Dynamic Music Composition in Video Games Using Pure Data

Tomo Buchberg, Tristen Khatibi, Keith Choi, Roberto Loya, Ahrin Meguerian, Austin Lucky*
{tomo.buchberg.330,tristen.khatibi.9,keith.choi.850,roberto.loya.852,ahrin.meguerian.355,austin.lucky.137}@my.csun.edu
Department of Computer Science
California State University, Northridge
Northridge, California

ABSTRACT

Our research addresses the question: is it possible to dynamically compose music for a video game in real-time? A critical area of video game development which is often neglected is sound design and music, although it is one of the most important elements when it comes to user experience in games. By algorithmically composing music in real time, we hope to enhance the player's experience and the replayability of our game. However, a common problem with algorithmically composed music is it often sounds mechanical and uninteresting rather than engaging and emotional.

We propose an adaptive approach for algorithmic music composition which avoids the repetitiveness of traditional linear game music and unemotional feeling of common algorithmic music, by striking a balance between human interaction and procedural generation. We drew inspiration from aleatoric music, where chance events and the performer's (in our case, the player's) decisions determine some elements of the composition. We found that by exploring this approach, not only was it feasible, but it greatly enhanced the interactivity of our game.

KEYWORDS

dynamic music composition, pure data, sequencer, video game music, aleatoric music, algorithmic music

1 INTRODUCTION

The goal of video game music is to engage and entertain the player. We chose to dynamically compose music in real-time in order to enhance the player's experience and the replayability of our game. Adaptive audio improves a player's experience by closely mirroring the events occurring in-game, such as heightened moments of suspense. It can be advantageous to use adaptive audio over static or linear audio [7].

In order to achieve immersive and compelling gameplay, video game developers often use audio middleware such as Wwise and FMOD in order to create adaptive sound effects and music for video games. However, it is uncommon for developers to utilize systems which compose and generate music entirely in real-time.

The focus of our project was to create an effective system for adaptive real-time music composition which enhances the player's overall experience. Our game uses a novel approach to procedurally generated music which tightly couples game events with music composition. This adaptive real-time approach allows users to have a unique and interesting experience on every play-through.

*All authors contributed equally to this research.

COMP491 Senior Design Project, California State University, Northridge,

Our video game was developed using the open source game engine Godot. Godot's lightweight engine allows us to simultaneously perform CPU-heavy audio processing in the background while running our game. The music is generated using the audio program, Pure Data. Within our game, state machines and signals update the 'state' of our Pure Data sequencer. The game is separated into various "areas," each with their own musical ambience. The player enters new areas by solving puzzles and unlocking doors, helping define gameplay and music progression and contributing to a feeling of exploration.

The rest of this paper is organized as following: Section 2 introduces similar works performed in this field of study, Section 3 discusses the framework, methodologies, and considerations that will be used for this project, and finally Section 4 illustrates our planned project's target timeline.

2 RELATED WORKS

When deciding that dynamically composed music in video games is a field we would like to further explore, we knew as a group having to do tons of research on the topic itself and ones surrounding it would be crucial to how achievable our goals would be. Throughout our research, there are categories at which our researched works can be divided into: The Evolution of Machines in Video Game Music, The Use of Neural Networks, Procedural Generation, and Real Time Audio Synthesis. Using these distinctions, we will be able to mark pinpoint moments that will help us in learning about what is possible and what possible errors or improvements can be done on the subject.

2.1 The Evolution of Machines in Video Game Music

Since the invention of the video game, machines have always been important to the rule the music plays. Traditionally, any music or extra sound is previously composed before being insert into the machine. Now, as time goes on, cultures, and deeper understanding of technology, we see a bigger incorporation of the use of machines in music. For example, there are two forms of music structures that are very common practices for video games: Linear and Adaptive. In a researched work published in 2020, the authors describe the linear music structure saying that the music itself begins playing when its associated level is loaded in. When the music reaches the end, it loops. When a new level is loaded, the music either abruptly changes or quickly fades out, and is replaced with the new level's associated music [7]. The linear music system can be paired with all of the classic video games you know including, Super Mario Bros., Legend of Zelda, Sonic the Hedgehog, etc.. The implementation of the system overall is very easy to both program and execute. We

all know that the soundtracks to these games are some of the most popular in video game history, but many games with this structure can miss the what emotional connections that the music can have with the gameplay. When directly compared to the linear music structure, the adaptive music structure is able to do more and allows the computer to be more involved with the music production. Along with the properties of the linear structure, an adaptive structure includes effects and triggers on different variables within a given level. In the same source as the paragraph above, it talks about changing musical features can include adding or removing instrumental layers, changing the tempo, adding or removing processing, changing the pitch content, etc. The adaptivity of music can be seen as an extra layer on top of the linear structure. Low levels of adaptivity may only adapt to a small set of in-game variables, while higher levels of adaptivity may adapt to tens or hundreds of in- game variables [7]. This makes the adaptive structure a better structure when wanting to compose dynamically as players would be exposed to non repetitive and emotion connecting music, which contributes to a better overall gameplay experience [3]. There multiple ways a adaptive structure can be used as well as a bunch of different algorithms it can be paired with. The following sections will address some of these findings.

2.2 The Use of Neural Networks

Neural networks can get implemented within a video game and more crucially the adaptive music structure in many different ways. Our researched helped in finding both of these possibilities and gave us the knowledge about the power that this technique can have within a video game. Here we will look at convolutional neural networks (CNN's) and artificial neural networks (ANN's).

2.2.1 Convolutional Neural Network (CNN). In a modern video game, high quality audio alerts is truly an important piece of information for the gameplay. Researchers were curious as to what kind of sounds are related to immersion and which are best suited for gameplay. The use of a CNN to train on raw audio data along with a Generative Adversarial Network or GAN is one technique that we found.[5]. They built off of models that already existed and that are quite popular, one being "Text to Speech", essentially adapting it for Audio Synthesis. In the end, the sounds were to be generate by the neural network. Even though the researchers weren't tied to a specific goal, a common theme about the audio samples emerged. The generated sounds did not exist in reality, while their quality is solely subjective to many who hear them. Even with this being the case, the researchers limited themselves to a single restriction. All data from the training sets involved files contain multiple different types of audio sounds (e.g. a car horn and the sound of traffic). This would overall lead to the combinations of multiple sounds rather than a truly unique sound.

2.2.2 Artificial Neural Network (ANN). Artificial neural networks (ANN's), are models that are meant to simulate the systems of neurons that make up a human brain so that a computer could be able to learn and make decisions in a human like manner. Often, they are used to map out how objects and game sounds behave. They can be trained using neuroevolution of augmented technologies(NEAT). The networks allow for a diverse experience because of the amount of possibilities that can be produced with the same minimal information. The use of ANNs in the gaming industry today can be see between the interactions between a user/player and NPC (Non Player Characters) as well as procedural content generation. We found research in a 2014 paper[2] which had done extensive research on the possibilities and wide variety of mappings that become available with and without using neural networks. Their researched worked specifically with guided and movable weapons and different frequencies that were paired to certain outcomes. At the conclusion of their article the expressed the great immersive qualities this type of work can have on a video game even if the overall recognition of the work may be lower than expectations.

2.3 Procedural Generation

Procedural generation (PG) or also referred to as procedural content generation (PCG) is content which has been created by a set of algorithms. Developing software and tools takes these algorithms and generates content based on any amount of parameters. This technique is commonly found in a video game genre known as "Rogue-Likes" or "Rogue-Lites". This sub genre of Video Games relies on PCG in which that all new plays are generally different every time you play. Procedural generated content in video games is to give the player the feeling that at any instance, their experience is different than the last time they played. PCG has gotten very popular and below we explored what potential uses that already exist and have been thought of.

2.3.1 Experience Based Procedural Content Generation. In a research article from 2012, the researchers explored creating dynamic music in video games which adapts to the player. According to the paper, procedural music is "composition that evolves in real time according to … control logics"[6]. PCG can employ adaptive or "evolutionary" algorithms to generate content. These algorithms used a fitness function that evaluates whether created content is appropriate. This process iterates over many generations where each generation introduces mutated content and discards unwanted content. This overall process is very similar to how a neural network model is trained. A collection of user ratings were then used for the study's adaptive algorithm. The overall research showed a larger appreciation and emotional connections once the algorithm was appropriately integrated.

2.3.2 General Use of PCG in Video Games. Being implemented into video games for at least 2 decades, with some examples of the massively successful indie game *Minecraft* (2011) to older titles like *Dwarf Fortress* (2006), PCG is starting to dominate the video game industry. In a piece of work by Sebastian Risi and Julian Togelius, [9] the two give a brief history of the implementations of PCG in video games along with looking at higher level techniques and how they can be implemented to increase generality of PCG. They wanted to try to apply higher level programming techniques such as Machine Learning and AI to better apply PCG in Video Games. They bring up searched-based AI and how it can be implemented to large tasks such as creating maps [9]. Knowing the current state of PCG and what is already being created is eye opening and is knowledge that none of us knew previously before going into this research.

2.4 Real Time Audio Synthesis

When making a game, a developer must find the best balance between desired effect, computational cost, storage cost, and performance. Real-time audio synthesis offers to cut down on storage cost by producing sounds as a program runs, rather than playing prerecorded sound files which may take up a lot of space on a disk. It also offers variability which prerecorded audio does not always have. Variability is of great interest to the player, whose motivation is to be entertained and excited by the game. On the other hand, real-time audio synthesis, as well as "mixing," the manipulation of audio data, can be computationally expensive.

2.4.1 Processing Power. When finding balance in regards to too much or too little input, groups of researchers of sound creation [5] and of video game learning agents [4] had a common theme. Both parties processed a lot of sounds that went in and come out of their system and in return having to find the exact balance between these sound was a big restriction. Based on this, all researchers needed to find the largest point where the learning agent wasn't overworking. More specifically, [5] had over 5000 hours of audio samples to get through in their training data set. When looking at either situation, you are able to feel the frustration that too many sounds can have on a development team. If a game is not able to run properly solely based on the amounts of sound files loaded into the system, the game as a whole must be reanalyzed and examined to figure a new storage and usage method.

2.4.2 Physics and Perception. A paper led by Raghuvanshi[8] proposed the use of physics-based sound synthesis in video games, as opposed to using traditional recorded audio for sound effects. This system of sound generation required a physics engine which informed a sound system of exact collision geometry and forces involved in the scene. It also required significant computational resources, but this was reduced using methods mentioned in the study. "Mode compression" is the method of replacing multiple frequencies close to each other with a single one representing all of them. This saved on computation because mixing one frequency is less expensive than mixing many. "Quality scaling" is a method that is useful when a scene contains multiple sounding objects. It is the process of mixing "foreground" sounds (sounds with higher priority and relevance) at high quality and "background" sounds at a lower quality. Higher quality mixing produces refined results at a higher cost and vice versa. Every object was mixed within an "assigned time quota" to assure consistent performance. While faithful reconstruction of sounds might be useful for many applications, a complex physics-based system is not necessary to invoke strong emotional effects on a listener. Sound design in games and other media often involves "tricking" auditory perception by creating recognizable sounds by unconventional means, or by methods which do not strictly emulate the physical real-world process for creating a particular sound.

3 DESIGN

3.1 Overview

As shown in Fig. 1, this project is designed to be split into two major engines: the Game Engine and the Music Engine. The Game Engine is responsible for running the actual game by keeping track of events, loading and playing animations/sounds, and interpreting user input. In addition, it manages gameplay events with several Finite State Machines to produce signals for the Music Engine to interpret. Within the Music Engine, sounds are then generated with a music controller, while its composition is determined by the sequencer. Once the music is composed by the sequencer, it is played alongside the game. Both of these engines work together to provide the necessary elements to run the game.

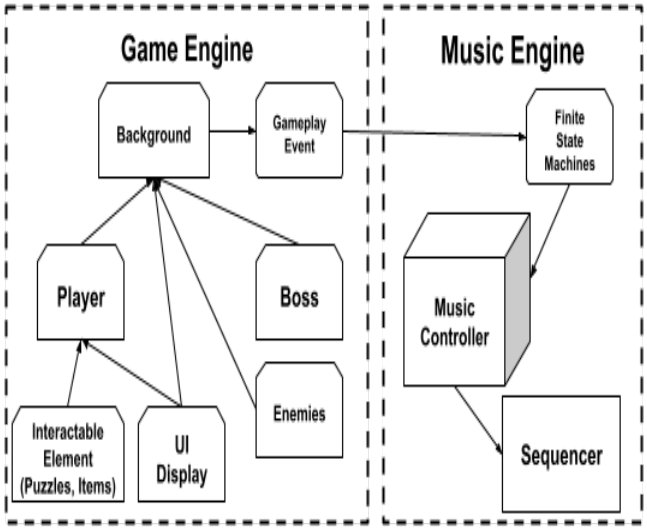

Figure 1: The Project's framework

3.2 Game Engine

The Game Engine is composed of the following components: Background, Gameplay Event, Player, Enemies, Boss, Interactable Element, and UI Display. The Background component represents the environment of the game and serves as the platform for the other components to coexist and interact. The Background would determine the boundaries of the game world, as well as the conditions necessary for the user to complete the game. The Background also allows for Gameplay Events to be sent to the Music Engine to be interpreted by Finite State Machines. The Player component is the character which would be controlled by inputs from the user, and has the ability to interact with items and puzzle elements. The Player also has an attacking function, as well as health in case the Player receives damage. This allows the Player to interact with the Boss and Enemies components. The Enemies and Boss do not receive input from the user; rather, it is controlled by the Game Engine. However, unlike the Player, the Enemies have different types that exhibit different attacks and movements. The Boss is a special component that shares many similar traits to the Enemies component, but it is much more difficult to fight than the Enemies. The Gameplay Event component represents all signals to be sent to the Music Engine. When certain conditions are met, such as the Player dropping to a certain health amount, or the Player encountering new Enemies, the Game Engine will emit these signals. The Interactable Element component is the foundation for other objects in the game that the player can interact with, and can be

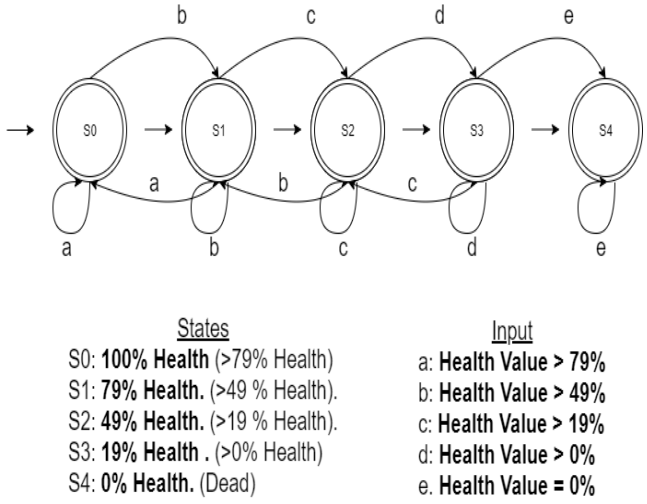

Figure 2: Health State Diagram

separated into two basic categories: Puzzles and Items. The Puzzle Elements consist of doors, gates, and keys that the Player interacts with to open up other areas to explore in the Background. Items are instant-use consumables that immediately affect the player, such as health potions. Finally, The UI Display module displays important Information such as the health of the Player, Boss, and Enemies as well as a menu for the user to interact with.

3.3 Music Engine

The Music Engine is composed of three components: the Finite State Machines (FSMs), a Music Controller, and the Sequencer. The FSMs are a series of data structures that keep track of what state of each music influencing element is in. As an example shown in Fig. 2, the Health State Manager makes updates to the states according to the given inputs (eg: player health value drops below 50% in Game Engine, influences the Minor Events Finite State Machine, to jump from state 1 to state 2).

The states influence the connected Music Controller which will transmit the received signals as data for the Sequencer to interpret. The Sequencer then makes necessary changes to the music such as more instruments, lower pitch, and higher tempo. This dynamically changes the song by adapting the music to the current game state.

4 IMPLEMENTATION

The below sections were implemented either using Godot's GD scripting language and Pure Data's visual based composing language. To conduct work in parallel, our group was divided into two groups. These groups were tasked with approaching the development of this project from the Musical and the Gameplay aspects, hence the "Music Group" and "Game Group".

4.1 Game Implementation

Due the the nature of the Godot language, we create all main items as a "node". From there these nodes are "instantiated" onto a "scene". In this way we can have several enemies or items by having several instances of the same node.

All characters (the player, enemies, and boss) all have basic stats such as health, attacking, and movement cooldowns. These stats are used to calculate mechanics such as how often an attacking animation can play, or how many hits a character can take.

Other common code that all characters have is their own set of signals. These signals are constantly emitting based on the current values of the stats of the character. Further details of these signals are described in the below sections.

All characters also have collision boxes and a raycast pointer in order to check if the character can attack or receive damage. The collision box prevents the player or enemy from walking out of bounds from the map. The raycast is similar to the collision box, however it is used in movement and in deciding directions. Viewing Figure 5 will illustrate the structure and organization of nodes for the Boss scene in Godot.

4.1.1 Player. The Player is the character that is controlled by the person playing the game. It receives input fed from the game and is converted into a direction. Based on this direction, we factor in the base speed, ongoing animations, and then convert this into physical movement which can be seen on screen. Depending on the direction playing, the player sprite is flipped to create the illusion of him turning around to each direction. When the player receives damage, the script sends the appropriate signals and plays an animation to visually see the change in health and damage.

Performing an attack is a matter of doing a few checks based on the current time and the cooldown time. If the current time has passed the cooldown time, it is possible to perform an simple Stab Attack. However if the player attacks again with a certain timing before the end of the previous animation, we can also do a Slash Attack, capable of doing more damage. We also have a simple check of charges remaning for the secondary, powerful attack alongside with the attack code.

4.1.2 Enemies. Each enemy has its own set of health, attacking cooldown, pursuit distances, and move speed. Although enemies are coded similar to the player, they must move and attack on their own. For this reason, we have added a "timer" node to constantly run in order to invoke the enemy's AI. This function performs a distance check to decide whether it should pursuit the player and what distance to stop following. Furthermore, its move speed is also affected by its attacking animation, allowing the player to weave in and out of fights dodging attacks. The enemies will "attack" at every available opportunity when the player's collision box collides with their raycast arrow. If the attacking animation successfully makes contact with the player during a certain frame, the player will receive the damage done by the enemy. Upon the enemy's death, the enemy will stop its AI function and play its death animation. After the death animation it will simply remove its instance to the game scene which essentially deletes itself.

4.1.3 Boss. The Boss is a heavily modified enemy with a few additional changes. Since the Boss can shoot several amount of fireballs, we carry a reference to a fireball scene on the Boss's node. As an indication of it charging fireballs, the Boss spawns several particles on itself. These particles are actually fireballs with no move speed

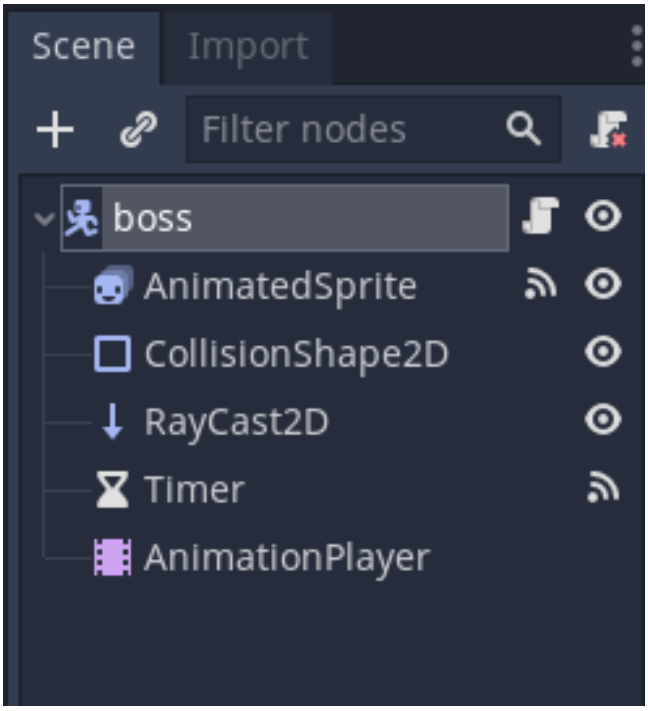

Figure 3: The Boss Node

and a triggered exploding animation. On top of the other checks an enemy may do, the Boss also checks if the Player is a certain range, in which it will shoot fireballs. The function responsible for this will spawn 2-5 fireballs, instance each of them randomly placed on the Boss, then adjust the direction towards the player and send them off. The Boss has a large hitbox so the attacking animation has him float in the air and attacks the Player during a certain frame of the attack cycle. The Boss does also have a cooldown where he will stay somewhat still in order to allow the Player to fight back, as is common in most video games.

4.1.4 Puzzles. In this game, we have several different puzzles.

The most common puzzles have a door "locking" a room so the Player must complete an objective. These objectives may include, fighting a number of enemies to leave the room or pushing boxes onto a certain pit. Fighting enemies in a room is composed of several nodes, the Enemy Spawner and the Door. The Enemy Spawner is started with the Main Game Scene where when the player reaches a certain specified region. This triggers the spawner to spawn its specified enemies. A timer node related to the spawner keeps track of the time passed. If the player was able to defeat the enemies in time, the player completes the "puzzle".

Another puzzle in the game is the box and pit puzzle. The player must slide and move the boxes onto a pit. Depending on the amount of available boxes or pits, the player may need to make some decisions to complete it. This puzzle is composed of several nodes, each going up on a layer of abstraction. Essentially the box and pit are separate nodes. The box has some properties allowing it to be pushed, and the pit will only accept collisions of the box's type. However there is a master node in which it spawns boxes and pits, keeps track of all pits that have a box on them, and emits the associated signals with the operation.

Finally we have the standard doors and keys in which a key collected will immediately send a signal and unlocks the door it is associated with. This is done by using the Main Game node that is keeping a reference between the two objects. Once opened, the door stays permanently opened.

4.2 Dynamic Music Generation

The following sections talk about the implementation of the Music Engine. Below in each sections are more details about the steps and the implementation done.

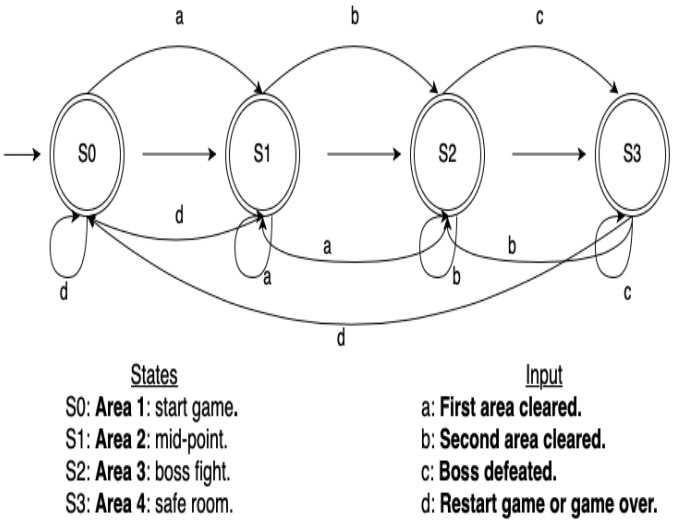

Figure 4: Areas State Diagram

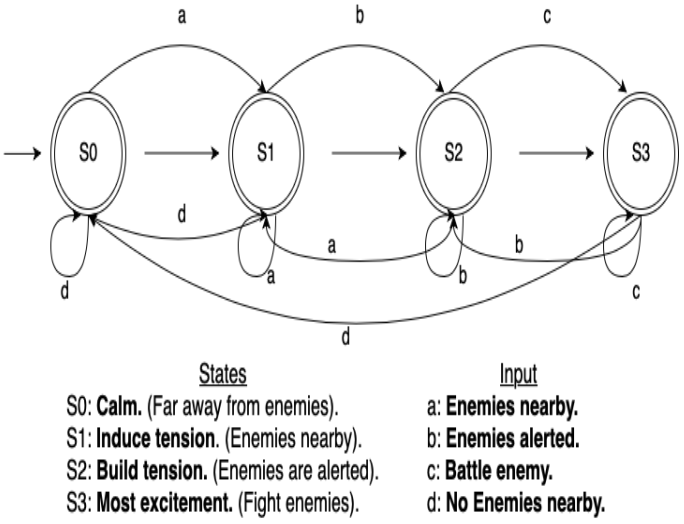

Figure 5: Distance State Diagram

4.2.1 Finite State Machines. The Finite State Machine is a major portion of the linkage between the Game and Music Engines. It is responsible for receiving all the signals in the game and converting them into "states" in which the Music Engine can process accordingly. As there are several state machines, the basic premise is to go from one state to the next based on the current input or status of the game. For example the Health State Machine is composed of several health nodes. These health nodes are essentially what signal, music bpm, and state the Game Engine is in. This vital information is then transmitted to the Music Engine through the packet connection for further processing. There is also an Minor Event State Machine handling the threat level of the Player's current progress [7]. This information is drawn from the Player location and progress through the game. Finally there is an Enemy Detection State Machine which receives signals from all the enemies and increases or decreases the associated state. This information is to present the Music Engine with the number of different enemies and their current status in relation to the Player as if they are attacking or chasing the player.

4.2.2 Sequencer. The sequencer is built using Pure Data, a visual scripting language and real-time audio generation software. Building upon Martin Brinkmann's [1]"patch," or audio program, we modified it to accept 23 parameters which control different parts of the sequencer: bpm (beats per minute), master volume, instrument volumes, instrument note lengths, scale selection, pitch, swing percent, rhythm sequence and probabilities, loop density, loop length, and note probabilities. For updating the bpm, there is also a "set" flag. This was needed to avoid "stuttering" of the sequencer because setting the bpm resets the musical measure ("loop") even when passing in the same value as before. The are five "instruments," or percussive voices which are interconnected and outputted to one channel.

4.2.3 Connecting to Pure Data. Building on the sequencer, we used a local network system in order to transmit data from Godot to Pure Data. In Godot, the "Music Controller" establishes a connection on local port 4242 upon starting the game. We used the FUDI protocol implemented by Pure Data in order to modify the patch to receive packets from Godot. We decided to send all 23 music parameters to the sequencer over TCP connection in one packet, rather than in separate packets (as in one packet for each parameter). This was done to avoid latency, ensuring that sudden changes in the game would affect the music immediately. There was no perceivable detriment in sending the 23 different parameters over the network at once and subsequently processing all of them.

4.2.4 Music Controller. The Music Controller is a script in Godot which drives the music composition. It takes signals sent by the state machines and game events, changes the global music parameters in response, and sends the new "state" of the sequencer to the Pure Data patch using the networking protocol every time the state is changed. The send function takes the message string containing the sequencer parameters, converts it to ASCII values, and sends the data using the established TCP socket.

5 EXPERIMENTS

We conducted many experiments during the development phase to aid the Team in making decisions. Primarily the experiments were conducted for the Music Group, but we also conducted experiments within the Game Group as this was our first game development experience.

Low Tension	Moderate Tension	High Tension
Enemy is far away from player.	Enemy is close to player.	Enemy is next to player.
0 enemies nearby.	1 or few enemies nearby.	Many enemies nearby.
Full health.	Moderate health.	Low health.
Area 1 (beginning), Area 4 (after boss defeated)	Area 2	Area 3 (boss room).
Main Menu, Game Over	Pause Menu	Victory

Figure 6: Music Evaluation Critera

To achieve "dynamic" music, we tested whether or not changes to the sequencer and the music controller made logical sense in relation to the game events they were representing, and whether those changes were aesthetically pleasing for the listener. The experiments for the Music Engine were done by trial and error in order to decide which parameters to change and what values to give them in response to game events. We assigned music changes for several game states: areas in the level (Fig. 4), the player's distance to enemies (Fig. 5), the number of enemies within the player's vicinity, the player's health, and UI changes (the main menu, pause menu, game over menu, and victory menu). Each state was tested to determine if musical changes met our evaluation criteria for tension levels (Fig. 6).

5.0.1 Game Development. For experiments relating to the Game development, we had to really take time to learn the basics of Godot and its associated scripting language.

In order to create an enemy, a node must be created with its associated script. Furthermore, while a node can use inheritance similar to Object Oriented Programming, they must still be instantiated and associated to a main, unrelated node.

5.0.2 Game Balancing. Another experiment the Game Group did was handling the way enemies are spawned onto the map. We had to investigate, what is the best way an enemy node can be instanced onto the game, while still being connected to the running state machines. This posed a few challenges such as connections of the signals not being made or enemy behavior not being consistent with what the script implements.

We also had to experiment on some game design choices such as deciding where and how many enemies to place in the game. Moreover all enemies had to be balanced in their stats in terms of their health, attack damage, and move speed. These large pieces of game balance greatly influence the game as the game can be more difficult or more easier to play because of these types of decisions.

5.1 Experimental Results

5.1.1 Music Results. Pitch modulation of the scale tonalities happens in reaction to the player's health parameter. Modulating to a higher pitch gives an immediate sense of urgency. Because changes in player health occur quite often, we needed to choose a parameter whose constant change would not disrupt the flow of the music. The number of enemies detected on the screen correlates directly to the number of instruments that are audible in the sequencer (five in total). Using this correlation, we were able to create heightened tension for moments when the player is surrounded by enemies. Similarly, as one or more enemies approach the player and get within certain distances, the length of the instruments' notes become shorter, creating more tension. Each "Area" in the level has its own settings that give it a unique feel. For example, when the player enters the boss area, the music reaches its climax by increasing the bpm and volume settings, decreasing the note lengths, and changing the scale tonalities to be more dissonant.

After experimenting with predefined note-length and volume configurations, we decided to add the ability to change each individual instrument volume and instrument note length. This feature adds more flexibility to the music composition. Because of the extensible nature of our Pure Data sequencer and music controller, it is possible to easily add more variables, such as unique note combinations (or scales) and rhythm probabilities, rather than choosing a pre-configured array of notes and rhythm sequences. Adding these new parameters could potentially increase the complexity of the music's control logic. Choosing a particular scale or set of tonalities that is predefined guarantees that a certain tonal "mood" will occur in response to a particular event. Also, the note probabilities can be used to change the "note probability" or frequency at which a certain tonality is played. So, the benefit in adding the ability to manually configure tonalities is limited. However, the ability to fine-tune the number and sequence of percussive hits in a measure, or changing the "rhythm probability," can be potentially beneficial.

5.1.2 Game Results. We found after some experimentation that by having a basic node was not sufficient for our purposes. So as a result instead of using inheritance, we simply broke out the same functionality and modified values into the different enemies. At a later date, we may revisit inheritance for enemies, but at this moment, all enemies use their own basic script with their own sets of values. We balanced their stats based on actual playthroughs of the game, seeing what would be the best set of stats for all characters.

In regards to spawning enemies onto the map, we had some difficulties in setting up the specific enemy spawners. Although it would be great if we could have code that spawned multiple enemies at once, we had difficulty with the AI not being activated upon enemy spawn. For this reason we integrated enemy spawns with a Main Scene Node which took over all spawning mechanisms and handled this in a lot more simpler manner. Although not the original planned route, this alternative solution, the Main Scene Node, also connected enemy information to the Finite State Machines, ultimately encapsulating everything and achieving the desired outcome we intended for.

The music group found success with the experiments with signals. We are able to send information from within the game to the sequencer. This allows the sequencer to produce dynamically generated music based on events within the game. There were set backs, a lot of the game functions were running constantly per millisecond so some signals would emit multiple times per action. We found a way to limit that using an incremental variable that would only allow the action to run on the initial signal emit. Overall the music group found success in our experiments with these signals.

6 CONCLUSION AND FUTURE WORK

By conducting various experiments, we were able to refine the gameplay and music of our video game. By testing each unique state using an evaluation criteria for "tension," we were able to produce music which adapts to the player's situation.

We were able to create a game which composes dynamic music in real-time by linking two different systems together over a local network connection. By employing the use of several Finite State Machines, we were able to aggregate the signals and create meaningful music responses based off the game. By associating specific game events with particular sounds and musical changes, we were able to strengthen the player's interaction and engagement with the game.

Although we succeeded in creating an engaging game with dynamic music, there are several improvements that can be made.

During our research phase, we experimented with producing real-time audio within Godot, using Godot's lower-level audio functions. However, this method was not ideal due to complex synchronization problems between the game and the audio generation. We decided to instead use two programs which communicated with each other: a game engine (Godot) and a music engine (Pure Data). Having Pure Data separate from the Godot engine introduced a new layer of complexity to our project.

The most obvious improvement that can be made is bundling Godot and Pure Data into one program. This can potentially be achieved using the libpd library and dynamic library binding via Godot's GDNative feature. The libpd library allows patches to be embedded into C and C++ programs. GDNative allows users to bind C++ programs to their game as dynamic libraries. However, by doing so, we would also require a component for interacting with lower-level audio APIs like Core Audio in order to produce sound. Audio frameworks like JUCE can be used to bridge the gap from application to lower-level audio functions. Rather than using lower-level audio APIs or frameworks like JUCE, another alternative for producing sound is to use (with GDNative and libpd) a patch which generates MIDI signals for Godot to interpret using a MIDI plugin.

Before even attempting to create an all-in-one application, a simpler solution might be to include batch commands to open the Pure Data Sequencer upon starting the game. This solution might be more intuitive for the user to understand the application, but it is not ideal since it still requires users to interact with two executables and one file (the game executable, the Pure Data executable, and the sequencer patch).

Improving the level of engagement is another area that can be explored further. The sequencer can be improved upon to include more types of instruments with different timbres. Experiments can be conducted, such as comparing the engagement and emotional

reactions of a test group towards the Pure Data solution versus traditional background music.

We believe creating art that was entirely unique to the video game theme would also have improved the player experience. We used sprites from free online game development resources to save on time and development. However we are interested to know if the player's engagement would be higher if we had also created a unique visual aesthetic to match better with the music.

One more area that could be extended would be introducing "styles of music". This opens up additional area for genres of music to be associated with levels. More levels created with their own "style" being composed would truly be unique and a great area for future work, development, and research.

7 ACKNOWLEDGMENTS

We thank all our Team members who helped us come up with the unique Finite State Machine idea. We developed a lot of time discussing it and trying to make it fit with logical sense in terms of the gameplay. Personal thanks to the Music Team (Tomo, Austin, and Robert) who lent much of their musical background for this project and to the Game Team (Tristen, Ahrin, and Keith) who put much of their time and energy in development of Animation, AI, and, programming skills.

REFERENCES

[1] Martin Brinkmann. 2013. *random percussion with trigger probability controlled by a sequencer.* http://www.martin-brinkmann.de/pd-patches.html

[2] William Cachia, Luke Aquilina, Hector P Martinez, and Georgios N Yannakakis. 2014. Procedural Generation of Music-Guided Weapons. (2014), 2–3.

[3] Alvaro E Lopez Duarte. 2020. Algorithmic interactive music generation in videogames. *SoundEffects-An Interdisciplinary Journal of Sound and Sound Experience* 9, 1 (2020), 38–59.

[4] Raluca D. Gaina and Matthew Stephenson. 2019. "did you hear that?" Learning to play video games from audio cues. *IEEE Conference on Computatonal Intelligence and Games, CIG* 2019-Augus (2019). https://doi.org/10.1109/CIG.2019.8848088

[5] Aoife McDonagh, Joseph Lemley, Ryan Cassidy, and Peter Corcoran. 2018. Synthesizing Game Audio Using Deep Neural Networks. *2018 IEEE Games, Entertainment, Media Conference, GEM 2018* (2018), 312–315. https://doi.org/10.1109/GEM.2018.8516448

[6] David Plans and Davide Morelli. 2012. Experience-driven procedural music generation for games. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 3 (2012), 192–198. https://doi.org/10.1109/TCIAIG.2012.2212899

[7] Cale Plut and Philippe Pasquier. 2020. Generative music in video games: State of the art, challenges, and prospects. *Entertainment Computing* 33, December 2019 (2020), 100337. https://doi.org/10.1016/j.entcom.2019.100337

[8] Nikunj Raghuvanshi, Christian Lauterbach, Anish Chandak, Dinesh Manocha, and Ming C. Lin. 2007. Real-time sound synthesis and propagation for games. *Commun. ACM* 50, 7 (2007), 66–73. https://doi.org/10.1145/1272516.1272541

[9] Sebastian Risi and Julian Togelius. 2020. Increasing generality in machine learning through procedural content generation. *Nature Machine Intelligence* 2, 8 (2020), 428–436. https://doi.org/10.1038/s42256-020-0208-z arXiv:1911.13071