ABSOLUTE: Player-Choice Determined Music in A Rhythm Game

Maggie Kraine
Northeastern University
360 Huntington Avenue
Boston, Massachusetts
kraine.e@northeastern.edu

Sage Langenfeld
Northeastern University
360 Huntington Avenue
Boston, Massachusetts
langenfeld.s@northeastern.edu

Alex Ma
Northeastern University
360 Huntington Avenue
Boston, Massachusetts
ma.al@northeastern.edu

ABSTRACT

ABSOLUTE is a rhythm game/RPG hybrid with player-choice driven gameplay mechanics & dynamic music which changes based on those choices. It's developed in the Unity Engine with WWISE integration to handle our adaptive audio. The game features overworld segments where the player can explore and rhythm game segments framed as boss battles. As a result our background research involved two different aspects, related titles' gameplay mechanics and the usage of generative and adaptive music in other games, rhythm or otherwise. Through our research we've further developed the gameplay and audio mechanics of our own rhythm game on a larger scale, and our own method for adaptive music and song design. Our song design uses horizontal restructuring and vertical layering to emphasize player-choice through audio. The resulting song of the game is a culmination of segmented song sections in which the timbre, emotion, and rhythm based off of the player's aggressive or passive choices throughout the levels.

1. INTRODUCTION

With the progression of video game technology over the years, the gaming industry has been rapidly moving towards a more immersive player experience. VR headsets, haptic feedback, and ever improving and realistic graphics are often at the forefront of conversation about the new worlds these games are building. However, game audio plays a large role in not only informing the player of gameplay and danger, but in immersing and entertaining the player in the emotions and movement in the world. With our background in Music Technology and Computer Science, one of our primary motivations in our game development is to continue to expand the intersection of gameplay and music creation. Our objective henceforth, is to create a game able to generate a composition determined by the path a player took. This path is defined by the player's choices in combat, either offensive or defensive movement. We aim to build a rhythm game with new complex mechanics, in which the music changes based on time, state, and choices. With a strong focus on sound design, we strive to develop a game that has a high replayability from the ability to generate new music based on player choice and adaptive audio.

2. BACKGROUND

For our project we wanted to develop a rhythm game that had unique rhythm mechanics and also showcase our knowledge of music technology by using generative music as well. As a result our background research involved two different aspects, related titles that have done something similar or different to what we wanted to accomplish, and about generative and adaptive music and looking at examples of it in other games, rhythm or otherwise.

2.1 Related Titles

One of the main inspirations for our project is the game Everhood [1]. This game revolutionized some of the limits of what a rhythm game could be, and is the most similar game to what we are trying to create in this project that is currently on the market. The main gameplay loop of the game is progressing through different encounters with other characters in the form of rhythm game levels. In these levels, the player is tasked to dodge the incoming notes that are played in time with the song, or absorb the notes by hitting the proper input at the right time to launch an attack at the enemy. This aligns with the gameplay loop we are trying to establish, with each level representing a specific character and the player must complete a rhythm game level in order to progress. In Everhood, most levels are completed twice, once normally and another time ending the song early by dealing enough damage to the enemy. This is still a somewhat linear approach to progression in the game, however in our game we plan on creating branching paths changes based on how the player completes the level. For this we plan on having a passive route where the player simply must complete the level normally, and a more violent route where the player deals enough damage to end the level before the song completes.

The main rhythm game mechanic in Everhood is also similar to our game, however instead of discrete lanes the player navigates over, our game will feature a slider that does not force notes or the player into specific points. In addition, the way Everhood syncs up music to notes on the screen differs from traditional rhythm games. In Everhood, the music syncs up with where notes appear, as if the character the song is for is playing notes and sending them as attacks to the player. However, in our game we plan to sync up the notes with where the player would interact with them, akin to more traditional rhythm games. Even so, the tutorial of Everhood could act as a great model for how we should tutorialize our game, with also narrative elements built into the tutorial in the character it introduces.

Another rhythm game worth looking into is the Rhythm Heaven series [2]. These games usually involve the player playing through around 50 minigames, each with their own music and mechanics. After every 4-5 normal minigames, there is a "remix" minigame which combines the mechanics of previous minigames to a new song (typically the ones immediately preceding it, however the final remix at the end of the game combines every standard minigame in fast-paced chaos). What makes this series especially interesting for us is its breadth of simple but varied rhythm mechanics with a focus on providing the player instructions through audio rather than visuals.

Making a rhythm game that is mostly played by listening shifts the player's attention towards the music rather than being glued to a stream of notes on a chart. We can group these mechanics into three categories: Simon Says, where a pattern is played to the player that they must then replicate, Audio Cues, where the player is taught to respond to certain sounds with certain rhythm patterns (typically a set of 2 or 3 cues), and Steady Patterns, where the player has a default behavior they return to (like playing backbeats) that is occasionally broken up or altered by cues. These different mechanics reflect differently in the accompanying music, like the music in simon says minigames having 7 notes of the call pattern, 1 beat rest, 7 notes of response where the player replicates the pattern, and then 1 more rest to round out the phrase. While the pitches can vary between call and response to make the repeated line more interesting & satisfying to listen to, the rhythm structure needs to uphold parity with the gameplay when the audio is intertwined. Another interesting thing to consider is how repeated rhythm patterns in the gameplay strengthen the identity of each minigame. One minigame might be well remembered for always finishing off each 4 bar phrase with two specific cues in a row. This comes into play in the game's remixes where, when cutting between different minigames mid-song, jumping into one of these "identity patterns" makes it easier for the player to recall the mechanics on the spot and can serve as a miniature representation of that minigame. In our game, we are planning on making the final gameplay segment mesh the rhythm mechanics of the previously played levels much like in a remix from Rhythm Heaven, and taking notes of what makes such an experience more understandable and enjoyable to the player is of value.

2.2 Generative and Adaptive Music

In addition to researching rhythm games to better understand the gameplay mechanics, we studied the usage of generative and adaptive music in games. We wanted to determine whether we had the time frame and technical ability to utilize generative music within our game, to help us create a highly unique experience for the player. One of the benefits of using generative music is alleviating the composer from some of the composition time that would be required to create such an amount of musical content. Spore and Doom(2016) accomplished this successfully, however, their developers also had the resources to spend ample time developing the system that generates the music. Developing a system at our level for our game would require us to utilize both Pure Data and the library libPD for it, which none of us have experience with. Besides the difficulty that may be faced in developing a system to generate the music, generative music itself doesn't always provide the best audio experience for games. "The audio director of *No Man's Sky*, Paul Weir, notes that generative music can also display the "10,000 bowls of oatmeal" problem, where the music is acceptable, but monotonous." [3] Initially we were focused on using Generative music within the game because of the ability of the player to personalize their experience. But, both the difficulty of creating the generative music system and the possible poor outcome suggest that it may be better to approach the audio from an adaptive music standpoint. We believe we can accomplish a similar effect of customization and personalization using adaptive music since we are working on a much smaller scale. Creating a music system that works adaptively with gameplay can be done through WWISE and careful composition surrounding musical transitions. Some techniques in creating adaptive music are increasing tension with an increasing rhythm, pitch, or instrumentation amount. Because our music is meant to follow the action of the game and the choices of the character to fight or pass, this system of increasing and decreasing tension will be effective for our use.

2.3 Research Conclusions

Through researching past games for both their rhythm gameplay mechanics, and the use of generative and adaptive music, we've further developed the structure of our own rhythm game on a larger scale, what makes for effective rhythm mechanics paired with audio, and what degree of adaptive music is both valuable and feasible for us.

3. PROJECT DESCRIPTION

3.1 Milestones

To organize our goals and progress we developed a plan of milestones to inform our next tasks and expansion of the project. We outlined our milestones in three sections, the Vertical Slice, Necessary Features, and Additional Features. Our targets, deliverables and timelines are defined below. All of the details of each milestone were divided on a smaller scale on Trello and design decisions were made in a separate document.

3.1.1 Vertical Slice

Our first major milestone was the Vertical Slice, which is essentially a small prototype and proof of concept of the game. We outlined our goals as creating and designing all required components for a vertical slice, a level that showcases all desired features. This slice's aim was to determine what features we want and what features are not necessary. Our deliverable for this slice was a playable build of an initial scene, with a guideline of two weeks.

3.1.2 Necessary features

This second major milestone: Necessary Features consisted of fleshing out all necessary features that are integral to gameplay or the project presentation as a contingency. The aim was to have main mechanics or features in a polished spot such that additional features can be built on top of it.The deliverable for this milestone was a complete prototype of the game with a generated composition, ready for the music portion of the demonstration. The timeframe for this milestone was 3 weeks.

3.1.3 Additional Features

This last milestone was optional, and dependent on the work completed in the last two milestones. The goals were to work on any additional features not necessary for the project presentation, mainly adding additional levels to add to narrative. But, this milestone also focused on refining the details of the game, with determining a title screen, credits screen, and other additional UI features. Our projected deliverable was a completed version of the game, a presentable product for the market. The timeframe for this milestone was 2 weeks.

Throughout the span of the project, the timeframes altered slightly as well as the length of the game. Though we had planned to create 11 levels, the pace of workflow and implementation of audio proved to be more difficult than anticipated. We shifted to accomplish the same audio and gameplay mechanics at a smaller scale of 4 levels. Some additional features that would make it a packaged product were left behind in order to focus on refining the mechanics of the game with a major focus on dynamic music creation through gameplay.

3.2 Rhythm Mechanics

3.2.1 Rhythm Design

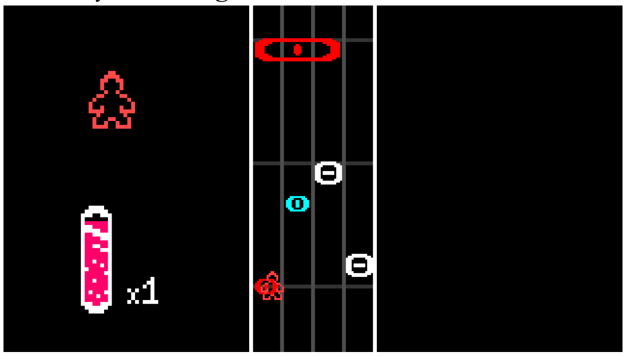
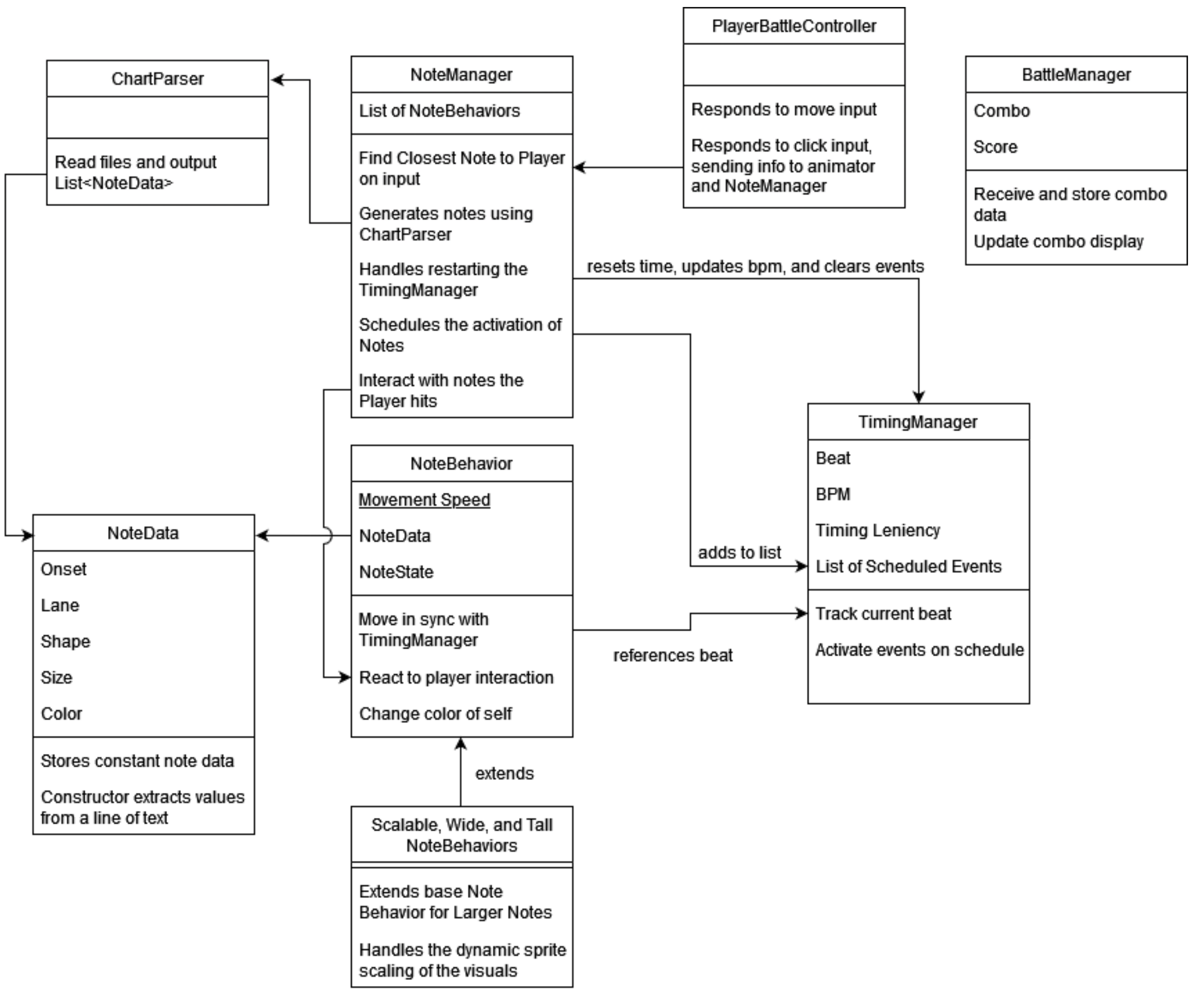


Figure 1: Battle Screenshot

Battles in ABSOLUTE play out as a rhythm game. This is structured similarly to the Guitar Hero series, where notes descend down from offscreen, appearing a few beats in advance, and then the player time their response for each note to match right when it hits a set line near the bottom of the screen. The player can control the small humanoid figure along the bottom by moving the mouse to slide it side to side within the horizontal constraints of the "staff", and can click with either the left or the right side of the mouse to cause the figure to flash red or blue. The notes that descend have various horizontal positions that ask the player to align themselves with in order to "hit" them. Once the player and note align both horizontally and vertically, the player may then click the mouse to cause the note to stop and fade away. If the input is too early or too late the note is unaffected, and if a note is allowed to progress offscreen it disappears and the player is penalized. The color red corresponds to "aggressive" attack types, which the player can do by left clicking, and the color blue corresponds to "passive" attack types. The majority of notes are the color white, meaning that the player can choose to interact with either attack type, and when hit they will change color to match. However, some notes will appear already as red or blue. These may only be interacted with using the corresponding click.

Each battle encounter comes with its own song, and to support dynamic music changes these songs are broken into subsections. For each subsection of the song, there is a corresponding "chart", a file containing all of the information used to generate the set of notes the player will interact with. The charts are designed to have notes which match significant rhythm patterns in the music, to generate interesting gameplay throughout each song.



PlayerBattleController
Responds to move input
Responds to click input, sending info to animator and NoteManager
BattleManager
Combo
Score
Receive and store combo data
Update combo display
ChartParser
Read files and output List<NoteData>
NoteManager
List of NoteBehaviors
Find Closest Note to Player on input
Generates notes using ChartParser
Handles restarting the TimingManager
Schedules the activation of Notes
Interact with notes the Player hits
resets time, updates bpm, and clears events
TimingManager
Beat
BPM
Timing Leniency
List of Scheduled Events
Track current beat
Activate events on schedule
adds to list
references beat
NoteBehavior
Movement Speed
NoteData
NoteState
Move in sync with TimingManager
React to player interaction
Change color of self
NoteData
Onset
Lane
Shape
Size
Color
Stores constant note data
Constructor extracts values from a line of text
extends
Scalable, Wide, and Tall NoteBehaviors
Extends base Note Behavior for Larger Notes
Handles the dynamic sprite scaling of the visuals

Figure 2: Block Diagram of Significant Rhythm Classes

3.2.1 Rhythm Technology

As this is a game made in Unity, the programming for the rhythm game mechanics is done is a series of C# scripts. At the center of the code structure is the TimingManager class. This class essentially operates as the metronome that other scripts can reference in order to synchronize behaviors with time, and to change the parameters of the metronome itself. While this class is primarily focused on tracking values for other classes to reference in actions, it also contains a list of scheduled actions where other classes can pass in a time and a function. This is useful as it allows the TimingManager to push updates as necessary with a singular time check each update, rather than needing each class to check its own time against the manager every update. The primary use of scheduled actions is to enable each of the note objects shortly before their time to move on screen.

Another foundational class is the NoteManager. This class handles the process of generating the notes that the player has to interact with. It parses the information for a note chart from a file of comma separated values, with each line representing one note. The values in each line respectively correspond to the timing of the note in beats, the horizontal position of the note, the type of note, for note types with variable sizes the size of the note itself, and finally the color of the note. The note manager uses each of these values to pick a prefab of the correct note type, generate a copy of it in the scene, and then assign the appropriate data to that note. It also schedules the activation of the note object in the timing manager, as notes don't need to be computed until it's time for them to come on the screen. Each note has an attached NoteBehavior component which handles shared behavior like moving towards the bottom of the screen in time and responding to being hit, and can be extended to support additional behavior like scaling the visuals of variable size notes.

The player sprite has its own script for behavior, PlayerBattleController. This handles moving the player back and forth based on the mouse position, and responding to the player clicking the mouse buttons. On a click, first the PlayerBattleController will call a function in the NoteManager which checks all of the active notes and finds the one that the player is closest to in the time dimension and also lined up with in the horizontal position. This function automatically tells the note, if appropriate, to react to being hit by the player, and the player then reports its success to the BattleManager, a class for storing things like the current combo (i.e. the length of the current chain of successfully hit notes, resetting to zero on both mis-input and inaction).

3.3 Sound Design

The Sound Design of our project consists of a few different technologies and methods. We outlined the artistic elements and goals in tangent with the technology of horizontal restructuring in Song Design. In WWISE implementation we describe the process of creating vertical layering, and unique spatial audio decisions for our game.

3.3.1 Song Design

We want to use Horizontal Resequencing in rhythm battle segments to show a sense of progression as well as keeping each iteration of playing through a song different and fresh. To do so, each song will have a sort of flowchart that depicts how the game will choose different sound modules to play.

The general flow of a song in our game is Introduction, Body, Outro. The introduction section plays once before branching out into the body portion. The body section is fairly open ended and can be broken up into multiple sub sections. This section contains the core motifs of the song, with as many subsections as the composer desires. This section will change dynamically with game states the most. For example a song could have 3 subsections. The first subsection will play after the intro. Then, it will transition to the 2nd subsection, with again 3 variations that play based off the first section. Finally the 3rd subsection follows that has changed based on if the player has dealt enough damage or not.

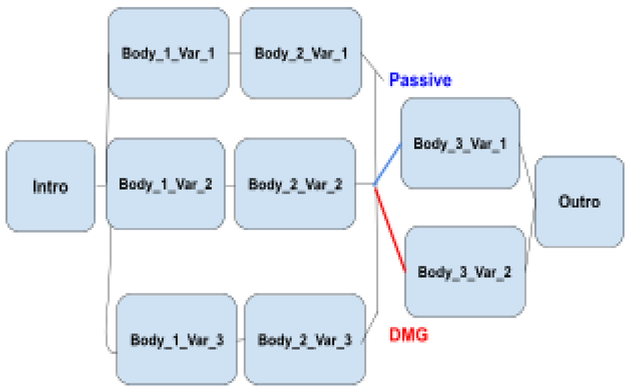


Figure 3: Example Song Sections

Segments may be looped depending on how the song is designed. Additional subsections such as bridges/ small interludes can be freely added. The Outro is the final section of the piece, similar to the introduction but can have multiple variations as well.

The tutorial level [ID:00] song flow is outlined in the figure below.

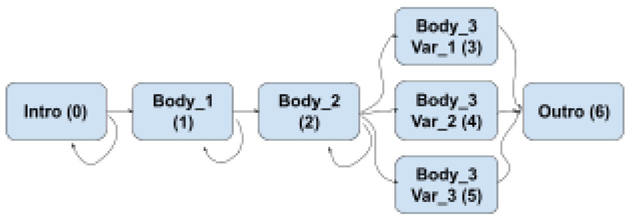


Figure 4: [ID:00] Tutorial Song Flow

The introduction of this level is a quick section to the song, teaching the player basic note hitting. If the player does not perform well enough the section loops. Body_1 is similar to the introduction, and teaches the player other note types. This section also loops if the player does not perform well enough. Body_2 teaches the player about different attack types [aggressive/passive] and will loop if the player does not perform well enough. Body_3 has 3 variations that play randomly if the player completes Body_2 successfully. From here on out, the player can fail. This is a longer section more representative of actual gameplay. The Outro is an outro sequence for the song, alluding back to the intro and bringing the song to a close. This section may be varied based on the player's actions in Body_3. The main motif in this tutorial song will be a thematic motif representative of the player that returns in later parts of the soundtrack.

Level 4 [ID: 03] song flow is outlined in the figure below. This level is unique in that it incorporates the players previous actions when determining which segment to play.

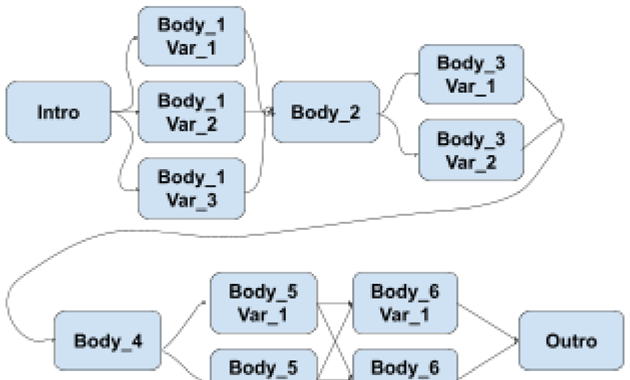


Figure 5: [ID: 03] Level 4 Song Flow

Body_1 is based on the random selection in Level 1. Body_2 is always the same. Body_3 is based on if the player was majorly peaceful or aggressive. Body_4 is used if a player finds a certain flag check interaction. Body_5 is based on if the player was peaceful or aggressive in Level 2. Body_6 is based on if the player was peaceful or aggressive in Level 3. These segments will also have the musical characteristics associated with the relevant stage where it comes from. For instance, Body_5 would call back to the song of Level 2, while Body_6 would call back to the song of Level 3. This behavior will be achieved through the GameManager class, which persists throughout the game and will keep track of major flags the player has reached, such as how they completed certain levels. This will also generate a final complete song for the player to have that reflects the final song in their journey.

3.3.2 Musical Song Design

To maintain a unified musical theme and feel throughout the game with 3 different composers, we outlined some musical restrictions for composition. A main goal for song composition because it is a rhythm based game is to have a strong beat somewhere in the song to help the player internalize the beat.

To preserve a state of changing emotion and personality we will use a leitmotif for the player to connect to specific characters and themes. This will be most prevalent in Level 1 and 4, but may have some connection to Level 2, as the boss in this level is a sibling of the player.

To emphasize the difference between passive/aggressive plays, we will utilize Note Density, Instrumentation, Key, and Transitions as tools to distinguish these attitudes. For passive characters and decisions we will have less note density, less harsh instrumentations(violin, piano, mellow percussion), Major key, and normal transitions. For aggressive characters and decisions we will utilize higher note density, harsher instrumentation(electronic instruments, louder percussion, brass), Minor key or non Major keys, and sudden transitions and strange cadences.

3.3.3 WWISE Implementation

This section will go over how we are implementing dynamic songs into the game specifically. There are several ways to go about this, so this section will make clear which method we are using and why we chose this method.

We will be using WWise callbacks on events in order to determine when a specific segment ends, as each segment will play out in its entirety. We will be using the AK_EndOfEvent and AK_Marker. AK_EndOfEvent tells Unity when an event has finished, and AK_Marker tells Unity when a specific marker that is defined in WWise has been reached. We choose these callbacks because AK_EndOfEvent lets us play the next segment of a song immediately after one ends, and AK_Marker to load in the next beatmap a measure before the song ends so that the transition in game is seamless for the player. These callbacks are all called from the SongManager class in Unity, which handles the playing of audio files from WWise which are related to the rhythm battle songs specifically.

We then define a specific Callback Function for each song that will be triggered when the Callback is reached. These vary from song to song based off of the Song Flows outlined in Figures 4 and 5 above. However, each of these functions will change the state of WWise which will change the current segment being played. On the WWise side, a Switch container is used to house all of the segments of a single song, which each segment linked to a different state. This container will be set to use a State group and its Play Mode will be Step.

A strict naming convention is used to make sure all variables are linked properly between WWise and Unity. The naming convention used is SongName_ID, and allows for future extension if desired. Segments will be named using a similar convention of SongName_ID_SegmentID.

Each song is stored in unity as a unique scriptable object to keep track of specific details of each song, such as its ID and its beats per minute. A scriptable object is used to make sure these values are global, and these scriptable objects will each have their own callback functions in order to produce the desired effects.

3.3.4 WWISE Sound Design

We also utilized WWISE itself to further design the sounds. Outside of DAWS, the WWISE software can be utilized to randomize and spatialize different sound effects within Unity to create a more realistic and dynamic world.

To create the sound of footsteps, within WWISE we created a random container that holds a sound clip of a footstep. This random container is set to start and stop a loop based on the character's stepping animation. The random container's parameters are set to randomize the pitch and a low pass filter between a range of values every time the sound repeats, creating a dynamic sense of movement. This allows the player to be further immersed within the overworld layer.

The overworld layer also contains a door to the rhythm battle that has a sound attached. The door is attached to the song container of the approaching battle. As the character 's distance to the door decreases, the volume increases and more of the pitches are heard as the filter is decreased with proximity. This effect creates a sense of change and danger in the player to alert them of an upcoming battle.

Using WWISE's built-in features we were able to make the world more immersive and informative with less sounds.

4. CONCLUSIONS AND FUTURE WORK

Through this process we determined a method both in concept and code of creating a game that player-choice determines the outcome of a song. The game mechanics and the player control the song dynamics through choice. We wanted to create a game with unique gameplay and sound design, that gave the player the ability to control their own "destiny".

We also determined that there were many ways of implementing such a system, with benefits and drawbacks that came with each method. Overall, these systems can become quite intricate and complex, so good documentation and planning can help the design process immensely. With that said there is a lot of room for more exploration and unique uses of dynamic audio.

In the future, we hope to expand this project with more time. The implementation process and design process took longer than anticipated, and we drafted levels that we could not execute by the deadline. We would also like to explore new and different gameplay mechanics in the rhythm battle, to further understand the player's interactions with the game.

5. ACKNOWLEDGMENTS

Thank you to Professor Victor Zappi for his continual support and guidance throughout the project.

6. REFERENCES

[1] Foreign Gnomes, Everhood, https://store.steampowered.com/app/1229380/Everhood/

[2] Nintendo, Rhythm Heaven (series), 2006-2016

[3] Cale Plut & Phillippe Pasquier, Generative music in video games: State of the art, challenges, and prospects https://www.sciencedirect.com/science/article/pii/S1875952119300795?via=ihub#