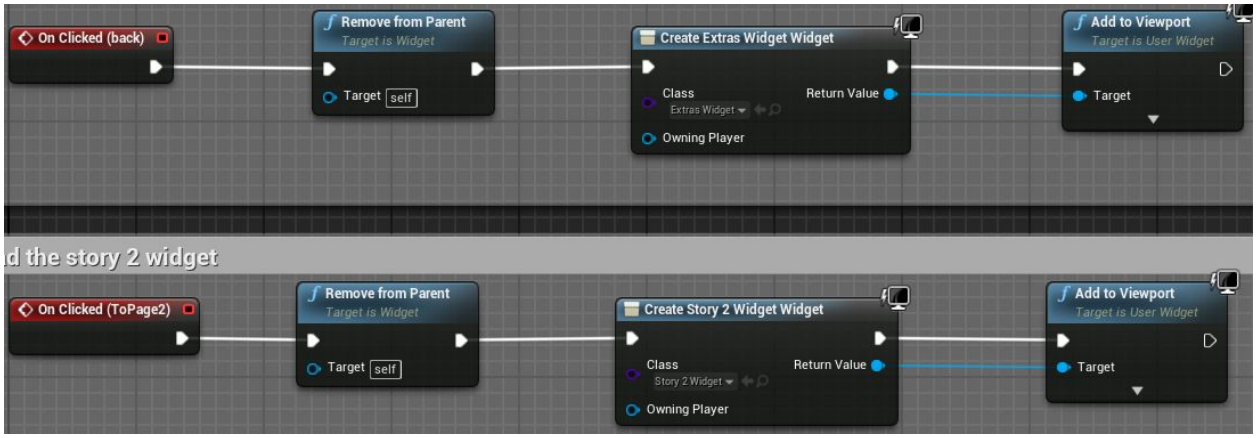

Figure 2.6.10-1

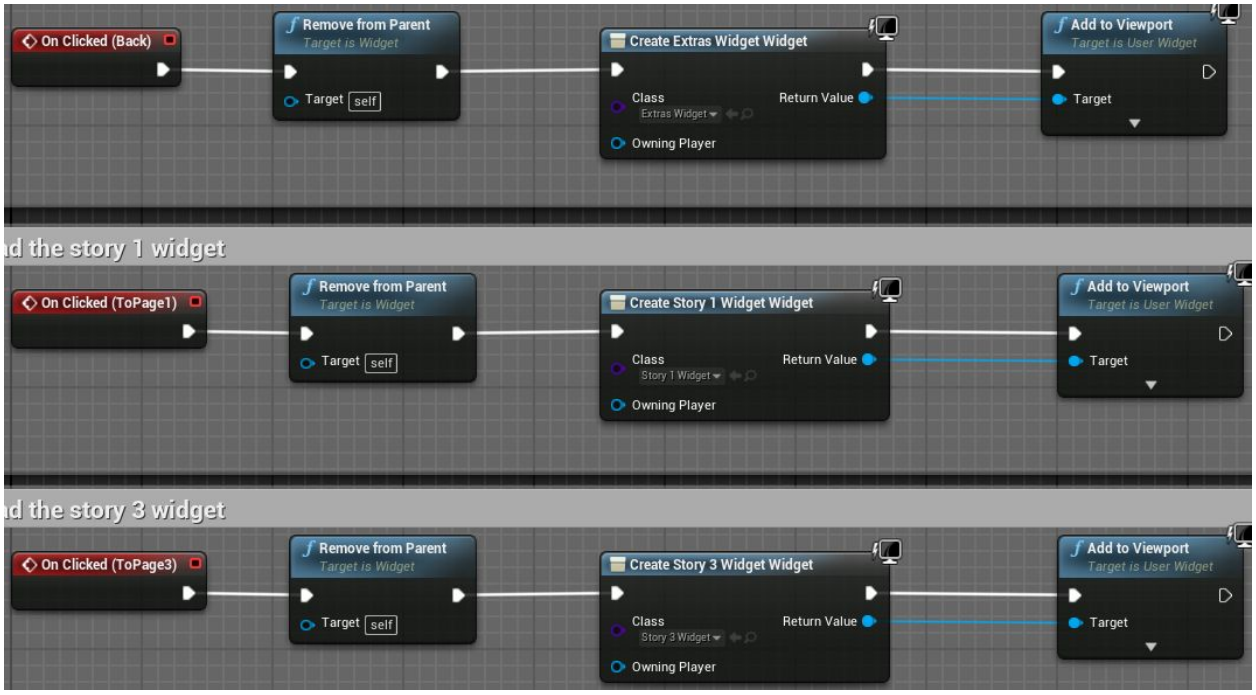

Figure 2.6.10-2

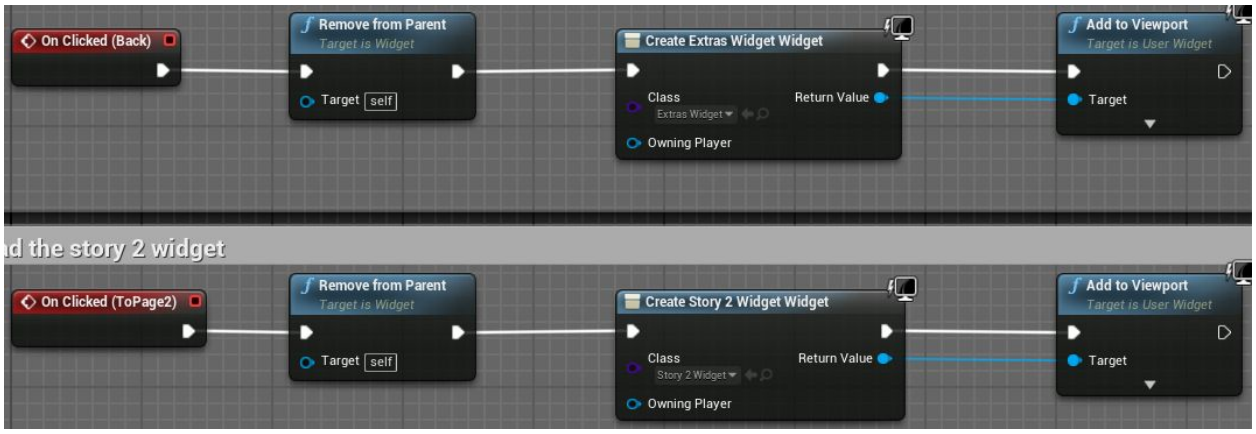

Figure 2.6.10-3

2.6.11 Character Select Screen

At the character select screen the players are able to select their characters. Player 1 selects their character first by clicking with the mouse cursor the character they want to play as. This sets the choice integer value to either a 0 or a 1 based on the character choice. Once player 1 chooses a character, the “Player1Selection” variable is set. Player 2 now selects their character choice and the “Player2Selection” variable is set. The variable for the branch condition is by default set to true so that the first character selection is for player 1. These variables are then saved into a save file. These player selection variables were used for both the player spawning mentioned in section 2.5.1 and the stage overlay in section 2.5.3. The code for saving the variables in the save file is the same as mentioned before. The player is able to traverse either

forward to the stage selection screen or back to the multiplayer screen. If the users progress to the stage select screen without selecting characters, default selections will be made for both players. Below are images showing the functionality for the back and stage select buttons in the top left and right hand corners of the screen and the player character selection.

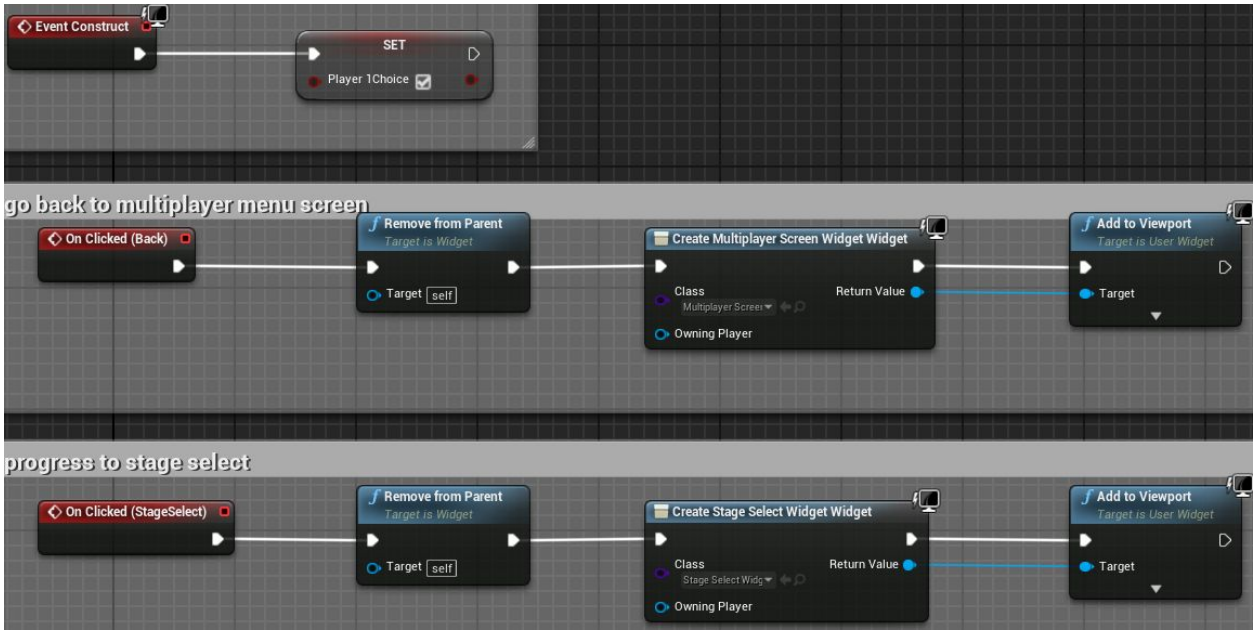

Figure 2.6.11-1

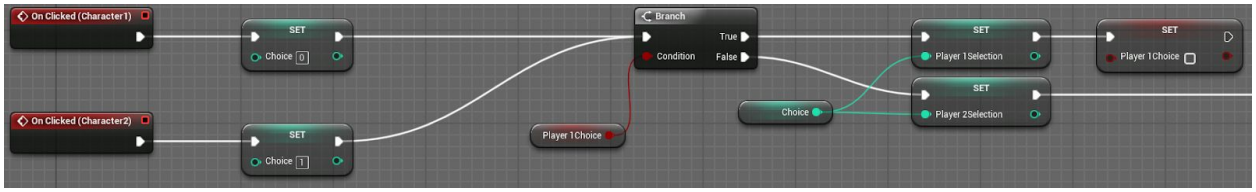

Figure 2.6.11-2

2.6.12 Stage Select Screen

The code for the stages first saves an integer holding the stage choice, either a 0 or a 1 and then loads the level of the stage selected. The value of 0 means the Cosmic Rift stage is selected and 1 means the coliseum stage is selected. Saving the selection of the stage choice is important in section 2.6.14 for the rematch feature. The player also has the ability to return to the character select screen and reselect characters again with the back button in the upper left corner of the screen. Below are the images for the code.

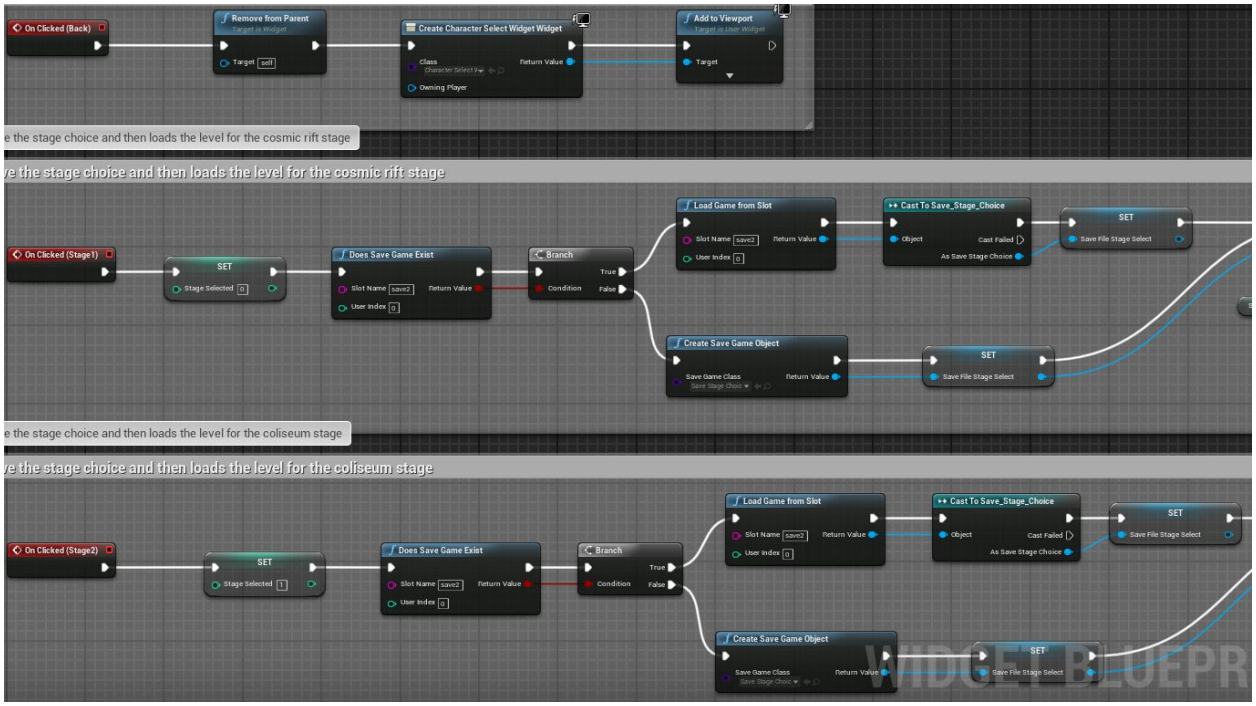

Figure 2.6.12-1

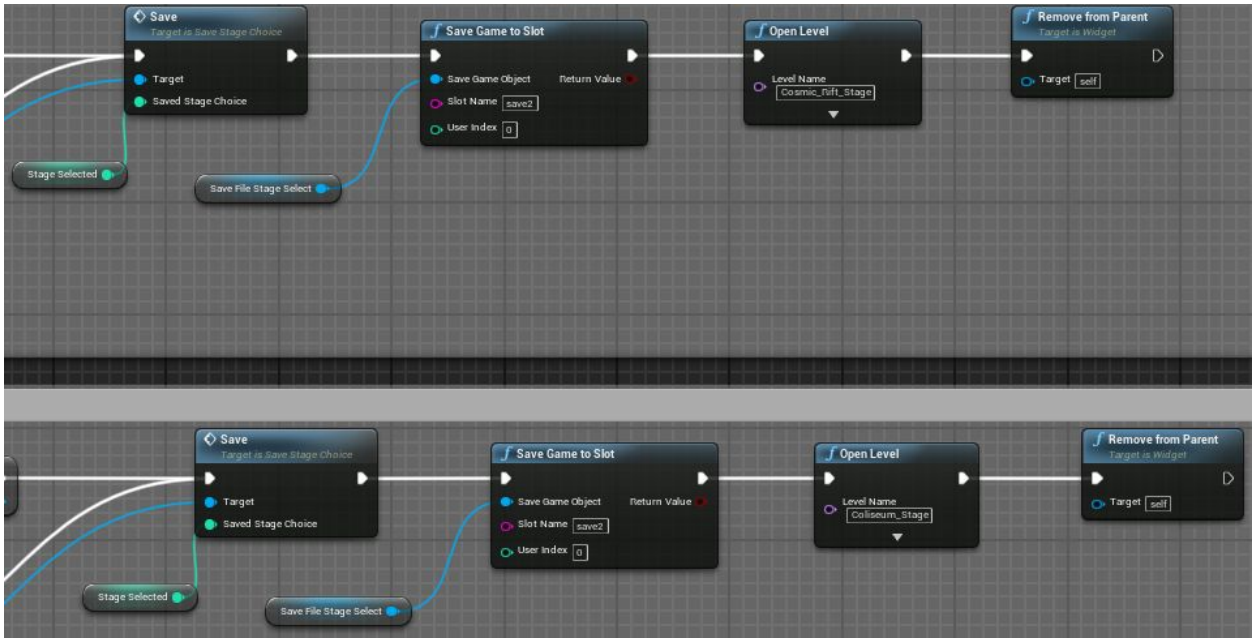

Figure 2.6.12-2

2.6.13 Pause Screen

In the paused state the players have different options they can select. The players can return to the character select screen and change their character choices. They can return to the stage select screen and change the stage or they can also return to the main menu. The functionality of the resume game feature just returns to the current game state. Either pressing the p key or pressing the resume game button with the mouse cursor both have the same effect where the current game state will resume. The images below display the code for all the pause screen choices.

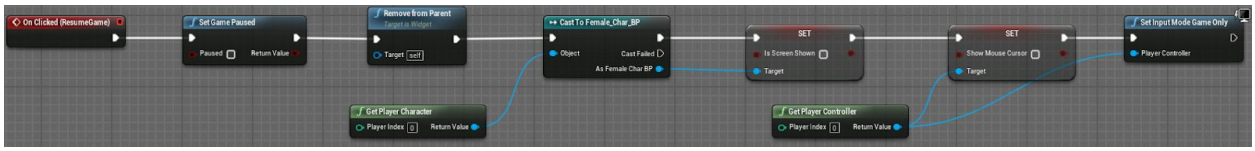

Figure 2.6.13-1

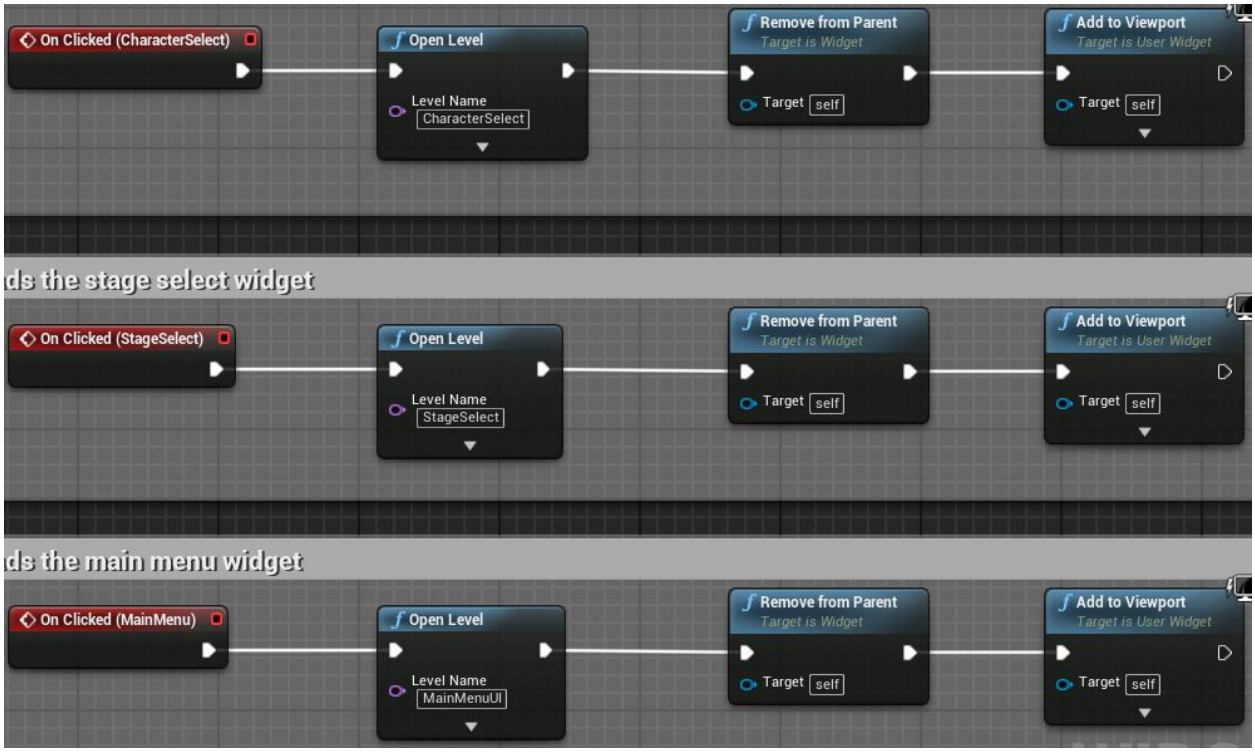

Figure 2.6.13-2

2.6.14 Game Over Screen

When the game ends and one player loses all three of their stocks, the game over screen will be displayed. The players can either rematch the other player where the character choices and the stage choice remains the same. They can return to the characters select screen where either one or both of the players can change characters. The players can return to the stage select screen and select a different stage or they can return to the main menu. The logic for the rematch feature uses the stage choice variable that was saved when selecting the stage. Using

a branch condition we can reload the same level that was played on previously. If the variable is equal to 0 then the cosmic rift stage is reloaded and if the variable is 1 then the coliseum stage is reloaded. The functionality for all of these options is displayed in the images below.

Figure 2.6.14-1

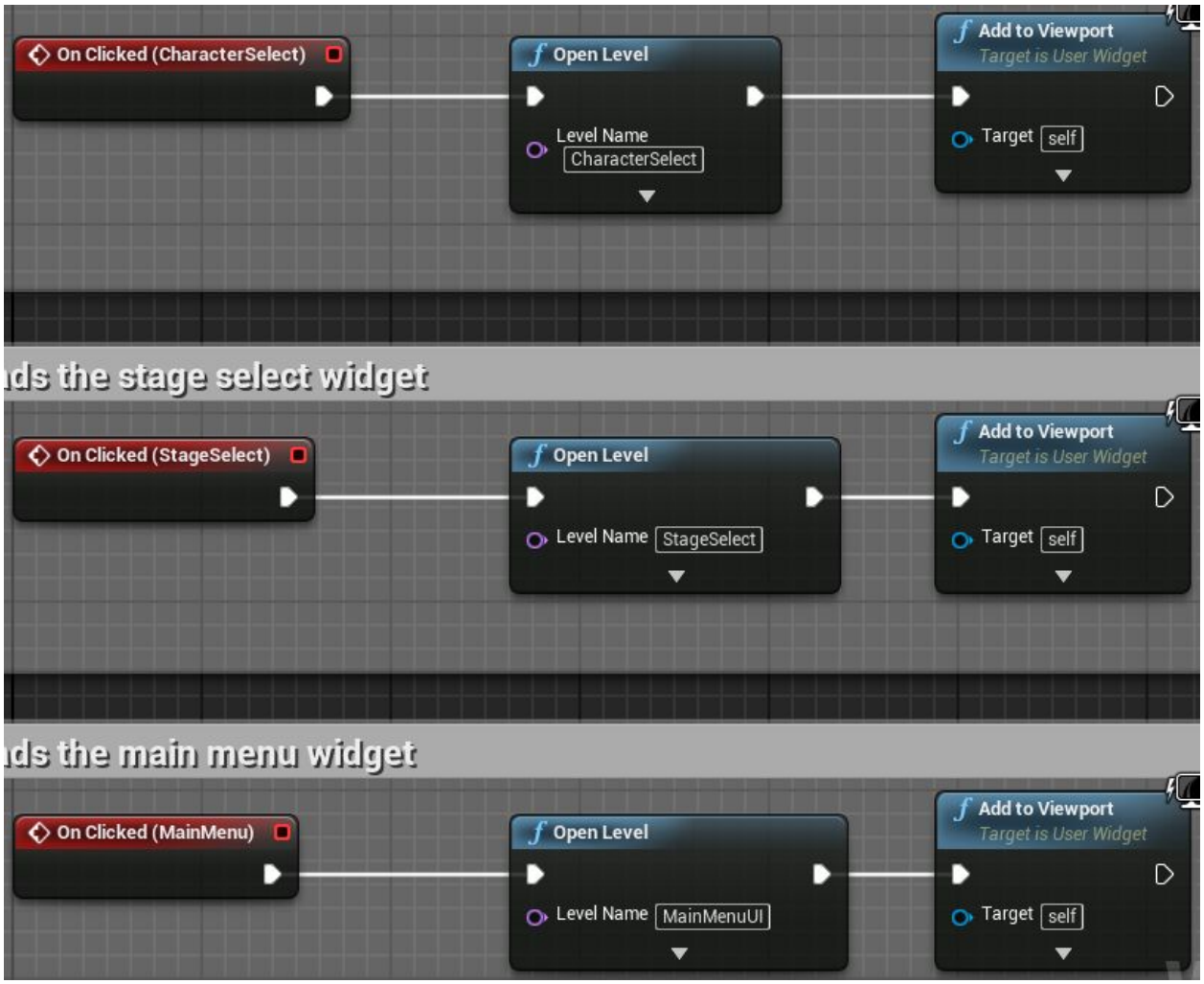

Figure 2.6.14-2