
Senior Projects Fall 2019

Bard Undergraduate Senior Projects

Fall 2019

Rhythm Quest: Creating a Music Video Game

Tanner Daniel Cohan
Bard College, tc1216@bard.edu

Follow this and additional works at: https://digitalcommons.bard.edu/senproj_f2019

 Part of the [Game Design Commons](#)



This work is licensed under a [Creative Commons Attribution-NonCommercial-No Derivative Works 4.0 License](#).

Recommended Citation

Cohan, Tanner Daniel, "Rhythm Quest: Creating a Music Video Game" (2019). *Senior Projects Fall 2019*. 51.

https://digitalcommons.bard.edu/senproj_f2019/51

This Open Access work is protected by copyright and/or related rights. It has been provided to you by Bard College's Stevenson Library with permission from the rights-holder(s). You are free to use this work in any way that is permitted by the copyright and related rights. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself. For more information, please contact digitalcommons@bard.edu.

Rhythm Quest:
Creating a Music Video Game

Senior Project Submitted to
The Divisions of Science, Mathematics, and Computing and Arts
of Bard College

by
Tanner Cohan

Annandale-on-Hudson, New York
December 2019

Acknowledgements

Special thanks to Keith O'Hara, Matthew Sargent, and Jennifer Triplett for helping me get through this project in one piece, I couldn't have done it without their support, assistance, and much more. Thanks to all of my friends and family who supported me even when I needed to delay the completion of my project, and special thanks to those friends who tested my game and allowed me to bounce ideas off of them. I couldn't have done this without support from all those around me.

Table of Contents

Abstract	1
Chapter I - Introduction	2
Chapter II - Previous Work	4
Chapter III: Tools	17
Chapter IV - Game Design	19
Chapter V - Conclusion	39
Chapter VI - Source Code	41
<i>CameraMover</i>	41
<i>PlayerControl</i>	42
<i>enemyspawner</i>	60
<i>Battle_Menu_Control</i>	64
<i>MusicControl</i>	72
<i>HorizReseq</i>	73
<i>VertRemixer</i>	77
<i>ArmAnimator:</i>	79
<i>Instructor_Move</i>	80
<i>DialogueManager</i>	90
<i>Dummy</i>	92
<i>Spot</i>	93
<i>FollowArrow</i>	95
<i>SpecialHeader</i>	95
<i>DialogueBox</i>	96
<i>DialogueTrigger</i>	97
<i>SpecialAttack</i>	98
<i>Tutorial Cutscene Guide</i>	104
Chapter VII - Bibliography	107

Abstract

This project was the creation of a music-centric video game named *Rhythm Quest*. *Rhythm Quest* is a combination of two very distinct video game genres, in a way that has not yet been done in the mainstream, particularly taking inspiration from the rhythm game *Rhythm Heaven* and the turn-based roleplaying-game (RPG) *Paper Mario*. It was a joint project between the computer science and music divisions, and involved the creation of all assets necessary to create a video game, including sprite creation, software development, writing, music, and sound effects. The game's tutorial, as well as the battle system, is complete.

Chapter I - Introduction

Two of the most well-known and often acclaimed genres of video games, music games and turn based RPGs have been around since the late 20th century. However, very little has been done to bridge the gap between the two genres. The closest game in mainstream to try a similar genre combination is *Crypt of the Necrodancer*, a rhythm game crossed with another popular video game genre: the “roguelike,” a type of video game involving exploring dungeons and collecting items. However, very few video games have attempted to combine turn based combat with rhythm and, as a composer and programmer, I wanted to create a game that could help showcase my pieces of music. Because of this, the rhythm genre seemed like the perfect choice. After all, rhythm games are all about paying attention to the music and performing actions in rhythm. But at the same time, I wanted to create an adventure, something akin to the beloved *Final Fantasy* series of games, in which you play a party of adventurers exploring and defeating creatures. Thus, the idea for a turn-based RPG rhythm game was created. This unexplored territory would be my first major venture into game development. Armed with video game developer software Unity, music software Logic, a working knowledge of the C# programming language, and a vision, I set off to create a video game.

As an avid lover of music, whether it's listening to it, playing it, or studying its theory, rhythm games have always held a special place in my heart. While they're extremely fun, colorful, and entertaining, amidst all over the color and spectacle, deep down they're a form of music education. Rhythm games encourage players to perform actions to the beat of the music,

which can help practice keep rhythm, an instrumental skill for any musician. In a way, the controller becomes something akin to an instrument. Combining rhythm games with other genres is a great way to teach music and rhythm without the player necessarily realizing it, which is one of the main reasons I decided to create *Rhythm Quest*. Sure, it focuses on being a fun experience for the player, but it also helps develop rhythmic skills in an entertaining, video game format.

Chapter II - Previous Work

My game is a cross between two longrunning genres of video games. One of those genres is the rhythm game genre, the other being the roleplaying game genre. Rhythm games (also known just as “music games”) are games where the main objectives revolve around performing a series of actions in time with music. The genre has a long history that dates back to the 1970s, most people citing the popular game *Simon* as the pioneer of the genre. *Simon* was a small electronic game that was, while not quite a video game, considered the first electronic music game. The toy was created by Ralph Baer, who is also known for creating the first video game console (Smithsonian). He took inspiration from the Atari game “Touch Me,” an old rhythm game that, according to Baer, was a great idea with a terrible execution (Smithsonian). With this in mind, in 1978, Baer created *Simon*, a circular toy with four colored buttons. *Simon* is a memory game in which a player must press the buttons in a particular sequence. Each subsequent sequence was the same as the previous one, with one new one added to the end of it. The buttons would light up for the player, who would then press the buttons in the order they had previously lit up. A couple decades later, in 1996, *PaRappa the Rapper* was released on the Sony Playstation, and is often referred to as the first true rhythm video game. Like *Simon* before it, *PaRappa the Rapper* also utilized a call-and-response style of gameplay, where players would spend part of the game watching the computer demonstration, then press buttons in synchronization with the beat (Digital Game Museum). *PaRappa the Rapper* sold quite well and helped popularize the genre of rhythm games, spawning spin-off titles, a sequel, and even a TV

series (Digital Game Museum). The game was also the 7th best selling game of 1997 in Japan (The-Magicbox). The year after *PaRappa the Rapper* released, video game company Konami released the arcade game *Beatmania*, developed by their brand new Games and Music division (Digital Game Museum). The game used a unique controller; a keyboard with 4 white keys and 3 black keys, as well as a turntable-like controller. Unlike *Simon* and *PaRappa* before it, *Beatmania* did not use a call-and-response system, but instead had players press buttons on the keyboard as they appeared on screen. *Beatmania* was also the first rhythm game to use licensed music, which would become a trend throughout the genre. Due to the game's success, Konami's music game division changed their name to "Benami."



(Left) the toy "Simon," created by Ralph Baer. The four colored panels light up in a specific order and afterwards the user must press them in the same order. (Right) the controller for the Konami game "Beatmania," featuring a turntable on the left and keyboard-like buttons on the right.



Screenshot of "PaRappa the Rapper" (1996 Sony Entertainment). The bar at the top of the screen dictates what button the player has to play when.

Beatmania was a major success, but, like *PaRappa the Rapper*, most of its success was contained to Japan. The first internationally popular rhythm game would also release in 1998, known as *Dance Dance Revolution*. Arguably the most well-known of the entire genre, *Dance Dance Revolution* (often shortened to *DDR*) was developed by Benami. Due to the success of *Beatmania*, Benami started to experiment with different controllers for rhythm games, including *DDR*'s well-known floor pad. A dancing game, *DDR*'s controller was on the floor, and primarily consisted of 4 buttons, one for each directional input. The screen displayed a series of arrows, and players would have to press the corresponding buttons with their feet. *Dance Dance Revolution* is credited as being the first rhythm game that had marginal success outside of Japan, and, even though American arcades were skeptical at first, the game soon found its way into arcades all over the country (Digital Game Museum). There have been countless rereleases of the rhythm game since its initial release in 1998, on practically every video game console imaginable, and the game is even recognized as an official sport in Norway (Digital Game Museum).



The controller and gameplay of Konami's "Dance Dance Revolution." The pad on the left features four arrow keys that the player must press with their feet, which corresponds to the arrows on the screen on the right.

Due to the resurgence of popularity of rhythm games worldwide, American video game company Harmonix started to experiment in the genre, starting out creating games inspired by *Parappa the Rapper*. They released the game *Frequency* in 2001 for the Playstation 2 and, despite being enjoyed by critics, the game didn't sell particularly well (Digital Game Museum). Around this time, RedOctane, a company whose main business relied on selling *Dance Dance Revolution* pads, had an idea for a game that used a guitar as the main controller, but needed a software company to develop it with. Together, Harmonix and RedOctane developed *Guitar Hero* in 2005, which was almost an instant success (Digital Game Museum). Player would wield a guitar-shaped controller consisting of five colored buttons in the place of frets and a "strum bar" at the base. Players would have to press the correct colored button and strum the strum bar at the same time to successfully play a note. The game used a large tracklist filled with different western rock and roll songs, and featured an adjustable difficulty level for each one. The game spawned multiple sequels, and by 2008 the series had made over one billion dollars in revenue (Edge). Harmonix also went on to develop *Rock Band*, a similar game, but with more peripheral controllers. Released in 2007, *Rock Band* was a multiplayer game in which players would each use a different instrument-themed controller, including a lead guitar, bass guitar, drum set, and microphone. Like *Guitar Hero*, *Rock Band* also sold well, producing another one billion dollars in revenue (RockBand.com). *Rock Band* was so influential that, in 2008, *Time Magazine* listed *Rock Band* creators Alex Rigopulos and Eran Egozy as two of their top 100 most influential people of the year, crediting the two as having "saved classic rock for years to come" (van Zandt). While *Guitar Hero* and *Rock Band* were two of the most popular franchises of the late

2000s, both dwindled in popularity shortly after the turn of the decade. *Guitar Hero* publisher Activision closed its *Guitar Hero* division in 2011, and *Rock Band*'s most recent game was in 2015 (Digital Game Museum). While both franchises still exist, their sales aren't nearly as high as they had been between 2005 and 2008.



Various controllers and Gameplay of Harmonix's "*Guitar Hero*". The colored buttons on the controller's neck correspond to the colored discs on the right, and the player must press the correct key as it lines up with the circles near the bottom of the right picture.

While the popularity of Rhythm games in the US were primarily contained to *Guitar Hero* and *Rock Band*, in Japan the genre was diverse, with many series of rhythm games maintaining popularity. Notable series include Nintendo's *Rhythm Heaven*, a collection of short rhythm games revolving around simple controls, iNiS's *Elite Beat Agents*, a Nintendo DS game where the player used the touch screen to keep the beat, and Namco's *Taiko no Tatsujin*, a series where a Japanese Taiko drum is used to keep rhythm. All franchises remain liked internationally, but sales indicate that each series is mostly popular in Japan.



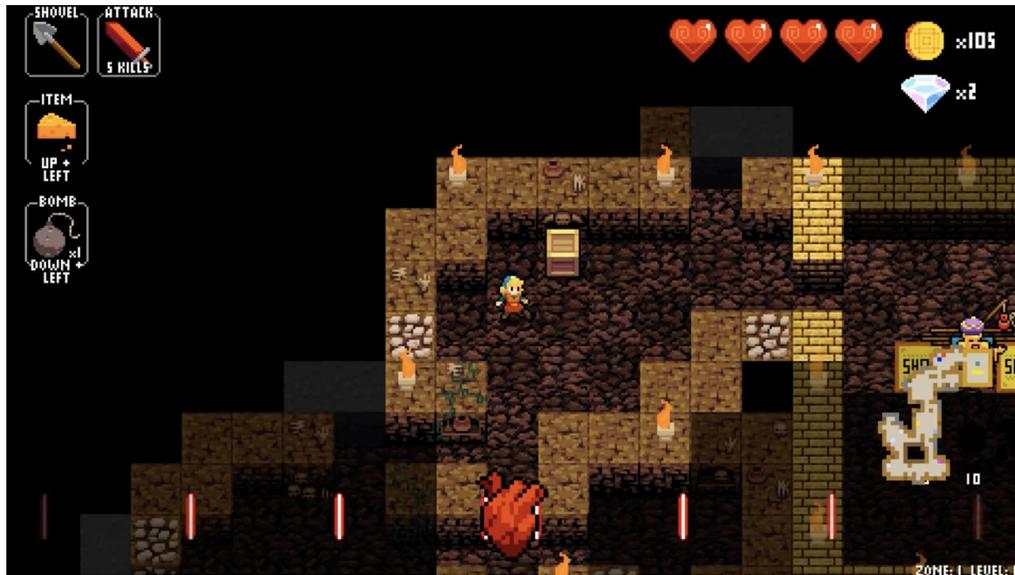
(Left) “Rhythm Heaven Fever,” (2009) developed by Nintendo SPD. In this minigame, the player (the monkey on the clock hand) must press buttons in time with the music to high-five the other monkeys. (Right) “Taiko no Tatsujin: Drum n Fun!” (2018) developed by Bandai Namco. In this game, the player must press buttons as the red circles approach the left side of the screen.

Rhythm games were losing popularity in the United States, but one game series remained popular throughout the 2010s, particularly Ubisoft’s *Just Dance* franchise. In 2009, the first *Just Dance* game released on the Nintendo Wii, featuring songs from a wide array of popular artists of the time. The game was not well critically received, with an aggregate review score of 49 out of 100 (Metacritic). However, the game was still a commercial success, selling 4.3 million units by 2010. The series continues to have an annual release, the most recent game being *Just Dance 2020*, which was released on November 5th, 2020, with 30 games total in the franchise. The game is motion based, and players participate by mimicking the dancers as they move on the screen.

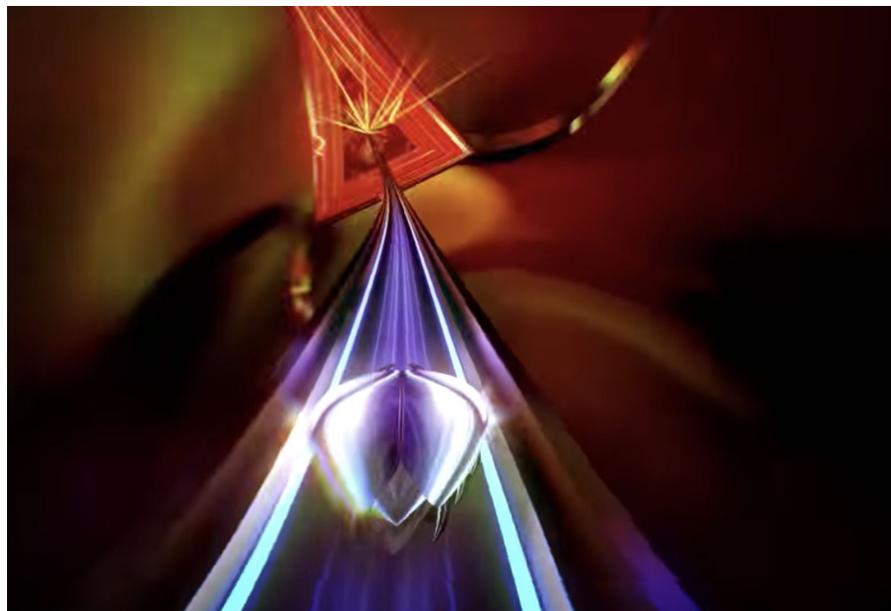


10
“Just Dance 2020” (2019), developed by Ubisoft Paris. As with other entries in the series, the players must dance in time with the music.

With the exception of the *Just Dance* franchise, rhythm games in the 2010s started to become more experimental, sometimes combining with other genres. Indie company “Brace Yourself Games” released *Crypt of the Necrodancer* in 2015, a combination of the rhythm game genre and the “roguelike” genre. Roguelikes are games that involve players exploring procedurally created dungeons and fighting enemies, typically based off of the fantasy genre. *Crypt of the Necrodancer* is a roguelike where the player explores dungeons while moving around and attacking to the beat of whatever music is playing. In 2016, developer “Drool” released *Thumper*, a highly acclaimed rhythm horror game, the game’s developer calling it a “Rhythm Violence Game” (Walker). The game features notes as physical obstacles the player encounters, and is available to play in virtual reality.

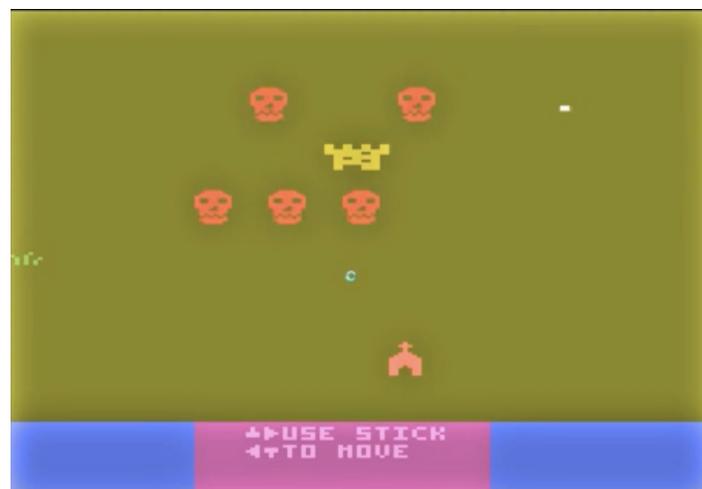


Brace Yourself Games' "Crypt of the Necrodancer" (2015). The tiles on the floor switch colors, and the player must jump between them in time with the music.



"Thumper" (2016) by Drool. The player controls the metallic creature at the bottom of the screen, and goes along the path, encountering rhythmic obstacles along the way.

While Rhythm games remain a staple for people who enjoy casual gaming, the other genre my game is based on, the turn based role-playing game genre, has a very different demographic. Role-playing games (or RPGs) are fantasy games in which the player assumes the role of another character, typically in a fantasy setting. RPGs aren't necessarily just video games, as the well-known tabletop game *Dungeons and Dragons* is also an RPG. Turn-based RPGs are a subset of role playing games where, when players enter combat with an enemy, the game shifts into a system where the player and enemy take turns performing actions in competition with each other, often with the goal of defeating each other. The first RPG on a video game console is often credited to the 1982 Atari 2600 game *Dragonstomper*, developed by Starpath (Vestal). It included many staples that turn based RPGs would have in the future, such as a money system, random enemy encounters, and, most importantly, turn based combat. While *Dragonstomper* was the first RPG, it remains relatively unknown to this day.



13

Gameplay of “*Dragonstomper*” (1982), developed by Starpath. The player controls the white dot in the top-right corner

In 1986, Enix published *Dragon Quest* for the Nintendo Entertainment System (NES), which is considered one of the most important RPGs of the entire genre. Developed by game designer Yuji Horii, *Dragon Quest* drew influence from several story based games before it, particularly the murder mystery game *The Portopia Serial Murder Case*. *Portopia Serial Murder Case* was developed by Horii in 1983, and was eventually ported to the NES in 1985, and this port is was mainly inspired *Dragon Quest*. The original version of *Portopia* was on a system with a keyboard, so when porting it over to the NES, a console that had a limited controller with only a few buttons, Horii and team had to create a completely different control scheme, and what they developed is what inspired the control scheme for *Dragon Quest* (1up.com). The NES itself also served as a large inspiration for *Dragon Quest*. Since before the NES most video games were played in arcades, they were designed for shorter style gameplay. However, when the NES came out, a system that was meant to be played in a household where games could be long and spanning, Horii wanted to make a game that required more of a time commitment and a more complicated system (1up.com). The game had a turn-based battle system, as well as an overworld map where the player could freely explore. While the game was not initially successful worldwide, the game sold very well in Japan, which allowed for the series to continue. *Dragon Quest* would go on to be one of the most prolific video game franchises, particularly in Japan, with 11 games in the main series, plus numerous spinoffs.



(Left) “The Portopia Serial Murder Case” (1983 Chunsoft). It features a screen showing the current scene, as well as text featuring options of things the player can do. (Right) A battle from “Dragon Quest” (1986 Enix). The enemy appears in a window in the center of the screen, with several options for the player in a menu on the left.



“Dragon Quest XI” (2017 Square Enix). The battle system remains similar to the first game in the series, with the enemy in the center of the screen, with options for the player on the bottom right.

The next major influential RPG was a game developed by Square in 1987 known as *Final Fantasy*. *Final Fantasy* had a lot in common with *Dragon Quest*, but it started a few trends that would become mainstream in turn-based RPGs, such as customizable playable characters, and the layout of the battles themselves. While in *Dragon Quest* you face the enemy head on without seeing the playable character on screen, in *Final Fantasy* the player stands on one side of the screen, while the enemy is on the other. This style of battle is the type that I based mine on. *Final Fantasy*, like *Dragon Quest*, is a long-running franchise, with 16 entries in the main series. In fact, *Dragon Quest* company Enix and *Final Fantasy* company Square merged in 2003, so both major turn based RPGs belong to the same developer company.



Battle screens from various games in the “Final Fantasy” series. The left is “Final Fantasy” (1987 Square), and the right is “Final Fantasy VI” (1994 Square). Both battles feature players on side and enemies on the other, facing each other.

Aside from hardware improvement, the core gameplay of turn based RPGs has not changed much since its creation in the 1980s. However, one other RPG would prove to be a major inspiration for my game in particular, and that’s Intelligent System’s 2000 turn based

RPG, *Paper Mario*. *Paper Mario* is a spinoff game from the prolific Mario series in which the titular plumber battles classic enemies in a turn-based RPG setting. While RPGs before this game simply had players selecting attacks from a menu and watching them carry through, *Paper Mario* was a bit more complex. Not only did one have to select their attack, but the player had to perform some sort of procedure to make sure the attack was successful, such as pressing buttons at exactly the right time, or pressing one button as many times as they could in a short amount of time. This added much more interactivity into the battles and gave the player more things to do, and this idea is something I incorporated into my game, adding the stipulation that the actions that needed to be performed had to be in time with a specific rhythm.



19



20

(Left) A boss battle from “*Paper Mario*” (2000 Intelligent Systems). The battle system has Mario and one ally on the left facing an enemy on the right, with multiple options to choose between. (Right) Mario performing a hammer attack in “*Paper Mario: The Thousand Year Door*” (2004 Intelligent Systems). To successfully perform the attack, the player must hold the control stick to the left while the three dots on the left side of the screen light up, and release it when the bigger dot lights up.

Chapter III: Tools

For the creation of my project, I used Unity for the majority of game development. Unity is a game engine developed by Unity Technologies that uses C# and GUI to create games and other audio/visual projects. Unity allows the user to create a list of game objects that each have a list of properties and components that can be applied to them. Each object automatically has a “transform” component that controls its position in the game, as well as its scale, rotation, and anything else pertaining to its location. Other examples include an audio source, which allows you to attach an audio source to the object to produce music or sound effects, a sprite renderer, which manages the appearance of in-game sprites, and animator controller, which manages an object’s animation. Arguably more important is the ability to create your own object components by writing scripts in C#. This is where the bulk of game creation is done.

Another important aspect of Unity’s user interface is the animator, which is where sprite animation is handled. Each object with an animator controller has an animator, which is represented by a flowchart of different boxes, starting with a box labeled “entry.” from there, one simply drags a set of png images into the game object to create a new animation, which manifests as another box in the animation flowchart. Transitions can be created as arrows between the boxes that can be triggered in scripts attached to the game object, or could have an animation transition after an animation had performed exactly once. This built in animation system was a crucial part of the game design, as having objects undergo animation was a necessary way of polishing the game.

Chapter IV - Game Design

Rhythm Quest is a game set in a fantasy world about a bard who's always wanted to go on an adventure finally getting an opportunity when a music-based enemy starts to attack the world. The player controls Wolfgang, a somewhat lazy guitarist from a small town of Preludia, which resides in the middle of a deep forest, who's grown up in the shadow of his brother, a world-renowned adventurer. However, once his music teacher disappears, Wolfgang sets out on a musical adventure, fighting enemies to the rhythm of the music. Once he finds his instructor, she tells him of a rhythmic demon named Cacophony, and that the only way to be able to even approach him would be to gather a choir of four specific singers, as well as their composer. The rest of the game involves Wolfgang traveling around the world to find the singers, visiting musically titled places such as Groveture Grove, Bandoneon Bay, Dixie's Land Casino, Minuet Manor, and the Temple of Tempo. The game is composed of 6 chapters, each with a boss at the end. Chapter one is the prologue, where Wolfgang rescues his music instructor (and soprano member of the choir) from a spider named Ella and a mouse called Micetro. Chapter two is on the oceanside where Wolfgang rescues the group's composer from a group of pirates called The Bandoneon Bandits. In chapter 3, Wolfgang travels up a mountain to a casino, where he battles King Swing and rescues the group's bass singer. In chapter 4, Wolfgang goes to a haunted mansion to fight a ghost named Spiritoso to rescue the group's alto, and chapter 5 he fights a demigod Atempol to save the group's tenor. Finally, chapter 6 has Wolfgang travel to a dark world where he finally faces off against Cacophony.

There are two parts to the game: the overworld and the battle system. The overworld is where the bulk of the story happens, and has the player moving from area to area. There are non-playable characters to talk to, as well as paths to navigate. Often, enemies will appear and, if one comes in contact with the player character, the player enters into the battle system.

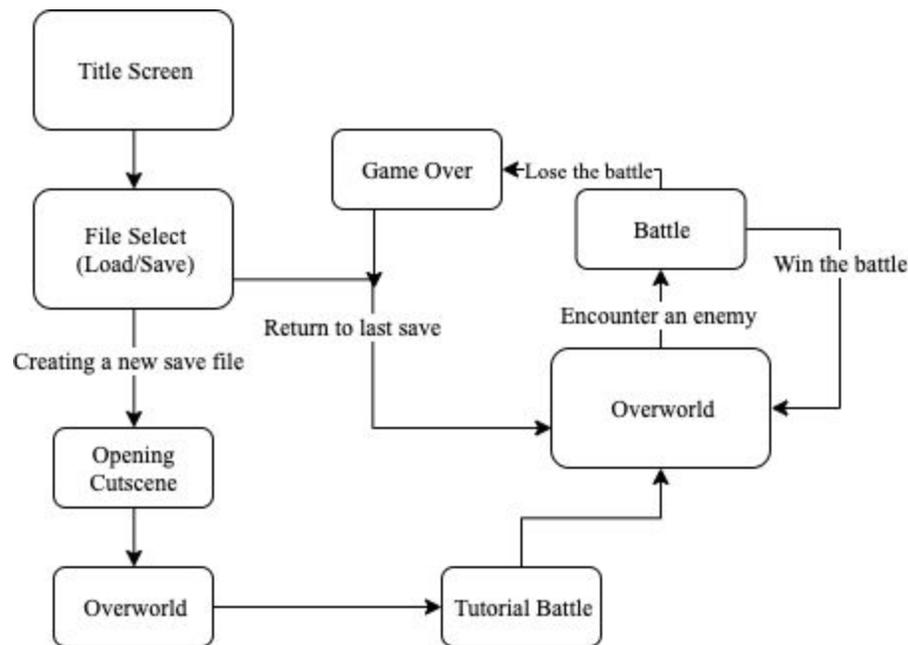


Figure 1: Game Flow

The battle system of *Rhythm Quest* is similar to many other turn based RPGs, particularly *Paper Mario*, with the addition of rhythm elements that the player must complete. In each battle, the player gets a drop down menu with different actions they can perform. “Attack” allows the player to perform a physical attack on a single enemy, “Items” allows the player to use an item from their inventory with varying effects depending on the chosen item (this feature has yet to be implemented), “Special” allows the player to perform special attacks that they can learn

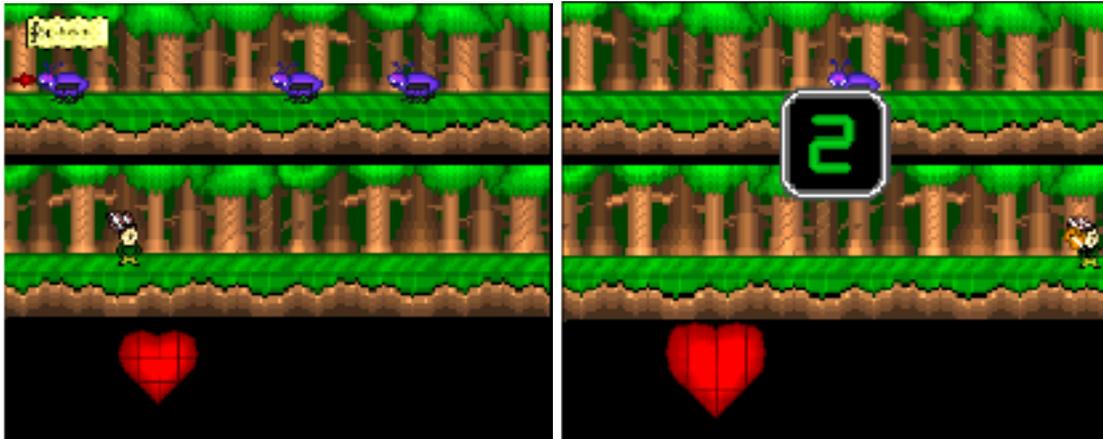
throughout the game with varying effects, and “Other” includes various choices, like running from battle. Each menu brings the player to a subsection of menus where further choices can be made. For example, once the attack menu is selected, the player can then select which enemy they want to attack. Creating the menu system was not too much of a challenge, mostly just involving assigning unique integers to menu choices and keeping track of which ones were picked. The screen is split into two connecting parts, with the bottom right connecting to the top left. The player stands in the bottom portion of the battle arena, while the enemies stand on the top. This is unorthodox for an RPG, as usually there’s only one screen showing the player on one side and the enemy on the other, but I split it up into two parts to distinguish between the player’s defending area and the enemy’s defending area.



Rhythm Quest's battle menu, featuring four options on a piece of sheet music.

When the player selects which enemy they plan on attacking, the player character runs to the bottom right side of the screen, waiting at the point where the battlefield loops around to the top. This is to make sure that, when the player does attack, the attack starts at the same time the next measure of music does to ensure that everything is synchronized. Once the player is waiting on the left side of the screen, a countdown timer appears and counts down from three, just to give the player a warning about when they'll be attacking. Once it reaches zero, the player starts to run into the enemy section of the battlefield and will approach where the enemy they picked is standing. The trick is that when the player reaches the enemy, they must attack in synchronization with the music. Enemies can stand in any 4 spots on the battlefield, each spot corresponding to a beat. The spot closest to the player is the first beat, second closest is the second beat, etc. Depending on where the chosen enemy is standing, the player must hit the enemy on that beat, as it takes the player character one full measure to run across the screen. The player must press the 'X' button to perform the attack, but they can also press the 'Z' button to do a jump attack. To successfully perform the jump attack, the player must press 'Z' exactly one beat before the player hits 'X'. For example, if the enemy is standing in the 3rd spot, the player must hit 'Z' during the second beat and 'X' during the third beat. The player also could decide to use a special attack on their turn. Special attacks are based on musical instruments, and there are 6 attacks in total. However, the section of the game that's completed only has one instrument unlocked, so there's currently one special attack available. The attack has the player character pull out their guitar, while 4 strings fall down over the enemy places. Circles will

randomly fall down the strings and overlap with the enemies. When they do, the player must press a button corresponding with each enemy spot.



(Left) the player selecting which of the 3 enemies to hit. Currently, the enemies are standing in attack spots one, three, and four. (Right) the player waiting to perform an attack on the selected enemy. The player waits at the bottom right of the screen until the countdown in the center reaches zero.

Once the player finishes their attacks, the enemy's turn begins. During this phase, each enemy currently in the battle gets the chance to perform an attack. When an enemy gets ready to attack, they wait at the top left part of the screen, right at the part where the screen loops down to the bottom. Like with the player, this is to make sure the enemy's attack is synchronized during the music. While the enemy is attacking, the player can move in 4 different directions. If the left arrow key is pressed, the character moves to the left for a few seconds, if the right arrow key is pressed, the character moves to the right, if the down arrow key is pressed the character ducks, and if the up arrow key is pressed, the character jumps. These are the four ways in which the player can dodge the enemy attacks. However, which way the player needs to dodge depends completely on the enemy. If the enemy attacks from above, for example, the player would need

to either dodge to the left or right. The player would also have to press the dodge button at the right time, and the key to doing that is visual clues from the enemy and audio clues from the music. Once the enemy phase ends, the player's turn begins again, and this repeats until either the enemies die or the player does.

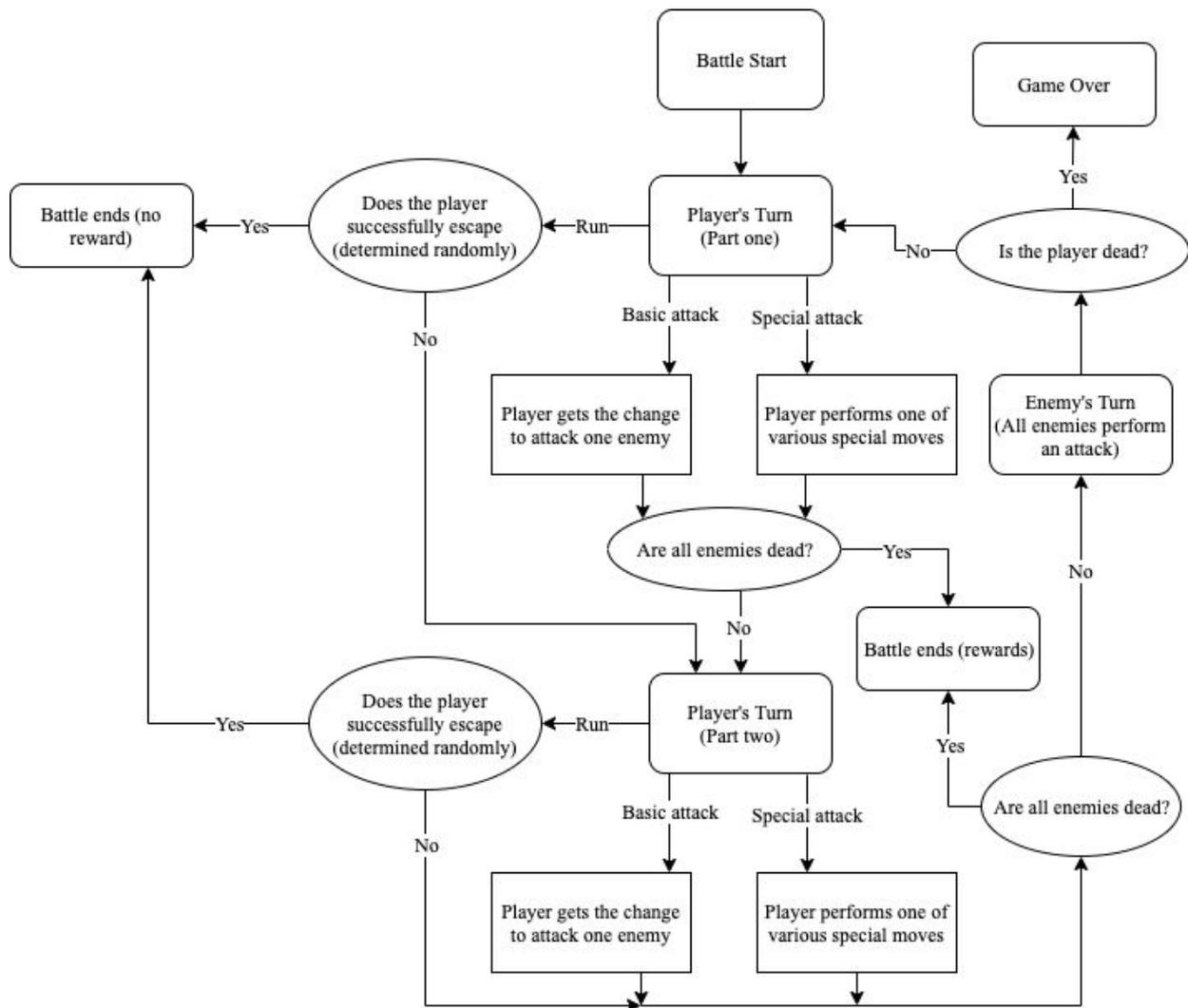


Figure 2: Battle Flow

The player controls the battles game with various buttons on the keyboard. The player makes all menu selections (including clicking through text) with the 'Z' key, and makes all deselections with the 'X' key. These two keys are also used in the enemy attacks, where 'X' is used to hit the enemy and 'Z' is used to jump beforehand. The arrow keys are also used in the battle, as the four inputs are the four directions that the player can dodge an enemy's attack.

Each chapter of the game has features that make the battle system for that specific chapter unique. While chapter one has only four available spots for the enemy to stand, chapter two introduces "offbeats" as a game mechanic. Here, instead of only being able to stand in spots "one two three four," enemies can stand in "one and two and three and four and," making for a more challenging timing. Taking place in a jazz-themed casino, chapter 3 changes the tempo slightly to account for swing. This offsets the enemies even further and offers a greater challenge. Chapter 4 eliminates one of the beats altogether. While the rest of the battles have music with 4 beats per measure, chapter 4 has only 3 beats per measure and uses a 3/4 time signature. Chapter 5 has the player dealing with a constantly shifting tempo, and chapter 6 features more arrhythmic music, to offer an even greater challenge. Besides the chapter differences, there are also status affects that can harm the player's ability to successfully time their attacks. For example, if a player is blinded by an enemy, the entire screen goes dark and their forced to rely completely on audio cues with no visual cues. If the player is deafened, the opposite is true, and the player will not be able to hear any music or sound cues, and must rely on visual cues. Finally, an item the player can purchase for use is the metronome, which adds a beat to the background of the music that can help the player successfully land attacks.

The visuals of the game were original, which was a challenge considering I am a very inexperienced artist. I did all of the sprites and backgrounds using the program “Piskel,” a simple pixel art creating program. Animations for the main player character were one of the most time consuming parts of the project, and I have over 100 different frames of animations for that character alone. Enemies also needed a fair amount of animation. Each one needed an idle animation, walking animation, an animation for each attack, and an animation for taking damage. Though they were bigger, the backgrounds were considerably easier to make than the rest of the sprites, as they were only static images that required no animation.

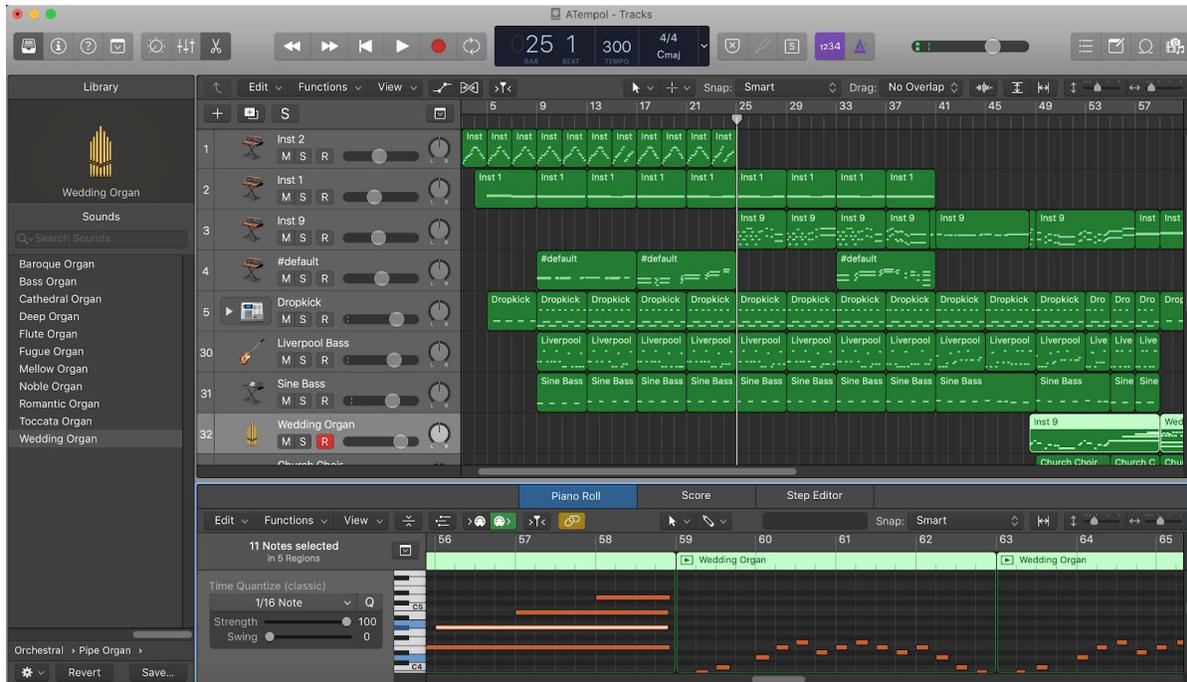


(Left) Piskel, the program used to create sprites. (Right) a spritesheet. This particular one is for the animation of the player character taking out his guitar in order to play it.

Another aspect of the game I created was the music. Since I was creating a music game, the composition of the songs in the game was an important aspect. I did the composition in *Logic Pro X*, while doing music editing in *Audacity*. I’ve written roughly 45 minutes of music for the game, 29 songs averaging around 90 seconds each. I used a mix of pre-existing software

instruments in Logic Pro, as well as some instruments I created myself, mostly instruments made to sound like 8-bit instruments. I used a synthesizer called “ESP” to create the majority of my instruments, particularly the ones made to sound like old nintendo games. I also used a synthesizer called “ES2” to create the voice synth used in a few songs, particularly in the choir piece. My composition strategy was to experiment with as many different styles of music as I could, while always keeping “old Nintendo music” as my base. Almost every song uses at least one instrument I created to sound like an old Nintendo game. Often times I’d use them at the beginning of a piece to give the impression of the song being purely retro-style, before adding in more software instruments, often a mix of synthesizers and classical instruments. Some songs only use a little bit of old Nintendo style music, while some songs (like the main battle theme) are almost completely composed of it. Some examples of styles and genres I tried to incorporate were Swing in the song “Swing Fling with King Swing,” Old West Folk in the song “Folk Ranch,” Funk in the song “Claviscious,” and Sea Shanty in “Bandoneon Bay,” and “Bend a Knee to the Bandoneon Bandits.” For a lot of songs I used tropes affiliated with the theme of the area they’re played in. For example, in the theme for the snowy town, I used a lot of bells, as well as a percussive jingle bell in the background. For the fight in a church, I had a harp playing in the background as a reference to the depiction of angels playing harps, as well as a section primarily featuring the Organ and choir, common amongst musical performances during a church sermon.

I did the bulk of my musical editing in Audacity, which mostly just involved cutting the clips of music to make sure they were exactly the correct length, as this was important for keeping the music and gameplay completely synchronized.



Logic Pro X, the program used to compose and produce music.

One of the first obstacles I had to overcome in game design was properly synchronizing time with the game's visuals and gameplay. For a rhythm game, the main gameplay centers around pressing buttons in time with the music, so keeping the rhythm constantly in synch with the gameplay was a crucial detail and, without it, the game simply wouldn't work. The entirety of the time-keeping is done within the script for the player character, as the player's movements are closely tied with the rhythm. The first attempt I made was to use Unity's built-in time function, which would return how much time had passed since the game had started, to keep

track of time. My goal was to have a variable set equal to that time, and have that reset when the music reached the end of the current measure. My first attempt seemed alright at first, but the music would quickly get out of synchronization with the rest of the game. My first thought was that the tempo could be creating problems. The main issue was that the time function was not as orderly as I needed it. I was calling it in update (a function that runs every frame), but the framerate of the game is inconsistent. With update's inconsistency, it would've been extremely difficult to successfully manage time and rhythm. However, Unity had another built-in function that would make it work: the fixed update. Unlike update, which ran every single frame, fixed update would only run every .02 seconds, giving it a consistent framerate. With fixed update, I was able to have a place where I could run my time keeping function consistently. However, even with this fix, the music was getting out of sync with the in-game actions. To correct this, I used a new variable called "biggerTime," which would act as a time reset. The music didn't desynchronize noticeably immediately, and, for a few seconds at a time, it wouldn't be noticeable at all. Because of this, my "biggerTime" fix worked. Essentially, "biggerTime" would reset itself and all of the other timekeeping tools whenever it reached a whole number.

In the end, my timeTracker function worked entirely in Fixed Update, except for a few checks during the start function, a function that only runs once during the game at the very beginning. The user of my script puts in what the tempo is through the Unity A.I, and the script handles the rest, the tempo becoming the variable "bpm." in the start function, more variables are assigned. A list of beats are determined, each one being a quarter of the bpm. For example, beat one was 60/bpm, beat two was $60/\text{bpm} * 2$, and so on. However, technically this is a misnomer,

as each beat is actually set to the value which is the end of the beat. While beat 1 technically begins at the beginning of the measure (or rather, 0 seconds after the measure begins) its set to equal the time 1/4th of the way through the measure, which is when beat 1 ends and beat 2 begins. Also in this part of the code, the variable “bigBeatTop” is assigned. This variable is the whole number that “biggerTime” must reach before it resets. “bigBeatTop” is first set equal to beat 4 (which is the full length of the measure). Then it gets put through a while loop where, until the remainder of “bigBeatTop” is 0, it keeps adding beat 4 to it. This is to ensure that “bigBeatTop” is equal to a whole number, which is needed to successfully reset time later on. In the 180 beats per minute example, “bigBeatTop” is equal to 4.

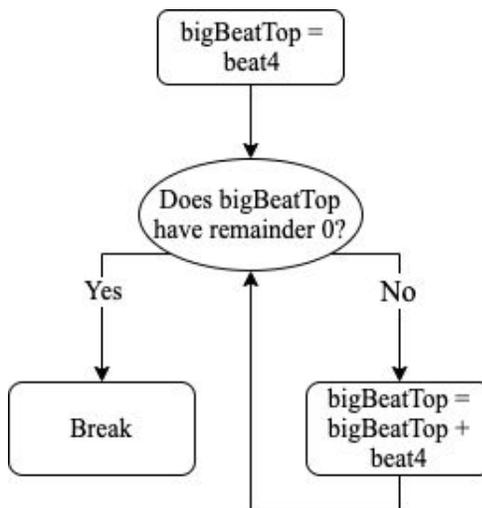


Figure 3: Determining bigBeatTop

After the start function, in fixed update, an if statement is used to check whether or not the time since the program has begun is 0, and if so, to start running both the music and the time tracker function at the same time. This is done to help keep the time function and the music in

complete synchronization. Everything else is done within the time tracker function, which is called every .02 seconds. This function manipulates 4 variables, all of which assist in keeping track of time, namely “oneBeat,” which tracks the length of one beat of music, “timeCounter,” which keeps track of the length of one measure of music, “biggerTime,” which keeps track of music up until a whole number of seconds has passed, and “segmentCounter,” which keeps track of 8 measures of the music before resetting. Unlike the other variables, “segmentCounter” doesn’t keep track of music specifically, but rather how many measures have passed before resetting after the 8th. At the beginning of the time tracker function, the first thing done is updating the first three variables. .02 is added to each of them, in accordance with the function being updated every .02 seconds. This is done with the round function, to make sure that the resulting number only has 2 decimal places. After that are a series of if statements to check to see the variables need to be reset. First “oneBeat” is checked to see if its equal to beat 1 and, if it is, its reset back to 0. Then an if statement is used to check whether “timeCounter,” is greater than or equal to “beat4” (the end of the measure) and if it is, “timeCounter” resets and segment counter is incremented. After that, “biggerTime” checks to see if it's greater than or equal to “bigBeatTop.” if it is, it resets “timeCounter” and “biggerTime” back to 0, as well as incrementing “segmentCounter”. Finally, the function checks to see if “segmentCounter” is greater than 8, and if it is, it resets it back to 1. That’s the entirety of the time tracker function, and with it, the game is able to keep time and action completely in synchronization.

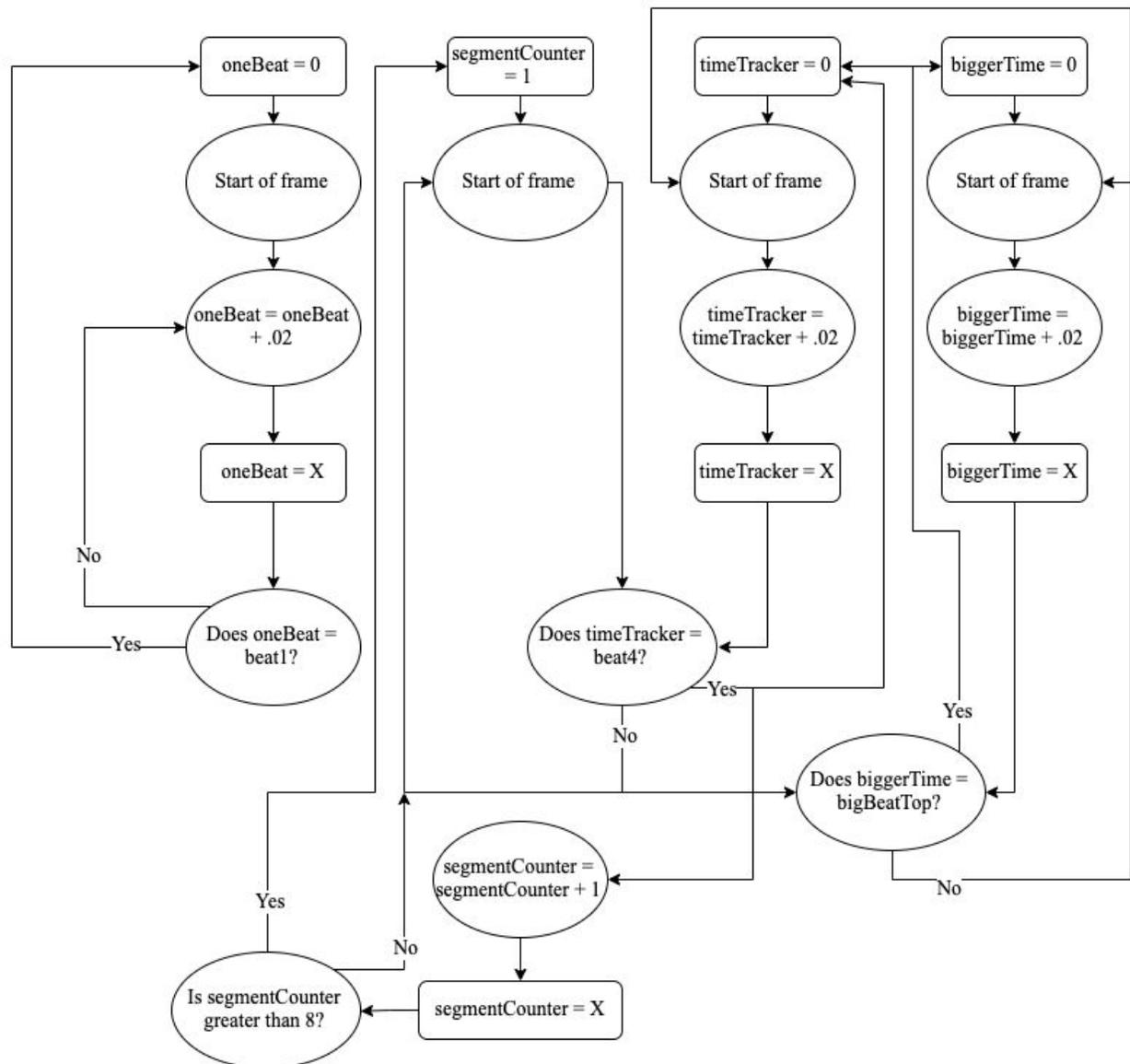


Figure 4: the TimeTracker function.

For the music aspect of the game, there were two classes I had to create first: the Horizontal Resequencer and the Vertical Remixer. The easier of the two is the vertical remixer. The vertical remixer's purpose is to control which audio layers are playing at the same time. For example, if there's a bassline in the audio and a melody line, they could both play at the same

time, or one could play while the other doesn't, or et cetera. The vertical remixer would control this, turning certain layers on and off in accordance of what's needed. This is done with a relatively simple script, starting by creating a struct called "audio layer." this contains each audio clip, a double for the clip's duration, and a float to control the starting volume of the clip. All audio layers start playing at the same time, but with differing volumes. The most important part of the script is the function "VertShift," which actually performs the vertical remixing. The function takes an audio source and a desired volume and set said audio source to whatever the desired volume is. This function helps add some life to the game and keeps the music interesting. One of the main places its used is during battles, while navigating menus. During each battle, a specific melody line plays over the rest of the music. However, in the special menu section of the menu, each menu option is a different instrument, and depending on which instrument you're highlighting, the melody line will change to a line played by the currently highlighted instrument. This exists purely for aesthetic reasons, but helps add some life to the music while doing a monotonous task such as navigating menus.

Unlike the vertical remixer, the horizontal resequencer is a more complicated class. While the vertical remixer dealt with single tracks of music on top of each other, the horizontal resequencer deals with reordering segments of music. Just like the vertical remixer, the horizontal resequencer starts by making a struct, this one called "AudioSegment," containing each audio clip, a double for the clip's duration, and an int called "nextSegment" which let's the script know which numbered segment should be played after it. The important function here is "ChangeScheduledTime," which takes two integers, each a reference to different segments in the

list of audio segments. The function changes the current segment to the audio segment the first int references, and makes the next segment the audio segment the second int references. Like the vertical remixer, the horizontal resequencer exists just to keep the music more dynamic and interesting. One example of it being used is the tutorial battle, where the music starts off as just a single woodblock keeping track of the beat, and every time the player does something successful in the battle, the music is advanced to something more interesting, usually adding more instruments along the way.

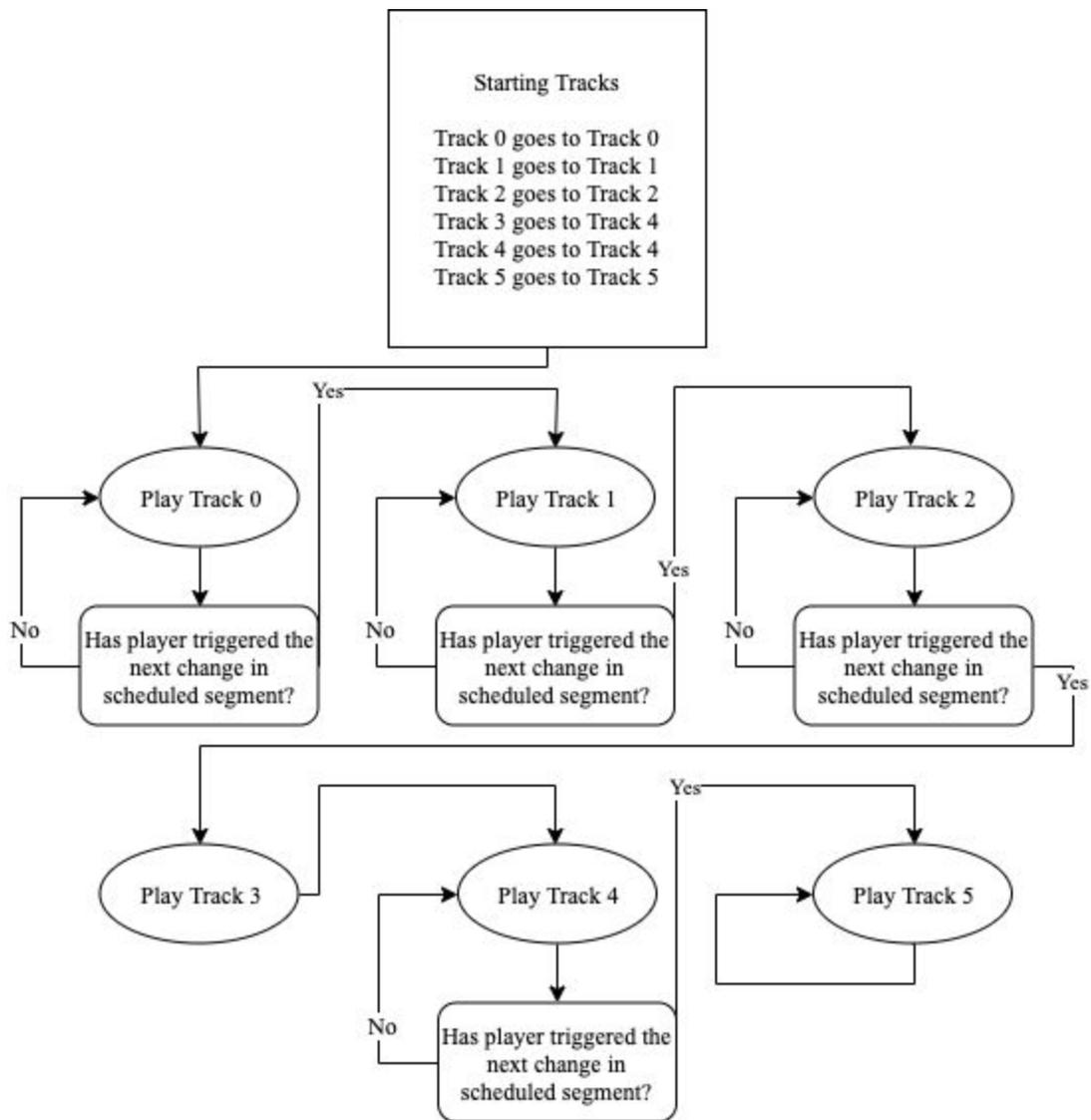


Figure 5: Horizontal Resequencing

Outside of the time tracking and music manipulation, the rest of the game's coding mainly controls object movement and in-game menu management. Most of the events that happen in battles are controlled by two scripts: BattleControl, which is contained within the game object for the menu cursor, and PlayerControl, which is the script for the player object. In

the case of the tutorial battle, however, a third script holds vital importance: the script contained inside of the “Instructor” object, the non-playable character who guides the player through the tutorial.

In the script for Battle Control, the basic structure for how the battles are run is kept, mostly through the use of state machines. Not only does it keep track of the specific menus, but also whether it's the player's turn, the enemy's turn, etc. While the Battle Control controls the basics of the battle, the Player Control controls almost everything pertaining to the player, as well as a few extras, including the time tracker. The Player Control controls where the player characters moves when, as well as what happens when the payer press any buttons while the menu isn't active, including doing damage to the enemy and performing a dodge during the enemy attack. The code for the Instructor is important because it houses the state counter that keeps the tutorial running. Since the tutorial is a scripted fight, a state machine was needed to keep everything running smoothly. The instructor code also manages a lot of when the dialogue is triggered.

Name	Description	Methods	Other classes used
CamerMover	Causes the camera to shake	CameraShake	none
PlayerContol	Controls how the player moves, and houses the time tracker function	timeTracker, AttackRun, Jump, JumpAttack, JumpBack, Fall, HitResolve, Dodge, Move,	Battle_Menu_Contr ol, ArmAnimator, CameraMover, enemyspawner, MusicControl, HorizReseq,

		PlayerTurnRed	
enemyspawner	Creates enemies for the player to fight, as well as controls their actions and attributes	enemyMove, EnemyTurnRed	PlayerControl, HorizReseq
Battle_Menu_Control	Controls the in-game battle menus	MainMenu, AttackMenu, SpecialMenu,	VertRemixer, DialogueBox, SpecialHeader, enemyspawner
MusicControl	Plays the game's music	playMusic	none
HorizReseq	Manages the game's horizontal resequencing	ChangeScheduledSegment	PlayerControl, VertRemixer,
VertRemixer	Manages the game's vertical remixing	VertShift	PlayerControl, HorizReseq
ArmAnimator	Manages Wolfgang's arms when they're a separate object from his body	none	none
Instructor_Move	Not only manages the movements of the instructor, but also controls the tutorial's dialogue	Move	HorizReseq, PlayerControl, Dummy, Battle_Menu_Control, DialogueManager,
DialogueManager	Controls the text that appears when a character is speaking	startDialogue, DisplayNextSentence, EndDialogue	MusicControl, DialogueBox, Battle_Menu_Control
Dummy	Controls the enemy used in the tutorial	Move	none
Spot	Controls the spot light that occasionally appears over objects	Darken, Lighten	Battle_Menu_Control
FollowArrow	Makes the enemy name follow the arrow while the player selects which enemy	SetEnemyNameText	none

	to attack		
SpecialHeader	Displays the text depending on what part of the special menu is selected	none	none
DialogueBox	Controls the appearance and disappearance of the dialogue box	Appear, Disappear	none
DialogueTrigger	Controls what happens if a new piece of dialogue is triggered	none	Instructor_Move, DialogueBox
SpecialAttack	Manages the player's special attacks	Fall, Rise, CheckSuccess	Battle_Menu_Control, PlayerControl,

Figure 6: Classes

Chapter V - Conclusion

Although I had to cut down on my goals, I'm still happy with the work I've done. I created a working system for a rhythm game that can be expanded upon into a fully fledged game. While I'm missing many key details, the core is present, and the most important aspects of the game are in tact. I plan on continuing work on *Rhythm Quest*, as I've only started to scratch the surface of what I believe I can create. I've developed skills that I believe will help me if I continue to pursue video game development, whether as an independent creator or as part of a larger team. I'm especially happy with the pieces of music I've created for this game to varying levels, even if there's still a few songs missing from the game's final tracklist. Overall, I'm content with what I've managed to accomplish, though there are still quite a few missing elements.

Despite this, there is still much work to be done until the game could be considered fully finished. The track list is near complete, but a few necessary songs have yet to be completed. Much of the game's story and dialogue has also yet to be written, and many enemies need sprites created for them. Most importantly, the coding for the overworld needs to be completed. While the code for the battle is almost near completion, that's only half the game. The other half is a top down overworld that brings the player from battle to battle, allows them to talk to non-playable characters (NPCs), and so forth. As far as the battle system goes, the only thing left to complete is the item system, as well as individual enemy attacks. The game ideally should have around 30-80 unique enemies, and each one would need its own set of attacks.

Aside from a few other smaller things, however, the battle portion of the game is essentially finished.

Game development was a deceptively strenuous project, but I'm happy to have gotten to the other side with a few things to show for myself. The video game industry has been growing and growing for decades, and I'm happy to have some experience in a field full of passionate creators, coders, and artists.

Chapter VI - Source Code

Source code is presented in the order it runs during the game's battle.

CameraMover

```
1. //Moves the camera in different ways
2.
3. using System.Collections;
4. using System.Collections.Generic;
5. using UnityEngine;
6.
7. public class CameraMover : MonoBehaviour {
8.
9.     //public Camera camera;
10.    public float cameraX;
11.    public float cameraY;
12.    public float cameraZ;
13.    public float cameraSize;
14.    public float shakeValueX;
15.    public float shakeValueY;
16.    public bool shakeState;
17.    public bool cameraShake;
18.    public int shakeCounter;
19.
20.    // Use this for initialization
21.    void Start () {
22.        //cameraCamera =
23.
24.    }
25.
26.    // Update is called once per frame
27.    void Update () {
28.    }
29.
30.    public void CameraShake() {
31.        if (shakeCounter < 30) {
32.            if (shakeState == false) {
33.                shakeValueX = Random.Range (-.5f, .5f);
34.                shakeValueY = Random.Range (-.5f, .5f);
35.                cameraX = cameraX + shakeValueX;
36.                cameraY = cameraY + shakeValueY;
```

```

37.         shakeState = true;
38.     } else {
39.         cameraX = cameraX - shakeValueX;
40.         cameraY = cameraY - shakeValueY;
41.         shakeValueX = 0;
42.         shakeValueY = 0;
43.         shakeState = false;
44.     }
45.
46.     transform.position = new Vector3 (cameraX, cameraY, cameraZ);
47.     shakeCounter++;
48.     //print (shakeCounter);
49.     //if (shakeCounter == 29) {
50.     //  shakeCounter = 0;
51.     //  player.cameraShake = false;
52.     //}
53. } else {
54.     shakeCounter = 0;
55.     cameraShake = false;
56. }
57. }
58. }

```

PlayerControl

```

1. //Controls most of the battle. The timer is in this script, as well as everything
   controlling the character
2.
3. using System.Collections;
4. using System.Collections.Generic;
5. using UnityEngine;
6.
7. public class PlayerControl : MonoBehaviour
8. {
9.
10.     public DialogueBox box;
11.
12.     public Battle_Menu_Control battleMenu;
13.     public float playerX, playerY, playerZ;
14.     public bool timedMovement; //If the player's movement has to be precise, this helps
   run it in 'Fixed Update'
15.     public float staticX, staticY;
16.     public Vector3 playerPos = new Vector3(0f, 0f, 0f);
17.     public ArmAnimator arms;
18.     public bool drawArms;

```

```

19.
20.     public Vector4 playerColors;
21.     public bool red, darken, lighten;
22.
23.     //for the animator:
24.     private Animator animator;
25.     public string currAnim, idleCheck;
26.
27.     //animator transition hashes
28.     int attackHash = Animator.StringToHash("Attack");
29.     int waitingHash = Animator.StringToHash("Waiting");
30.     int attackRunHash = Animator.StringToHash("AttackRun");
31.     int jumpHash = Animator.StringToHash("Jump");
32.     int attackJumpHash = Animator.StringToHash("JumpAttack");
33.     int attackLandHash = Animator.StringToHash("AttackLand");
34.     int attackJumpBackHash = Animator.StringToHash("AttackJumpBack");
35.     int jumpToStartHash = Animator.StringToHash("JumpToStart");
36.     int jumpAttackFailHash = Animator.StringToHash("JumpAttackFail");
37.     int upHash = Animator.StringToHash("Up");
38.     int downHash = Animator.StringToHash("Down");
39.     int leftHash = Animator.StringToHash("Left");
40.     int rightHash = Animator.StringToHash("Right");
41.     int guitarGrabHash = Animator.StringToHash("GuitarGrab");
42.     int guitarPlayHash = Animator.StringToHash("GuitarPlay");
43.
44.     //animator state hashes
45.     int idleHash = Animator.StringToHash("PlayerIdle");
46.
47.     public int specialSuccessCount;
48.
49.     public bool unpause;
50.     public bool attackRun;
51.     public float attackDistance;
52.
53.     public bool keyPress;
54.
55.     public CameraMover cam;
56.
57.     //info on the beats. This changes depending on the bpm
58.     public float bpm, beat1, beat2, beat3, beat4, bigbeatTop;
59.     public Vector4 beats;
60.     public float leniency; // the offset for how correct the player has to be when
        hitting a beat
61.
62.     //For the attack countdown
63.     public GameObject Countdown;

```

```

64.  public Animator countdownAnim;
65.  public float countDownX;
66.  public float countDownY;
67.  public float countDownZ;
68.  public bool displayCountdown;
69.  public int frameCounter;
70.  public float countdownVolume;
71.  public int animHash = Animator.StringToHash("AttackCountdown");
72.
73.  //For time synchronization
74.  public float oneBeat;
75.  public float timeCounter;
76.  public float biggerTime;
77.  public int segmentCounter;
78.
79.  public enemyspawner other;
80.
81.  //for music and sound effects
82.  public MusicControl music;
83.  public AudioClip countdownSound, guitarHit, guitarSwing, attackJump, thud,
    attackFail;
84.  private AudioSource sfxSource;
85.  public HorizReseq horizReseq;
86.
87.  //For the jump
88.  public float jumpRateOfChange = .06f;
89.  public bool jumpTrue, jumpDown;
90.  public float jumpFactor;
91.  public float jumpHeight = 1.4f;
92.
93.  //For the jump attack
94.  public bool jumpAttackTrue, jumpAttackUp, jumpAttackDownSuccess, jumpAttackFail;
95.  public float jumpAttackFactorX, jumpAttackFactorY;
96.  public float jumpAttackX1Deriv, jumpAttackY1Deriv; //rate of change for the jump
    attack
97.  public float jumpAttackX2Deriv, jumpAttackY2Deriv;
98.
99.  //For the jump back
100.     public bool jumpBackTrue, jumpBackDown;
101.     public float jumpBackFactor;
102.     public float jumpBackRateOfChange;
103.     public float jumpBackHeight;
104.
105.     //For a fall
106.     public bool fallTrue;
107.

```

```

108.     //For a succesful attack
109.     public int successHit;
110.     public int successPause;
111.
112.     //For a failed attack
113.     public int failHit;
114.
115.     //For the damage of the attack
116.     public int damage;
117.
118.
119.     //For dodging
120.     public bool dodgeTrue, shortDodge, dodgeSuccess;
121.     public bool enemyHit;
122.     int dodgeCounter, pauseCounter;
123.     float addX, addY, maxX, maxY;
124.     KeyCode currentKey;
125.
126.     //For the chosen enemy
127.     public float enemyPos;
128.     public float enemyX;
129.
130.     // Use this for initialization
131.     void Start()
132.     {
133.
134.         playerColors = new Vector4(1, 1, 1, 1);
135.         darken = true;
136.         lighten = false;
137.
138.
139.         dodgeSuccess = true;
140.         beat1 = (60 / bpm);
141.         beat2 = (60 / bpm) * 2;
142.         beat3 = (60 / bpm) * 3;
143.         beat4 = (60 / bpm) * 4;
144.         beats = new Vector4(beat1, beat2, beat3, beat4);
145.         bigbeatTop = beat4;
146.         while (bigbeatTop % 1 != 0) bigbeatTop = bigbeatTop + beat4;
147.         leniency = .14f;
148.
149.         box = FindObjectOfType<DialogueBox>();
150.         animator = GetComponent<Animator>();
151.         Vector3 countDownPos = new Vector3(countDownX, countDownY, countDownZ);
152.         Countdown.transform.position = countDownPos;
153.         countdownAnim = Countdown.GetComponent<Animator>();

```

```

154.         sfxSource = GetComponent();
155.         jumpAttackX1Deriv = .02f;
156.         jumpAttackY1Deriv = .02f;
157.         jumpAttackX2Deriv = .002f;
158.         jumpAttackY2Deriv = .002f;
159.         keyPress = false;
160.         jumpBackHeight = .8f;
161.         jumpBackRateOfChange = .06f;
162.
163.
164.     }
165.
166.     void FixedUpdate()
167.     {
168.         if (Time.time == 0)
169.             music.playMusic();
170.         timeTracker();
171.         if (timedMovement == true)
172.         {
173.             Move(.2f);
174.         }
175.
176.         if (oneBeat == 0 && currAnim == idleCheck)
177.         {
178.             animator.Play(idleHash, 0, 0);
179.         }
180.     }
181.
182.
183.     // Update is called once per frame
184.     void Update()
185.     {
186.         //print(battleMenu.specialStateCount);
187.         if (red == true) PlayerTurnRed();
188.
189.
190.         //gets the name of the current animation every frame for comparison to
         other animation states
191.         currAnim = animator.GetCurrentAnimatorClipInfo(0)[0].clip.ToString();
192.
193.         //Along with the short snippet of code below, this manages Wolfgang
         grabbing the guitar
194.         if (battleMenu.cutsceneCounter > 9)
195.         {
196.             animator.SetBool(guitarGrabHash, false);
197.         }

```

```

198.
199.     //During the tutorial cutscene, makes wolfgang show off his guitar
200.     if (battleMenu.cutsceneCounter == 9)
201.     {
202.         animator.SetBool(guitarGrabHash, true);
203.         battleMenu.cutsceneCounter++;
204.     }
205.
206.     //manages what the player does during the enemy portion of the battle
207.     if (other.enemyPhase == true)
208.     {
209.         dodgeSuccess = true;
210.         if (Input.GetKeyDown(KeyCode.DownArrow) ||
Input.GetKeyDown(KeyCode.UpArrow) ||
211.             Input.GetKeyDown(KeyCode.LeftArrow) ||
Input.GetKeyDown(KeyCode.RightArrow))
212.         {
213.             print("time of dodge: " + timeCounter);
214.             dodgeTrue = true;
215.         }
216.         if (enemyHit == true && dodgeSuccess == true)
217.         {
218.             if (Input.GetKey(KeyCode.LeftArrow)) dodgeSuccess = true;
219.             else dodgeSuccess = false;
220.         }
221.         if (dodgeSuccess == false)
222.         {
223.             red = true;
224.             cam.cameraShake = true;
225.             other.enemyPhase = false;
226.         }
227.     }
228.     if (dodgeTrue)
229.     {
230.         Dodge();
231.     }
232.     if (drawArms == true)
233.     {
234.         arms.transform.position = new Vector3(playerX, playerY, playerZ - .5f);
235.     }
236.     else
237.     {
238.         arms.transform.position = new Vector3(100, 100, playerZ - .5f);
239.     }
240.     animator.speed = 1f;
241.

```

```

242.         //Sets the player position from the bottom of the screen to the top of the
        screen
243.         if (playerX >= 15f && playerY < 0)
244.         {
245.             playerX = -15f;
246.             playerY = 7.5f;
247.         }
248.         else if (playerX > 16f && playerY > 0)
249.         {
250.             playerX = -11f;
251.             playerY = -0.87f;
252.         }
253.
254.
255.         //Makes the player run across the screen
256.         if ((battleMenu.cutsceneCounter == 22 || battleMenu.cutsceneCounter == 30
        || battleMenu.cutsceneCounter == 39) &&
257.             battleMenu.attackBool == true && (playerX < 13f || unpaused == true) &&
258.             attackRun == false && successHit == 0 && failHit == 0)
259.         {
260.
261.             animator.ResetTrigger(waitingHash);
262.             animator.SetTrigger(attackHash);
263.             playerX = playerX + .2f;
264.             if (unpaused == true)
265.             {
266.                 arms.transform.position = new Vector3(playerX, playerY, playerZ -
        .5f);
267.
268.             }
269.
270.             //Makes the player wait if they're on the right half of the screen
271.         }
272.         else if (playerX >= 13)
273.         {
274.             animator.ResetTrigger(attackHash);
275.             animator.SetTrigger(waitingHash);
276.             //Displays the counter on the correct beat
277.             if (timeCounter == 0.00f && unpaused == false)
278.             {
279.                 displayCountdown = true;
280.                 countdownAnim.Play(animHash, 0, 0);
281.                 sfxSource.PlayOneShot(countdownSound, countdownVolume);
282.                 //Takes the counter away on the right beat
283.             }
284.             else if (timeCounter > beat3 && displayCountdown == true)

```

```

285.         {
286.             displayCountdown = false;
287.             unpause = true;
288.             attackRun = true;
289.             animator.ResetTrigger(waitingHash);
290.             animator.SetTrigger(attackRunHash);
291.             drawArms = true;
292.             staticX = -9f;
293.             staticY = 7.45f;
294.         }
295.     }
296.
297.     //print(battleMenu.specialAttackStart);
298.     if (battleMenu.specialStateCount == 1)
299.     {
300.         horizReseq.ChangeScheduledSegment(8, 7);
301.         animator.SetInteger(guitarPlayHash, 1);
302.     }
303.     //Makes the player wait until the end of the beat to begin the special
    attack
304.     if (battleMenu.specialStateCount == 1 && segmentCounter == 4 && timeCounter
    == 0.00f)
305.     {
306.         displayCountdown = true;
307.         countdownAnim.Play(animHash, 0, 0);
308.         sfxSource.PlayOneShot(countdownSound, countdownVolume);
309.     }
310.     else if (timeCounter > beat3 && displayCountdown == true)
311.     {
312.         displayCountdown = false;
313.         battleMenu.specialStateCount = 2;
314.     }
315.     if (battleMenu.specialStateCount == 2 && timeCounter == 0.00f)
316.     {
317.         battleMenu.specialStateCount = 3;
318.         battleMenu.vert.VertShift(battleMenu.vert.audioSources[0], false);
319.         animator.SetInteger(guitarPlayHash, 2);
320.     }
321.     if (battleMenu.specialStateCount == 12)
322.     {
323.         battleMenu.vert.VertShift(battleMenu.vert.audioSources[0], true);
324.         animator.SetInteger(guitarPlayHash, 0);
325.         battleMenu.specialStateCount++;
326.     }
327.
328.     transform.position = new Vector3(playerX, playerY, playerZ);

```

```

329.         if (cam.cameraShake == true) cam.CameraShake();
330.         //print(Time.time % (1f+(1f/3f)));
331.
332.         //Manages the location of the countdown
333.         if (displayCountdown == true)
334.         {
335.             countDownX = 0f;
336.             countDownY = 4;
337.         }
338.         else
339.         {
340.             countDownX = -100;
341.         }
342.         Vector3 countDownPos = new Vector3(countDownX, countDownY, countDownZ);
343.         Countdown.transform.position = countDownPos;
344.         if (attackRun == true) AttackRun();
345.         else if (successHit > 0 || failHit > 0) HitResolve();
346.
347.     }
348.
349.
350.     //keeps track of the beat and rhythm, dependant on how much time has passed
351.     public void timeTracker()
352.     {
353.         biggerTime = Mathf.Round((biggerTime + .02f) * 100f) / 100f;
354.         timeCounter = Mathf.Round((timeCounter + .02f) * 100f) / 100f;
355.         oneBeat = Mathf.Round((oneBeat + .02f) * 100f) / 100f;
356.         if (oneBeat == beat1) oneBeat = 0;
357.         if (timeCounter == 0 || timeCounter == beat4)
358.         {
359.             timeCounter = 0;
360.             segmentCounter++;
361.         }
362.         if (biggerTime >= bigbeatTop)
363.         {
364.             timeCounter = 0;
365.             biggerTime = 0;
366.         }
367.         if (segmentCounter > 8)
368.         {
369.             segmentCounter = 1;
370.         }
371.         //print (oneBeat+" "+timeCounter+" "+biggerTime+" "+segmentCounter);
372.     }
373.
374.

```

```

375.
376.
377.
378.     //Manages the running part of the attack
379.     void AttackRun()
380.     {
381.         enemyX = (other.GetEnemyStats(other.enemyList[battleMenu.enemyChoice])[0]);
382.
383.
384.         //Determines the current enemies position
385.         if (enemyX == battleMenu.enemyPlaces[0]) enemyPos = 1;
386.         else if (enemyX == battleMenu.enemyPlaces[1]) enemyPos = 2;
387.         else if (enemyX == battleMenu.enemyPlaces[2]) enemyPos = 3;
388.         else if (enemyX == battleMenu.enemyPlaces[3]) enemyPos = 4;
389.
390.         if (Input.GetKey(KeyCode.Z) && jumpTrue == false && jumpAttackTrue == false
&& battleMenu.cutsceneCounter > 10)
391.         {
392.             timedMovement = false;
393.             if ((enemyPos == 2 && (timeCounter > beat4 - leniency || timeCounter <
0 + leniency)) ||
394.                 (enemyPos == 3 && (timeCounter > beat1 - leniency && timeCounter <
beat1 + leniency)) ||
395.                 (enemyPos == 4 && (timeCounter > beat2 - leniency && timeCounter <
beat2 + leniency)) ||
396.                 (enemyPos == 1 && (timeCounter > beat3 - leniency && timeCounter <
beat3 + leniency)))
397.             {
398.                 sfxSource.PlayOneShot(attackJump, .5f);
399.                 damage++;
400.                 jumpAttackTrue = true;
401.                 jumpAttackX1Deriv = jumpAttackX1Deriv * (((5 -
battleMenu.enemyChoice) + 1) - .5f);
402.                 jumpAttackUp = true;
403.                 animator.SetTrigger(attackJumpHash);
404.                 animator.ResetTrigger(attackRunHash);
405.                 drawArms = false;
406.                 arms.transform.position = new Vector3(-100, -100, -10);
407.             }
408.             else
409.             {
410.                 drawArms = false;
411.                 attackRun = false;
412.                 sfxSource.PlayOneShot(attackFail, 2f);
413.                 failHit++;
414.             }

```

```

415.     }
416.     else if (Input.GetKey(KeyCode.X))
417.     {
418.         timedMovement = false;
419.         sfxSource.PlayOneShot(guitarSwing, 1f);
420.         print("time of hit: " + timeCounter);
421.         //If the hit is succesful...
422.         if ((enemyPos == 1 && (timeCounter > beat4 - leniency || timeCounter <
0 + leniency)) ||
423.             (enemyPos == 2 && (timeCounter > beat1 - leniency && timeCounter <
beat1 + leniency)) ||
424.             (enemyPos == 3 && (timeCounter > beat2 - leniency && timeCounter <
beat2 + leniency)) ||
425.             (enemyPos == 4 && (timeCounter > beat3 - leniency && timeCounter <
beat3 + leniency)))
426.         {
427.             sfxSource.PlayOneShot(guitarHit, 1f);
428.             animator.ResetTrigger(attackRunHash);
429.             damage++;
430.             other.hitEnemy(other.enemyList[battleMenu.enemyChoice], damage);
431.             damage = 0;
432.             cam.cameraShake = true;
433.             attackRun = false;
434.             successHit++;
435.             if (jumpAttackTrue == true)
436.             {
437.                 jumpAttackDownSuccess = true;
438.                 animator.SetBool(attackJumpHash, false);
439.                 animator.SetBool(attackLandHash, true);
440.             }
441.
442.         }
443.         else
444.         {
445.             if (jumpAttackTrue == true) jumpAttackFail = true;
446.             else
447.             {
448.                 drawArms = false;
449.                 attackRun = false;
450.                 sfxSource.PlayOneShot(attackFail, 2f);
451.                 failHit++;
452.             }
453.         }
454.     }
455.

```

```

456.     if (((enemyPos == 1 && (timeCounter > 0 + leniency && timeCounter <
      beat1)) ||
457.         (enemyPos == 2 && timeCounter > beat1 + leniency) ||
458.         (enemyPos == 3 && timeCounter > beat2 + leniency) ||
459.         (enemyPos == 4 && timeCounter > beat3 + leniency)) ||
460.         jumpAttackFail == true) && jumpAttackTrue == true)
461.     {
462.         jumpAttackFail = false;
463.         timedMovement = false;
464.         jumpAttackTrue = false;
465.         attackRun = false;
466.         drawArms = false;
467.         sfxSource.PlayOneShot(attackFail, 2f);
468.         jumpAttackFactorX = 0f;
469.         jumpAttackFactorY = 0f;
470.         jumpAttackX1Deriv = .02f;
471.         jumpAttackY1Deriv = .02f;
472.         failHit++;
473.     }
474.     if (playerX > enemyX + .5f && playerY > 0)
475.     {
476.         timedMovement = false;
477.         drawArms = false;
478.         attackRun = false;
479.         sfxSource.PlayOneShot(attackFail, 2f);
480.         failHit++;
481.     }
482.     if (jumpAttackTrue) JumpAttack();
483.     else
484.     {
485.         //Update the player's location
486.         timedMovement = true;
487.         //playerX = playerX + .2f;
488.         playerY = playerY + jumpFactor;
489.     }
490. }
491.
492.
493.
494.
495.
496. //What happens if the player jumps
497. void Jump()
498. {
499.     //turns jumpDown on if the difference between the Y before the jump - the Y
      after the jump is of a certain height

```

```

500.         if (playerY - staticY > jumpHeight) jumpDown = true;
501.         //If the player's current Y is back to the same level as the original Y
           before the jump, reset everything and stop jumping
502.         if (playerY < staticY + .04f && jumpDown == true)
503.         {
504.             jumpDown = false; //Stop jumping down
505.             animator.ResetTrigger(jumpHash); //Bring animation back to a run
506.             jumpFactor = 0f; //Reset the rate of change
507.             jumpTrue = false; //Get out of this small jump function
508.         }
509.         //Change the value by which Y is increased
510.         if (jumpDown == true)
511.         {
512.             jumpFactor = jumpFactor - jumpRateOfChange;
513.         }
514.         else if (jumpTrue == true)
515.         {
516.             jumpFactor = jumpFactor + jumpRateOfChange;
517.         }
518.     }
519.
520.
521.
522.
523.
524.     //What happens if the player jumps at the right time for an attack
525.     void JumpAttack()
526.     {
527.
528.
529.         playerX = playerX + jumpAttackFactorX;
530.         playerY = playerY + jumpAttackFactorY;
531.
532.         if (playerY - staticY > jumpHeight && jumpAttackUp == true)
533.         {
534.             jumpAttackUp = false;
535.             jumpAttackFactorX = 0;
536.             jumpAttackFactorY = 0;
537.         }
538.         if (jumpAttackUp == true)
539.         {
540.             jumpAttackFactorX = jumpAttackFactorX + jumpAttackX1Deriv;
541.             jumpAttackFactorY = jumpAttackFactorY + jumpAttackY1Deriv;
542.
543.             jumpAttackX1Deriv = jumpAttackX1Deriv - jumpAttackX2Deriv;
544.             jumpAttackY1Deriv = jumpAttackY1Deriv + jumpAttackY2Deriv;

```

```

545.         //print (playerX + " " + playerY + " " + jumpAttackFactorX + " " +
           jumpAttackFactorY + " " + jumpAttackX1Deriv + " " + jumpAttackY1Deriv);
546.     }
547.     else
548.     {
549.         if (jumpAttackDownSuccess == true)
550.         {
551.             animator.ResetTrigger(attackJumpHash);
552.             animator.SetTrigger(attackLandHash);
553.             jumpAttackFactorX = .3f;
554.             jumpAttackFactorY = -.3f;
555.             if (playerY < staticY + .04 && jumpAttackDownSuccess == true)
556.             {
557.                 jumpAttackDownSuccess = false; //Stop jumping down
558.                 animator.ResetTrigger(attackLandHash); //Bring animation back
           to a run
559.                 jumpAttackFactorX = 0f; //Reset the rate of change
560.                 jumpAttackFactorY = 0f;
561.                 jumpAttackX1Deriv = .02f;
562.                 jumpAttackY1Deriv = .02f;
563.                 jumpAttackTrue = false; //Get out of this small jump function
564.                 jumpTrue = false;
565.             }
566.         }
567.     }
568. }
569.
570. void JumpBack()
571. {
572.     playerX = playerX - .15f;
573.     playerY = playerY + jumpBackFactor;
574.     //turns jumpDown on if the difference between the Y before the jump - the Y
           after the jump is of a certain height
575.     if (playerY - staticY > jumpBackHeight) jumpBackDown = true;
576.     //If the player's current Y is back to the same level as the original Y
           before the jump, reset everything and stop jumping
577.     if (((playerY < staticY + .04 && jumpBackDown == true && successHit < 3))
           || ((playerY < .5f) && jumpBackDown == true))
578.     {
579.         jumpBackDown = false; //Stop jumping down
580.         animator.ResetTrigger(attackJumpBackHash); //Bring animation back to a
           run
581.         jumpBackFactor = 0f; //Reset the rate of change
582.         jumpBackTrue = false; //Stop running this function
583.         successHit++;
584.         if (playerY < 1)

```

```
585.         {
586.             playerX = -7.75f;
587.             playerY = -0.87f;
588.             playerZ = -3f;
589.         }
590.     }
591.     //Change the value by which Y is increased
592.     if (jumpBackDown == true)
593.     {
594.         jumpBackFactor = jumpBackFactor - jumpBackRateOfChange;
595.     }
596.     else if (jumpBackTrue == true)
597.     {
598.         jumpBackFactor = jumpBackFactor + jumpBackRateOfChange;
599.     }
600.     if (playerY > 10)
601.     {
602.         playerZ = -.2f;
603.     }
604. }
605.
606.
607.
608.
609. void Fall()
610. {
611.     if (fallTrue)
612.     {
613.         playerY = playerY - .2f;
614.     }
615.     if (playerY < (staticY + .04f))
616.     {
617.         fallTrue = false;
618.         sfxSource.PlayOneShot(thud, .5f);
619.         failHit++;
620.     }
621. }
622.
623. void HitResolve()
624. {
625.     timedMovement = false;
626.     if (jumpAttackTrue) JumpAttack();
627.     else if (successHit == 1)
628.     {
629.         jumpBackHeight = .8f;
630.         jumpBackRateOfChange = .06f;
```

```
631.         animator.ResetTrigger(attackLandHash);
632.         drawArms = false;
633.         animator.SetTrigger(attackJumpBackHash);
634.         jumpBackTrue = true;
635.     }
636.     if (jumpBackTrue)
637.     {
638.         JumpBack();
639.     }
640.     else if (successHit == 2)
641.     {
642.         successPause++;
643.         if (successPause == 10)
644.         {
645.             successHit++;
646.             sfxSource.PlayOneShot(attackJump, .5f);
647.         }
648.     }
649.     else if (successHit == 3)
650.     {
651.         successPause = 0;
652.         animator.SetTrigger(jumpToStartHash);
653.         jumpBackRateOfChange = .1f;
654.         jumpBackHeight = 5;
655.         jumpBackTrue = true;
656.         animator.ResetTrigger(attackHash);
657.     }
658.     else if (successHit == 4)
659.     {
660.         animator.ResetTrigger(jumpToStartHash);
661.         battleMenu.attackBool = false;
662.         battleMenu.menuBool = true;
663.         battleMenu.turnCounter++;
664.         successHit = 0;
665.         unpause = false;
666.         if (battleMenu.cutscene == true) battleMenu.cutsceneCounter++;
667.     }
668.
669.     if (failHit == 1)
670.     {
671.         animator.ResetTrigger(attackJumpHash);
672.         animator.SetTrigger(jumpAttackFailHash);
673.         fallTrue = true;
674.     }
675.     if (fallTrue)
676.         Fall();
```

```
677.         else if (failHit == 2)
678.         {
679.             successHit = 1;
680.             failHit = 0;
681.             animator.ResetTrigger(jumpAttackFailHash);
682.         }
683.     }
684.
685.     public void Dodge()
686.     {
687.
688.         if (Input.GetKeyDown(KeyCode.LeftArrow) && dodgeCounter == 0)
689.         {
690.             addX = -.5f;
691.             maxX = playerX - 1.5f;
692.             animator.SetTrigger(leftHash);
693.             currentKey = KeyCode.LeftArrow;
694.         }
695.         else if (Input.GetKeyDown(KeyCode.DownArrow) && dodgeCounter == 0)
696.         {
697.             animator.SetTrigger(downHash);
698.             currentKey = KeyCode.DownArrow;
699.         }
700.         else if (Input.GetKeyDown(KeyCode.RightArrow) && dodgeCounter == 0)
701.         {
702.             addX = .5f;
703.             maxX = playerX + 1.5f;
704.             animator.SetTrigger(rightHash);
705.             currentKey = KeyCode.RightArrow;
706.         }
707.         else if (Input.GetKeyDown(KeyCode.UpArrow) && dodgeCounter == 0)
708.         {
709.             currentKey = KeyCode.UpArrow;
710.             addY = .60f;
711.             maxY = playerY + 1.8f;
712.             animator.SetTrigger(upHash);
713.         }
714.
715.
716.
717.
718.         if (dodgeCounter >= 5)
719.         {
720.             if (Input.GetKey(currentKey) && pauseCounter != 12)
721.             {
722.                 dodgeCounter--;
```

```
723.         pauseCounter++;
724.     }
725.     else
726.     {
727.         playerX = playerX - addX;
728.         playerY = playerY - addY;
729.     }
730. }
731. else if (playerX != maxX && playerY <= maxY)
732. {
733.     playerX = playerX + addX;
734.     playerY = playerY + addY;
735. }
736. if (dodgeCounter == 7)
737. {
738.     dodgeCounter = 0;
739.     playerY = -0.82f;
740.     animator.ResetTrigger(downHash);
741.     animator.ResetTrigger(upHash);
742.     animator.ResetTrigger(leftHash);
743.     animator.ResetTrigger(rightHash);
744.     addX = 0f;
745.     addY = 0f;
746.     maxX = 0f;
747.     maxY = 0f;
748.     dodgeTrue = false;
749.     pauseCounter = 0;
750. }
751. if (dodgeTrue == true)
752. {
753.     dodgeCounter++;
754. }
755.
756. }
757.
758. public void Move(float speed)
759. {
760.     playerX = playerX + speed;
761. }
762.
763. public void PlayerTurnRed()
764. {
765.     SpriteRenderer spriteRenderer = this.GetComponent<SpriteRenderer>();
766.     if (spriteRenderer.color[1] > 0 && darken == true)
767.     {
768.         playerColors[1] = playerColors[1] - .2f;
```

```

769.         playerColors[2] = playerColors[2] - .2f;
770.     }
771.     else if (spriteRenderer.color[1] < 1 && lighten == true)
772.     {
773.         playerColors[1] = playerColors[1] + .2f;
774.         playerColors[2] = playerColors[2] + .2f;
775.     }
776.     if (darken == true && spriteRenderer.color[1] < .1f)
777.     {
778.         darken = false;
779.         lighten = true;
780.     }
781.     else if (lighten == true && spriteRenderer.color[1] > .9f)
782.     {
783.         lighten = false;
784.         darken = true;
785.         red = false;
786.     }
787.     spriteRenderer.color = playerColors;
788.
789. }
790.
791. }

```

enemyspawner

```

1. //Script controls all of the enemies in the battle. First it spawns them,
2. //and then controls their individual actions on their turn. It also manages their
   placement, turn order, etc
3.
4. using System.Collections;
5. using System.Collections.Generic;
6. using UnityEngine;
7.
8. public class enemyspawner : MonoBehaviour {
9.
10.     public GameObject enemy01;
11.     public GameObject enemy02;
12.     public GameObject currentEnemy;
13.     public Transform enemyParent;
14.     public PlayerControl player;
15.     public float test = 6.3f;
16.     public List<GameObject> enemyList = new List<GameObject>();
17.     public List<string> enemyNameList = new List<string> ();
18.     public List<string> tempEnemyNameList = new List<string> ();

```

```

19. public List<float> startingX = new List<float> ();
20. public bool enemyPhase;
21. public int enemyTurn;
22. public int attackPhase;
23. public int randomEnemyType;
24. public int randomEnemy;
25. public int enemiesGone;
26. public Vector4 enemyColors;
27. public bool red, darken, lighten;
28. EnemyAnimate enemyAnimate;
29.
30. public bool specialBattle; //Determines whether enemies are spawned in the typical
way
31. public GameObject specialEnemy;
32.
33. public int currEnemy;
34.
35. //number of enemies
36. public int enemyCount;
37.
38. //for enemy attack
39. public bool run;
40.
41. //for the enemy attack's hit time
42. public float enemyHit;
43.
44. //for individual enemies
45. public float enemyX, enemyY, enemyZ;
46.
47. //for triggering enemy music
48. public HorizReseq horizReseq;
49.
50. // Use this for initialization
51. public void Start () {
52.     enemyColors = new Vector4(1, 1, 1, 1);
53.     darken = true;
54.     lighten = false;
55.     if (specialBattle == false) {
56.         List<GameObject> enemyBase = new List<GameObject>();
57.         enemyBase.Add(enemy01);
58.         enemyBase.Add(enemy02);
59.         int numberOfEnemies = UnityEngine.Random.Range(1, enemyCount);
60.
61.         for (int i = 0; i < numberOfEnemies; i++)
62.             {
63.                 int randomEnemyType = UnityEngine.Random.Range(0, enemyBase.Count);

```

```

64.         GameObject addedEnemy = enemyBase[randomEnemyType];
65.         string enemyName = addedEnemy.name;
66.         float newX = (6.5f - (float)(i - 1) * 3f);
67.         enemyList.Add(
68.             Instantiate(
69.                 addedEnemy,
70.                 new Vector3(
71.                     newX,
72.                     7.5f,
73.                     -1),
74.                 Quaternion.identity,
75.                 enemyParent)
76.             );
77.         enemyList[i].name = enemyName;
78.         enemyNameList.Add(enemyList[i].name);
79.     }
80. }
81.
82. else {
83.     enemyList.Add(specialEnemy);
84.     enemyNameList.Add(specialEnemy.name);
85. }
86.
87. }
88.
89. // Update is called once per frame
90. void Update () {
91.     if (red == true) EnemyTurnRed(specialEnemy);
92.     if (enemyPhase == true && specialBattle == false) {
93.         enemyAttack (currentEnemy);
94.     }
95. }
96.
97.
98.
99.
100. //Return a list of the enemies stats. [Xpos, Ypos]
101. public List<float> GetEnemyStats (GameObject enemy) {
102.     List<float> enemyStats = new List<float>();
103.     enemyStats.Add (enemy.transform.position.x);
104.     enemyStats.Add (enemy.transform.position.y);
105.     return(enemyStats);
106. }
107.
108. public void hitEnemy (GameObject enemy,int damage) {
109.     //EnemyAnimate enemyAnimate = enemy.GetComponent<EnemyAnimate>();

```

```

110.         //enemyAnimate.hit = true;
111.         //enemyAnimate.takeDamage (damage);
112.         red = true;
113.
114.     }
115.
116.     public void enemyAttack (GameObject enemy) {
117.         enemyAnimate = enemy.GetComponent<EnemyAnimate>();
118.         if (attackPhase == 1) {
119.             enemyAnimate.attack = true;
120.             if (player.segmentCounter == 8) {
121.                 run = true;
122.             }
123.             if (run == true) {
124.                 enemyMove (enemy);
125.                 enemyAnimate.walk = true;
126.             }
127.         } else if (attackPhase == 2) {
128.             float enemyX = GetEnemyStats(enemy)[0];
129.             float enemyY = GetEnemyStats(enemy)[1];
130.             enemyAnimate.attack2(enemyX, enemyY);
131.         }
132.     }
133.
134.     public void enemyMove (GameObject enemy) {
135.         enemyAnimate = enemy.GetComponent<EnemyAnimate>();
136.         float enemyX = GetEnemyStats(enemy)[0];
137.         float enemyY = GetEnemyStats(enemy)[1];
138.
139.         //Making the enemy loop around on the screen, from the top to the bottom
140.         if (enemyX < -12.1f && enemyY > 3) {
141.             enemyX = 12f;
142.             enemyY = -0.87f;
143.         } else if (enemyX > 12.1f && enemyY < 3) {
144.             enemyX = -12f;
145.             enemyY = 7.5f;
146.
147.         } else if (enemyAnimate.attack == true) {
148.             if (enemyX <= -9 && player.segmentCounter != 8) {
149.                 horizReseq.ChangeScheduledSegment (6, horizReseq.nextSegment);
150.                 run = false;
151.             } else if (player.segmentCounter == 8) {
152.                 enemyX = enemyX - .08f;
153.             } else if (player.segmentCounter == 1) {
154.                 run = false;
155.                 attackPhase = 2;

```

```

156.         } else {
157.             enemyX = enemyX - .2f;
158.         }
159.     } else {
160.         if (enemyX >= enemyAnimate.startingX && enemyY > 3) {
161.             enemyAnimate.goBack = false;
162.         } else {
163.             enemyX = enemyX + .6f;
164.         }
165.     }
166.     enemy.transform.position = new Vector3 (enemyX, enemyY, -1);
167. }
168.
169. public void EnemyTurnRed(GameObject enemy){
170.     SpriteRenderer spriteRenderer = enemy.GetComponent<SpriteRenderer>();
171.     if (spriteRenderer.color[1] > 0 && darken == true){
172.         enemyColors[1] = enemyColors[1] - .2f;
173.         enemyColors[2] = enemyColors[2] - .2f;
174.     }
175.     else if (spriteRenderer.color[1] < 1 && lighten == true){
176.         enemyColors[1] = enemyColors[1] + .2f;
177.         enemyColors[2] = enemyColors[2] + .2f;
178.     }
179.     if (darken == true && spriteRenderer.color[1] < .1f) {
180.         darken = false;
181.         lighten = true;
182.     }
183.     else if (lighten == true && spriteRenderer.color[1] > .9f){
184.         lighten = false;
185.         darken = true;
186.         red = false;
187.     }
188.     spriteRenderer.color = enemyColors;
189. }
190. }
191. }

```

Battle_Menu_Control

```

1. //This controls the menu and the flow of the battle. its essentially the backbone of
   the battle
2.
3. using System.Collections;

```

```
4. using System.Collections.Generic;
5. using UnityEngine;
6. using UnityEngine.UI;
7.
8. public class Battle_Menu_Control : MonoBehaviour {
9.
10.     //The following variables determine where in the series of menus the player is
11.     public bool mainMenuBool;
12.     public bool menuBool;
13.     public int attackMenu;
14.     public bool attackBool;
15.     public int specialStateCount;
16.
17.     public VertRemixer vert;
18.
19.     public bool cutscene; //Controls when the battle starts
20.
21.     public int itemMenu;
22.     public int specialMenu;
23.     public int otherMenu;
24.
25.     public float turnCounter;
26.
27.     //For scripted battles, this controls what scripted actions happen when
28.     public int cutsceneCounter;
29.     public int lastCutsceneCounter;
30.     public DialogueBox box; //This turns off player control when the dialogue box is on
    screen
31.
32.     //For the main menu
33.     public GameObject menu;
34.     int menuNum = 0;
35.     public int mainMenuItems = 4;
36.     public Vector3 menuPos;
37.     public float menuAttackY;
38.     public float menuItemY;
39.     public float menuSpecialY;
40.     public float menuOtherY;
41.
42.     //For the special menu
43.     public GameObject special;
44.     public Vector3 specialPos;
45.     public SpecialHeader specialHeader;
46.
47.
48.     //For the pointer
```

```

49.     public Vector3 arrowPos;
50.     public AudioClip sfxClip, menuSelect, menuCancel;
51.     private AudioSource sfxSource;
52.     public float audioVolume;
53.     public int arrowLocation; //What number enemy the arrow is pointing at
54.     public enemyspawner spawn;
55.
56.     //For the special menu curser
57.     public GameObject curser;
58.     public Vector3 curserPos;
59.     public int curserLocation;
60.
61.     //For the enemy name paper
62.     public FollowArrow enemyName;
63.     public EnemyTextPosition enemyText;
64.     public float yOffset = 2.5f;
65.     float enemyNameX;
66.     float enemyNameY;
67.     public int numberOfEnemies;
68.     public int attackMenuCounter;
69.
70.     //Enemy positions
71.     public Vector4 enemyPlaces;
72.
73.     public bool activate;
74.
75.     public int enemyChoice;
76.
77.     // Use this for initialization
78.     void Start () {
79.         enemyNameX = -20;
80.         enemyNameY = -20;
81.         box = FindObjectOfType<DialogueBox>();
82.         menuPos = new Vector3(-100, -100, menu.transform.position[2]);
83.         specialPos = new Vector3(30, -2, special.transform.position[2]);
84.         arrowPos = new Vector3(-100, -100, this.transform.position[2]);
85.         curserPos = new Vector3(100, 0, 0);
86.         arrowLocation = 1;
87.         cutsceneCounter = 0;
88.         sfxSource = GetComponent<AudioSource> ();
89.
90.
91.
92.
93.     }
94.

```

```

95.
96. // Update is called once per frame
97. void Update () {
98.     //print(cutsceneCounter);
99.     //print(specialStateCount);
100.     menu.transform.position = menuPos;
101.     special.transform.position = specialPos;
102.     this.transform.position = arrowPos;
103.     curser.transform.position = curserPos;
104.     if (mainMenuBool == true) MainMenu();
105.     if (box.dialogueSession == false && (cutsceneCounter == 19 ||
cutsceneCounter == 29 ||
106.         cutsceneCounter == 38 ||
cutsceneCounter == 78)
107.         && menuBool == true) mainMenuBool = true;
108.     if (attackMenu > 0) AttackMenu();
109.     else if (specialMenu > 0) SpecialMenu();
110.
111.     if(specialStateCount == 14){
112.         specialStateCount = 0;
113.         menuBool = true;
114.         cutsceneCounter++;
115.     }
116.
117.     if (cutscene == false)
118.     {
119.
120.         List<int> menuNums = new List<int>();
121.         menuNums.Add(attackMenu); menuNums.Add(itemMenu);
122.         menuNums.Add(specialMenu); menuNums.Add(otherMenu);
123.
124.         //Starts the enemy attack phase of the battle
125.         if (turnCounter >= 2 && activate == true)
126.         {
127.             spawn.enemyPhase = true;
128.             spawn.currentEnemy = spawn.enemyList[0];
129.             spawn.run = true;
130.             activate = false;
131.         }
132.
133.         if (turnCounter >= 2 && spawn.currEnemy > spawn.enemyNameList.Count)
134.         {
135.             spawn.currEnemy = 0;
136.             spawn.attackPhase = 1;
137.             turnCounter = 0;
138.             attackMenu = 0;

```

```
138.         arrowLocation = 1;
139.         menuNum = 0;
140.         spawn.enemyPhase = false;
141.         activate = true;
142.     }
143.
144.
145.     if (menuNums[0] > 0 || cutscene == true) mainMenuBool = false;
146.     else if (menuBool == true && turnCounter < 2) mainMenuBool = true;
147.     else if (menuNums[0] > 0) AttackMenu();
148.
149.     //Move the enemy paper to the correct spot
150.     Vector3 enemyNamePos = new Vector3(enemyNameX, enemyNameY,
enemyName.transform.position[2]);
151.     enemyName.transform.position = enemyNamePos;
152.
153.     //Move the enemy text to the right spot
154.     Vector3 enemyNameTextPos = new Vector3(enemyNameX + .8f, enemyNameY +
-.2f, -9);
155.     enemyText.transform.position = enemyNameTextPos;
156.
157.     //Create the arrow
158.     transform.position = arrowPos;
159. }
160. //how the battle menu should function in the case of a cutscene
161. else {
162.
163.     Vector3 enemyNamePos = new Vector3(enemyNameX, enemyNameY,
enemyName.transform.position[2]);
164.     enemyName.transform.position = enemyNamePos;
165.
166.     //during part 12, the menu finally appears
167.     if (cutsceneCounter == 12 || cutsceneCounter == 71)
168.     {
169.         MainMenu();
170.         arrowPos[1] = 3.5f;
171.         cutsceneCounter++;
172.     }
173.
174.
175. }
176.
177.
178.
179.
180.
```

```

181.     }
182.
183.     void MainMenu()
184.     {
185.         specialPos[0] = 100;
186.         menuPos[0] = 0;
187.         menuPos[1] = 0;
188.         arrowPos[0] = -5.75f;
189.         enemyNameX = -20;
190.         enemyNameY = -20;
191.
192.         if (box.dialogueSession == false)
193.         {
194.             if (Input.GetKeyDown(KeyCode.UpArrow) && menuNum > 0)
195.             {
196.                 menuNum--;
197.                 sfxSource.PlayOneShot(sfxClip, audioVolume);
198.             }
199.             else if (Input.GetKeyDown(KeyCode.DownArrow) && menuNum < mainMenuItems
- 1)
200.             {
201.                 menuNum++;
202.                 sfxSource.PlayOneShot(sfxClip, audioVolume);
203.             }
204.             else if (Input.GetKeyDown(KeyCode.Z))
205.             {
206.                 if (menuNum == 0) attackMenu++;
207.                 else if (menuNum == 2)
208.                 {
209.                     specialMenu = 1;
210.                     specialPos[0] = 7;
211.                     curserLocation = 1;
212.                     curserPos[0] = specialPos[0] - 2;
213.                     curserPos[1] = specialPos[1] + 1;
214.                 }
215.                 sfxSource.PlayOneShot(menuSelect, audioVolume);
216.             }
217.         }
218.
219.         //Move the arrow to the right spot
220.         List<float> menuOptionYs = new List<float> ();
221.         menuOptionYs.Add(menuAttackY); menuOptionYs.Add(menuItemY);
        menuOptionYs.Add(menuSpecialY); menuOptionYs.Add(menuOtherY);
222.         arrowPos[1] = menuOptionYs[menuNum];
223.
224.     }

```

```

225.
226.     void AttackMenu () {
227.
228.         numberOfEnemies = spawn.enemyList.Count;
229.         arrowPos[1] = 7.3f;
230.
231.
232.         if (Input.GetKeyDown (KeyCode.LeftArrow) && arrowLocation > 1) {
233.             arrowLocation--;
234.             sfxSource.PlayOneShot (sfxClip, audioVolume);
235.         } else if (Input.GetKeyDown (KeyCode.RightArrow) && arrowLocation <
numberOfEnemies) {
236.             arrowLocation++;
237.             sfxSource.PlayOneShot (sfxClip, audioVolume);
238.         } else if (Input.GetKeyDown (KeyCode.X)) {
239.             attackMenu--;
240.             attackMenuCounter = 0;
241.             sfxSource.PlayOneShot (menuCancel, audioVolume);
242.         }
243.         menuPos[0] = 500;
244.         enemyNameX = (spawn.GetEnemyStats (spawn.enemyList
[numberOfEnemies-arrowLocation]) [0])-1.5f;
245.         enemyNameY = arrowPos[1] + yOffset;
246.         enemyName.SetEnemyNameText ((spawn.enemyNameList
[numberOfEnemies-arrowLocation]).ToString());
247.
248.         //Move the arrow to the correct spot
249.         arrowPos[0] = (spawn.GetEnemyStats (spawn.enemyList
[numberOfEnemies-arrowLocation]) [0])-1.5f;
250.         attackMenuCounter++;
251.
252.         if (Input.GetKeyDown (KeyCode.Z) && attackMenuCounter > 10) {
253.             enemyChoice = numberOfEnemies-arrowLocation;
254.             attackMenu=0;
255.             attackMenuCounter = 0;
256.             menuBool = false;
257.             enemyNameX = -20;
258.             enemyNameY = -20;
259.             arrowPos[0] = -100;
260.             mainMenuBool = false;
261.             sfxSource.PlayOneShot (menuSelect, audioVolume);
262.             attackBool = true;
263.             if (cutsceneCounter == 19 || cutsceneCounter == 29 || cutsceneCounter
== 38) cutsceneCounter++;
264.         }
265.

```

```
266.     }
267.
268.     void SpecialMenu() {
269.         specialPos[0] = 7;
270.         arrowPos[0] = 100;
271.         specialHeader.curMenuItem = curserLocation;
272.
273.         if (curserLocation == 1)
274.         {
275.             vert.VertShift(vert.audioSources[0], false);
276.             vert.VertShift(vert.audioSources[1], true);
277.
278.         }
279.         else
280.         {
281.             vert.VertShift(vert.audioSources[0], true);
282.             vert.VertShift(vert.audioSources[1], false);
283.         }
284.
285.         if (Input.GetKeyDown(KeyCode.LeftArrow) && curserLocation != 1 &&
curserLocation != 4){
286.             curserLocation--;
287.             curserPos[0] = curserPos[0] - 2;
288.         }
289.         else if (Input.GetKeyDown(KeyCode.RightArrow) && curserLocation != 3 &&
curserLocation != 6){
290.             curserLocation++;
291.             curserPos[0] = curserPos[0] + 2;
292.         }
293.         else if (Input.GetKeyDown(KeyCode.DownArrow) && curserLocation < 4){
294.             curserLocation = curserLocation + 3;
295.             curserPos[1] = curserPos[1] - 2;
296.         }
297.         else if (Input.GetKeyDown(KeyCode.UpArrow) && curserLocation > 3){
298.             curserLocation = curserLocation - 3;
299.             curserPos[1] = curserPos[1] + 2;
300.         }
301.         else if (Input.GetKeyDown(KeyCode.X)){
302.             attackMenuCounter = 0;
303.             vert.VertShift(vert.audioSources[0], true);
304.             vert.VertShift(vert.audioSources[1], false);
305.             specialMenu--;
306.             curserPos[0] = 100;
307.             sfxSource.PlayOneShot(menuCancel, audioVolume);
308.         }
309.
```

```

310.         if (Input.GetKeyDown(KeyCode.Z) && attackMenuCounter > 10)
311.         {
312.             menuPos[0] = 500;
313.             specialPos[0] = 500;
314.             curserPos[0] = 100;
315.             specialMenu = 0;
316.             attackMenuCounter = 0;
317.             menuBool = false;
318.             vert.VertShift(vert.audioSources[0], true);
319.             vert.VertShift(vert.audioSources[1], false);
320.             curserPos[0] = 100;
321.             mainMenuBool = false;
322.             sfxSource.PlayOneShot(menuSelect, audioVolume);
323.             specialStateCount = 1;
324.         }
325.         attackMenuCounter++;
326.
327.     }
328. }

```

MusicControl

```

1. using System.Collections;
2. using System.Collections.Generic;
3. using UnityEngine;
4.
5. public class MusicControl : MonoBehaviour {
6.
7.
8.     public AudioSource musicSource;
9.     public PlayerControl player;
10.
11.     // Use this for initialization
12.     void Start () {
13.     }
14.
15.     // Update is called once per frame
16.     void Update () {
17.
18.     }
19.
20.     public void playMusic() {
21.         musicSource.Play ();
22.     }
23. }

```

HorizReseq

```
1. using System.Collections;
2. using System.Collections.Generic;
3. using UnityEngine;
4.
5. // Namespace for audio
6. using UnityEngine.Audio;
7.
8. public class HorizReseq : MonoBehaviour {
9.
10.     [Header("Target AudioManager.")]
11.
12.     public AudioManager myOutputMixerGroup;
13.
14.     public PlayerControl player;
15.     public VertRemixer vertRemixer;
16.     private bool vertChecker;
17.
18.     public bool musicChange;
19.
20.     // First let's make a custom struct
21.     // which will allow us to group several variables together
22.
23.     // Tagging it with System.Serializable allows us to see this in the inspector
24.     [System.Serializable]
25.
26.     // We'll call the struct "AudioSegment" to be different from AudioClip and
    AudioSource
27.     public struct AudioSegment {
28.
29.         // each audio segment will contain a clip
30.         public AudioClip clip;
31.
32.         // it will also contain a double-precision duration
33.         // allowing us to have audio-sample level time accuracy
34.         public double duration;
35.
36.         // we'll also have an integer that represents which segment to play next
37.         // (assuming we'll be using a list or array of these later)
38.         public int nextSegment;
39.     }
40.
```

```

41.     [Header("The following is an array of AudioSegments that can be sequenced in any
order. ")]
42.     [Header("Each AudioSegment has a Clip, a Duration, and a Next Segment.")]
43.     [Header("Clip: the audio file to be triggered.")]
44.     [Header("Duration: the duration in seconds before the next segment is triggered.")]
45.     [Header("Next Segment: this is the segment to go to next.")]
46.     [Header("Note that the Next Segment is specified by its Element number.")]
47.
48.     // Now let's create an array of these segments
49.     // Arrays are just places to store multiple variables of the same type,
50.     // which you can access by index
51.     // The size of the array must be known in advance!
52.     public AudioSegment[] audioSegments = new AudioSegment[4];
53.
54.
55.     // we also want a variety of integers to control various things:
56.     public int nextSource, curSource, nextSegment, curSegment;
57.
58.     // this public int will be our polyphony
59.     // or the number of AudioSources we want to use
60.     [Header("Number of voices to use (minimum is 2)")]
61.     [Range(2,16)]
62.     public int polyphony = 4;
63.
64.     [Header("Initial delay in seconds to give the computer time to spool audio off the
hard drive.")]
65.     public double initialDelay = 1;
66.
67.     // A list is like an Array but you don't need to know the length of a list in
advance.
68.     // We want the user to specify polyphony dynamically, so we'll use a list instead
of an array
69.     private List<AudioSource> audioSources = new List<AudioSource>();
70.
71.     // let's declare a double (64-bit high precision number)
72.     // to specify the absolute dsp time when the next audio segment comes in
73.     public double scheduledTime;
74.
75.     // bool for an initialization flag
76.     private bool doOnce;
77.
78.     // When start is called
79.     void Start() {
80.
81.         vertChecker = true;
82.         // we'll use a while loop to add audio sources to our horiz reseq player

```

```

83.     // start at 0
84.     int i = 0;
85.
86.     // then keep adding audio sources while i is less than the number of sources we
    want
87.     while (i < polyphony) {
88.
89.         // audioSources is our List, and we add to it each of
90.         // the AudioSources that we add using AddComponent
91.         audioSources.Add (this.gameObject.AddComponent<AudioSource> ());
92.
93.         // increment counter
94.         i++;
95.     }
96.
97.     foreach (AudioSource audioSource in audioSources) {
98.         audioSource.outputAudioMixerGroup = myOutputMixerGroup;
99.     }
100.
101.     // Set the scheduled event time to NOW (as soon as Start() is done)
102.     scheduledTime = AudioSettings.dspTime;
103.
104. }
105.
106. // each video frame
107. void Update() {
108.
109.     //print(curSegment);
110.
111.     // we check and see if we've reached the scheduled event time
112.     if (AudioSettings.dspTime >= scheduledTime) {
113.
114.         if (curSegment > 0 && vertChecker == true){
115.             vertRemixer.vertBool = true;
116.             vertChecker = false;
117.         }
118.
119.         musicChange = true;
120.
121.         // if so, and this is the first time this happens (no clip playing
    already)
122.         if (!doOnce) {
123.
124.             // Delay this slightly into the future to give the CPU time
125.             // to start streaming audio off the hard drive
126.             scheduledTime = scheduledTime + initialDelay;

```

```

127.
128.         // Assign the next clip to be scheduled
129.         audioSources [nextSource].clip = audioSegments [nextSegment].clip;
130.
131.         // schedule the next audio source to play at a point in the future
132.         audioSources [nextSource].PlayScheduled (scheduledTime);
133.
134.
135.         // flag that we've done this initialization
136.         doOnce = true;
137.
138.         // if we've reached the scheduled event time and we've already done our
initialization
139.         } else {
140.
141.             player.timeCounter = 0f;
142.             player.biggerTime = 0f;
143.             player.oneBeat = 0f;
144.             player.segmentCounter = 1;
145.
146.
147.             // Assign the next clip to be scheduled
148.             audioSources [nextSource].clip = audioSegments [nextSegment].clip;
149.
150.             // calculate when to play it based on current clip duration
151.             scheduledTime = audioSegments [curSegment].duration +
scheduledTime;
152.
153.             // schedule the next audio source to play at a point in the future
154.             audioSources [nextSource].PlayScheduled (scheduledTime);
155.
156.         }
157.
158.         // get the next curSegment from nextSegment
159.         curSegment = nextSegment;
160.
161.         // get the next nextSegment by looking it up in the segment array
162.         nextSegment = audioSegments[nextSegment].nextSegment;
163.
164.         // get the current source from the old next source
165.         curSource = nextSource;
166.
167.         // update next source
168.         nextSource = (nextSource + 1) % polyphony;
169.     }
170.

```

```

171.     }
172.
173.     public void ChangeScheduledSegment (int segment, int next) {
174.
175.         // Future (t + 1)
176.         // Change the currently scheduled segment based on input
177.         curSegment = segment;
178.
179.         // Find the segment that follows that
180.         nextSegment = next;
181.
182.         // stop the audio source that is scheduled to play the currently scheduled
    segment
183.         audioSources [curSource].Stop ();
184.
185.         // assign its new target clip
186.         audioSources [curSource].clip = audioSegments [curSegment].clip;
187.
188.         // set play time (time to play next is still accurate
189.         // because it is based on a clip already playing).
190.         audioSources [curSource].PlayScheduled (scheduledTime);
191.
192.     }
193.
194.
195. }

```

VertRemixer

```

1. using System.Collections;
2. using System.Collections.Generic;
3. using UnityEngine;
4.
5. using UnityEngine.Audio;
6.
7. public class VertRemixer : MonoBehaviour
8. {
9.
10.     public AudioManager myOutputMixerGroup;
11.     public int nextSource, curSource;
12.     public PlayerControl player;
13.     public HorizReseq horizReseq;
14.     public bool vertBool;
15.
16.

```

```

17. // Tagging it with System.Serializable allows us to see this in the inspector
18. [System.Serializable]
19.
20. // We'll call the struct "AudioSegment" to be different from AudioClip and
    AudioSource
21. public struct AudioLayer
22. {
23.
24.     // each audio segment will contain a clip
25.     public AudioClip clip;
26.
27.     // it will also contain a double-precision duration
28.     // allowing us to have audio-sample level time accuracy
29.     public double duration;
30.     public float vol;
31. }
32. public AudioLayer[] audioLayers = new AudioLayer[4];
33.
34. [Header("Number of voices to use (minimum is 2)")]
35. [Range(2, 16)]
36. public int polyphony = 4;
37.
38. public List<AudioSource> audioSources = new List<AudioSource>();
39. public double scheduledTime;
40.
41. // Start is called before the first frame update
42. void Start()
43. {
44.     int i = 0;
45.
46.     // then keep adding audio sources while i is less than the number of sources we
    want
47.     while (i < audioLayers.Length)
48.     {
49.
50.         // audioSources is our List, and we add to it each of
51.         // the AudioSources that we add using AddComponent
52.         audioSources.Add(this.gameObject.AddComponent<AudioSource>());
53.         audioSources[i].clip = audioLayers[i].clip;
54.         audioSources[i].loop = true;
55.         audioSources[i].volume = audioLayers[i].vol;
56.
57.         // increment counter
58.         i++;
59.     }
60.

```

```

61.     foreach (AudioSource audioSource in audioSources)
62.     {
63.         audioSource.outputAudioMixerGroup = myOutputMixerGroup;
64.     }
65.
66.     // Set the scheduled event time to NOW (as soon as Start() is done)
67.     scheduledTime = AudioSettings.dspTime;
68. }
69.
70. // Update is called once per frame
71. void Update()
72. {
73.
74.     if (Input.GetKeyDown(KeyCode.P)) VertShift(audioSources[0], true);
75.     if (Input.GetKeyDown(KeyCode.O)) VertShift(audioSources[0], false);
76.
77.     if (vertBool == true)
78.     {
79.         foreach (AudioSource audioSource in audioSources)
80.         {
81.             audioSource.Play();
82.         }
83.     }
84.     vertBool = false;
85.
86.
87.
88.     // get the current source from the old next source
89.     curSource = nextSource;
90.
91.     // update next source
92.     nextSource = (nextSource + 1) % polyphony;
93. }
94.
95. public void VertShift(AudioSource audioSource, bool on){
96.     if (on == true) audioSource.volume = 1f;
97.     else audioSource.volume = 0f;
98. }
99. }

```

ArmAnimator:

```

1. //Manage's Wolfgang arms when they're a separate object from his body
2.
3. using System.Collections;

```

```

4. using System.Collections.Generic;
5. using UnityEngine;
6.
7. public class ArmAnimator : MonoBehaviour {
8.
9.     public Animator animator;
10.    public int waitHash = Animator.StringToHash("Wait");
11.    public AudioClip guitarSwing;
12.    //private AudioSource sfxSource;
13.
14.    // Use this for initialization
15.    void Start () {
16.        animator = GetComponent<Animator>();
17.    }
18.
19.    // Update is called once per frame
20.    void Update () {
21.        animator.ResetTrigger (waitHash);
22.        if (Input.GetKeyDown (KeyCode.X)) {
23.            Vector3 armPos = new Vector3 (transform.position [0], 0, -10);
24.            if (transform.position [1] > 0) {
25.                AudioSource.PlayClipAtPoint (guitarSwing, armPos, 20f);
26.                animator.SetTrigger (waitHash);
27.            }
28.        }
29.    }
30. }

```

Instructor_Move

```

1. //Not only manages the movements of the instructor, but also controls the tutorial's
   dialogue
2.
3. using System.Collections;
4. using System.Collections.Generic;
5. using UnityEngine;
6.
7. public class Instructor_Move : MonoBehaviour
8. {
9.
10.    public HorizReseq horizReseq;
11.    public PlayerControl player;
12.    public Battle_Menu_Control menuControl;
13.    public Dummy dummy;
14.    public float walkSpeed;

```

```

15.     public Vector3 instructorPos;
16.     public int dialogueCount; //determines which dialogue to run
17.     public DialogueManager dialogue;
18.     public Animator animator;
19.     public SpriteRenderer spriteRenderer;
20.     public bool successfulHit, noJump; //these are for checking what the player does
    during the attack
21.     public string idleCheck, runCheck, kick1Check, currAnim; //These strings are for
    animation states
22.     public int moveHash = Animator.StringToHash("Move");
23.     public int pickUpHash = Animator.StringToHash("PickUp");
24.     public int kickHash = Animator.StringToHash("Kick");
25.
26.     public int idleHash = Animator.StringToHash("Instructor_Idle");
27.
28.     public float offsetX, offsetY; //This is the offset between the professor and the
    dummy when the professor is holding the dummy
29.     private int count; //for counting what part of the scripted movements need to be
    brought up.
30.     // Start is called before the first frame update
31.     void Start()
32.     {
33.         walkSpeed = .25f;
34.         instructorPos = new Vector3(6, -.5f, this.transform.position[2]);
35.         animator = GetComponent<Animator>();
36.         spriteRenderer = GetComponent<SpriteRenderer>();
37.         idleCheck = animator.GetCurrentAnimatorClipInfo(0)[0].clip.ToString();
38.
39.
40.     }
41.
42.     private void FixedUpdate()
43.     {
44.         if (player.oneBeat == 0 && currAnim == idleCheck) {
45.             animator.Play(idleHash, 0, 0);
46.         }
47.
48.     }
49.
50.     // Update is called once per frame
51.     void Update()
52.     {
53.
54.         currAnim = animator.GetCurrentAnimatorClipInfo(0)[0].clip.ToString();
55.         //if (currAnim == idleCheck && player.oneBeat == 0)
    animator.ForceStateNormalizedTime(0);

```

```

56.     if (dummy.dummyFollow == true) {
57.         dummy.dummyX = instructorPos[0] + offsetX;
58.         dummy.dummyY = instructorPos[1] + offsetY;
59.     }
60.     count = menuControl.cutsceneCounter;
61.     //In count 1, the instructor moves from the bottom of the screen to enemy
position one
62.     if (count == 1)
63.     {
64.         Move(walkSpeed, true);
65.         runCheck = animator.GetCurrentAnimatorClipInfo(0)[0].clip.ToString();
66.         if (instructorPos[0] > menuControl.enemyPlaces[0] && instructorPos[1] > 0)
67.         {
68.             instructorPos[0] = menuControl.enemyPlaces[0];
69.             spriteRenderer.flipX = false;
70.             menuControl.cutsceneCounter++;
71.             animator.SetInteger(moveHash, 2);
72.             dialogueCount = 1;
73.         }
74.     }
75.
76.     //In count 3, the instructor moves to enemy position four
77.     else if (count == 3)
78.     {
79.         Move(walkSpeed, true);
80.         if (instructorPos[0] > menuControl.enemyPlaces[3] && instructorPos[1] > 0)
81.         {
82.             instructorPos[0] = menuControl.enemyPlaces[3];
83.             spriteRenderer.flipX = false;
84.             menuControl.cutsceneCounter++;
85.             animator.SetInteger(moveHash, 2);
86.             dialogueCount = 2;
87.         }
88.     }
89.     //In count 5, the instructor goes offstage
90.     else if (count == 5)
91.     {
92.         Move(walkSpeed, true);
93.         if (instructorPos[0] > 16f)
94.         {
95.             spriteRenderer.flipX = false;
96.             menuControl.cutsceneCounter++;
97.             dialogueCount = 3;
98.         }
99.     }
100.    //In count 7, the Instructor takes the dummy to enemy position one

```

```
101.         else if (count == 7)
102.         {
103.             Move(walkSpeed * -1, false);
104.             dummy.dummyFollow = true;
105.             if (instructorPos[0] < menuControl.enemyPlaces[0] - offsetX &&
instructorPos[1] > 0)
106.             {
107.                 instructorPos[0] = menuControl.enemyPlaces[0] - offsetX;
108.                 dummy.pos = 1;
109.                 spriteRenderer.flipX = false;
110.                 menuControl.cutsceneCounter++;
111.                 //menuControl.cutsceneCounter = 35;
112.                 //horizReseq.ChangeScheduledSegment(0, 5);
113.                 animator.SetInteger(moveHash, 2);
114.                 dialogueCount = 4;
115.                 dummy.animator.SetTrigger("DropHandle");
116.             }
117.         }
118.
119.         //Continues the instructor's dialogue after Wolfgang shows off his guitar
120.         else if (count == 10)
121.         {
122.             horizReseq.ChangeScheduledSegment(1, 1);
123.             dialogueCount = 5;
124.             menuControl.cutsceneCounter++;
125.
126.         }
127.
128.         else if (count == 13)
129.         {
130.             dialogueCount = 6;
131.             menuControl.cutsceneCounter++;
132.         }
133.
134.         else if (count == 16)
135.         {
136.             dialogueCount = 7;
137.             menuControl.cutsceneCounter++;
138.         }
139.
140.         else if (count == 20)
141.         {
142.             dialogueCount = 8;
143.             menuControl.cutsceneCounter++;
144.         }
145.
```

```

146.     else if (count == 23 && player.playerY < 0)
147.     {
148.         if (successfulHit == true)
149.         {
150.             dialogueCount = 9;
151.         }
152.         else dialogueCount = 10;
153.         menuControl.cutsceneCounter++;
154.     }
155.
156.     else if (count == 25 || count == 34 || count == 43 || count == 67)
157.     {
158.         successfulHit = true;
159.         dummy.Animator.ResetTrigger("DropHandle");
160.         dummy.Animator.SetBool("PickUpHandle", true);
161.         animator.SetBool(pickUpHash, true);
162.         menuControl.cutsceneCounter++;
163.     }
164.
165.
166.     //Moves the instructor and dummy to enemy position 4, but only after the
167.     //last animation has played out.
168.     //This is done by comparing the current state of animation to the run state
169.     else if (count == 26 || count == 68)
170.     {
171.         dummy.pos = 4;
172.         animator.SetBool(pickUpHash, false);
173.         animator.SetInteger(moveHash, 1);
174.         if (currAnim == runCheck) Move(walkSpeed, true);
175.         dummy.dummyFollow = true;
176.         if (dummy.dummyX > menuControl.enemyPlaces[3] && instructorPos[1] > 0)
177.         {
178.             spriteRenderer.flipX = false;
179.             menuControl.cutsceneCounter++;
180.             animator.SetInteger(moveHash, 2);
181.             dummy.dummyFollow = false;
182.         }
183.     }
184.
185.
186.     else if (count == 27 || count == 69)
187.     {
188.         dummy.dummyX = menuControl.enemyPlaces[3];
189.         instructorPos[0] = dummy.dummyX - offsetX;
190.         dummy.Animator.SetBool("PickUpHandle", false);

```

```
191.         dummy. animator. SetBool("DropHandle", true);
192.         if (count == 27) dialogueCount = 11;
193.         else if (count == 69) dialogueCount = 24;
194.         menuControl. menuBool = true;
195.         menuControl. cutsceneCounter++;
196.     }
197.
198.     else if (count == 31)
199.     {
200.         if (successfulHit == false) dialogueCount = 12;
201.         else
202.         {
203.             dialogueCount = 13;
204.             horizReseq. ChangeScheduledSegment(2, 2);
205.         }
206.         menuControl. cutsceneCounter++;
207.     }
208.
209.     else if (count == 33)
210.     {
211.         if (successfulHit == false)
212.         {
213.             menuControl. cutsceneCounter = 29;
214.         }
215.         else
216.         {
217.             menuControl. cutsceneCounter++;
218.         }
219.         successfulHit = true;
220.     }
221.
222.     else if (count == 35)
223.     {
224.         dummy. pos = 3;
225.         animator. SetBool(pickUpHash, false);
226.         animator. SetInteger(moveHash, 1);
227.         if (currAnim == runCheck) Move(walkSpeed * -1, false);
228.         dummy. dummyFollow = true;
229.         if (dummy. dummyX < menuControl. enemyPlaces[2] && instructorPos[1] > 0)
230.         {
231.             menuControl. cutsceneCounter++;
232.             animator. SetInteger(moveHash, 2);
233.             dummy. dummyFollow = false;
234.         }
235.     }
236.
```

```
237.     }
238.
239.     else if (count == 36)
240.     {
241.         dummy.dummyX = menuControl.enemyPlaces[2];
242.         instructorPos[0] = dummy.dummyX - offsetX;
243.         dummy.Animator.SetBool("PickUpHandle", false);
244.         dummy.Animator.SetBool("DropHandle", true);
245.         dialogueCount = 14;
246.         menuControl.menuBool = true;
247.         menuControl.cutsceneCounter++;
248.     }
249.
250.     else if (count == 40)
251.     {
252.         if (successfulHit == false) dialogueCount = 15;
253.         else if (noJump == true) dialogueCount = 16;
254.         else
255.         {
256.             dialogueCount = 17;
257.             horizReseq.ChangeScheduledSegment(3, 4);
258.         }
259.         menuControl.cutsceneCounter++;
260.     }
261.
262.     else if (count == 42)
263.     {
264.         if (successfulHit == false || noJump == true)
265.             menuControl.cutsceneCounter = 38;
266.         else menuControl.cutsceneCounter++;
267.         successfulHit = true;
268.         noJump = true;
269.     }
270.
271.     else if (count == 44)
272.     {
273.         animator.SetBool(pickUpHash, false);
274.         animator.SetInteger(moveHash, 1);
275.         if (currAnim == runCheck) Move(walkSpeed * -1, false);
276.         dummy.dummyFollow = true;
277.         if (dummy.dummyX < -13 && instructorPos[1] > 0)
278.         {
279.             menuControl.cutsceneCounter++;
280.             animator.SetInteger(moveHash, 2);
281.             dummy.dummyFollow = false;
```

```

282.     }
283.   }
284.
285.   else if (count == 45){
286.     player.other.enemyPhase = true;
287.     dummy.Animator.SetBool("PickUpHandle", false);
288.     dummy.Animator.SetBool("DropHandle", true);
289.     Move(-.2f, false);
290.     if (instructorPos[0] <= -11.7f){
291.       instructorPos[0] = -11.7f;
292.       animator.SetInteger(moveHash, 2);
293.       if (player.timeCounter == player.beat3)
294.       {
295.         animator.SetBool(kickHash, true);
296.         menuControl.cutsceneCounter++;
297.       }
298.     }
299.     else Move(-.2f, false);
300.   }
301.
302.   else if (count == 46 || count == 47 || count == 56 || count == 57){
303.     kick1Check = animator.GetCurrentAnimatorClipInfo(0)[0].clip.ToString();
304.     if(player.timeCounter == 0) menuControl.cutsceneCounter++;
305.   }
306.
307.   else if (count == 48 || count == 58){
308.     animator.SetBool(kickHash, false);
309.     if(currAnim != kick1Check) {
310.       dummy.Animator.SetInteger(dummy.attackHash, 1);
311.       if (player.timeCounter > 0 && player.timeCounter < player.beat1)
dummy.dummySpeed = -.375f;
312.       else if (player.timeCounter > player.beat1 && player.timeCounter <
player.beat2) dummy.dummySpeed = -.2f;
313.       else if (player.timeCounter > player.beat2 && player.timeCounter <
player.beat3) dummy.dummySpeed = -.1f;
314.       else if (player.timeCounter > player.beat3) {
315.         dummy.dummySpeed = 0;
316.         menuControl.cutsceneCounter++;
317.         dummy.Animator.SetInteger(dummy.attackHash, 2);
318.       }
319.       dummy.Move(dummy.dummySpeed);
320.     }
321.   }
322.
323.   else if (count == 49 || count == 50 || count == 59 || count == 60){
324.     dummy.dummySpeed = -.375f;

```

```

325.         if(player.timeCounter >= .14f && player.timeCounter < player.beat1/2 &&
dummy.dummyY < 1){
326.             player.enemyHit = true;
327.             if (count == 49 || count == 59) menuControl.cutsceneCounter++;
328.         }
329.         else{
330.             player.enemyHit = false;
331.             if (count == 50 || count == 60) menuControl.cutsceneCounter++;
332.         }
333.     }
334.
335.     else if (count == 51){
336.         if (player.dodgeSuccess == false) dialogueCount = 18;
337.         else dialogueCount = 19;
338.         horizReseq.ChangeScheduledSegment(5, 6);
339.         player.other.enemyPhase = false;
340.         menuControl.cutsceneCounter++;
341.     }
342.
343.     else if (count == 53 || count == 63){
344.         dummy.Animator.SetInteger(dummy.attackHash, 3);
345.         dummy.Move(dummy.dummySpeed*-1f);
346.         if(dummy.dummyY > 1 && dummy.dummyX >= -13.25f){
347.             dummy.dummyX = -13.25f;
348.             dummy.Animator.SetInteger(dummy.attackHash, 4);
349.             if (count == 53) dialogueCount = 20;
350.             menuControl.cutsceneCounter++;
351.         }
352.     }
353.
354.     else if (count == 55){
355.         player.other.enemyPhase = true;
356.         if (player.timeCounter == player.beat3)
357.         {
358.             animator.SetBool(kickHash, true);
359.             menuControl.cutsceneCounter++;
360.         }
361.     }
362.
363.     else if (count == 61){
364.         if (player.dodgeSuccess == false) dialogueCount = 21;
365.         else
366.         {
367.             dialogueCount = 22;
368.             menuControl.vert.VertShift(menuControl.vert.audioSources[0], true);
369.         }

```

```
370.         player.other.enemyPhase = false;
371.         menuControl.cutsceneCounter++;
372.     }
373.
374.     else if (count == 64){
375.         if (player.dodgeSuccess == false) menuControl.cutsceneCounter = 55;
376.         else {
377.             dialogueCount = 23;
378.             menuControl.cutsceneCounter++;
379.         }
380.     }
381.
382.     else if (count == 66){
383.         Move(.2f, false);
384.         if(instructorPos[0] >= dummy.dummyX - offsetX){
385.             instructorPos[0] = dummy.dummyX - offsetX;
386.             animator.SetInteger(moveHash, 2);
387.             menuControl.cutsceneCounter++;
388.         }
389.     }
390.
391.     else if(count == 72){
392.         dialogueCount = 25;
393.         menuControl.cutsceneCounter++;
394.     }
395.
396.     else if (count == 75){
397.         dialogueCount = 26;
398.         menuControl.cutsceneCounter++;
399.     }
400.
401.     else if (count == 79){
402.         if (player.specialSuccessCount == 0) dialogueCount = 27;
403.         else if (player.specialSuccessCount < 4) dialogueCount = 28;
404.         else if (player.specialSuccessCount < 8) dialogueCount = 29;
405.         else dialogueCount = 30;
406.         menuControl.cutsceneCounter++;
407.     }
408.
409.     else if (count == 81){
410.         if (player.specialSuccessCount < 4) menuControl.cutsceneCounter = 78;
411.         else
412.         {
413.             dialogueCount = 31;
414.             menuControl.cutsceneCounter++;
415.         }
```

```

416.         player.specialSuccessCount = 0;
417.     }
418.
419.
420.     if (instructorPos[0] > 15f && instructorPos[1] < 0)
421.     {
422.         instructorPos[0] = -15f;
423.         instructorPos[1] = 7.7f;
424.     }
425.
426.     if (player.failHit > 0) successfulHit = false;
427.     if (player.jumpAttackTrue == true) noJump = false;
428.
429.     transform.position = instructorPos;
430. }
431.
432. public void Move(float speed, bool right){
433.     animator.SetInteger(moveHash, 1);
434.     instructorPos[0] = instructorPos[0] + speed;
435.     if (right == true) spriteRenderer.flipX = true;
436. }
437.
438.
439. }

```

DialogueManager

```

1. using System.Collections;
2. using System.Collections.Generic;
3. using UnityEngine;
4. using UnityEngine.UI;
5.
6. public class DialogueManager : MonoBehaviour
7. {
8.
9.     public MusicControl music;
10.    public DialogueBox box;
11.    public AudioClip dialogueNoise;
12.    private AudioSource sfxSource;
13.    public Battle_Menu_Control menuControl;
14.    public bool ender;
15.
16.
17.    public Text nameText;
18.    public Text dialogueText;

```

```
19.
20.     public Queue<string> sentences;
21.
22.     void Start()
23.     {
24.         sentences = new Queue<string>();
25.         sfxSource = GetComponent<AudioSource>();
26.
27.     }
28.
29.     void Update()
30.     {
31.         if (Input.GetKeyDown(KeyCode.Z))
32.         {
33.             DisplayNextSentence();
34.         }
35.     }
36.
37.     public void startDialogue(Dialogue dialogue, bool end)
38.     {
39.
40.         nameText.text = dialogue.name;
41.
42.         ender = end;
43.
44.         sentences.Clear();
45.
46.         foreach (string sentence in dialogue.sentences)
47.         {
48.             sentences.Enqueue(sentence);
49.         }
50.
51.         DisplayNextSentence();
52.     }
53.
54.     public void DisplayNextSentence()
55.     {
56.         if (sentences.Count == 0)
57.         {
58.             if (box.dialogueSession == true) EndDialogue();
59.             return;
60.         }
61.         string sentence = sentences.Dequeue();
62.         StopAllCoroutines();
63.         StartCoroutine(TypeSentence(sentence));
64.     }
```

```

65.
66.     IEnumerator TypeSentence(string sentence) {
67.         dialogueText.text = " ";
68.         foreach (char letter in sentence.ToCharArray()){
69.
70.             dialogueText.text += letter;
71.             //sfxSource.PlayOneShot(dialogueNoise, .2f);
72.             yield return null;
73.         }
74.     }
75.
76.     public void EndDialogue() {
77.         if (ender == true) box.Disappear();
78.         ender = false;
79.         menuControl.cutsceneCounter++;
80.
81.     }
82. }

```

Dummy

```

1. //Script for controlling the tutorial enemy
2.
3. using System.Collections;
4. using System.Collections.Generic;
5. using UnityEngine;
6.
7. public class Dummy : MonoBehaviour
8. {
9.     public Animator animator;
10.    public SpriteRenderer spriteRenderer;
11.    public int dropHandle = Animator.StringToHash("DropHandle");
12.    public int pickUpHandle = Animator.StringToHash("PickUpHandle");
13.    public int attackHash = Animator.StringToHash("Attack");
14.    public float dummyX, dummyY, dummyZ;
15.    public float dummySpeed;
16.    public bool dummyFollow;
17.    public int pos;
18.
19.
20.    // Start is called before the first frame update
21.    void Start()
22.    {
23.        animator = GetComponent<Animator>();
24.        spriteRenderer = GetComponent<SpriteRenderer>();

```

```

25.     }
26.
27.     // Update is called once per frame
28.     void Update()
29.     {
30.
31.         if (dummyX > 15f && dummyY < 1)
32.         {
33.             dummyX = -15f;
34.             dummyY = 8.2f;
35.         }
36.         else if (dummyX < -15f && dummyY > 1){
37.             dummyX = 15f;
38.             dummyY = 0;
39.         }
40.
41.         transform.position = new Vector3(dummyX, dummyY, dummyZ);
42.
43.     }
44.
45.     public void Move(float speed){
46.         dummyX = dummyX + speed;
47.     }
48. }

```

Spot

```

1. //Controls the spotlight that occasionally appears over objects
2.
3. using System.Collections;
4. using System.Collections.Generic;
5. using UnityEngine;
6.
7. public class Spot : MonoBehaviour
8. {
9.
10.     public Battle_Menu_Control menuControl;
11.     public int counter;
12.     public Vector3 spotPos;
13.     public float transparency;
14.     public bool darkenBool;
15.     public bool lightenBool;
16.     SpriteRenderer spriteRenderer;
17.
18.     // Start is called before the first frame update

```

```
19. void Start()
20. {
21.     spotPos = new Vector3(0, 0, this.transform.position[2]);
22.     spriteRenderer = this.GetComponent<SpriteRenderer>();
23.     transparency = 0;
24.     darkenBool = false;
25.     lightenBool = false;
26. }
27.
28. // Update is called once per frame
29. void Update()
30. {
31.     counter = menuControl.cutsceneCounter;
32.     if (darkenBool == true) Darken();
33.     else if (lightenBool == true) Lighten();
34.     if (counter == 15) {
35.         spotPos[0] = -1;
36.         spotPos[1] = -.6f;
37.         darkenBool = true;
38.         menuControl.cutsceneCounter++;
39.     }
40.
41.     if (counter == 18) {
42.         lightenBool = true;
43.         menuControl.cutsceneCounter++;
44.     }
45.
46.     if (counter == 74){
47.         spotPos[0] = -1;
48.         spotPos[1] = -6.5f;
49.         darkenBool = true;
50.         menuControl.cutsceneCounter++;
51.     }
52.
53.     if (counter == 77){
54.         lightenBool = true;
55.         menuControl.cutsceneCounter++;
56.     }
57.
58.     this.transform.position = spotPos;
59.     spriteRenderer.color = new Vector4(255, 255, 255, transparency);
60.
61. }
62.
63. public void Darken(){
64.     if (spriteRenderer.color[3] < .75f) transparency = transparency + .1f;
```

```

65.     else darkenBool = false;
66.   }
67.
68.   public void Lighten(){
69.       if (spriteRenderer.color[3] > 0) transparency = transparency - .1f;
70.       else lightenBool = false;
71.   }
72.
73. }

```

FollowArrow

```

1. //Makes the enemy name follow the arrow while the player selects which enemy to attack
2.
3. using System.Collections;
4. using System.Collections.Generic;
5. using UnityEngine;
6. using UnityEngine.UI;
7.
8. public class FollowArrow : MonoBehaviour {
9.
10.    public Text enemyNameText;
11.    public float enemyNameX = -100;
12.    public float enemyNameY = -100;
13.    public float enemyNameZ;
14.
15.    // Use this for initialization
16.    void Start () {
17.        enemyNameZ = -2;
18.        Vector3 startingPos = new Vector3 (enemyNameX, enemyNameY, enemyNameZ);
19.        transform.position = startingPos;
20.    }
21.
22.    public void SetEnemyNameText (string nameText) {
23.        enemyNameText.text = nameText;
24.    }
25. }

```

SpecialHeader

```

1. //Displays the text depending on what part of the special menu is selected
2.
3. using System.Collections;

```

```

4. using System.Collections.Generic;
5. using UnityEngine;
6. using UnityEngine.UI;
7.
8. public class SpecialHeader : MonoBehaviour
9. {
10.
11.     public Text specialMenuTitle;
12.     public Text specialMenuDesc;
13.     public int curMenuItem;
14.
15.     // Start is called before the first frame update
16.     void Start()
17.     {
18.         curMenuItem = 1;
19.     }
20.
21.     // Update is called once per frame
22.     void Update()
23.     {
24.         if (curMenuItem == 1)
25.         {
26.             specialMenuTitle.text = "Steel String Swipe";
27.             specialMenuDesc.text =
28.                 "press a string's key when the falling button overlaps the still one.
29.                 It'll be either z, x, c, or v.";
30.         }
31.         else
32.         {
33.             specialMenuTitle.text = "???";
34.             specialMenuDesc.text = "????????????????????????????";
35.         }
36.     }

```

DialogueBox

```

1. //Manages the dialogue box
2.
3. using System.Collections;
4. using System.Collections.Generic;
5. using UnityEngine;
6.
7. public class DialogueBox : MonoBehaviour
8. {

```

```

9.
10.     public bool dialogueSession;
11.
12.     private void Start()
13.     {
14.         dialogueSession = false;
15.     }
16.     public void Appear(float x, float y, float scaleX, float scaleY) {
17.         transform.position = new Vector3(x, y, transform.position[2]);
18.         transform.localScale = new Vector3(scaleX, scaleY, 1);
19.         dialogueSession = true;
20.     }
21.
22.     public void Disappear()
23.     {
24.         this.transform.position = new Vector3(-1000, -1000, transform.position[2]);
25.         dialogueSession = false;
26.     }
27. }

```

DialogueTrigger

```

1. using System.Collections;
2. using System.Collections.Generic;
3. using UnityEngine;
4.
5. public class DialogueTrigger : MonoBehaviour
6. {
7.     public Instructor_Move instructor;
8.     public DialogueBox box;
9.     public Dialogue dialogue;
10.    public int dialogueNumber;
11.    public bool first, last;
12.
13.
14.    void Start()
15.    {
16.        instructor = FindObjectOfType<Instructor_Move>();
17.        box = FindObjectOfType<DialogueBox>();
18.    }
19.    void Update()
20.    {
21.        if (dialogueNumber == 0 && Input.GetKeyDown(KeyCode.S)) {
22.            box.Appear(0f, -7f, 1.7f, 1.5f);
23.            TriggerDialogue();

```

```

24.     }
25.     else if (dialogueNumber > 0 && dialogueNumber == instructor.dialogueCount) {
26.         TriggerDialogue();
27.         instructor.dialogueCount = 0;
28.     }
29. }
30.
31.
32. public void TriggerDialogue()
33. {
34.     if(first){
35.         box.Appear(0f, -7f, 1.7f, 1.5f);
36.     }
37.     FindObjectOfType<DialogueManager>().startDialogue(dialogue, last);
38. }
39. }

```

SpecialAttack

```

1. //Manages the player's special attacks
2.
3. using System.Collections;
4. using System.Collections.Generic;
5. using UnityEngine;
6.
7. public class SpecialAttack : MonoBehaviour
8. {
9.     private List<GameObject> strings = new List<GameObject>();
10.    private List<GameObject> buttonOffs = new List<GameObject>();
11.    private List<GameObject> buttonOns = new List<GameObject>();
12.    private float offset = 1.46f;
13.    public GameObject string1;
14.    public GameObject string2;
15.    public GameObject string3;
16.    public GameObject string4;
17.    public GameObject buttonOff1;
18.    public GameObject buttonOff2;
19.    public GameObject buttonOff3;
20.    public GameObject buttonOff4;
21.    public GameObject buttonOn1;
22.    public GameObject buttonOn2;
23.    public GameObject buttonOn3;
24.    public GameObject buttonOn4;
25.    public GameObject currentBeat;
26.    public Battle_Menu_Control battleMenu;

```

```
27. public PlayerControl player;
28. public Vector3 stringPos, onPos, offPos;
29. public float stringYDown = 8.36f;
30. public float stringYUp = 14.2f;
31. public float fallingY;
32. public bool stringFall, stringRise;
33. public bool downCheck, upCheck;
34. public bool keyDown;
35. public int audioCounter;
36. public int count;
37. public bool success;
38. public bool fail;
39. public bool buttonFall;
40. public float buttonHeight;
41.
42. public AudioSource sfxSource;
43. public AudioClip failClip;
44. public List<AudioClip> successSound = new List<AudioClip>();
45.
46. // Start is called before the first frame update
47. void Start()
48. {
49.     sfxSource = GetComponent<AudioSource>();
50.     strings.Add(string1);
51.     strings.Add(string2);
52.     strings.Add(string3);
53.     strings.Add(string4);
54.     buttonOffs.Add(buttonOff1);
55.     buttonOffs.Add(buttonOff2);
56.     buttonOffs.Add(buttonOff3);
57.     buttonOffs.Add(buttonOff4);
58.     buttonOns.Add(buttonOn1);
59.     buttonOns.Add(buttonOn2);
60.     buttonOns.Add(buttonOn3);
61.     buttonOns.Add(buttonOn4);
62.     buttonHeight = 13;
63.
64.     stringPos[1] = stringYUp;
65.     stringPos[2] = string1.transform.position[2];
66.     offPos = new Vector3(0, stringPos[1] - offset,
buttonOn1.transform.position[2]);
67.     onPos = new Vector3(0, stringPos[1], buttonOff1.transform.position[2]);
68.
69.     for (int i = 0; i < 4; i++)
70.     {
71.         stringPos[0] = battleMenu.enemyPlaces[i];
```

```

72.         strings[i].transform.position = stringPos;
73.         onPos[0] = stringPos[0];
74.         offPos[0] = stringPos[0];
75.         buttonOffs[i].transform.position = offPos;
76.         buttonOns[i].transform.position = onPos;
77.     }
78. }
79.
80.
81.
82. void FixedUpdate()
83. {
84.
85.     count = battleMenu.specialStateCount;
86.     if (battleMenu.specialStateCount == 1 && downCheck == false) stringFall = true;
87.     if (stringFall == true) Fall();
88.     if (count == 3)
89.     {
90.         currentBeat = buttonOns[UnityEngine.Random.Range(0, 4)];
91.         battleMenu.specialStateCount++;
92.         fallingY = buttonHeight;
93.     }
94.     if (count > 3 && count < 12)
95.     {
96.         float time = player.timeCounter;
97.         Vector4 beats = player.beats;
98.         if (((time >= 0 && time < beats[0] + player.leniency) ||
99.             (time >= beats[1] && time < beats[2] + player.leniency)) && success ==
100. false) fallingY = fallingY - .2f;
101.         currentBeat.transform.position = new
102. Vector3(currentBeat.transform.position[0], fallingY, onPos[2]);
103.         if ((time > beats[0] + player.leniency && time < beats[1])
104.             || (time > beats[2] + player.leniency)) success = false;
105.         if (success == false &&
106.             ((time > beats[0] - player.leniency && time < beats[0] +
107. player.leniency) ||
108.              (time > beats[2] - player.leniency && time < beats[2] +
109. player.leniency)))
110.         {
111.             fail = true;
112.             CheckSuccess();
113.         }
114.         else if (fail == true){
115.             currentBeat.transform.position = new
116. Vector3(currentBeat.transform.position[0], stringYUp,
117. currentBeat.transform.position[2]);

```

```
112.         fallingY = 12;
113.         sfxSource.PlayOneShot(failClip);
114.         audioCounter++;
115.         battleMenu.specialStateCount++;
116.         if (audioCounter > 3) audioCounter = 0;
117.         fail = false;
118.         currentBeat = buttonOns[UnityEngine.Random.Range(0, 4)];
119.     }
120.
121. }
122. else if (count == 13){
123.     if (upCheck == false) stringRise = true;
124.     if (stringRise == true) Rise();
125.     if (upCheck == true) battleMenu.specialStateCount++;
126. }
127.
128.
129. for (int i = 0; i < 4; i++)
130. {
131.     stringPos[0] = battleMenu.enemyPlaces[i];
132.     strings[i].transform.position = stringPos;
133.     offPos[1] = stringPos[1] - offset;
134.     buttonOffs[i].transform.position = new
Vector3(buttonOffs[i].transform.position[0], offPos[1], offPos[2]);
135. }
136.
137. }
138.
139. void Update(){
140.
141.     print(player.specialSuccessCount);
142.     if (Input.GetKeyDown(KeyCode.Z) && keyDown == false)
143.     {
144.         TurnColor(buttonOffs[0], "red");
145.         keyDown = true;
146.     }
147.     else if (Input.GetKeyDown(KeyCode.X) && keyDown == false)
148.     {
149.         TurnColor(buttonOffs[1], "red");
150.         keyDown = true;
151.     }
152.     else if (Input.GetKeyDown(KeyCode.C) && keyDown == false)
153.     {
154.         TurnColor(buttonOffs[2], "red");
155.         keyDown = true;
156.     }
```

```
157.         else if (Input.GetKeyDown(KeyCode.V) && keyDown == false)
158.         {
159.             TurnColor(buttonOffs[3], "red");
160.             keyDown = true;
161.         }
162.
163.         if (Input.GetKeyUp(KeyCode.Z))
164.         {
165.             TurnColor(buttonOffs[0], "white");
166.             keyDown = false;
167.         }
168.         else if (Input.GetKeyUp(KeyCode.X))
169.         {
170.             TurnColor(buttonOffs[1], "white");
171.             keyDown = false;
172.         }
173.         else if (Input.GetKeyUp(KeyCode.C))
174.         {
175.             TurnColor(buttonOffs[2], "white");
176.             keyDown = false;
177.         }
178.         else if (Input.GetKeyUp(KeyCode.V))
179.         {
180.             TurnColor(buttonOffs[3], "white");
181.             keyDown = false;
182.         }
183.     }
184.
185.     public void Fall()
186.     {
187.         if (string1.transform.position[1] >= stringYDown) stringPos[1] =
188.             stringPos[1] - .2f;
189.         else
190.         {
191.             stringPos[1] = stringYDown;
192.             stringFall = false;
193.             downCheck = true;
194.             upCheck = false;
195.         }
196.     }
197.     public void Rise()
198.     {
199.         if (string1.transform.position[1] <= stringYUp) stringPos[1] = stringPos[1]
200.             + .2f;
201.         else
```

```
201.     {
202.         stringPos[1] = stringYUp;
203.         stringRise = false;
204.         upCheck = true;
205.         downCheck = false;
206.     }
207. }
208.
209. public void CheckSuccess()
210. {
211.     if ((Input.GetKeyDown(KeyCode.Z) && currentBeat == buttonOns[0] && keyDown
212. == false) ||
213.         (Input.GetKeyDown(KeyCode.X) && currentBeat == buttonOns[1] && keyDown
214. == false) ||
215.         (Input.GetKeyDown(KeyCode.C) && currentBeat == buttonOns[2] && keyDown
216. == false) ||
217.         (Input.GetKeyDown(KeyCode.V) && currentBeat == buttonOns[3] && keyDown
218. == false))
219.     {
220.         currentBeat.transform.position = new
221. Vector3(currentBeat.transform.position[0], stringYUp,
222. currentBeat.transform.position[2]);
223.         fallingY = buttonHeight;
224.         sfxSource.PlayOneShot(successSound[audioCounter]);
225.         audioCounter++;
226.         battleMenu.specialStateCount++;
227.         if (audioCounter > 3) audioCounter = 0;
228.         success = true;
229.         fail = false;
230.         currentBeat = buttonOns[UnityEngine.Random.Range(0, 4)];
231.         player.specialSuccessCount++;
232.     }
233. }
234.
235. public void TurnColor(GameObject button, string color){
236.     SpriteRenderer spriteRenderer = button.GetComponent<SpriteRenderer>();
237.     if (color == "red") spriteRenderer.color = new Vector4(1, 0, 0, 1);
238.     if (color == "white") spriteRenderer.color = new Vector4(1, 1, 1, 1);
239. }
```

Tutorial Cutscene Guide

Throughout the script for the tutorial battle, there are quite a few steps for completing the scene, all controlled by the variable "cutscene counter". This document is to function as a glossary for the variable, and to detail what happens at each value. The value always increases by one unless it's otherwise specified.

Value of "Cutscene Counter"	What class it operates in	How it operates
0	DialogueTrigger	Controls the first section of dialogue. Variable is increased in DialogueManager's "EndDialogue" function.
1	Instructor	makes the instructor move to enemy position one, then increases the counter
2	DialogueManager	Controls dialogue
3	Instructor	The instructor moves to enemy position four
4	DialogueManager	Controls dialogue
5	Instructor	The instructor goes offstage in search of the dummy
6	DialogueManager	Controls dialogue
7	Instructor	The instructor returns onstage with the dummy, and takes it to enemy position one.
8	DialogueManager	Controls dialogue
9	PlayerControl	Triggers the player animator, having him pick up his guitar.
10	Instructor	Changes the music to part 2 of the tutorial theme and triggers the next dialogue
11	DialogueManager	controls dialogue
12	Battle_Menu_Control	Makes the battle menu appear
13	Instructor	Triggers the next dialogue
14	DialogueManager	controls dialogue
15	Spot	makes the spotlight appear over the attack button
16	Instructor	Triggers the next dialogue
17	DialogueManager	controls dialogue
18	Spot	makes the spotlight disappear

19	Battle_Menu_Control	Lets the player control the battle menu for the first time. Only increases when the player selects attack.
20	Instructor	manages the next bit of dialogue
21	DialogueManager	controls dialogue
22	Player	Active while the player performs their first attack
23	Instructor	manages the next bit of dialogue. Different depending on whether the last attack hit or missed
24	DialogueManager	controls dialogue
25	Instructor	Instructor picks up the dummy, using both the instructor and dummy's animators
26	Instructor	Instructor takes the dummy to enemy position 4
27	Instructor	handle drops and dummy's exact position is set
28	DialogueManager	controls dialogue 11
29	Battle_Menu_Control	lets the player manipulate the menu for the second attack
30	Player	Active while the player performs their second attack.
31	Instructor	controls dialogue. Different depending on attack success
32	Dialogue Manager	controls dialogues 12 and 13
33	Instructor	sets the counter back to 29 if the last attack was a failure.
34	Instructor	does the same thing as count 25
35	Instructor	Similar to count 26, the instructor picks up the dummy and takes it to spot 3
36	Instructor	Drops the dummy's handle and triggers dialogue 14
37	Dialogue Manager	controls dialogue 14
38	Battle_Menu_Control	lets the player manipulate the menu for the third attack
39	Player	Active while the player performs the third attack.
40	Instructor	Triggers either dialogue 15, 16, or 17 depending on the player's action
41	Dialogue Manager	controls dialogues 15, 16, and 17
42	Instructor	Sends the player back to step 38 if they didn't do the last attack correctly
43	Instructor	same step as 25 and 34

44	Instructor	Similar to 35, the instructor picks up the dummy and takes it to the enemy attack position
45	Instructor	The instructor moves into position to kick the dummy towards the player.
46 and 47	Instructor	this step is run twice because of issues with the animator. In order to set "kick" to the proper animation state, I needed to run it twice, as it doesn't set it properly the first time. The instructor doesn't start the kick until the third beat.
48	Instructor	Makes the dummy start its attack, but only after the Instructor's kick animation is finished. During this step, the dummy slows down slightly every beat.
49 and 50	Instructor	Manages when the dummy's hit is active. When it first activates, it increases the timer to 51, then increases again when it deactivates.
51	Instructor	If the player fails to dodge, it runs dialogue 18. If the player does, it runs dialogue 19
52	Dialogue Manager	controls dialogues 18 and 19
53	Instructor	the dummy comes back to the enemy attack position and sets up dialogue 20
54	Dialogue Manager	controls dialogue 20
55	Instructor	has the instructor wait until beat 3 before kicking the dummy again
56, 57, 58, 59, and 60	Instructor	the same as 46, 47, 48, 49, and 50
71	Battle Menu Control	same as 12
72	Instructor	triggers dialogue 25
73	Dialogue Manager	controls dialogue 25
74	Spot	Makes the spotlight appear over the special button
75	Instructor	triggers dialogue 26
76	Dialogue Manager	controls dialogue 26
77	Spot	Makes the spotlight disappear
78	Battle Menu Control	same as 19. Also manages the entire special attack
79	Instructor	triggers either dialogue 27, 28, 29, or 30
80	Dialogue Manager	controls either 27, 28, 29, or 30

Chapter VII - Bibliography

- Digital Game Museum. "Rhythm Games." *Digital Game Museum*, 2018, www.digitalgamemuseum.org/rhythm/. Accessed 18 Nov. 2019.
- "East and West, Warrior and Quest: A Dragon Quest Retrospective." *Iup.com*, IGN Entertainment, 2011, web.archive.org/web/20121108214936/http://www.1up.com/features/dragon-quest-retrospective?pager.offset=1. Accessed 19 Nov. 2019.
- Edge Staff. "Guitar Hero Breaks \$1 bln." *Edge*, Future Publishing Limited, 21 Jan. 2008, archive.ph/20120906082152/http://www.next-gen.biz/news/guitar-hero-breaks-1-bln. Accessed 18 Nov. 2019.
- Metacritic*. CBS Interactive, www.metacritic.com/. Accessed 18 Nov. 2019.
- "1997 Top 30 Best Selling Japanese Console Games." *The-Magicbox*, 2008, the-magicbox.com/Chart-BestSell1997.shtml. Accessed 18 Nov. 2019.
- Nutt, Cristian. "Ubisoft: Just Dance Passes 4 Million, Katy Perry Joining the Party." *Gamasutra*, Informa, 6 Oct. 2010, www.gamasutra.com/view/news/121534/Ubisoft_Just_Dance_Passes_4_Million_santi%20house%20friday%20approved%20Katy_Perry_Joining_The_Party.php. Accessed 18 Nov. 2019.
- "Rock Band® Franchise Officially Surpasses \$1 Billion Dollars in North American Retail Sales, According to the NPD Group Over 40 Million Paid Individual songs Via Download to Date On Rock Band® Platform." *RockBand.com*, MTVnetworks, 26 Mar. 2009, web.archive.org/web/20090328174823/http://www.rockband.com/news/one_billion_dollars. Accessed 18 Nov. 2019.
- Smithsonian National Museum of American History. "Simon Electronic Game, 1978." *Smithsonian National Museum of American History*, Smithsonian, americanhistory.si.edu/collections/search/object/nmah_1302005. Accessed 18 Nov. 2019.
- Van Zandt, Steven. "Alex Rigopoulos & Eran Egozy." *Time*, Time USA, 12 May 2008, content.time.com/time/specials/2007/article/0,28804,1733748_1733752_1735901,00.html. Accessed 18 Nov. 2019.
- Vestal, Andrew. "The History of Console RPGs." *Gamespot*, CBS Interactive, web.archive.org/web/20090802000558/http://www.gamespot.com/features/vgs/universal/rpg_hs/index.html. Accessed 19 Nov. 2019.
- Walker, John. "How Thumper Made a Lot of People Very Uncomfortable." *Rock Paper Shotgun*, Gamer Network, 9 Mar. 2015, www.rockpapershotgun.com/2015/03/09/how-thumper-made-a-lot-of-people-very-uncomfortable/.

Image Credits

1. Taylor, Wilton. *Simon Says Memory Game*. 28 Sept. 2012. *flickr*, SmugMug, 10 Feb. 2004, https://live.staticflickr.com/8322/8050771801_87b4e448a0_b.jpg. Accessed 8 Dec. 2019.
2. Tmkn. *Beatmania IIDX controller for PlayStation*. 3 Dec. 2006. *Wikimedia Commons*, Wikimedia Foundation, 19 Nov. 2019, https://upload.wikimedia.org/wikipedia/commons/e/e7/Beatmania_IIDX_controller_for_PlayStation.jpg. Accessed 8 Dec. 2019.
3. "Parappa the Rapper (PSX) - Perfect All Stages Playthrough (Tool-Assisted) by Sabih." *YouTube*, uploaded by DarkFulgore, 5 June 2015, www.youtube.com/watch?v=IbYdRZ8FiuA. Accessed 8 Dec. 2019.
4. RadioActive. *Dance Dance Revolution Solo Bass Mix arcade machine stage*. 15 Mar. 2005. *Wikimedia Commons*, Wikimedia Foundation, 19 Nov. 2019, https://upload.wikimedia.org/wikipedia/commons/8/88/Dance_Dance_Revolution_Solo_Bass_Mix_arcade_machine_stage.jpg. Accessed 8 Dec. 2019.
5. "Dance Dance Revolution - Butterfly." *YouTube*, uploaded by Xortex Estados Unidos, 4 Sept. 2012, www.youtube.com/watch?v=4tsDb3CeFp4. Accessed 8 Dec. 2019.
6. Y2kcrazyjoker4. *Guitar Hero series controllers*. 31 July 2007. *Wikimedia Commons*, Wikimedia Foundation, 19 Nov. 2019, https://upload.wikimedia.org/wikipedia/commons/8/80/Guitar_Hero_series_controllers.jpg. Accessed 8 Dec. 2019.
7. "Guitar Hero (PS2 Gameplay)." *YouTube*, uploaded by GXZ95, 20 June 2012, www.youtube.com/watch?v=BVyWcUHPWUU. Accessed 8 Dec. 2019.
8. "[Rhythm Heaven Fever] ~ Monkey Watch (Perfect)." *YouTube*, uploaded by Pablo Acevedo, 13 Feb. 2012, www.youtube.com/watch?v=kBIg7LEHs-0. Accessed 8 Dec. 2019.
9. "Taiko no Tatsujin | Taiko Drum N Fun - 33 Minutes of English Gameplay (Nintendo Switch) [HD]." *YouTube*, uploaded by Scott E, 1 Nov. 2018, www.youtube.com/watch?v=v_CCI6PVqAQ. Accessed 8 Dec. 2019.
10. "Just Dance 2020 - Launch Trailer - Nintendo Switch." *YouTube*, uploaded by Nintendo, 7 Nov. 2019, www.youtube.com/watch?v=9BrAT_o7yWA. Accessed 8 Dec. 2019.
11. "Crypt of the Necrodancer Gameplay @1080p60fps (No Commentary)." *YouTube*, uploaded by Austin Dunn, 27 June 2016, www.youtube.com/watch?v=Rs4Sc6-PjQI&t. Accessed 8 Dec. 2019.
12. "Thumper Gameplay Level 1." *YouTube*, uploaded by SirJames I Gamerfuzion, 11 Oct. 2016, www.youtube.com/watch?v=3CDw4i09ZN4. Accessed 8 Dec. 2019.
13. "dragonstomper walkthrough." *YouTube*, uploaded by Yizhak shachar, 16 Feb. 2016, www.youtube.com/watch?v=9jdB_1iavW0. Accessed 8 Dec. 2019.
14. "Portopia Renzoku Satsujin Jiken (FC)." *YouTube*, uploaded by Nenriki86, 4 May 2011, www.youtube.com/watch?v=hcQhnJxP7qU. Accessed 8 Dec. 2019.
15. "Dragon Warrior Part 11: Metal Slime Hunting." *YouTube*, uploaded by NintendoCapriSun, 6 Jan. 2014, www.youtube.com/watch?v=3all1YM90B8. Accessed 8 Dec. 2019.

16. "DRAGON QUEST XI - English Walkthrough Part 1 - Prologue (Full Game) PS4 PRO." *YouTube*, uploaded by Shirrako, 13 Sept. 2018, www.youtube.com/watch?v=QmjSoL5y4YA. Accessed 8 Dec. 2019.
17. "Final Fantasy 1 Battle Music (FDS Remix)." *YouTube*, uploaded by DeltaDragonoid225, 4 Oct. 2015, www.youtube.com/watch?v=fOx9h0ymKHs. Accessed 8 Dec. 2019.
18. "SNES Final Fantasy VI (III US) Full Gameplay 1080p." *YouTube*, uploaded by Yoshiyukiblade, 30 Dec. 2013, www.youtube.com/watch?v=XF2cski7Q7M. Accessed 8 Dec. 2019.
19. "Paper Mario (N64) Boss Battle #1: Koopa Bros." *YouTube*, uploaded by William Dearth, 1 Feb. 2014, www.youtube.com/watch?v=oWFyHwEoN6A. Accessed 8 Dec. 2019.
20. "Paper Mario: The Thousand Year Door (Full Game Walkthrough, 10 HP challenge, Everything)." *YouTube*, uploaded by NintendoMovies, 19 Dec. 2018, www.youtube.com/watch?v=LPAbAR6guM8. Accessed 8 Dec. 2019.