April 2018

DOLDRUM

Kelly Ellen Zhang
Worcester Polytechnic Institute

Doldrum

A Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree in Bachelor of Science
in
Computer Science & Interactive Media and Game Development

By

David Allen

Kent Fong

Matt Szpunar

Henry Wheeler-Mackta

Kelly Zhang

Date: 4/26/18
Project Advisors:

Professor Gillian Smith, Advisor

Professor Ralph Sutter, Advisor

ABSTRACT

Doldrum is a Virtual Reality rhythm game that provides the player with a sense of flow and a feeling of triumph as they overcome a towering boss. The game mixes traditional rhythm game elements with reaction-based combat. The player battles a villainous, animated Cuckoo Boss via beating on a mystical xylophone. They must keep rhythm, avoid enemy attacks, and perform specific actions to combat their opponent.

This report describes the design process, production, and testing results of the game. *Doldrum* underwent significant changes to its design over the course of its development. The final product is ultimately a vertical slice that demonstrates the concept of a rhythm-action game. Playtesters enjoyed *Doldrum's* mix of rhythm and action game elements while complimenting its visual and audio components.

Doldrum was developed during the 2017 – 2018 academic year at Worcester Polytechnic Institute. The game was developed in *Unreal Engine 4* for use with the HTC Vive virtual reality headset.

TABLE OF CONTENTS

Abstract i
Table of Contents ii
List of Figures vi
1 Introduction 1
2 Background 2
2.1 Unreal Engine 4 4
2.2 Game Inspirations 4
2.2.1 Space Pirate Trainer 5
2.2.2 Audio Shield 5
2.2.3 Thumper 6
2.2.4 Taiko no Tatsujin 6
2.2.5 Patapon 7
2.2.6 Guitar Hero 7
2.3 Art Inspirations 8
2.3.1 Horror Games 9
2.3.2 Various Anime 10
2.3.3 Guillermo Del Toro's House of Horrors 11
3 Gameplay 11
3.1 Experience Goal 11
3.2 Gameplay Loop 11
3.3 Progression 13
3.4 End Condition 15
4 Design 15
4.1 Platform 15
4.2 Mallets 15
4.3 Notes 15
4.4 Note Highway 16
4.5 Xylophone 17
4.6 Cuckoo 17
4.7 Chimes 18
4.8 User Interface 19
4.8.1 Main Menu 19

4.8.2 Tutorial 20
4.8.3 Pause Menu 21
4.8.4 HUD 21
4.9 Original Scope 23
4.9.1 PlayStation VR 23
5 Artistic Implementation 24
5.1 Cuckoo 24
5.2 Note Highway 30
5.3 Notes 30
5.4 Platform 33
5.5 Mallets 34
5.6 Xylophone 35
5.7 Environment 36
5.7.1 Chimes 38
5.7.2 Lighting 39
5.8 User Interface 40
6 Technical Implementation 40
6.1 Metronome 40
6.2 Cuckoo 42
6.3 Note Highway 44
6.4 Notes 45
6.4.1 Popping 46
6.4.2 Guaranteeing On-Beat Sound 46
6.5 Note Generation 47
6.5.1 Procedural Generation 47
6.5.2 Grammar-based generation 48
6.5.3 Scripted Generation 51
6.5.4 Original System 53
6.6 Phases 54
6.7 Player 55
6.7.1 VR Interfacing 55
6.7.2 Platform 55
6.7.3 Walking Out of Bounds 55
6.7.4 Mallets 55

6.7.5 Height Calibration 56
6.8 Xylophone 56
6.8.1 Note Hitting 57
6.8.2 Actions 59
6.9 User Interface 61
7 Sound 62
8 Music 63
9 Project Management 66
9.1 Trello 66
9.2 Perforce 67
9.3 Google Drive 67
9.4 Slack 68
9.5 Discord 68
10 Testing 68
10.1 Methodology 68
10.1.1 Overview of the Experiment/Design 68
10.1.2 Population 68
10.1.3 Location 69
10.1.4 Restrictions 69
10.1.5 Technique 69
10.1.6 Materials 70
10.1.7 Entrance and Setup Procedure 70
10.1.8 Study Procedure 71
10.1.9 Ending Procedure 71
10.1.10 Variables 71
10.1.11 Statistical Treatment 72
10.2 Results 72
10.2.1 Playtesting Demographics 72
10.2.2 Physical Assessment 72
10.2.3 Usability Assessment 73
10.2.4 Gameplay Assessment 74
10.2.5 Emotional Assessment 77
10.2.6 Sentiment Analysis 77
10.2.7 Improvements Made 80

10.2.8 PAX East 80
10.3 Release 81
11 Post-mortem 82
11.1 Design 82
11.2 Future Work 83
11.3 What Went Wrong 83
11.4 What Went Right 83
12 References 85
13 Appendix 87
13.1 Additional Concept Art 87
13.1.1 Trumpet Boss 87
13.1.2 Audience Boss Environment 87
13.1.3 Owl Mentor Environment 88
13.1.4 Note Highway Player View 88
13.1.5 Note Highway Side View 89
13.1.6 Menu Concept 90
13.1.7 Logo Iterations 90
13.2 Playtest Informed Consent Form 91
13.3 Post-Playtest Survey Questions 95
13.4 Header Files 97
13.4.1 BossCharacter.h 97
13.4.2 BossState.h 99
13.4.3 BossStateMachine.h 101
13.4.4 MetronomeController.h 102
13.4.5 MetronomeListenerComponent.h 104
13.4.6 NoteActor.h 105
13.4.7 NoteGeneratorFunctionLibrary.h 107
13.4.8 NoteGrammar.h 108
13.4.9 PhaseChangeListenerComponent.h 112

LIST OF FIGURES

Figure 1: Screenshot of Unreal Engine 4's Blueprint logic 4
Figure 2: Promotional Image for Space Pirate Trainer (2016) 5
Figure 3: Promotional image for Audio Shield (2016) 5
Figure 4: Screenshot of Thumper (2016) 6
Figure 5: Screenshot of Taiko no Tatsujin (2001) 6
Figure 6: Screenshot of Patapon (2007) 7
Figure 7: Screenshot of Guitar Hero 3 (2007) 8
Figure 8: Screenshot of Silent Hill 2 9
Figure 9: Example of a "boss" in Puella Magi Madoka Magica (2011) 10
Figure 10: Photograph of one room of Guillermo Del Toro's House of Horrors 11
Figure 11: The first view that players will see when they start the game 12
Figure 12: The Cuckoo Boss readying an attack 13
Figure 13: Chimes surrounding the Cuckoo Boss 14
Figure 14: Note Highway with Beat Bar (left) and Input Bar (right) 16
Figure 15: Unity Editor view of BossAttackState instance 18
Figure 16: The Main Menu 19
Figure 17: The first slide of the tutorial 20
Figure 18: The Pause Menu 21
Figure 19: The full health display 22
Figure 20: The Note Streak Display 22
Figure 21: Fully depleted frenzy meter (left) and fully charged frenzy meter (right) 22
Figure 22: Concept art for two abandoned boss designs 23
Figure 23: Hydra boss concept art 25
Figure 24: Early concept art of the Cuckoo Boss 25
Figure 25: Concept art of the Cuckoo Boss redesign 26
Figure 26: Final render of the Cuckoo Boss model 26
Figure 27: Iterations of the Cuckoo Boss' wings 27
Figure 28: Texture sets for the Cuckoo's diffuse, normal, and specular maps 27
Figure 29: Rigged and skinned Cuckoo mesh in 3ds Max 28
Figure 30: The Cuckoo wing's range of motion 29
Figure 31: The Cuckoo attack animation flow 29
Figure 32: Notes travelling down the Note Highway 30
Figure 33: Default Note (left) and Healing Note (right) 31
Figure 34: Material graph for the base Note material 31
Figure 35: Note material viewed in the Unreal material editor 32
Figure 36: Final render of Platform mesh 33
Figure 37: Comparison between the original unicorn-head mallets (left) and the final mallet model (right) 34
Figure 38: Concept art of unicorn head mallets(right) with Monster Hunter and Alice inspirations (left) 35
Figure 39: Early concept for the drums 35
Figure 40: Final render of a singular xylophone key 36
Figure 41 Final render of entire xylophone 36

Figure 42: Concept art of the Cuckoo's environment 37
Figure 43: Grayboxed stage by Laurie Mazza 37
Figure 44: The full environment 38
Figure 45: The Cuckoo readying an attack, surrounded by the Chimes 39
Figure 46: Early stage layout with basic lighting 39
Figure 47: Screenshot of a MetronomeController's editable properties 41
Figure 48: Screenshot of an actor's MetronomeListener component listening to a MetronomeController actor 41
Figure 49: OnBeat delegate as seen in Blueprints 42
Figure 50: Screenshot of the Cuckoo's state transition map property 43
Figure 51: Screenshot of a part of the Cuckoo's attack logic in Blueprints 44
Figure 52: Note's MoveDown Timeline Blueprint node 45
Figure 53: The Highway Valves that the Notes are struck by 46
Figure 54: Sequences generated by Tracery prototype 49
Figure 55: JSON ruleset for Doldrum's note generation 50
Figure 56: Early prototype of the Note Editor 52
Figure 57: Screenshot of MIDI sequence in Maschine 2 53
Figure 58: Screenshot of the Note Editor's final version 53
Figure 59: DrumInputStruct Blueprint node with all values set to the default 58
Figure 60: The player's lightning attack orb 59
Figure 61: The five move locations as seen from above. 60
Figure 62: The UI Flowchart 61
Figure 63: "Cuckoo Hurt" sound effect, created in Reaper 62
Figure 64: Maschine 2 Digital Audio Workstation 64
Figure 65: Music arrangement in Bitwig Studio 2 65
Figure 66: A screenshot of the Doldrum Trello board 66
Figure 67: Circle of Affect used to categorize and measure positive and negative words and moods of user 70
Figure 68: Post-Playtest Survey, Physical Assessment Results: Headset comfort 73
Figure 69: Post-Playtest Survey, Usability Results: Performing attack action 74
Figure 70: Post-Playtest Survey, Usability Results: Performing dodge action 74
Figure 71:Post-Playtest Survey, Gameplay Results: Note predictability 75
Figure 72: Post-Playtest Survey, Gameplay Results: Pacing 75
Figure 73: Chart on the number of tries per playtester 76
Figure 74: Post-Playtest Survey--Emotional Assessment, chart on moods felt during gameplay 77
Figure 75: Chart of playtester's sentiment during audio recording 78
Figure 76: Post-Playtest Survey--Chart of playtester's sentiment on game visuals 79
Figure 77: Post-Playtest Survey--Chart of playtester's sentiment on game music 79
Figure 78: Post-Playtest Survey--Chart of playtester's sentiment on gameplay 80
Figure 79: itch.io Analytics Panel, Visits to Doldrum's itch.io page from the following sites 81

1 INTRODUCTION

Doldrum is a virtual reality rhythm-action game for the HTC Vive. Above all else, Doldrum attempts to be a fun and continuously engaging rhythm game. By utilizing mechanics typically found in combat-centric games, we were able to achieve these goals. The player stands in a fixed location and battles a massive Cuckoo Boss. Procedurally generated note sequences travel down a note highway towards the player. The player uses the Vive motion controllers to strike a xylophone-like instrument in time with the approaching notes. At a consistent interval, flashing input bars travel down the highway. These allow the player to interact with the boss in one of two ways: they can either attack the boss or dodge the boss's incoming fireballs. As the game progresses, the notes become more rapid and more complex. Furthermore, additional mechanics are introduced such as targets that the player must strike in specific locations in order to damage the boss.

Doldrum was designed specifically to utilize the HTC Vive's motion controllers and the added full-body engagement of VR. [36] Our goal was to induce a flow state for players using music and rhythm as the means of action. In Mihaly Csikszentmihalyi's talk on Flow, *The Secret to Happiness,* he defines flow as a state in which a person is in intense focus and when they know exactly what to do from one moment to the other. This heightened state of cognitive focus can make gameplay more engaging. [5, 10] Taking inspiration from the game *Thumper (2017)* we wanted to capture the excitement of an action game using the heightened engagement of VR. [14] Unlike *Thumper,* which was originally designed as a non-VR game and then ported to VR, *Doldrum* is a rhythm game specifically for VR. *Doldrum* functions better as a VR game than a non-VR game as it relies on precise motion controls.

Original sound effects and music were created by the project team's sound designer. As with all rhythm games, the music is crucial for the experience. Rather than creating a complex and intricate soundtrack, the backing track is simple and accessible with a strong focus on rhythmic elements. As the backing track plays, the player performs randomly generated melodies over it. The notes are all a part of a pentatonic scale: a scale that will not create dissonant harmonies when played over the backing track. Additionally, the notes of the scale can be played in any order while still sounding melodic. All sound effects were created by layering and processing samples and synthesis.

Doldrum was built using Unreal Engine 4, via C++ and Blueprint scripting. The systems were built on two levels. The C++ level contains most of the systems that drive the abstracted behavior of the game such as note generation, boss state, and phase state. The Blueprint level contains the majority of the logic for gameplay, presentation, and player input handling. Utilizing Unreal's C++ API, the systems level of

the game includes reflected functions that can be called as Blueprint nodes, bridging the communication between the two levels. While the technical implementations of the game's components are conceptually interconnected, they are designed and built in a modular manner for ease of fixing and changing functionality.

During production of *Doldrum,* we faced a number of challenges related to scoping and time constraints. Despite these issues, we were able to make a complete and polished vertical slice of the game's original design. Following production of our game, we performed user playtesting on the WPI campus and showcased our game at the WPI PAX East booth. During this testing, we noticed aspects of our game that could be improved upon; primarily the user experience and the user interface. Many people chose to play *Doldrum* multiple times and found the gameplay to be difficult but rewarding when successful. With the overall positive reception of our game, we consider the final product a success.

2 BACKGROUND

We wanted to develop a Virtual Reality game in order to experiment and gain familiarity with the medium as we were inexperienced with it. The project was initially pitched to be a VR game but we were given creative control to dictate what type of VR game we would make. We were excited to develop a game with this novel technology.

There were a variety of challenges that must be considered when designing a Virtual Reality game. Virtual Reality games have limited player movement due to many locomotion methods causing simulation sickness [25]. Additionally, VR games focus on motion controls in order to heighten engagement through physical action. Ultimately, the final product was a result of us attempting to create an engaging action-oriented game within the constraints of the medium.

In order to not restrict ourselves to the limited performance of smartphone powered head mounted displays (HMDs), we initially narrowed our headset selection to the Oculus Rift, PlayStation VR (PSVR), and HTC Vive. Early complications related to obtaining a PSVR dev-kit ruled out that platform as a possibility. We ultimately chose the Vive over the Rift due to its superior technology. The Vive offers room-scale virtual reality while the Rift is restricted to a single standing location. While *Doldrum* does not utilize room-scale VR, we did not want to restrict ourselves. The final deciding factor was certain team members' previous familiarity with the Vive.

While some Vive games are designed for a traditional gamepad, most are designed for the included motion controllers. Because of this, we self-imposed a constraint to use the motion controllers.

We chose to not use the motion controllers' touch pads as core control due to their poor haptic feedback and their inferiority to a traditional joystick. In order to match the expectations of the platform, we attempted to make a game that was controlled significantly through motion controls.

We wanted to use the full-body engagement offered by VR to induce a heightened flow state in players. In order to achieve this, another self-imposed constraint for the game was to make it combat-centric. We believed that players would be more engaged if there was a physical consequence to their actions (i.e. the player character taking damage). Many VR games are already combat-heavy, but the primary means of action is either melee combat or shooting guns. We chose to avoid melee combat as it can problematic as the player's weapon could phase through the enemies, causing inconsistency and potentially breaking immersion. We decided against using guns in the game early on as the current VR market is saturated with shooting games. The desire to make an action game while avoiding traditional means of combat lead us to the rhythm game design. We are able to elicit the same emotions as a traditional combat game without needing to adhere to the conventions of the genre.

A significant consideration in current VR design is artificial locomotion. [25] The most common solution for locomotion limitations is teleportation. [30] Typically, a player will point at the ground and press a button; the screen will briefly fade to and from a black screen and then the player will be in the location they had pointed to. According to Oculus, a leading VR hardware and VR software developer, this locomotion method significantly improves comfort with a majority of players. [30] Though it helps reduce nausea, it can make players spatially disoriented and adversely affect the balance of game actions. As we wanted to avoid the problems of teleportation and the discomfort of other types of artificial locomotion, we chose to design *Doldrum* as a standing-only game. Standing-only refers to a VR game which is played standing but involves no movement of the player's legs.

Another best practice for VR developers is clear haptic feedback. Haptic feedback is essential for an immersive VR experience as it conveys object interaction. [30] Both motion controllers vibrate when a xylophone face is struck in order to give the illusion of the player striking a solid object.

We decided to use simple colors and shapes rather than more realistic graphics early in the creative process. Due to the low effective resolution of HMDs, we believe that stylized aesthetics are more readable than realistic ones. [16] We adhered to this in creating the visuals for *Doldrum* by using bright, flat colors and basic shapes. Basic shapes were used to achieve the stylized aesthetic as it is less realistic.

2.1 UNREAL ENGINE 4

Two primary engines were considered during pre-development for *Doldrum*: Unity and Unreal Engine 4. The latter was chosen due to increased team familiarity and because it featured an immediately ready project template for VR support. Additionally, Unreal Engine 4's Blueprinting, material, and particle systems allowed for accelerated development. While much of the game is programmed in C++, the majority is implemented exclusively in Blueprints.

In Unreal Engine 4, Blueprints are a visual scripting system that can be used as an alternative to more traditional programming. Most of the game's VR functionality and visual elements are implemented via blueprints due to their ease of implementation and quick compilation time. The latter allowed for the game's intractable elements, particularly VR, to be implemented and fine-tuned quickly and efficiently.

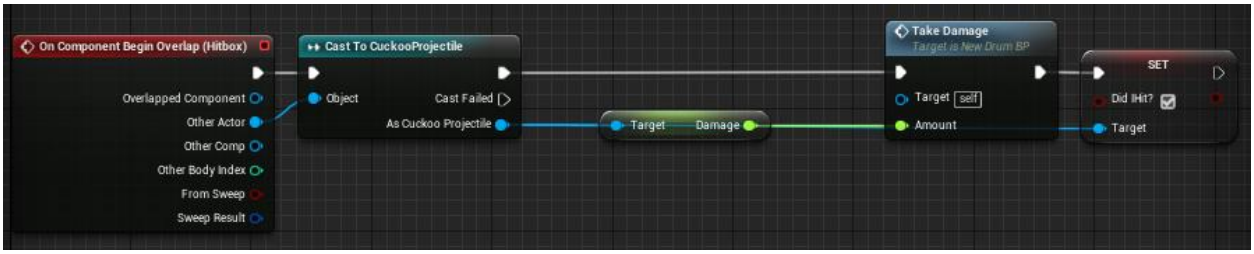


Figure 1: Screenshot of Unreal Engine 4's Blueprint logic

Several elements of the Unreal Engine 4 development process are unique to the engine. For instance, all Blueprints that are placeable in the game world are considered Actors. Examples of Actors in *Doldrum* include the player, the mallets that they hold, the xylophone that they hit, etc.

2.2 GAME INSPIRATIONS

In preparation for designing *Doldrum*, we researched other games in order to determine best practices for VR and rhythm design.

2.2.1 Space Pirate Trainer

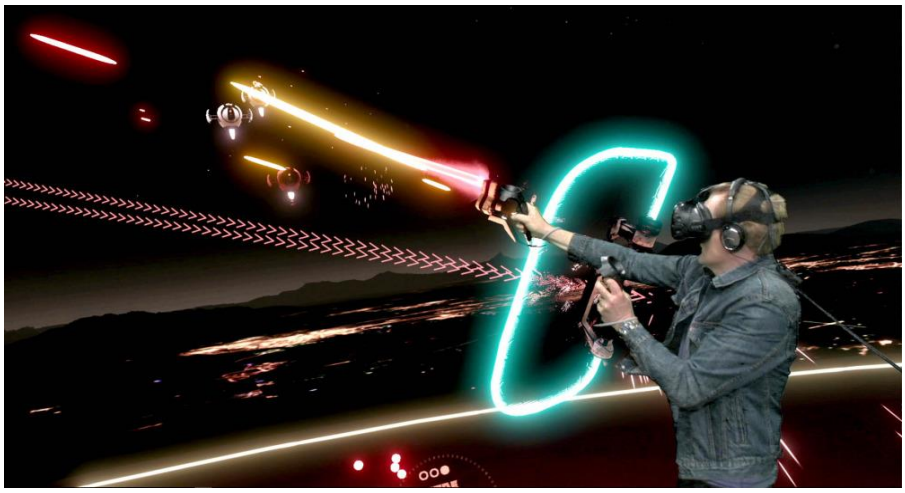


Figure 2: Promotional Image for Space Pirate Trainer (2016) [22]

Space Pirate Trainer is a polished VR shooter in which the player shoots increasingly dangerous waves of flying robots. [22] *Space Pirate Trainer* is a simple wave-shooter, a genre that we did not want to develop. Notably, the game gives the player many different weapons that enable different strategies despite the player not being able to move. Despite *Space Pirate Trainer* featuring simple gameplay, the experience is highly engaging due to the elegance of the mechanics and the game's polish. The gunplay is responsive with satisfying audiovisual feedback. The game also features small details such as dynamic lighting and a slowdown effect when dodging enemy projectiles. When creating *Doldrum*, we wanted to achieve same level of polish and elegance found in *Space Pirate Trainer*.

2.2.2 Audio Shield

Figure 3: Promotional image for Audio Shield (2016) [21]

Audio Shield is a VR rhythm game in which the player uses their controllers as shields to block incoming projectiles in beat with the music of the game. The player remains stationary while playing this game. *Audio Shield's* usage of motion controls have the player to move physically in time with the game's

music. The game's usage of motion controls to encourage rhythmic actions was a heavy inspiration for the gameplay of Doldrum. [39]

2.2.3 Thumper

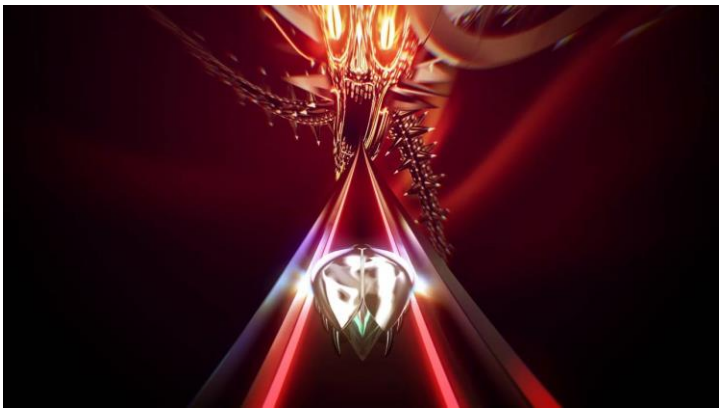


Figure 4: Screenshot of Thumper (2016) [11]

Thumper is a rhythm action game where the player guides a space beetle on a track through an abstract void to confront an enemy. The player presses buttons in beat as they run down the track, occasionally fighting large enemies. [11, 14] *Thumper* can be played in or out of VR. The aesthetics, music, and boss components appealed to us and influenced the design of *Doldrum*. In particular, we drew influence from percussion-heavy music and the dark environment with abstract visuals. Additionally, we found that the boss battle levels would increase the intensity of the gameplay. We played *Thumper* both in and out of VR. We noted that the VR version was more immersive and intense as it blocked out external distractions and cause players to befocused on the game.

2.2.4 Taiko no Tatsujin

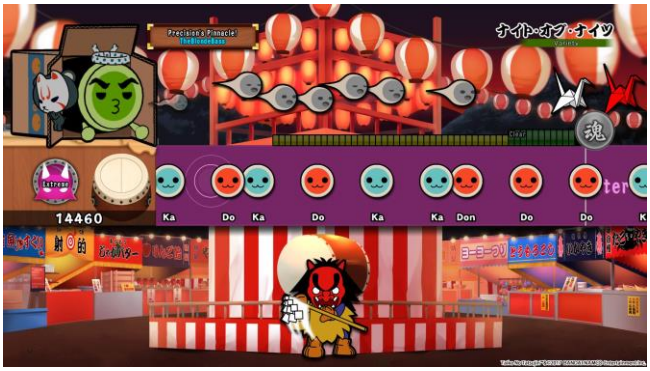


Figure 5: Screenshot of Taiko no Tatsujin (2001) [6]

Taiko no Tatsujin is a rhythm game where the player hits notes scrolling across the screen in time with specific music pieces. Notes come in mainly two colors, red and blue. Red notes require a hit on the drum face while blue notes require a hit on the rim. [6]

Arcade versions of *Taiko no Tatsujin* features an electronic Taiko drum and drumsticks as input controls for the player. As a team, we liked the idea of a more tactile game feel through the use of a physical percussion instrument. We attempted to emulate the feeling of a percussion instrument in VR using motion controls. [6]

2.2.5 Patapon

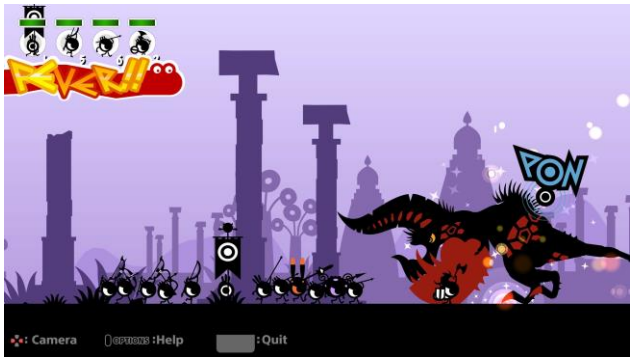


Figure 6: Screenshot of Patapon (2007) [31]

Patapon is a rhythm game released on the PSP. The player controls a tribe of warriors via specific drum sequences. Different input sequences made by the player determines what the tribe of warriors do. [31]

Specific drum inputs that determine what happens was part of the early design of *Doldrum*. However, these proved overly complicated during the production stage and were scrapped. Instead, we settled on having three specific but simple input controls for the player. The first is hitting the middle xylophone key with both mallets to perform an attack. The other inputs are dodging left and right which is two hits on either the leftmost or rightmost xylophone key.

2.2.6 Guitar Hero

Figure 7: Screenshot of Guitar Hero 3 (2007) [28]

Guitar Hero is a rhythm game in which the player uses a guitar-styled controller to hit notes that come towards them on a *Note Highway*. These notes are five different colors that correspond to their position on the highway. Upon reaching the player, the player must hit the note by holding down the corresponding button and strumming their guitar controller. This game was a major inspiration for *Doldrum* and we implemented a similar Note Highway for it. [28]

2.3 Art Inspirations

Art from several mediums have inspired the overall aesthetic of *Doldrum*. The initial aesthetic goal was to introduce horror elements into the game. Therefore, several horror games, movies, and TV shows were referenced. When looking through references, we found that the surreal aspect of these works was much more desirable than their scare factor.

2.3.1 Horror Games

Figure 8: Screenshot of Silent Hill 2 [24]

Other inspirations came from various survival horror games, specifically the *Silent Hill* series and *Rule of Rose*. The team played through *Silent Hill 2* to gather artistic and narrative inspiration for *Doldrum*. One of the major takeaways from the game was the monster design. [24, 34] While many of the monster designs were fleshy and grungy, the core theme was that the designs were reflections of the main character's inner psyche. In the original design of the game, the bosses also reflected the player character's inner worries. However, the element of surrealism is still present in the final game.

Similarly, the monster designs from the *Shin Megami Tensei*: *Persona* series also played a part in inspiring the boss designs. Though the genre is not horror, many of artistic designs have uniquely surreal elements that can elicit a feeling of unease. [3]

2.3.2 Various Anime

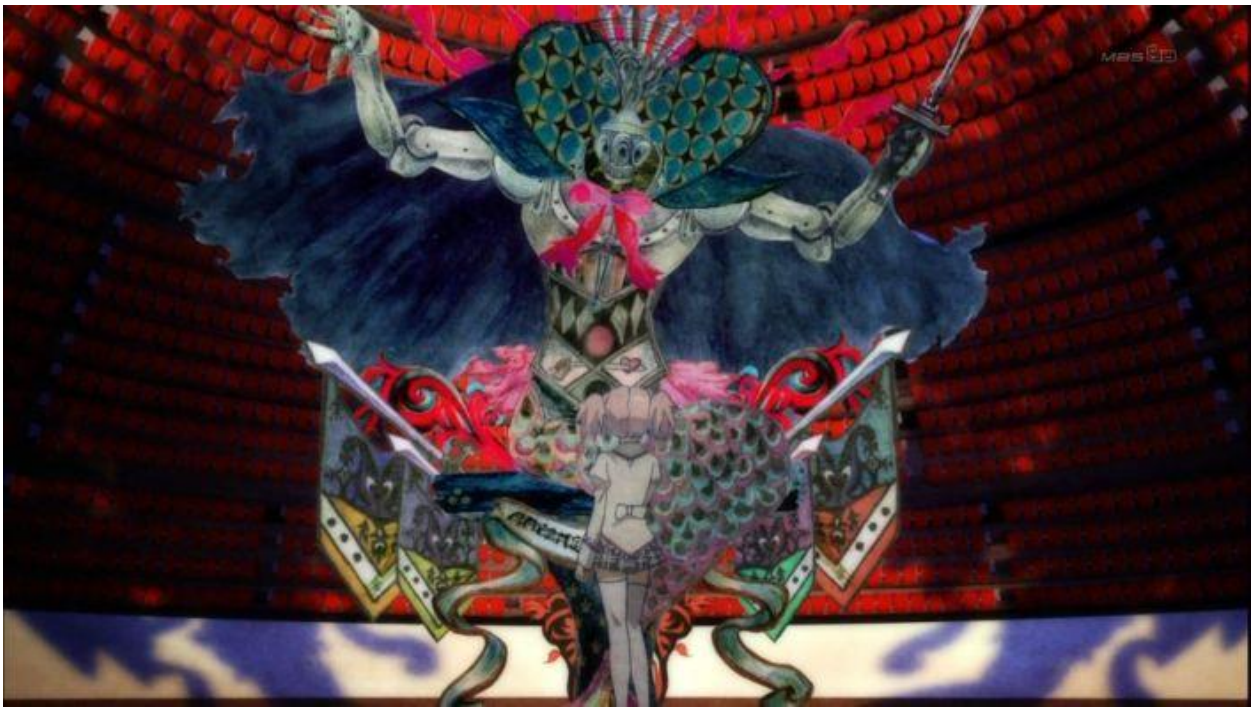


Figure 9: Example of a "boss" in Puella Magi Madoka Magica (2011) [29]

Other inspirations with surreal aesthetics were from various anime such as *Puella Magi Madoka Magica*. [29] The general theme of these shows are main characters that face against a “boss.” In *Madoka*, the boss fights turned the surrounding environment into that of a dark fairy tale. These two shows influenced the stage design of the game, where the area the player plays in an enclosed area where the elements are thematic to the boss’ design.

2.3.3 Guillermo Del Toro's House of Horrors

Figure 10: Photograph of one room of Guillermo Del Toro's House of Horrors [33]

Guillermo Del Toro's *House of Horrors* also served as inspiration for the general aesthetic and feel of the Cuckoo and the stage. The house is filled with many unique objects; some are props from his own directed films and others are his favorite characters from other media. Pictures of Guillermo's house influenced the Cuckoo's body shape and color. The brown and red color palette along with gold accents can be seen on the Cuckoo's body. The Cuckoo's body also features wood paneling similar to the wood that is visible throughout the furniture pieces in his house.

3 GAMEPLAY

3.1 EXPERIENCE GOAL

Doldrum's experience goals are to provide the player with a sense of rhythmic flow and a feeling of triumph as they overcome a towering boss.

3.2 GAMEPLAY LOOP

When the player puts on the HTC Vive headset and starts the game, they are surrounded by a dark environment. The only visible elements in front of them are a xylophone and a pair of mallets. The player

then grabs the mallets with the trigger buttons on the Vive controllers. If the player does not do this within a short amount of time, the game will prompt them to grab the mallets.

Figure 11: The first view that players will see when they start the game

Once the player grabs the mallets, a main menu screen will appear in front of them. In front of the xylophone faces are menu options: xylophone height calibration, tutorial, and starting the game. Hitting the faces of the xylophone correspond to entering its menu option.

When the player hits the “Start Game” option, the music begins playing and the environment lights up, revealing the Cuckoo Boss. Notes will then begin coming down the Note Highway.

From now until the end condition of the game, players are to hit the Notes as they reach the player by striking against the faces of a in-game xylophone using their motion controllers. Successful hits will build the player’s Note Streak Display and special attack meter. When players miss a Note, a small amount of health is lost and both meters reset to zero. Occasionally, health Notes come down the Note Highway. Successfully striking these Notes restores a small amount of player health.

Every four measures, a colorful bar spawns. When it reaches the player, they can perform an Action Input by hitting both mallets on one of the xylophone faces. The actions that the player performs allows them to interact with the boss in front of them. These actions include attacking the boss with a lighting ball and relocate the player. If the player’s special attack meter is full when they perform an attack action, a powerful lighting attack is shot from the player towards the boss, dealing more damage than the regular attack.

Figure 12: The Cuckoo Boss readying an attack

Every couple of measures, before the bar spawns, the boss will begin to charge a fireball attack. When this happens, the fog in the stage turns orange, further telegraphing the attack. After charging for a measure, the boss will launch the fireball towards the player. If the player performs a dodge action at the end of the measure, they will move away from the attack and avoid the fireball.

3.3 PROGRESSION

The player's battle with the Cuckoo is broken down into nine phases. Each phase will loop until the player damages the boss down to a certain threshold. When the boss health is equal to or below that threshold, the game will transition to the next phase. It should also be noted that these thresholds are predefined.

There are three distinct phase types: pre-scripted, generated, and chime. Pre-scripted phases will output a note sequence that will stay the same between gameplay sessions. A pre-scripted phase ends when the note sequence ends. There is one pre-scripted phase that plays at the very beginning of the game to introduce players to hitting notes at a slow pace.

Figure 13: Chimes surrounding the Cuckoo Boss

Chime phases are similar to generated phases, except the condition to transition to the next phase is for the player to hit specific Chimes. These phases add variety to the gameplay as it forces the player to move around the stage in particular directions. Without the Chime phases, movement would exclusively be used for dodging boss attacks. Chime phases happen twice in the game-- once in the middle of the game and another one at the very end of the game.

Phase Number	Phase Type
1	Pre-scripted Phase
2	Generated Phase
3	Generated Phase
4	Generated Phase
5	Chime Phase
6	Generated Phase
7	Generated Phase
8	Chime Phase
9	Game End

Table 1: Game's phase progression

Generated phases will output note sequences that are procedurally generated by the game. The game will continue to generate Notes until the boss' health goes below a threshold. Phases that are not pre-scripted or chime phases are by generated phases.

3.4 End Condition

In order to win the game, players must reduce the boss's health to zero by performing attack actions. If the player's health reaches zero from missing Notes, getting hit by boss attacks, or a combination of both, they lose. When the player reaches either end condition, they are provided the option to return to the main menu, where they can choose to play the game again.

4 Design

For the first two terms, our team focused heavily on the design of *Doldrum*. Our goal was to create a game that put players in a rhythmic state while still engaged in the gameplay. We went through several iterations in designing the rhythm aspect of the game, and boss fight structure. During this process, many gameplay elements were made, then redesigned. Because our team had a very large scope in the very beginning, we were also very conscientious in cutting any elements that were not core to the experience goal.

4.1 Platform

Doldrum features a platform that the player stands on at all times. This platform conveys the player's limited play space and the player, xylophone, and Highway are all bound to. When the player moves via the respective input command, the platform and all of its bound components will move--carrying the player and xylophone with it. Likewise, the Highway, Boss, and some stage elements will rotate to ensure that they are always facing the platform and thus the player.

4.2 Mallets

As *Doldrum* is primarily a game about hitting a xylophone, the players' hands are instead represented as mallets. These mallets serve several purposes:

1. Hitting the xylophone faces
2. Indicating when the player is doing well
3. Providing haptic feedback when the player hits the xylophone

4.3 Notes

Notes are the small circular objects that travel down the Note Highway. When Notes reach the player, they must strike the respective xylophone face. With Note hitting as the main action players are doing, successful hits needed to elicit strong player feedback. For visual feedback, Notes are "popped" off

of the Note Highway. For audio feedback, an airy and tonal xylophone sound effect is played. For haptic feedback, the Vive controllers vibrate.

4.4 NOTE HIGHWAY

Doldrum features a standard gameplay loop that is similar to that of other rhythm games such as *Guitar Hero*. In both games, small circular notes travel from their spawn point towards the player and the player must hit these notes as they reach them. These notes spawn at the base of the Cuckoo Boss and travel along what is referred to as the Note Highway. The Highway is one of the core elements of *Doldrum's* design and implementation due to the Note Highway moving generated notes to the player.

The Note Highway was designed to fulfill three distinct purposes. First, it had to provide the player with easily identifiable rhythmic actions to perform. Second, the Note Highway had to provide a stable platform for the player to be able to more easily identify the musical timing of the note. Finally, it needed to clearly indicate when the player can make either an attack or move action. These three tasks were all accomplished in a similar way. Rhythmic actions are indicated via the Notes. Musical timing is indicated via thin Beat Bars that travel down the highway. A Beat Bar reaches the player on every beat of the Metronome. Additionally, certain Beat Bars spawn as Input Bars. These Input Bars allow the player to make an Action Input when they reach the player.

Figure 14: Note Highway with Beat Bar (left) and Input Bar (right)

The highway is a long, linear path that stretches from the Cuckoo Boss to the player. Its subcomponents travel along its path from the Cuckoo Boss to the player, thus allowing the Highway to serve as a visual bridge between the player and their opponent. Its major interactable subcomponents include the Notes and Input Bars.

Oncoming Notes provide the player with constant rhythmic actions to perform. Their purpose is to reduce downtime between the player's Action Inputs. The Notes require that the player constantly hits the xylophone. A given Note will correlate to exactly one of the player's five xylophone faces.

When an Input Bar is spawned and travels down the Highway, it is an indicator to the player that they must make an Action Input. These bars are visually flashy, colorful, and noticeably larger than the Notes and decorative Beat Bars. They were designed this way to indicate that the player is able to interact with the Cuckoo Boss.

In early designs, *Doldrum* did not feature a Note Highway. Rather, all of the player's rhythmic actions were dictated by the player. This was abandoned and the Note Highway was implemented in order to provide more structure for the gameplay.

4.5 XYLOPHONE

The player strikes a xylophone using motion controls. As Notes travel down the Note Highway, the player must strike the corresponding xylophone face. This is conveyed through the color of the xylophone matching the color of the Note. Additionally, the Notes travel down linear lanes of the Highway. Each lane leads to the corresponding xylophone key.

Initially, the xylophone was planned to be a cube-like entity that the player could strike to activate different actions. In this early design, the cube's faces acted as the individual faces of a drum, such that the top and sides of the cube were all separate faces. This structure was quickly abandoned in favor of a more traditional drum kit-like structure due to the difficulty of reliably hitting flat surfaces that faced different directions. However, when building out the Note Highway, we found multiple issues with using the drum kits. By emulating the positioning of a real drum kit, some faces of the drum were harder to hit than others. We attempted to make the drum faces surround the player at a uniform distance but ran into issues with aligning the faces to the Highway's lanes. With Notes coming down the Note Highway, it was hard to discern which face went with which Note. Later, the drum kit design was replaced with the current xylophone appearance. With the xylophone design, all of the faces the players hit is at a uniform height and neatly lines up with each lane in the Note Highway, allowing for overall neatness and readability.

4.6 CUCKOO

The boss character, the Cuckoo Boss, is the singular opponent the player plays against in *Doldrum*. The Cuckoo constantly faces the player and attacks them randomly when an input bar spawns. It also has an internal health bar. As its health depletes from player attacks, it advances the progression of the game. As the game progresses, the procedurally generated note sequences become more challenging and Chimes are introduced as a mechanic.

The initial prototype for the behavior setup was built in Unity. Unity was more familiar to Kelly Zhang and she was able to program basic boss behavior in a few hours. This prototype featured a top-

down perspective of the boss character, a player, and three lanes representing the Note Highway. The player was able to attack, dodge, shield, and parry using keyboard controls. The boss was able to turn to face the player, attack, and parry. Since the prototype was played from a top-down view, gameplay did not provide much insight on how players would feel performing the same actions in virtual reality with a first-person view. Though the gameplay between the boss and the player were not useful, the main takeaway ended up being how the boss' behavior could be technically implemented.

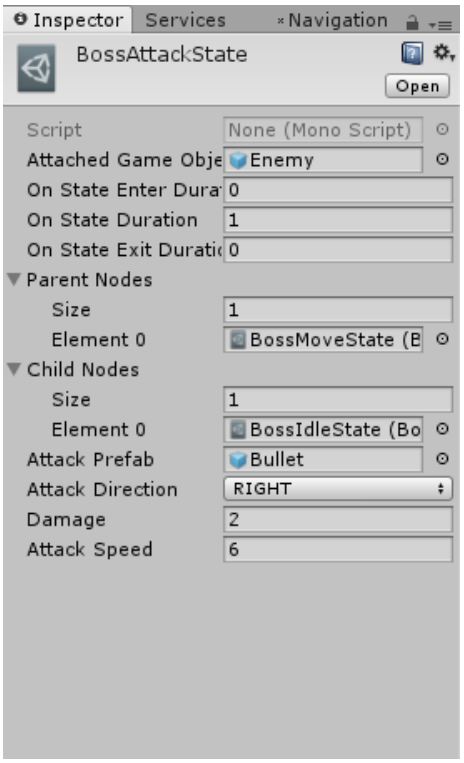


Figure 15: Unity Editor view of BossAttackState instance

4.7 CHIMES

During chime phases, a circular set of chimes descend from the top of the game's level and surround the Cuckoo. These chimes protect the Cuckoo from getting hurt by player attacks. However, the game will select one chime as vulnerable to the player's attack. When a defined number of vulnerable chimes are damaged, the game will progress to the next phase. Players will have to perform dodge actions for their attacks to reach the vulnerable chime. When designing the boss fight mechanics, we wanted players to utilize the actions that they have in response to the boss's actions. The shield action was originally the action that encouraged players to shift their platform around the stage to attack vulnerable areas on the boss. Since cutting the shield action from the Cuckoo, we incorporated our original design goals into an environment piece, the chimes.

4.8 User Interface

The player interacts with the user interface (UI) in *Doldrum* by striking the xylophone faces. Each face is labeled with a menu command such as “left”, “right”, and “select”. UI elements such as text and images are displayed to the player on a 3D panel in the game world.

In non-VR games, the UI is typically positioned so that it is constantly rendered in the foreground of the screen. This approach does not work in VR, as our eyes are unable to focus on text and UI elements that are close to the camera. [40] Therefore, the UI had to be displayed in the game world as a 3D element. The simplest way to implement this was to create 2D panels that would display our UI and place them in a 3D space.

Additionally, we wanted a way for the players to interface with UI elements without having to teach them additional controls for menu navigation. The solution to this problem was to integrate menu navigation to the existing xylophone faces.

4.8.1 Main Menu

Figure 16: The Main Menu

The main menu is the interface the player sees upon booting the game. The player enters the game in a dark room. The only items visible are the xylophone and the mallets. The mallets are encased in a floating bubble above the xylophone. This interface is to encourage the player to reach their controllers to grab the mallets. Once the player grabs both mallets, they will stay attached to the player’s hands for the remainder of the gameplay duration. The main menu interface will then automatically open,

introducing the player to the game's title. In the main menu, the player can select from the following options:

- Start Game
- Tutorial
- Calibrate Drum

The start game option is purposefully not set to the center, forcing the player to get used to hitting the xylophone faces before playing the actual game, instead of potentially and accidentally hitting the center to start.

4.8.2 Tutorial

Figure 17: The first slide of the tutorial

The tutorial menu is accessed via the main menu and is a series of slides with images of how to play the game. The player toggles between pages by hitting the leftmost and rightmost faces of the xylophone. When the player is satisfied with reading the contents of the tutorial, they hit the center face to go back to the main menu. Originally, the tutorial section was intended to be interactive and walked the player through the controls of the game. Due to time constraints, it was scrapped for the slides.

4.8.3 Pause Menu

Figure 18: The Pause Menu

The pause menu is accessed by pressing the menu button on the HTC Vive controller. This menu button is located above the touchpad. This menu is only accessible when the player has already started the game to avoid collisions with other menus. As the name indicates, the menu pauses the current game and provides the following menu options:

- Resume Game
- Main Menu
- Calibrate

4.8.4 HUD

To inform the player on how well they're performing during the game, a HUD (heads-up display) was created. The HUD is placed to the left of the player's peripheral vision, in order to not obscure the Note Highway or xylophone. A consequence of placing the HUD outside direct vision is that they are forced to look away from the Note Highway in order to see it. This may lead to undesired behavior from the player where they only focus on the highway and do not notice their performance. While it's a minimal amount of impact to the overall gameplay experience, the HUD should supply the player information in a comfortable manner rather than a stressful one.

4.8.4.1 Health Display

Figure 19: The full health display

The health display is a red bar that depletes when the player misses a note and when it is attacked by the boss. A heart is displayed to the left of the bar, indicating to the player that it represents health. The heart plays a beating animation, and it serves a secondary purpose of indicating the rhythm of the game.

4.8.4.2 Note Streak Display

Figure 20: The Note Streak Display

The Note Streak Display indicates how many consecutive notes a player has hit without missing. It resets to zero when the player misses a note. While it does not impact gameplay, it was added with the intent of providing a sense of satisfaction to players who are performing well.

4.8.4.3 Frenzy Meter Display

Figure 21: Fully depleted frenzy meter (left) and fully charged frenzy meter (right)

The frenzy meter is a blue zig-zag meter that indicates how close the player is to entering the Frenzied State. Since Frenzied attacks are lightning-based, the appearance of this meter is jagged. In addition, when the meter is filled, a lightning effect appears on the meter to indicate that it is “charged up”. A sound effect also plays in case the player’s active gaze is not on the HUD.

4.9 ORIGINAL SCOPE

Doldrum has changed greatly over the course of development. The game originally featured five boss fights, each of which would represent a person or step in a musician's journey towards becoming great. This theme was inspired by the film *Whiplash,* and was meant to be unsettling to the player, leaving them with a feeling of inadequacy. [37] The idea was to create a horror game that played on existential fears instead of traditional scare techniques.

During the fights, the player would be able to play along with the rhythm of the song to build up power for their next action. The actions would be triggered by a measure-long series of drum hits that would trigger an *action*. The *actions* included: a multi-directional attack, relocation of the player, and a shield that would form around the player. The player would choose which *action* they wanted to perform based on the current situation. The act of performing a series of inputs in order to trigger a discrete action was inspired by *Patapon*.

Ultimately, four of the five bosses were removed due to time constraints. With the removal of the bosses, the narrative was no longer needed. Additionally, the complex *action* system was replaced with a simpler system with less possible actions.

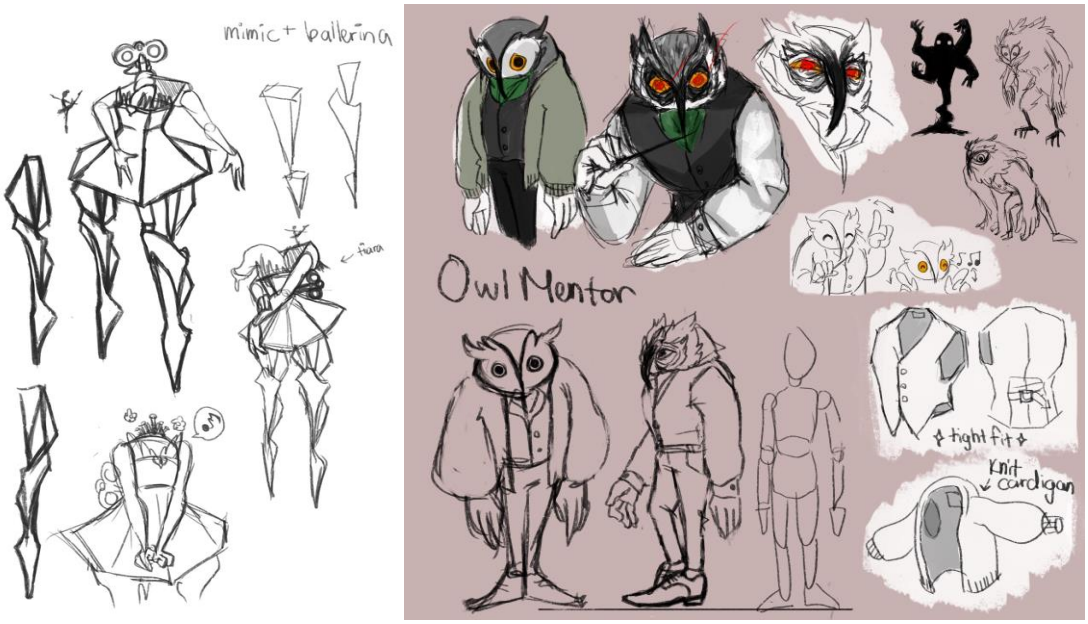


Figure 22: Concept art for two abandoned boss designs

4.9.1 PlayStation VR

Originally, we wanted to gain experience and develop for a console system. The only headset that can be used with a console is the PlayStation VR, a peripheral to the PlayStation 4. There are kits

available to develop for the device using the Unreal Engine. The game would have a potentially larger audience of players and a smaller pool of games to compete against.

When investigating how we could obtain a development kit from Sony, we found out that it was difficult to achieve as a student team. The registration process required us to either be a corporate entity or a university program. Upon requesting assistance from the IMGD department for a development kit at the beginning of summer 2017, we ran into issues primarily due to miscommunication. The PSVR equipment was purchased but there was no word on whether or not an application was submitted. If we were to continue pursuing a development kit, it would have significantly cut into our development time. As a result, we did not use the PSVR. This was a misuse of the IMGD program's budget and we sent a formal apology to the department head, Professor deWinter.

5 ARTISTIC IMPLEMENTATION

With specific aesthetic goals in mind, it was necessary for there to be a uniform artistic direction. The process generally starts off with Kelly Zhang creating 2D concept art for the asset to be created in 3D. The concept is then passed to Kent Fong to model and texture. If the asset required animation, it was then rigged and animated. During the various stages of the 3D production pipeline, several passes of the asset were imported into Unreal to make sure it looked right in engine. Once it was in engine, Matt Szpunar worked on the lighting that set the tone and mood for the game.

When designing the visuals, we decided to create very few objects of concern for the player in order to provide clarity and prevent confusion. For example, the notes are bright and contrast the dark highway. The Cuckoo is mainly warm colored, with browns and dark reds. The xylophone contrasts the Cuckoo with cool colors like turquoise and blue. Additionally, when the player has to react the Cuckoo's attack, the entire scene begins to glow red as a signifier. The simple colors and shapes provide a clear experience.

Over the course of two terms, Laurie Mazza also helped us with 3D modeling environmental assets, allowing Kent to focus on the Cuckoo.

5.1 CUCKOO

The Cuckoo Boss design can be broken down into three sections. The body is based off a cuckoo clock, the wings and feet are based off birds, and the head and arms were inspired from one of our scrapped bosses as shown in *Figure 23*. We liked the look and feel of this scrapped boss so we decided to incorporate into with our Cuckoo Boss.

Figure 23: Hydra boss concept art

Figure 24: Early concept art of the Cuckoo Boss

Early concept art of the Cuckoo shows a very symmetrical and stiff boss model. A redesign early in the modelling process was proposed to introduce more curves into the body. A curvier body makes the boss more interesting to look at and also makes it a more organic creature.

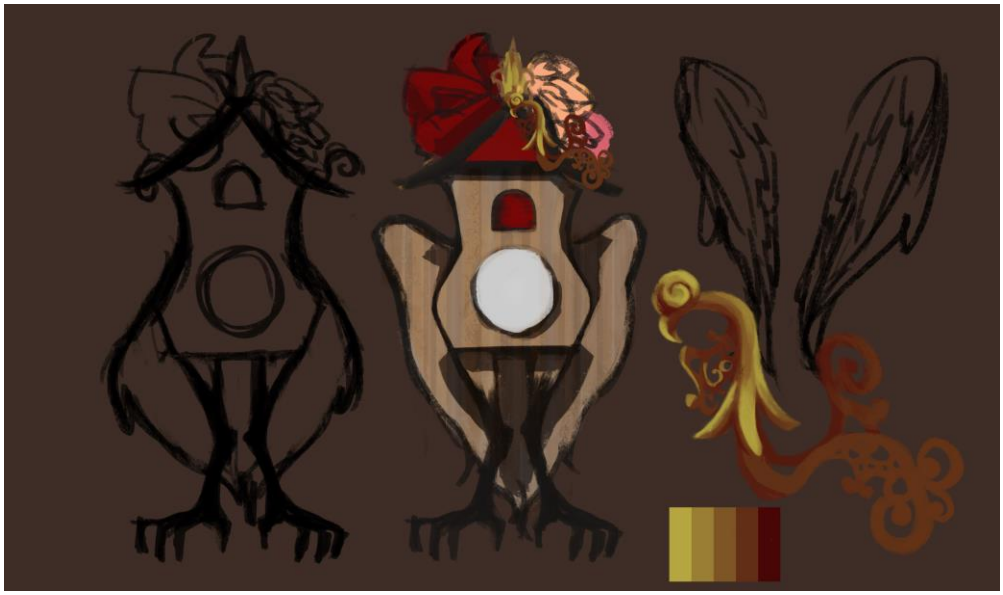


Figure 25: Concept art of the Cuckoo Boss redesign

The Cuckoo body started out as a rectangle in *Autodesk Maya* with simple extrusions to block out the clock face, mouth, and roof. It was then brought into *Pixologic Zbrush* to add finer details and variance to the body for a more organic feel. Inside Zbrush, various tools were utilized to create the curves and wood panel texture on the body. The roof was done by duplicating and layering a single rectangle tile multiple times. Since the extremities were based on living creatures, they were done entirely in Zbrush for an organic look.

Figure 26: Final render of the Cuckoo Boss model

For the Cuckoo, bad wing deformation could ruin the animation. Therefore, special attention went into the wings. The wings went through several iterations. The wings first started out as a single complete

mesh with modelled feathers but did not look realistic. In Zbrush, *nanomesh* was used to populate the wings with feathers that did look nice, but still did not achieve the desired deformation during animation. The single mesh did not bend in a convincing way for the player to understand. These wings were eventually scrapped after studying birds and seeing how they have layered feathers. The layered feathers were mimicked during modelling. Four different feathers were created in Maya and Zbrush to give variation between the primary and secondary feathers. The final wings with individual feathers resulted in aesthetically pleasing wings that correctly deformed.

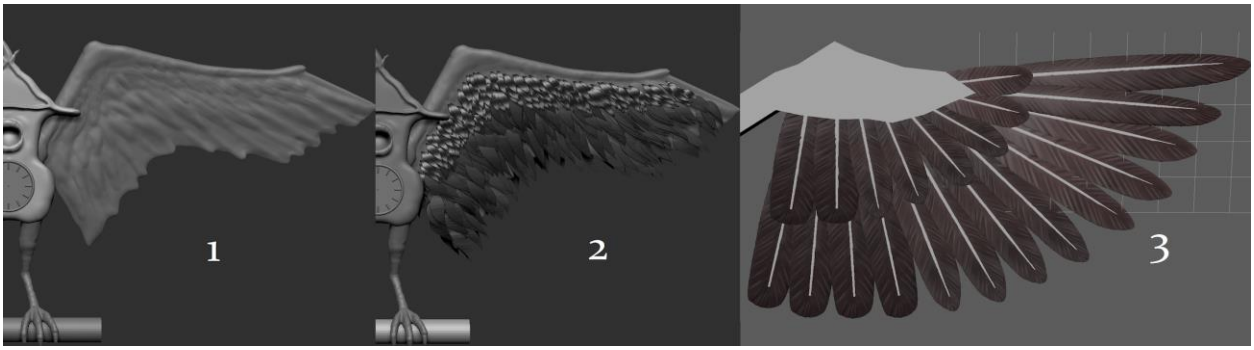


Figure 27: Iterations of the Cuckoo Boss' wings

Textures for the Cuckoo were done entirely in Zbrush. All diffuse textures for the Cuckoo were hand painted for a unique and cartoon-like design. The body wood panels were painted with a tileable wood texture that was manipulated in *Adobe Photoshop* to match the overall aesthetic. Diffuse maps for different components were painted and generated in Zbrush. Normal maps were baked from high poly meshes using *xNormal*. Specular maps were made in Photoshop.

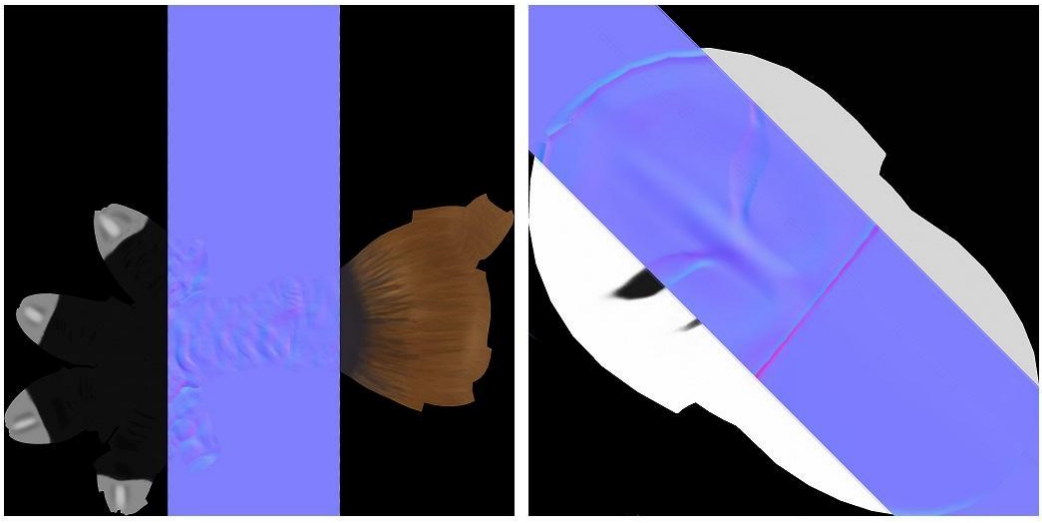


Figure 28: Texture sets for the Cuckoo's diffuse, normal, and specular maps

Rigging for the Cuckoo was done entirely in *3ds Max* utilizing their CATRig system. 3ds Max was chosen over other software because Kent already had experience with it. Also, CATRigs are fairly intuitive to create and they work seamlessly for animating with 3ds Max.

First, the pelvic bone is placed around the Cuckoo's center and then the spine and other extremities follow. CATRigs are simple to use and come with built-in Inverse Kinematics and Forward Kinematics for the arms and feet. In the Cuckoo's case, the wings were rigged as an extra set of arms, with extra bones for each feather group. The head and neck are rigged as a second spine while the roof panels are just extra bones. After placing the bones in the correct places, the mesh was ready to be skinned.

Figure 29: Rigged and skinned Cuckoo mesh in 3ds Max

The skin weighting process took a few days due to both the asymmetrical shape of the Cuckoo and the fact that each feather had to be skinned to a specific bone. Each primary feather is skinned to a specific bone because it allows for greater control of the feathers while also providing realistic

deformation of the wing. This method of skinning each feather was inspired by a wing rig test by the *YouTube* user 3D_guy_2008 [1], who created wings with individual feathers.

Figure 30: The Cuckoo wing's range of motion

The animations were done in 3ds Max because a CATRig was used. Animations for the Cuckoo included an idle, attack, hit, and death. All animations besides the death are done with the Cuckoo hanging from a bar. This meant most of the core motions came from the hips and spread outwards. The idle is a simple sway from left to right that loops and is what the player sees the majority of the time. The attack animation was inspired slightly by both *Ho-oh* from *Pokemon* and the *Hadouken* from *Street Fighter*[18, 23]. The attack had to be split into buildup, attack loop, and follow through. Splitting the animation allowed the programmers to decide the length of the attack for any attack variations. In engine, the animations gradually blend depending on the game event. Animations were done in 96 frames per second because the HTC Vive's refresh rate is 90 hertz.

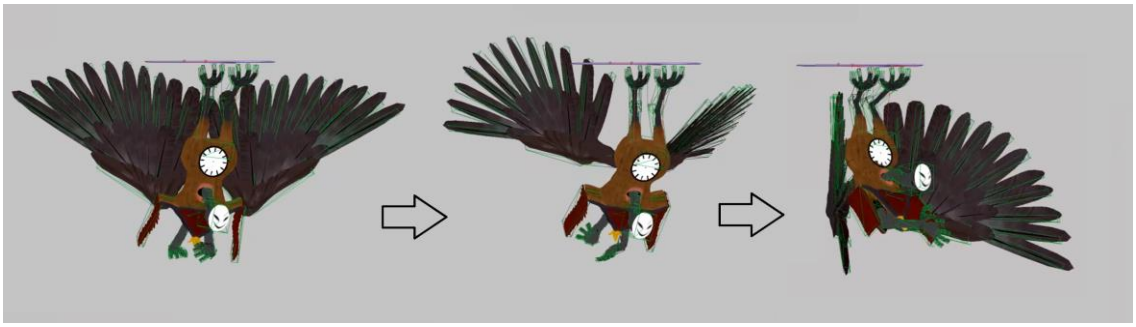


Figure 31: The Cuckoo attack animation flow

5.2 NOTE HIGHWAY

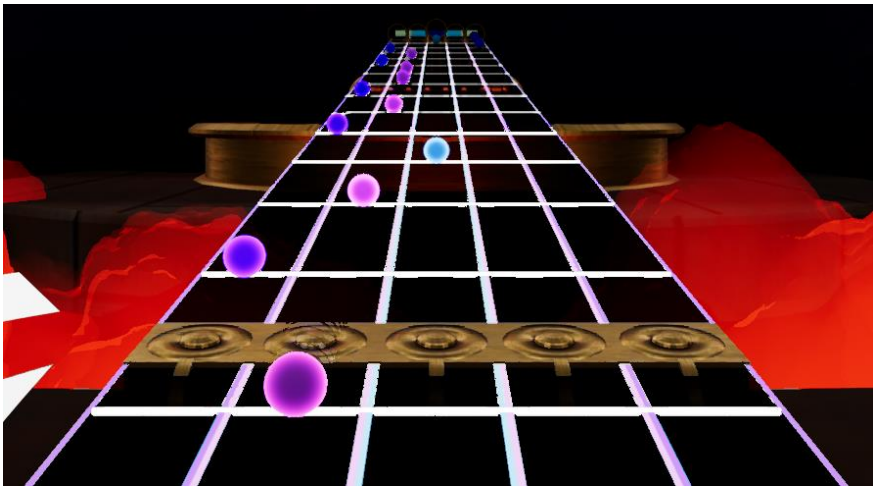


Figure 32: Notes travelling down the Note Highway

The Note Highway's appearance was designed to be easily noticeable by the player without being too visually loud. Additionally, its appearance had to convey a sense of musical time to the player. This was accomplished via the Beat Bars, with one bar reaching the player on every quarter-note of the metronome.

Beat Bars are visually distinct from the Highway's base color of semi-transparent black. They are thin, fully-opaque white bars. However, the bars are not overly bright nor do they glow. The Highway's simple and dark color scheme was chosen due to it being visually distinct from its surroundings without taking the attention away from the Boss, Notes, and other interactables. Likewise, the bars' white color was chosen to ensure that the player notices them.

The Highway itself is made up of five distinct lanes that each correlate to a given xylophone face. These lanes are separated by colorful dividers that mirror the coloration of the xylophone faces. A given lane's dividers share a color with that lane's respective Notes, thus allowing for players to easily distinguish which Xylophone face an oncoming Note correlates to.

5.3 NOTES

The visual design of the Notes accomplishes two primary goals. First, the Notes' simplistic spherical appearance and light border ensures that they are easily noticeable. Second, a given Note's color will match its target xylophone face. Thus, players can easily discern which xylophone face to hit from Note appearance alone.

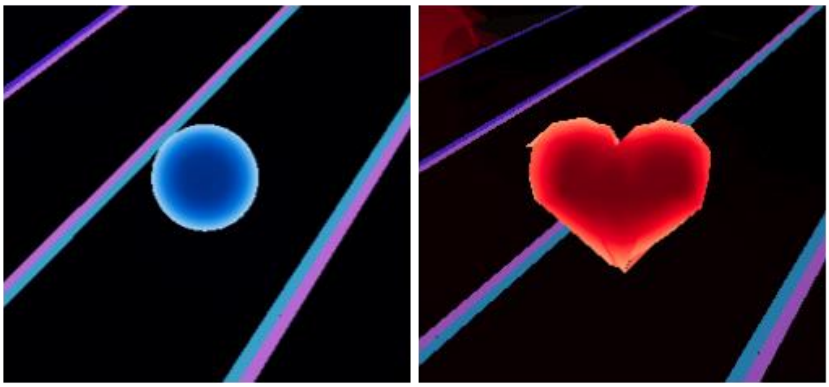


Figure 33: Default Note (left) and Healing Note (right)

Healing notes have a unique shape and color regardless of their assigned lane. All healing notes are bright red and shaped like a simple heart. This drastically different appearance ensures that players are aware of which Notes are healing notes. Similarly, the player's own health bar features a bright red heart, thus allowing for players to easily associate the uniquely shaped Notes with health restoration.

The appearance of the Notes was accomplished by applying a material instance to a basic sphere mesh. The material instance is derived from a base material that features a definable color, a Fresnel that gives the Note its colored border, and a constantly fluctuating emissive intensity. This base material has several child instances that allow for easy modification to the material's base color, Fresnel color, and intensity. There is a separate instance for each of the five lanes and additional instances for missed notes and healing notes.

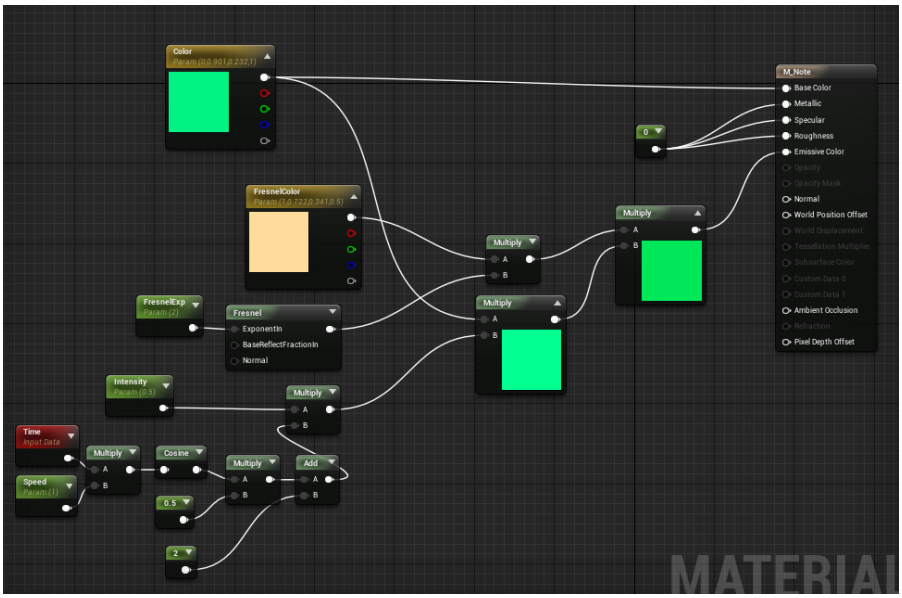


Figure 34: Material graph for the base Note material

These material instances are dynamically applied to a given Note when that Note is spawned. When a note is spawned, it is assigned a target lane and whether-or-not it is a healing note. If it is a healing note, the Note's color and shape are changed to a bright red heart. If the Note is not a healing note, its color is set to the color of its respective xylophone face and lane.

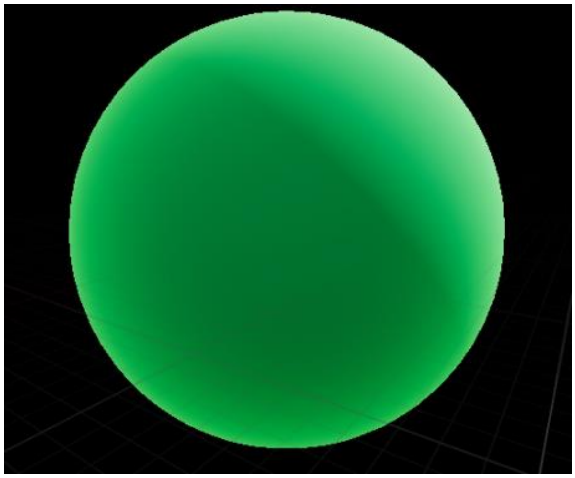


Figure 35: Note material viewed in the Unreal material editor

5.4 PLATFORM

Visually, the platform matches the brass appearance of the xylophone and Highway valves. The platform is designed to imitate the appearance of a brass instrument in order to match the musical aesthetic of the game. It has piping that borders the edges of the platform that meets and joins into a bell which resembles one of a brass instrument. These pipes act as railings that serve as indicators of where the playspace ends. Additionally, the attacks exit through the opening of the bell.

Figure 36: Final render of Platform mesh

5.5 MALLETS

Figure 37: Comparison between the original unicorn-head mallets (left) and the final mallet model (right)

The mallets were modeled to appear identical to real xylophone mallets. While the design fits with the aesthetic of the game, we original experimented with a louder and more intricate design . Since the Cuckoo Boss was bird themed with fire elements, we experimented with the player representing an animal that fit with the lightning element. When browsing inspiration for lightning elemental creatures, we found the *Kirin* from the *Monster Hunter* series, as well as weapons from *Alice: Madness Returns* as examples of unicorns with lightning.

Figure 38: Concept art of unicorn head mallets(right) with Monster Hunter and Alice inspirations (left)

Unfortunately, while making the mallets in 3D, we felt that the unicorn mallets would be too distracting and it was unclear how the mallets should hit the xylophone faces. Following this concept, we ended up reverting to a more traditional mallet.

5.6 XYLOPHONE

Early renditions of the xylophone were rectangular before changing to a more traditional circular look.

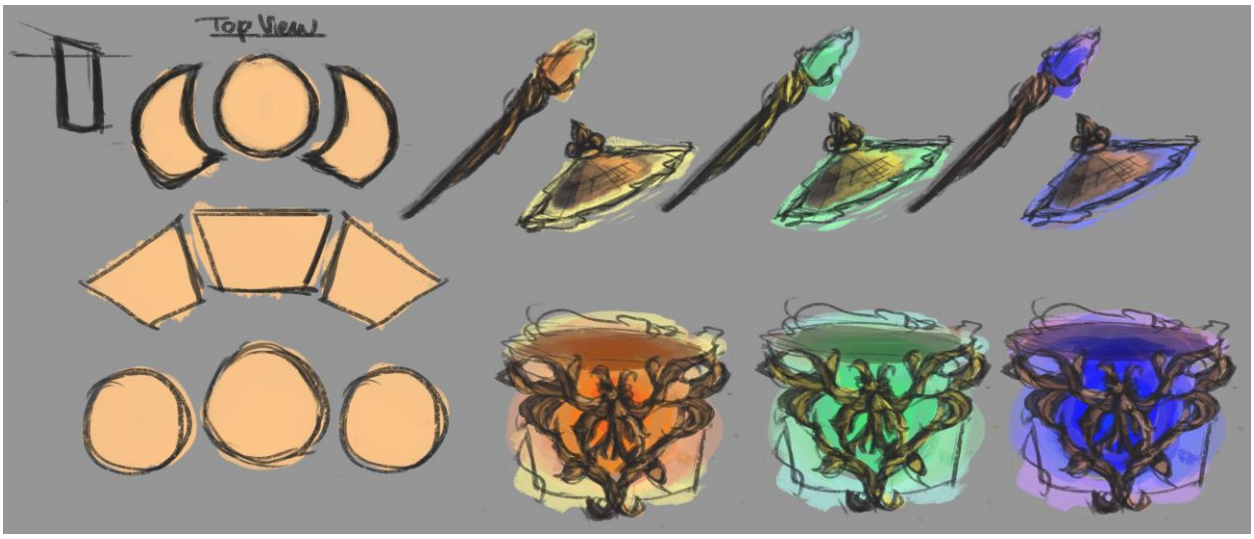


Figure 39: Early concept for the drums

The decision was made to move forward with a more rectangular drum face and we decided on a xylophone-inspired drum. Following the xylophone theme, the three drum faces and two cymbals were

turned into five simple xylophone keys. We decided on a cool color palette for the xylophone bars as a contrast for the warm colors of the Cuckoo.

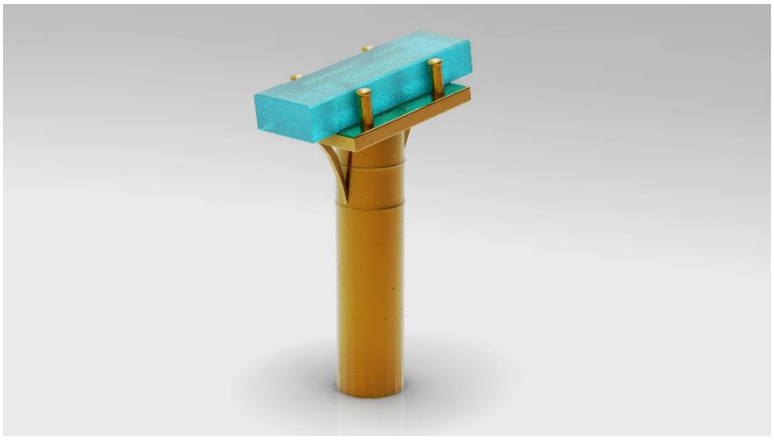


Figure 40: Final render of a singular xylophone key

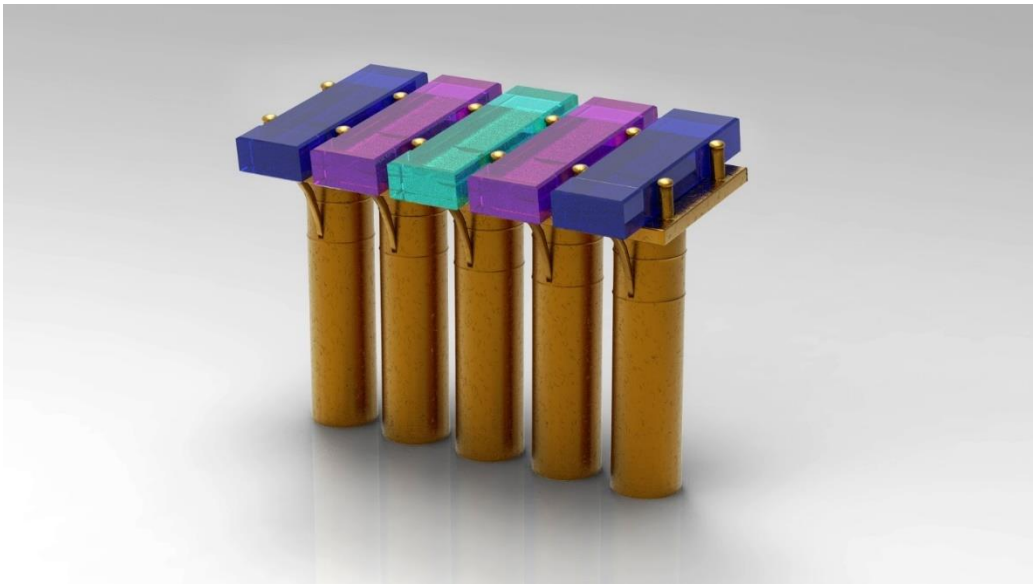


Figure 41 Final render of entire xylophone

5.7 ENVIRONMENT

The goal of the environment in *Doldrum* is to emphasize and complement the Cuckoo Boss. To achieve this, the centerpiece of the game world is a large stage where the Cuckoo hangs.

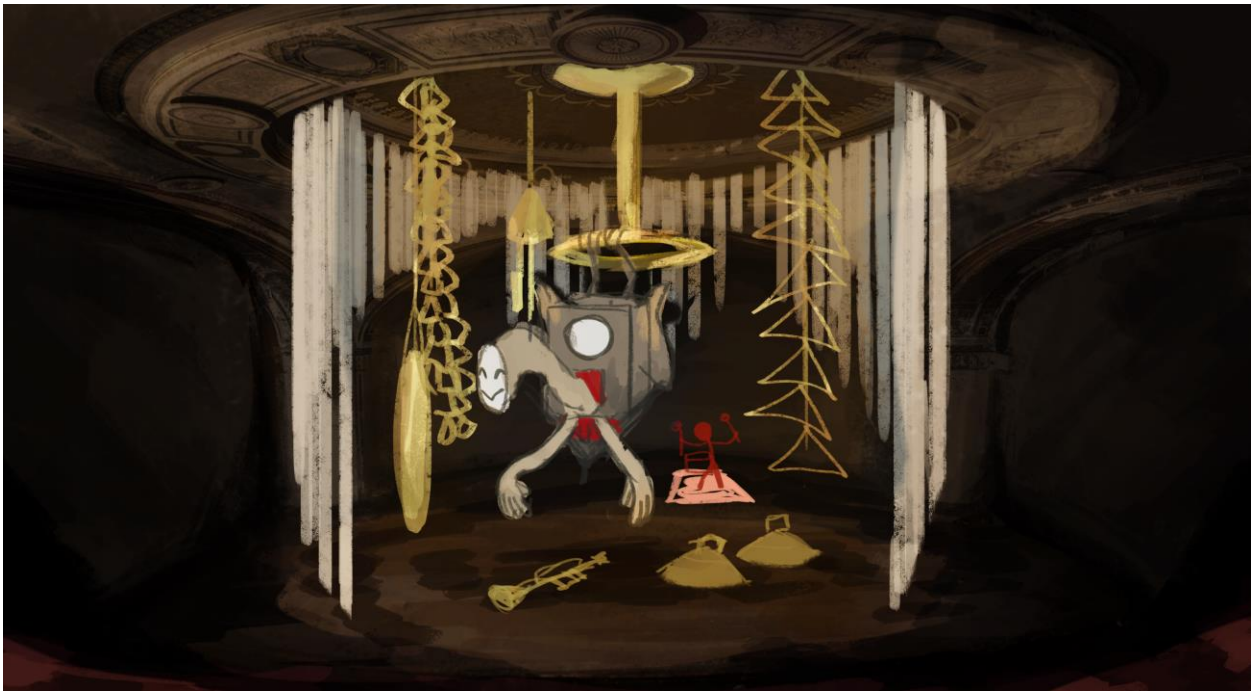


Figure 42: Concept art of the Cuckoo's environment

This stage is modeled after a theater. It consists of the stage floor and the backdrop. The floor is wooden with a raised metal platform in the center. On the outside ring of this stage, two spotlights are placed, pointed at the Cuckoo. The backdrop is a half-dome structure where the Cuckoo and chimes hang. All meshes that make up the stage (i.e. chimes, spotlights, backdrop) were created in Maya by Laurie Mazza.

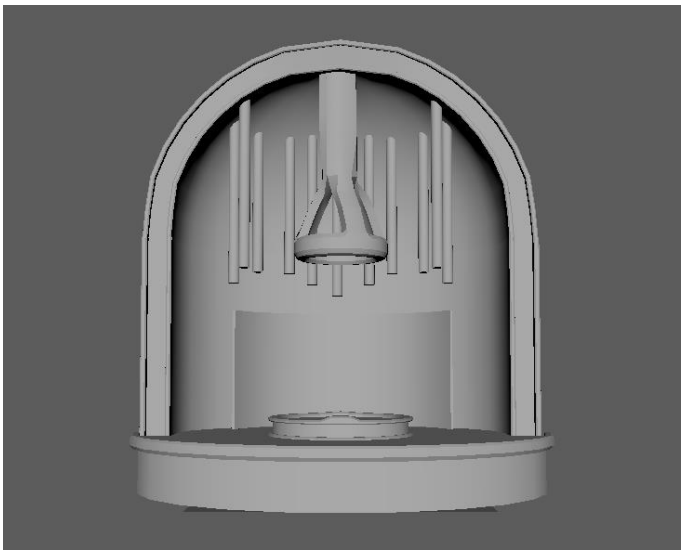


Figure 43: Grayboxed stage by Laurie Mazza

Figure 44: The full environment

Between the inner and outer sections of the stage there is empty space which is filled with a purple fog effect. The fog clouds provide a feeling of mystery and malevolence. These cloud effects were created using a material that uses noise in conjunction with the Time node to create a material that resembles a cloud when applied to a sphere. A constant rotation was then applied to these cloud objects which gives the appearance of an ever-shifting layer of fog when placed next to each other in large volumes.

5.7.1 Chimes

The chimes hang from the backdrop of the stage and are used by the Cuckoo during certain stages of the fight. These are designed to look like the chimes of a grandfather clock and are arranged to mimic the bars of a bird cage. The chimes are surrounded by an orange vortex of fire. These elements come together to match the visual design of the Cuckoo.

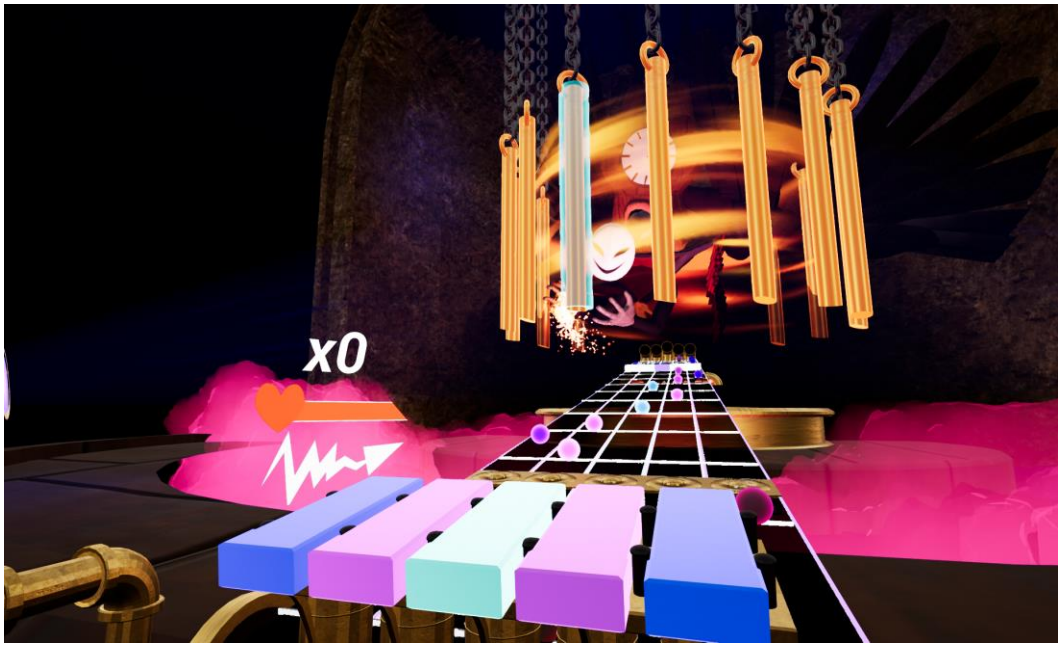


Figure 45: The Cuckoo readying an attack, surrounded by the Chimes

5.7.2 Lighting

The lighting was designed to frame the Cuckoo as the focal point of the scene. The overall environmental lighting is dim with the exception of two spotlights that are aimed directly the Cuckoo. These emit a light indigo colored light towards the Cuckoo, illuminating the Cuckoo and casting shadows onto the backdrop of the stage.

Until the player begins the fight, all of the lights are off. Only the fog, xylophone, and mallets are visible. Once the game has begun, the rest of the lights turn on and reveal the environment and Cuckoo.

Figure 46: Early stage layout with basic lighting

5.8 USER INTERFACE

The appearance of the HUD was meant to be simple and stylistic for player readability. We wanted players to be able to understand the UI at a glance. Despite efforts to blend the UI with the environmental aesthetic, the use of a painted UI and stylized serif fonts for the HUD made elements harder to read from a distance.

6 TECHNICAL IMPLEMENTATION

The approach to implementing features into *Doldrum* was iterative. We would often build a small piece of a large concept. Then we would test the piece to make sure the piece was functional and adhered to our design guidelines. Most importantly, if the piece was a gameplay attribute it had to feel good to play.

Additionally, by drawing the technical strengths from each programmer, Henry with Blueprints and Kelly with C++, we were able to construct two technical levels that neatly separated out gameplay logic and systems logic. By separating the logic, we were able to iterate on existing functionality with ease and break existing features. In this section, we will address the various systems, gameplay, and player interactions. For systems components such as the Phase System, Note Generation System and Cuckoo behavior, we will explain the inner workings and how it drives the gameplay loop. Gameplay components such as the Note Highway and Notes will cover techniques used to enhance game feel. With features involving player interaction, we will address how we ensured that VR interactions worked smoothly and reliably for all players.

6.1 METRONOME

Nearly all of *Doldrum*'s events and interactions with the player are dependent on musical timing. As such, it was very important that we develop a flexible and robust metronome system early in the development process.

The metronome system needed to satisfy the following core requirements:

1) Reliable timing
2) Ability to broadcast to objects when a beat happens
3) Easy to fix and change based on future design changes

To ensure reliable timing, we decided to depend on Unreal's Time Management system. The system allows programmers to set timers that would reliably countdown in real time, and have a custom

function be called when the timer reaches zero. While we could write our own timers, we would have to account for factors such as jumps in framerate. To keep the system implementation as simple as possible, using the Time Manager was our best option.

To satisfy the second requirement, an observer pattern was used. There is a `MetronomeController` object that controls the timing. `MetronomeListener` objects can register to a controller and a developer can assign custom functions to be called through the listener's OnBeat delegate. When the `MetronomeController` starts to countdown seconds for each beat, it iterates through its list of registered listeners and calls the OnBeat delegate. This pattern ensured that all objects dependent on beat-based timing are being triggered from the same source(s). It also allowed for flexibility for changes in implementation, since it would only need to change functionality in two objects. The flexibility here satisfies our third requirement.

The metronome system was implemented primarily in C++ and makes use of Unreal's C++ API. `MetronomeController` is derived from the `AActor` class, whereas the MetronomeListener is derived from the `UActorComponent` class. This allows the system's components to be implemented and managed on the Blueprint level, while still having access to Unreal C++ API commands.

Figure 47: Screenshot of a MetronomeController's editable properties

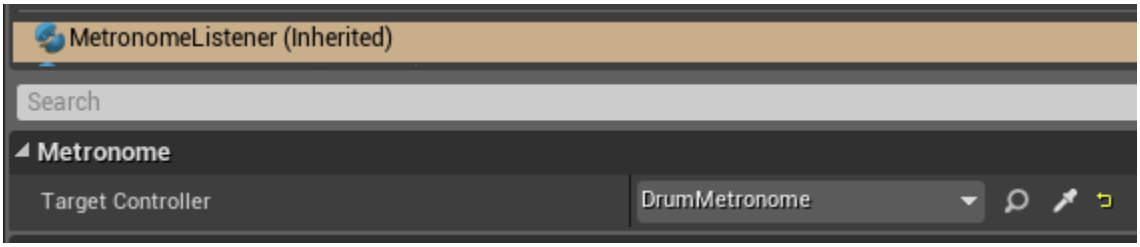


Figure 48: Screenshot of an actor's MetronomeListener component listening to a MetronomeController actor

For example, a programmer working on the Blueprints level begins by creating a Blueprint that derives from the `MetronomeController` class. The class will reflect its properties that are flagged as editable onto the Blueprint's property window. Moving this Blueprint actor to the active level will allow listener objects in the same level to register to it.

To enable objects to listen to the controller, the programmer attaches the `MetronomeListener` as a component of the objects. In the Blueprint Event Graph, they can initialize the OnBeat event as an event node.

Figure 49: OnBeat delegate as seen in Blueprints

As development continued, we realized that getting OnBeat calls from the listener is often not enough. Sometimes the game receives player input that is offbeat. The game needs to respond to this feedback such that is still on beat. This is explored further in the *Notes* section. The Unreal Time Management system offers useful function calls like `GetTimerElapsed` and `GetTimerRemaining` to get time values of when the next beat is or how long it has been since a beat event has been fired. Wrapping these calls around functions in `MetronomeListener` allows us to expose our wrapper functions for Blueprint use, where most of the game logic for *Doldrum* is. All objects that have the `MetronomeListener` attached will now have access to these functions.

6.2 CUCKOO

The Cuckoo acts and reacts to the following game events: metronome ticks, phase changes, and player actions (e.g. Attack, Dodge). The core of the Cuckoo's gameplay logic is built from Blueprints and extended from a class called `BossCharacter`. Its primary goal is to keep track of the boss' state.

The state machine in the base `BossCharacter` class was designed in a way to allow as much flexibility as possible for Blueprints to implement custom state behavior that was specific to a boss. This was before our team decided to scope the boss count down to one but is still viable to extend from if we were to continue implementing more boss characters.

At a high level, state objects dictate how long the boss takes to transition into the state, how long the boss stays in a state, and how long the boss takes to transition out. It also has a list of possible states to transition to. In the prototype, it also held a list of states it can traverse from. That was removed during the actual implementation into Unreal. Keeping a list of parent states added complexity to the state machine. Based on the boss design, it was only necessary for the state machine to know which states it

needed to traverse to. In the Unity prototype, there were parameters specific to a state, as shown in *Figure 28*. In the Unreal implementation, this was translated into a two-level system. These two levels are the C++ systems level and the Blueprints level. The systems level runs the transitioning of one state to another based on preset transition rules. The system can then make calls up to the Blueprints level which runs custom logic.

To define state transition rules, `BossCharacter` uses a state transition map, where the key is a Boss' starting state and the value is a custom data structure. This data structure holds the number of metronome ticks for a state to enter a transition, to stay in a transition, and to exit in a transition. This structure also allows us to have fine control over the timing of state transitions.

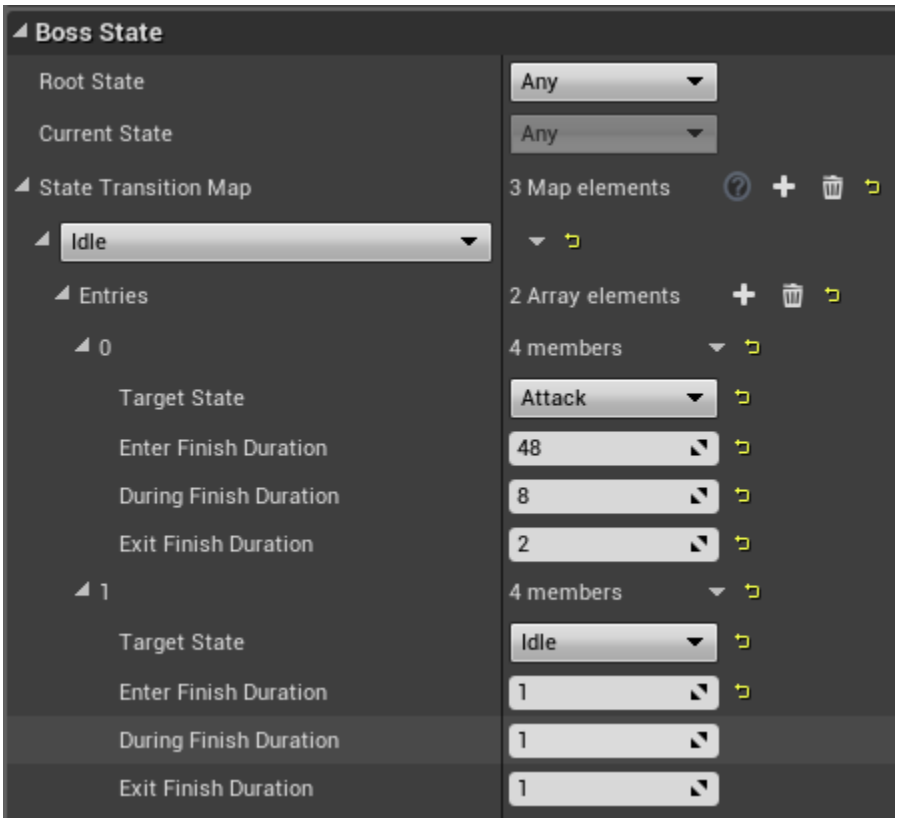


Figure 50: Screenshot of the Cuckoo's state transition map property

This map as shown in *Figure 29* can be exposed to Blueprints, allowing programmers and designers to easily adjust timing values and transition rules from the Unreal editor. When the state machine begins during runtime, these values can be referenced on the systems level and the state machine. It will begin at the defined root state and make the appropriate transitions as defined in the map. When the machine changes to a different state, it will make calls up to the Blueprints level with additional parameter values such as sub-state and transition stage. For the Cuckoo, we only made use of the

transition stages. Based on design decisions made during the implementation of the state machine, we decided not to introduce additional mechanics tied to sub-states.

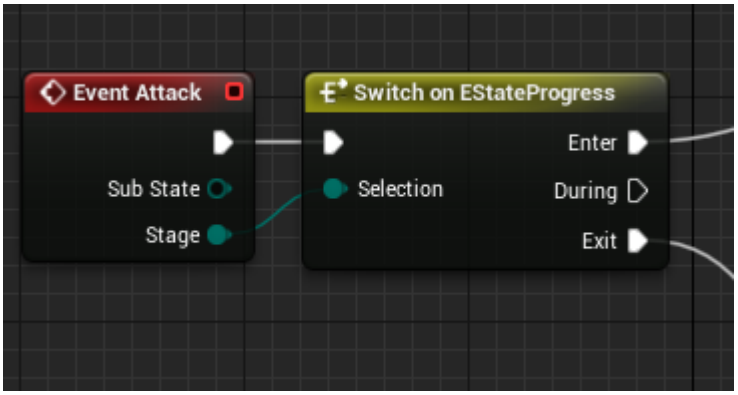


Figure 51: Screenshot of a part of the Cuckoo's attack logic in Blueprints

An example seen in the game is the Cuckoo's attack. When the state machine begins to transition to the `ATTACK` state, we set its enter transition duration to 48 ticks (3 measures). In its Blueprint, the logic for entering the `ATTACK` state is the following:

1) Play Cuckoo's buildup animation
2) Spawn a fireball object
3) Play buildup sound effect
4) Change environment colors and lighting

As the Cuckoo's attack is primarily buildup and fire, there was no need for us to implement additional logic for the `DURING` transition stage. When the Cuckoo is about to transition out of attacking the following is run:

1) Play Cuckoo's launch animation
2) Launch the spawned fireball object at Player's location
3) Play fireball launch sound effect
4) Reset environment colors and lighting
5) Object cleanup

6.3 NOTE HIGHWAY

The `NoteHighwayBP` class contains the functionality of the Note Highway. The Note Highway has five lanes in which Notes travel down. These lanes are constructed using Unreal's splines. By using the splines, we are able to dynamically change the shape of the Highway without needing an artist to

create a static Highway mesh. Elements such as Notes also make use of this mechanism by aligning themselves to positioning of the splines in the level's world space.

To provide the player with easily identifiable rhythmic actions, the `NoteHighwayBP` makes use of the `MetronomeListenerComponent` attached. On every beat, the Note Highway receives an `OnBeatEvent`, various calls are made, depending on which beat in the measure the game is in. On every fourth beat, beat bars are spawned to convey a sense of musical timing to the player. On every beat, the Note Highway will check to make sure the timing is in sync. If the timing is in sync, it will pull a `NoteStruct` produced by the Note Generation system, mentioned in a later section. When a `NoteStruct` gets pulled, a `NoteHighwayNoteBP` object is spawned, further discussed in the Notes implementation.

6.4 NOTES

Note objects are spawned as a `NoteHighwayNoteBP` by the Note Highway. A Note moves down the Highway via a Blueprint `timeline` node that begins when that Note is spawned into the game world. Each Note has its own `timeline` that updates the Note's location and color intensity over its duration. Additionally, the `timeline` features several individual event notifiers that trigger additional functionality when they are reached.

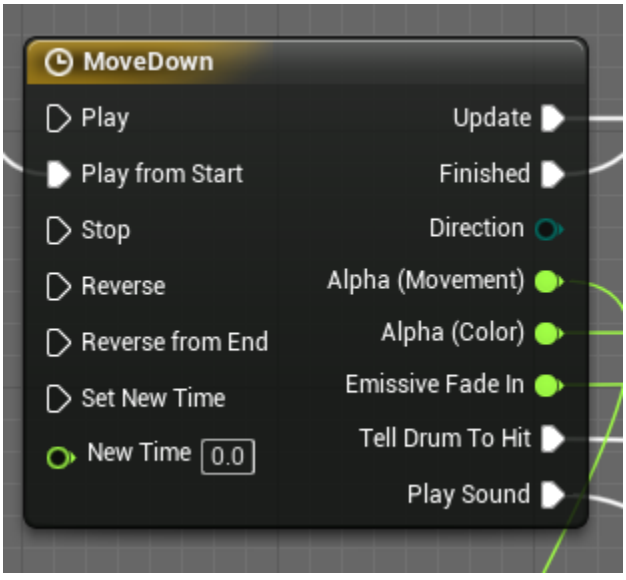


Figure 52: Note's MoveDown Timeline Blueprint node

The `timeline` features three separate alpha values for the Note's location along the Highway, its color over the course of its journey, and for its gradual increase in emissive intensity. The `timeline` also has three separate event calls for when the Note reaches the end of the Highway, when it should tell the

xylophone to expect a player input, and when it should play its musical sound. These events are all called at explicitly defined times.

The event that tells the xylophone to expect a player input is called significantly earlier than when the Note finishes its movement down the Highway. This is because players are expected to hit the xylophone faces when a given Note reaches a line of bronze valves, thus allowing for the Note to travel past these valves if the player misses.

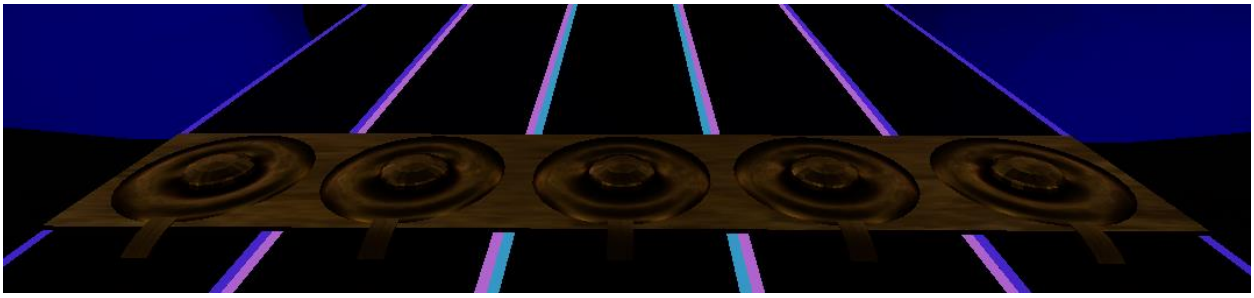


Figure 53: The Highway Valves that the Notes are struck by

If the Note's respective xylophone face is not hit by the player when the Note reaches the valves, the Note will continue down the Highway. When the `timeline` finishes and the Note is at the bottom of the Highway, it is despawned and the player loses some health. However, if the player does hit the xylophone face successfully, the Note's `timeline` is stopped and the Note is instead popped upwards.

6.4.1 Popping

Whenever a note is correctly hit a combination of visual effects occurs, known as Note Popping. The act of popping cancels the Note's movement down the Highway and instead causes the Note to launch into the air. This launch is handled by Unreal Engine 4's native physics support. Thus, after the note is airborne for a few seconds the Note will fall back down.

Normally, Notes are not considered to be physics objects; they do not abide by gravity nor do they have any collision with other objects. Until a Note is popped, it is only moved via its movement `timeline`. When a Note is popped, the Note's physics properties are enabled for a short time to allow for the launch effect. Additionally, the Note is also set to gradually shrink in size after it is popped. This allows for the Note to smoothly exit the game world after it is no longer needed.

6.4.2 Guaranteeing On-Beat Sound

Successful Note hits result in a xylophone sound. The player has approximately one-third of a second to hit the respective xylophone face when a Note reaches the valves. Thus, it is possible for players to hit the Notes slightly earlier or slightly later than what would be considered to be on-beat. To

prevent the resulting sound from sounding off-beat due to imperfect timing, the xylophone sound plays on-beat and independently from player input.

When a Note reaches the valves its xylophone sound is automatically played regardless of player input. This sound component is then stored as a temporary variable within the Note. If the player does not successfully hit the Note, this sound component is stopped and a *missed note* sound is played. All subsequent Notes will not automatically play their xylophone sound until a note is successfully hit. This sound system ensures that players are rewarded despite having imperfect reaction time. Through this system, successful Note hits will always sound as if they are being played on beat.

6.5 NOTE GENERATION

The generation of the note sequences in *Doldrum* is mostly procedural. It relies on a set of rules and patterns to dictate Notes placement and frequency on the Note Highway. However, the beginning of the game features a simple hand-written sequence for the player to play out. This simple sequence is done through the note generator's pre-scripted system.

The Note Generation system interfaces with other game components by using Blueprint Function Libraries. Unlike the Metronome System, and the Boss State Machine, Blueprint Function Libraries can be accessed by any Blueprint in the project. The libraries do not need to be instantiated as an actor or component in the game's level. Typically, the libraries are used to contain complex helper functions. The Note Generator was a single entity and its sole purpose was to generate Note sequences and provide them to other systems, both on the Blueprints level and the C++ level. On the C++ level, we can simply reference the system by getting its static instance. Blueprint Function Library acts as the bridge to access the instance on the Blueprints level.

Using Blueprint Function Libraries, we were also able to write C++ functions that were especially useful to the Note Highway. Such functions like `PeekNextNoteSet` allowed the Highway to do a sync check on the next note before popping it from the queue.

6.5.1 Procedural Generation

With procedural generation of Note sequences, we were able to provide variability to the gameplay's linear structure. From a design perspective, we had already scoped down gameplay to a guided experience where players are to strike Notes when they reach a certain point on the Note Highway. Since we had already scoped down the project significantly e, there was little room to introduce variability for the experience to feel fresh every playthrough. By procedurally generating the Note sequences, players can play different sets of melodies in each session. The trade-off to procedurally

generating sequences, is that some sets of Notes may sound out of place while others sound as if a human had composed it.

6.5.2 Grammar-based generation

Tracery, a tool built by Kate Compton, was used to prototype the ruleset language used for *Doldrum*'s Note generation [12]. The language uses context-free grammars to generate text. This grammar entails a start symbol, production rules, terminal, and non-terminal symbols. Non-terminal symbols can be replaced by other rules, and terminal symbols generate pieces of the final result.

For the purpose of creating sets of Notes that adhered to design goals, the following ruleset was built:

```
<<start>>          -> <measureA><measureB>
<measureA>         -> <four>
<measureB>         -> <three><action>
<four>             -> {<three><one>, <one><three>, <two><two>}
<three>            -> {<two><one>, <one><two>}
<two>              -> <one><one>
<one>              -> <fourth>
<fourth>           -> {<eighth><eighth>, fourthNote}
<eighth>           -> {<sixteenth><sixteenth>, eighthNote}
<sixteenth>        -> sixteenthNote
<action>           -> actionBar
```

Code Listing 1: Production rules for Note generation

To enhance readability, the terminating symbols `fourthNote`, `eighthNote`, and `sixteenthNote` were replaced with Unicode note symbols to emulate sheet music. The following sets were generated using the above ruleset. All sets are valid measures.

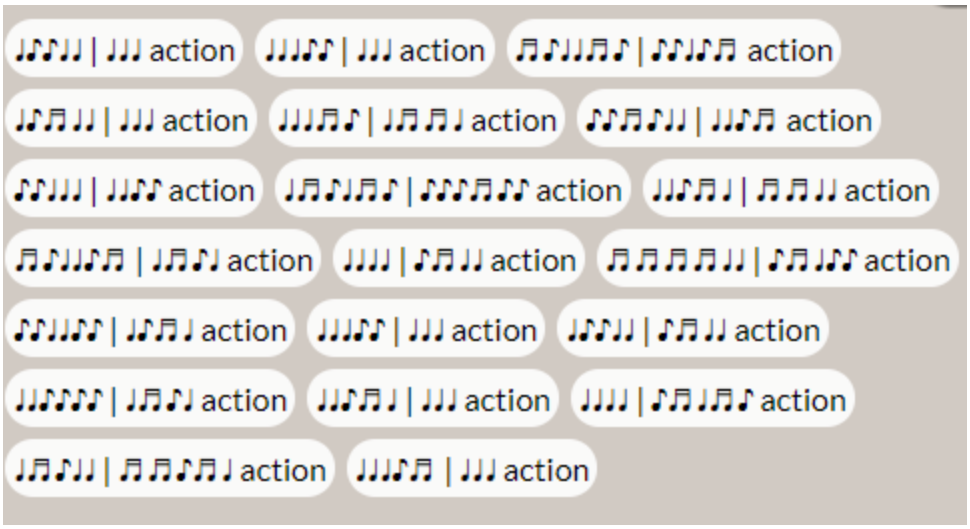


Figure 54: Sequences generated by Tracery prototype

Following the prototyping, a C++ system was built that reads a JSON ruleset, creates a graph, and traverses through it in order to generate Note.

6.5.2.1 Writing to C++

The core requirements needed for *Doldrum*'s Note generation are the following:

1. Ability to read in a ruleset
2. Ability to generate Notes based on ruleset
3. Have parameters that factor difficulty into generation
4. Ability to pass generated Notes to other systems (such as Note Highway)

To read in a ruleset, we first needed to figure out how the ruleset was structured. Since the prototype's ruleset was fairly easy to create and modify, it was used as an inspiration. The ruleset built in Tracery was also in JSON format. A similar JSON format was used for the C++ version. Unreal's C++ API had a JSON parsing library, which helped parse the ruleset data. Following parsing, for each rule found in the ruleset, it was mapped to custom defined data structures. One of the structures represented the non-terminating symbol (@) and held information about its sub-rules (also referred to as tokens in the code) and probability for being generated. The other represented the terminating symbol (*) and held information about the note type it was supposed to represent.

```
    "@origin": [
        {
            "tokens": "@measureOne,@measureOne,@measureOne,@measureTwo",
            "prob": 1
        }
    ],
    "@measureOne": [
        {
            "tokens": "@four",
            "prob": 0.75
        },
        {
            "tokens": "*spreadFour",
            "prob": 0.50
        }
    ],
    "@measureTwo": [
        {
            "tokens": "@three,@action",
            "prob": 0.75
        },
        {
            "tokens": "*spreadThree,@action",
            "prob": 0.50
        }
    ],
```

Figure 55: JSON ruleset for Doldrum's note generation

Once the rules were mapped, the next step was to build the algorithm that generates the Note sets. The high-level pseudo-code for the algorithm is the following:

```
Define Result as empty array of notes
Recurse into start rule
      For each token in rule’s token list
            If token is non-terminating (@):
                  Select a subrule based on additional parameters
                  Recurse into subrule and define result as sublist
                  Append sublist to current list
            If token is terminating (*):
                  Initialize note based on sub-rule type
                  Add note to the current list
```

```
Return Result
```

Code Listing 2: Procedural Note Generation Algorithm Pseudocode

In selecting a subrule, a helper function is called to run a heuristic function based on: desired Note density (game difficulty), and probability of rule getting selected. This function is run on all possible rules that derive from the current evaluated rule. The rule with the highest heuristic is then selected. In many cases when testing the system, there were rules that had the same heuristic value. The solution was to then randomly select from this list, so that there was still some randomness involved in the algorithm. The end product of the algorithm will produce a list of Note objects that are then pushed into a queue for use by other systems like the Note Highway.

6.5.3 Scripted Generation

While making the entire gameplay procedurally generated is an option, there are moments where we want the melodies generated to follow the music or be scripted in a specific fashion for the players to get used to gameplay. Using the same `NoteGrammar` system, it loads in a JSON file with the sequence of Notes, parses the sequence, and loads the sequence into memory. When the game needs to use the scripted sequence, the system clears out of the queue of existing notes and pushes the scripted note sequence in. From the Note Highway's perspective, there is no difference in which kinds of Notes are being generated, and therefore can use the same system to spawn them.

6.5.3.1 Note Editor

To write scripted sequences, a possible option would be to write them out note by note in a text editor, but this option is very tedious and error prone. To simplify this process, a web-based note editor was built. Kelly had previous knowledge of building web-based applications, so it took less time to build it than it would have taken if it were made as an Unreal plugin. In the Note Editor, a team member could select which notes they want to play and when. A graph on the left side will display the sequence. The user can then press *Save JSON*, to save the scripted sequence in JSON format. They can then drag the file into the Unreal project's *Contents/Data* folder, where the note generation system looks for scripted sequence files.

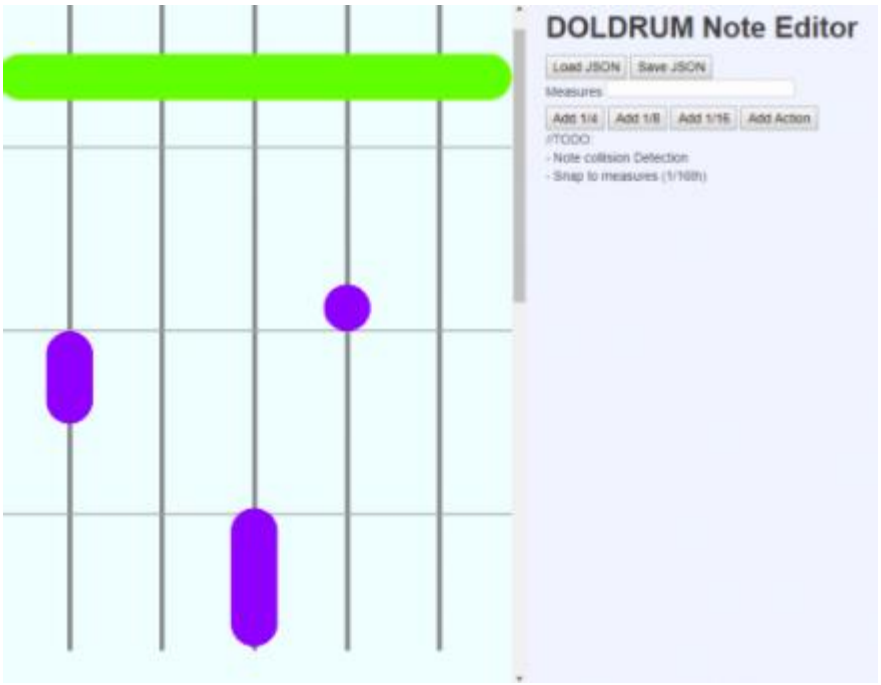


Figure 56: Early prototype of the Note Editor

6.5.3.2 JSON Generation

The purpose of the editor was for the team to be able to construct a visual sequence of Notes into a structure that the Note Generator is able to understand. To do that, JSON conversion functions were implemented in addition to the visual editing. While it was desirable for everything to be done client-side, modern browsers only allow for the saving of files to be done via the server. Due to this constraint, the conversion functions were moved onto a simple Node.js based web server.

When the client makes a download call, it sends the scripted sequence of notes. The server takes that data, converts it to the structure the Note Generation system uses, saves it as a JSON file, and pushes that file to the client for download. When making an upload call from the client, the opposite is done. The server takes in the JSON file, parses it, and converts it to the data structure the editor uses, and sends the parsed and converted data back to the client for it to display.

6.5.3.3 MIDI Parsing

MIDI parsing was added to the Note Editor upon the sound designer, David Allen's suggestion. This sped up the workflow of making pre-scripted files and allowed for flexibility on the audio implementer's end.

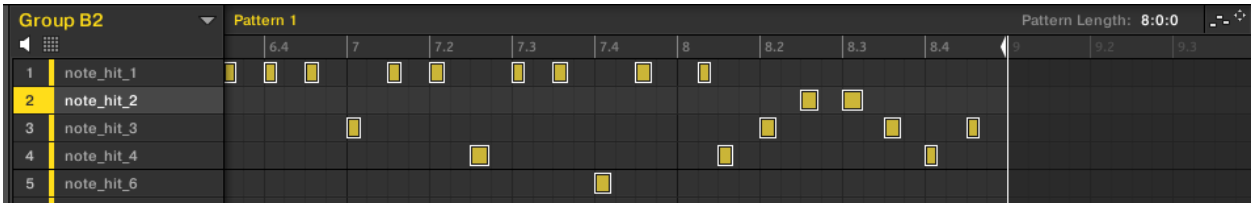


Figure 57: Screenshot of MIDI sequence in Maschine 2

To implement, *midi-parser-js* was used on the server side. [9] On upload, the server will receive the uploaded MIDI file as part of the client's request. The server then runs it through the parser and sends a JSON back to the client in the response call. The client then goes through the contents of the object, extracts the necessary information, and populates the editor for display. The user can decide to make additional changes to the sequence via the interface and save it as a JSON file.

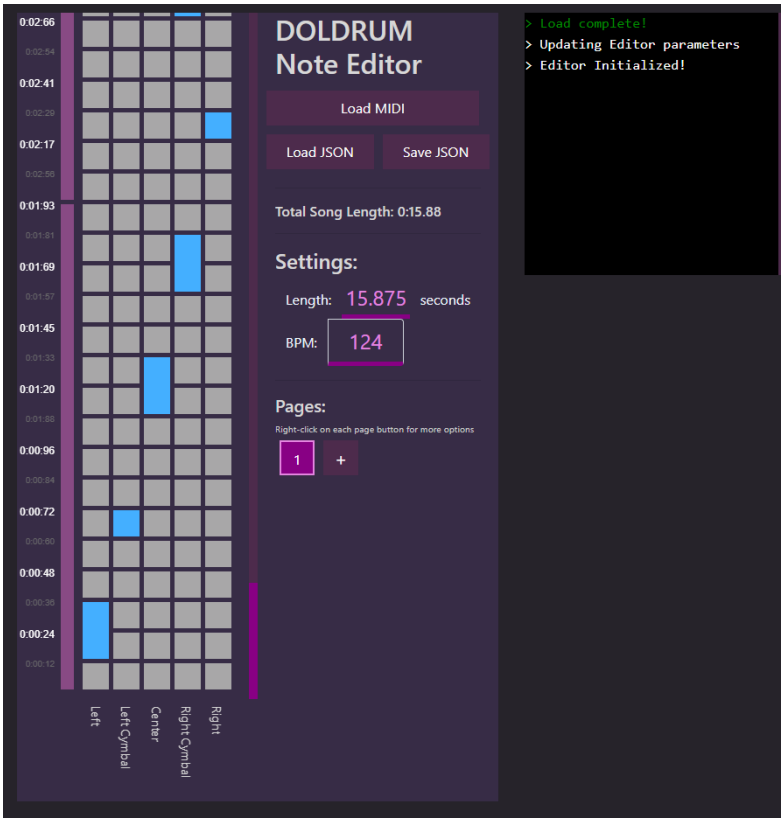


Figure 58: Screenshot of the Note Editor's final version

6.5.4 Original System

Originally, generation of music by players would be dependent on a MIDI file containing chord information at various timestamps. This MIDI file would correspond to the soundtrack playing in the game. To keep track of which possible notes could play at any given time, there would be a system that holds a map of the timestamps, with a list of valid notes to play. A `MidiManager` was then built to hold

this information. To save time on parsing the data, we used a C++ MIDI parsing library, *Midifile.* [8] This library parsed out the information in our file, and provided information on the duration of the track, when each note was played, and which notes were being played. The intention, was to use this library to hold a collection of possible notes to play at a given timestamp. Whenever the player would strike their instrument, it would play a random note within the collection, ensuring that it sounds correct with the backing soundtrack. However, due to scoping and having the main gameplay be less freeform and more guided, we moved towards a grammar-based system.

6.6 PHASES

Doldrum's Phase System is responsible for the game's progression. It acts similarly to the Boss State Machine system in that there are requirements defined by a designer on how to progress. Structurally, it is similar to the Metronome System as there is a manager object with listeners attached to it. The Phase System's ruleset is tied to a `BossCharacter`, so if the game was to progress to the next boss, a new set of phase rules specific to that boss would come into play.

Additional requirements for the Phase System were the following:

- Ability to delay changing of phases to allow for object cleanup and subsequent transition effects.
- Rules to allow for phase progression could differ depending on the phase

The Cuckoo listens to phase changes since there are specific phases that either trigger environmental changes (e.g. Chimes) or the Cuckoo's state behavior. The Note Highway listens to phase changes to know when to clear the Highway of Notes, as well as knowing what type of Notes to generate for the next phase.

To satisfy the requirements defined for the Phase System in an efficient manner, much of the previously implemented systems were referenced (Boss State Machine, Metronome). The difference with the Phase System is that its controller object is a singleton, since the game will only need one running at all times.

To implement the delaying of phases, a listener is attached to an actor. Every listener is called by the Phase System when it leaves a phase and when it enters a phase. When leaving the phase, the blueprint logic must explicitly tell the Phase System that it is ready to proceed to the next phase. The caveat to this approach, is that all objects using the `PhaseControllerListener` needed the programmer to implement the call that tells the system that it is ready to go to next phase. This meant that any object that did not implement the call would prevent the `PhaseSystem` from progressing.

6.7 PLAYER

The `PlayerBP` class handles all interfacing between the HTC Vive and the game world. Additionally, the platform is used both in player movement and to indicate to the game's play space to the player. All Unreal VR interfacing support is contained within the `PlayerBP` class.

6.7.1 VR Interfacing

The HTC Vive features three trackable components: The headset and the two motion controllers. The majority of basic motion tracking and other Vive-specific functionality was provided by the VR Template starter project. This project was officially introduced in version 4.13 of Unreal Engine [15]. The template also provided functions for both attaching hand actors to the controllers' locations and making the controllers vibrate.

6.7.2 Platform

Functionally, the platform is the actor that moves when the player makes a movement input command. Several actors, including the player and xylophone, have their locations parented to the platform. Thus, when the platform moves, they also move. Additionally, the platform also handles collision checking for whenever the player is standing outside of the playspace. The movement of the platform and what happens when the player walks outside of the playspace are handled by the xylophone blueprint and player blueprint, respectively.

6.7.3 Walking Out of Bounds

When the player steps outside of the playspace the game will automatically pause. This playspace is represented as an invisible collision box that contains the platform that the player stands on. If the player leaves that collision box then the game will automatically pause until they walk back into the box.

6.7.4 Mallets

The mallets are long and narrow which allows players to easily hit xylophone faces in a natural and realistic way. The mallets' size and length were modelled after real xylophone mallets. Only the spherical tip of the mallet can hit the xylophone faces.

In addition to providing the player with a way to hit the xylophone faces, the spherical tip will also glow with electrical sparks when the player is doing well. After hitting a certain number of consecutive Notes, the player will enter the Frenzy state. During this state, electrical particles will spawn from and follow the tip of the mallet.

Whenever the mallets collide with the xylophone faces the motion controllers will vibrate. As the xylophone only exists in the game world, there is no way to prevent players from moving their controllers

straight through the xylophone faces. Therefore, the vibration is the only way for players to discern when the mallets collide with the xylophone.

VR functionality such as allowing the player to rotate their head to look around was provided by the VR template and was not significantly modified. The hands, however, were significantly adjusted to act as mallets.

6.7.5 Height Calibration

Doldrum features an option to adjust the height of the xylophone based on the current height of the Vive headset. This allows for players of all heights to easily hit the xylophone with the mallets without having to overreach.

Height calibration is accomplished via updating the xylophone's current location to be a set distance on the z-axis—the vertical axis in Unreal Engine 4—from wherever the player's camera is located. This camera is bound to the current location of the Vive headset. Headset rotation and location aside from its position on the z-axis is ignored.

6.8 XYLOPHONE

The xylophone actor is where the majority of the player-specific logic and functionality is contained. This actor blueprint is titled `NewDrumBP` due to a significant overhaul to how the xylophone works that occurred early on in development. The reason for the name discrepancy is because the xylophone was a drum for a majority of the game's development. `NewDrumBP` contains logic and functionality for: receiving and responding to incoming Note notifications from the Highway, action inputting, handling of gameplay systems such as health and Frenzy, and handling collisions with the player's mallets. `NewDrumBP` is one of the largest blueprints in *Doldrum*, and it has undergone significant changes and modifications over the course of the game's development.

While the player class handles most VR interfacing, the `NewDrumBP` handles most game-specific logic. This includes health, Frenzy handling, and Note handling.

The player begins the game with a certain amount of health and certain events can reduce it. If the health ever reaches zero, the boss stops performing actions, the Highway stops sending Notes, and the player sees a "Game Over" screen. Health loss and gain both have their own respective functions within `NewDrumBP` that must be called within the actor in response to certain game events. These functions modify the actor's health variable while also calling other events that result from the loss or gain of health. For example, losing health will reset the player's Frenzy counter.

Frenzy is a system contained within the `NewDrumBP` actor. This system rewards the player for correctly striking Notes without missing. On every successful Note hit, the Frenzy counter is incremented. When that counter surpasses the defined `FrenzyMax` variable the player will enter the frenzied state. During this state, player attacks gain additional visual effects and deal increased damage. If the player ever takes damage from any source, their Frenzy is reset back to zero and they lose all benefits from high frenzy.

Functions that add or subtract health or frenzy also update the player's HUD (heads-up display). Additionally, the function that reduces player health also causes the Note Highway to briefly pulse red.

6.8.1 Note Hitting

The `NewDrumBP` contains several functions that pertain to incoming Notes. The xylophone contains a Boolean variable for each of its five faces that refers to whether that specific xylophone face is expecting an input. The act of setting a xylophone to expect an input is referred to as "priming". When a Note arrives at the xylophone, that `primed` Boolean is set to true and the player is expected to hit that face within a set amount of time. When a Note arrives at the xylophone, the following sequence of events occur:

1. The *`PrimeDrum`* function is called. This function takes the target xylophone face as an input and sets the respective `primed` Boolean to true.
2. A timer for that respective xylophone face is started. After a set duration, the timer is completed and another function is called
    a. If the player hits the xylophone face before the timer concludes, the timer is cancelled and the respective Note is popped. This is considered a successful hit.
    b. If the timer concludes, the player is considered to have missed the Note and the player is dealt a small amount of damage.
3. After either outcome, the `primed` Boolean for that xylophone face is set to false until another Note arrives at that face.

The `primed` Boolean for a respective xylophone face is used for when the player's mallets collide with a given xylophone face. If that xylophone face is `primed` then the respective timer is stopped. The player is determined to have successfully hit that Note. Player hits to a given xylophone face that are performed while that xylophone face is `primed` will also increment the Frenzy counter.

6.8.1.1 Action Input Checking

The player's sole way of interacting with the Boss is via the action inputs that can be performed whenever an Input Bar reaches the player. These action inputs require that the player hit a single xylophone face with both mallets. Depending on the face hit, one of three actions can occur:

1. Two hits on the center face will result in an attack. A lightning orb will travel towards the boss and deal damage to it
2. Two hits on the leftmost xylophone face will cause the player to move left. This can be used to dodge attacks or reposition their attacking point
3. Two hits on the rightmost xylophone face will cause the player to move right.

Action Input checking is done in several steps. Firstly, hitting any xylophone face after an Input Bar has reached the player will create a *`DrumInputStruct`*, a struct that contains information pertaining to a possible action.

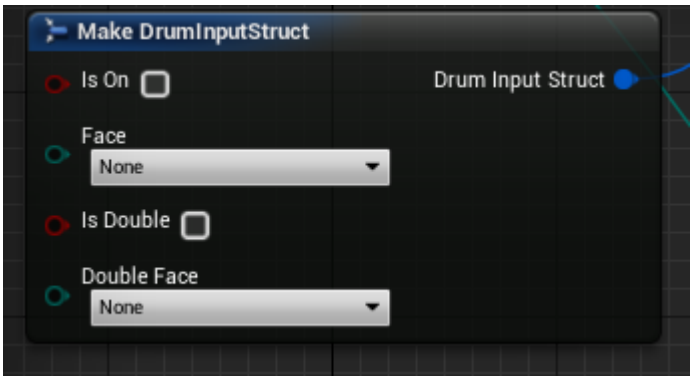


Figure 59: DrumInputStruct Blueprint node with all values set to the default

A *`DrumInputStruct`* represents a single beat's worth of information. The *`DrumInputStruct`* contains a Boolean for whether-or-not the beat features a hit xylophone, an enumeration for which face was hit, whether-or-not the beat featured another simultaneous hit, and an enumeration for which xylophone face was simultaneously hit.

After the *`DrumInputStruct`* is constructed from the xylophone faces that the player hits, it is compared to the entries in an array of other *`DrumInputStruct`*s. This array contains a separate struct for each of the three possible inputs. Each struct within this array is compared to the newly created *`DrumInputStruct`*. If it matches any entry in that array, the respective action is performed and no further entries in the array are compared.

Comparing one *`DrumInputStruct`* to another is facilitated via a helper function titled *`CompareDrumInputStruct`*. This function is used several times throughout the Action Input system.

Specifically, comparisons occur when the struct is first made and when it is being compared to the array of action inputs. The *DrumInputStruct* is immediately created as soon as the player makes any hit while an input is expected. Thus, due to players always hitting one xylophone face slightly before another even on simultaneous hits, the hit that occurs after the initial will instead result in a check to see if a *DrumInputStruct* already exists. If that struct exists, NewDrumBP will instead only populate the variables of the struct that pertain to simultaneous hits. Likewise, if the struct's variables are all equal to the default, it can be assumed that this is the first hit of the input.

6.8.2 Actions

After the *DrumInputStruct* is successfully compared to one of the three given actions, the respective action is performed. This action occurs immediately after the player's input.

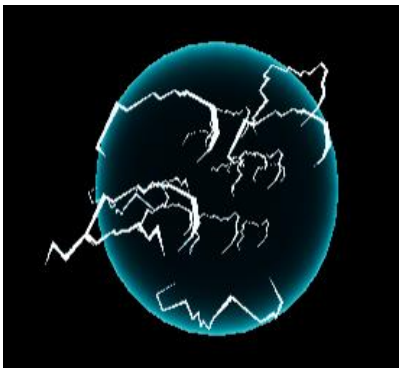


Figure 60: The player's lightning attack orb

The *Attack* action fires a lightning orb that travels directly to the Boss. When it collides with the Boss, it takes damage and its health depletes. If the player is Frenzied, this *Attack* will deal increased damage and have the added visual effects of some arcing lightning beams.

The *Move* actions will relocate the Platform that the player stands on in whichever direction was inputted. The player can move either left or right. Movement will smoothly relocate the player to one of five *MoveLocation* actors that are statically placed in the game world.

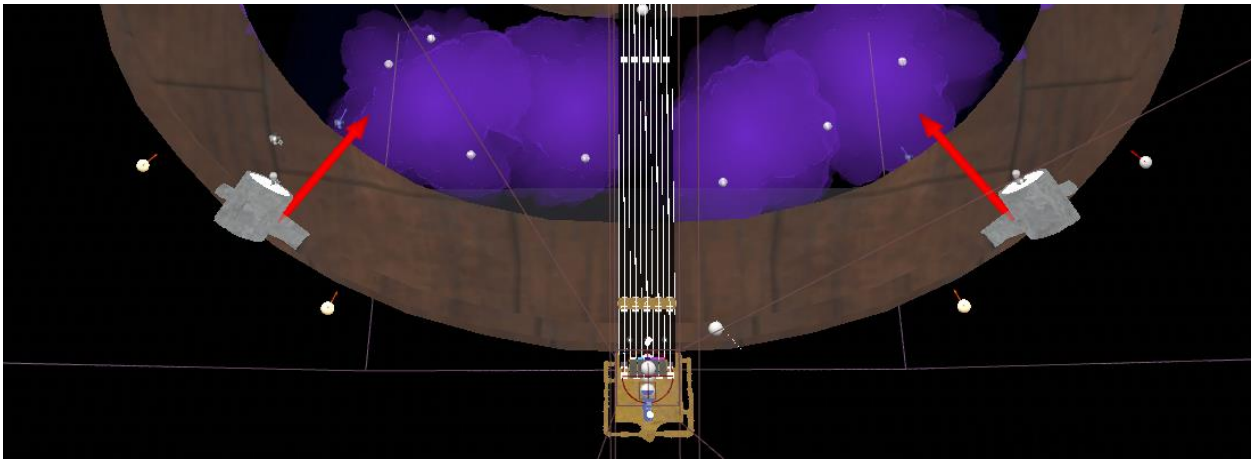


Figure 61: The five move locations as seen from above.

`MoveLocation` actors are invisible during gameplay but are represented in the Unreal Engine 4 editor as small white spheres with red arrows. The chosen `MoveLocation` actor is dependent on which actor the Platform is currently located on. For instance, in the above image, the Platform is currently at the centermost `MoveLocation` actor. If they were to move right, the Platform would travel to the closest actor towards the right.

6.9 USER INTERFACE

Figure 51 outlines the flow of the game via the UI interface. The UI is mostly linear with a few options that allow the player to navigate to other menus.

Doldrum uses Unreal's Motion Graphics (UMG) UI to display UI elements to the player. Three UI widget Blueprints are hooked up to the xylophone's `NewDrumBP`. One widget, the HUD, is always active in world space. The other two are only active whenever `NewDrumBP` is called to display the menu UI. When the menu UI is active, all drum hit events that occur will redirect to the menu widget Blueprints, triggering menu-specific logic.

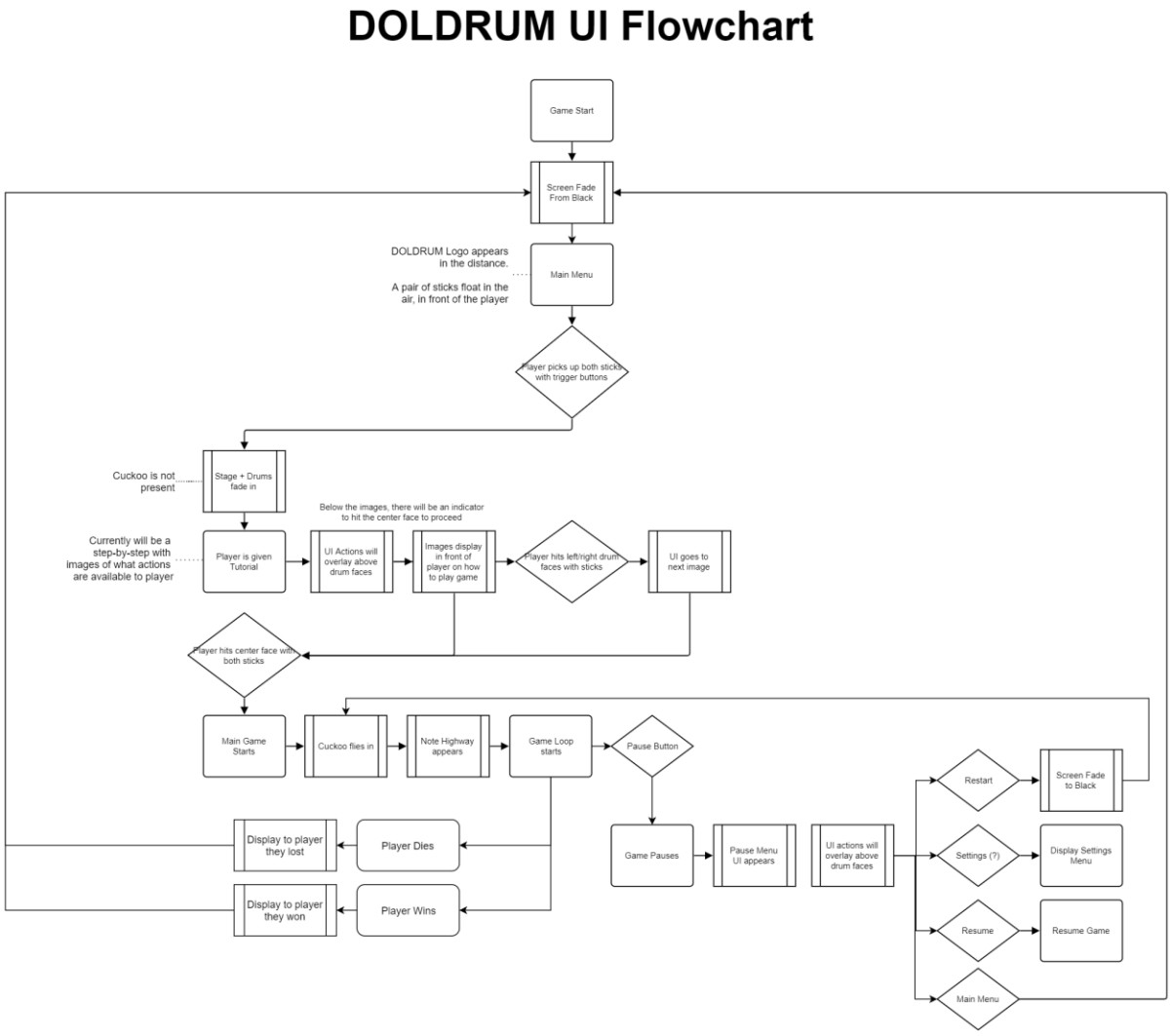


Figure 62: The UI Flowchart

7 SOUND

The sound of the game was wholly original. Sound effects can be broken down into a few categories. First, there are the player-centric sounds. This includes the xylophone, the Notes, the damage sounds, and the movement sounds. Another category would be the sounds related to the Cuckoo Boss which includes a death sound clip which is synced to an animation. Finally, we have the non-diegetic sounds such as UI.

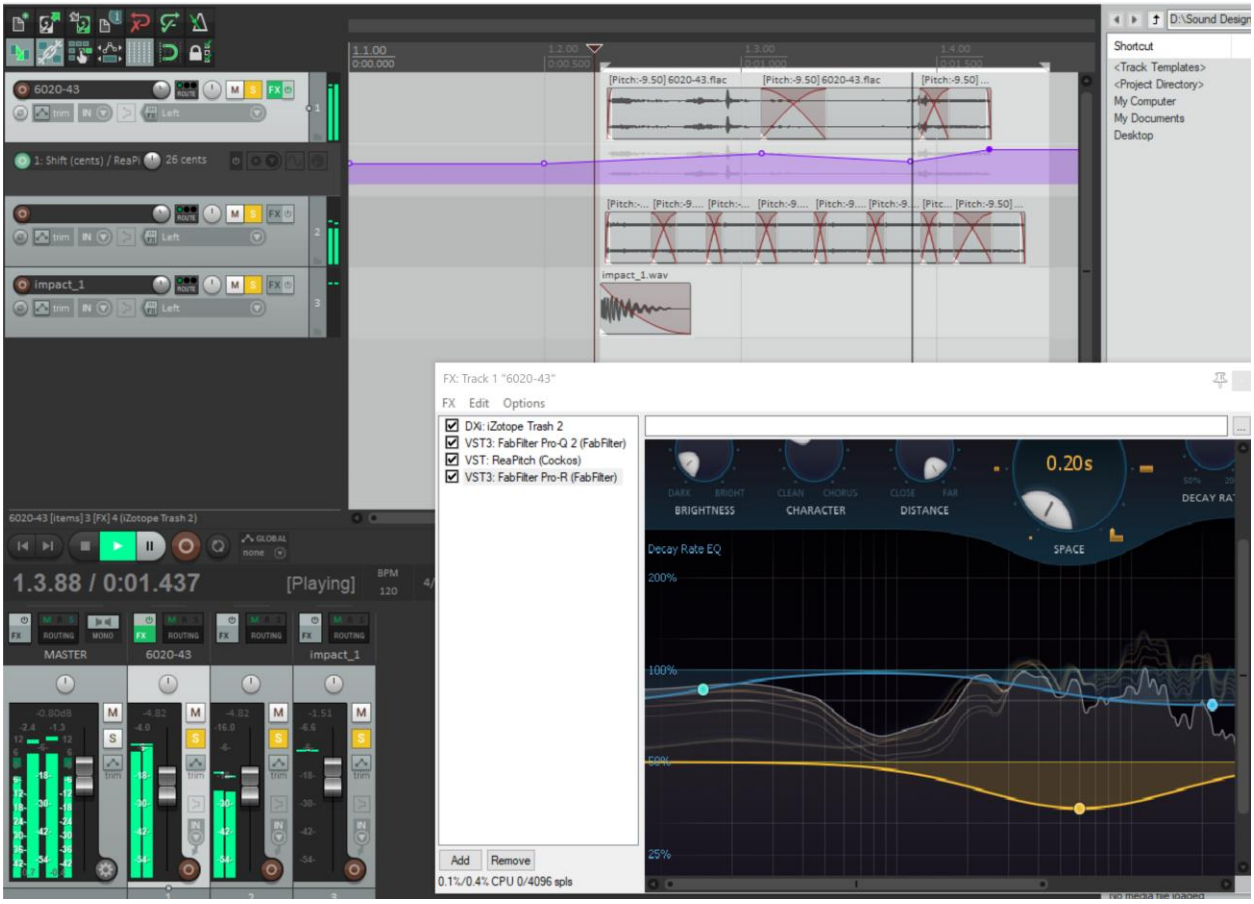


Figure 63: "Cuckoo Hurt" sound effect, created in Reaper

The process of creating sound effects was entirely based on editing, layering, and modulating existing sample or synthesizers. None of the sound effects were recorded through field recording or Foley. The typical process for creating sound effects began with browsing a sample library for a base sound. Examples of base sounds include impacts or a texture. The base sound would serve as a foundation. Then, additional sound effects would be added on top of the foundational sound effect, depending on what was needed. For example, a stronger impact, a longer tail, or textures such as

brightness or warmth would be needed. From there, effects plugins would alter some or all of the sounds. Software used included *Reaper* as the digital audio workstation, and various plugins from the following developers: *Soundtoys*, *Fabfilter*, *Izotope*, *Cockos*, and *Native Instruments*. Typically, equalization and compression would be applied to the tracks followed by a "creative" effect such as distortion or pitch shifting. Through the creative effects, the sounds would unify into a distinct sound effect.

After creation of the sounds, they were implemented into Unreal. The sound designer, David Allen, imported the sounds into the engine and created a "Sound Cue" for each sound. Sound Cues are a type of asset in Unreal. They function similarly to small Blueprints but are innately cast as sound assets. Within them, a sound designer can set up modulations for sound effects such as randomizing pitch or volume. More advanced actions include the concatenation of consecutive sounds or the random selection of a single sound from a pool. Using Cues, the sound designer configured exactly how sounds should behave without touching any of the blueprints. Once the cues were created, he worked with one of the programmers to find where to trigger the cues inside of the blueprints.

During play, the player plays five different notes, based on the pentatonic scale. The pentatonic scale consists of the 1st, 3rd, 4th, 5th, and 7th tones of a scale. These notes will typically not conflict with the game's backing track and will sound correct in any sequence. We originally had the five tones of the pentatonic scale corresponding to each of the five xylophone keys. However, we found that the 7th tone would create unresolvable tension as the player could never hit the octave (8th tone). Because of this, we removed the 7th tone and replaced it with the octave, which removed those issues.

Rather than use real xylophone samples for the instrument, we used synthesized sounds mixed with drum samples. The reasoning for this was to unify the sound of the instrument with the instrumentation of the backing track. As the backing track is nearly entirely comprised of synthesizers, a real xylophone would sound out of place. Rather, a short synthesizer was designed in Native Instruments' *Massive* and paired with the attack of a kick drum, making the sound appear both percussive and tonal. For testing, the sound designer would improvise to the backing track using the designed sounds to ensure they fit.

8 MUSIC

The game features an original backing track to accompany the player's procedurally generated melodies. The music was composed with a few constraints in mind: a strong emphasis on beats, simple rhythms, and unobtrusive melodies. In order to help the players maintain their rhythm, the whole beats were emphasized in the backing track. The song begins with a strong pulse on the whole beats from a kick

drum. This plays for two measures with a simple bass track accompanying it before the player has to play any notes. This gives the player enough to time to develop a firm grasp on the beat, allowing them to stay on-beat for the rest of their playtime.

The music features four distinct sections that loop until another section begins. The four two-minute long sections feature many of the same instruments and melodic sequences. However, the intensity of the arrangement increases with subsequent sections. Each section is used for two consecutive phases of the boss fight for a total of eight phases. As mentioned, the phases always transition upon the Cuckoo's health hitting a certain threshold and the Cuckoo's health will only change upon taking damage from an Attack input. As the Attack input will always come at the end of a musical phrase, the phase will always transition at the end of a phrase, creating a seamless transition between musical sections.

The music was written to be fun and upbeat while maintaining a serious and focused tone. In order to achieve this, inspiration was drawn from jazz music and features a typical jazz drum kit. As the game's only track and the foundation for the player's performance, the music had to be accessible and digestible. It features a catchy and repetitive bass line which is iterated on across the different sections. The lead synthesizers were meant to be more rhythmic than melodic. While there is a clear tone to all of them, they feature a significant punchiness, adding to the rhythmic pulse of the other tracks. Additionally, there are sampled mechanical toys and pot lids which create a playful and hypnotic tone. This track plays quick, staccato rhythms and add to the playful goal of the track. Later sections of the music feature a fast hi-hat which imitates a bird quickly chirping through heavy modulation. This sound is not particularly melodic but was another way to impart a pulse and the beat to the player.

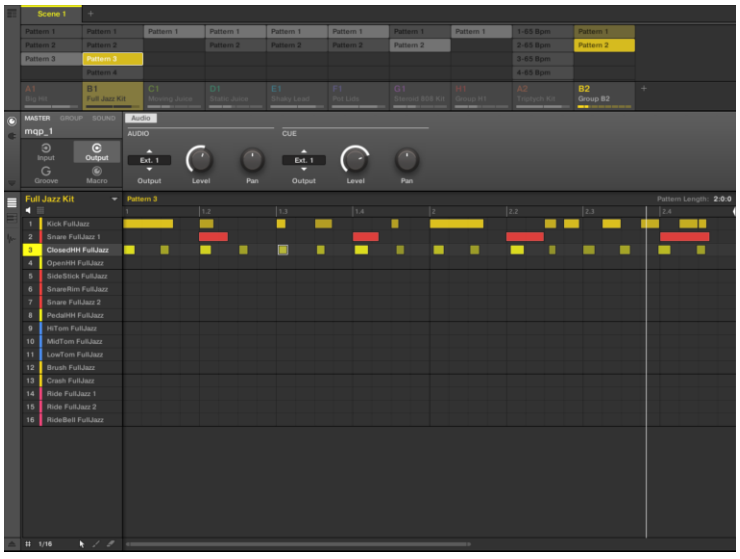


Figure 64: Maschine 2 Digital Audio Workstation

All music was composed in the *Maschine 2* DAW (digital audio workstation) with dedicated *Maschine mk3* hardware and later arranged in Bitwig Studio 2. All of the music was performed live into the DAW using the hardware. Sections of the song would loop and each individual instrument would be sequenced on top of each other. The continuous play of the music and the organic means of creating the layers heightened the improvised feeling of the music. [2]

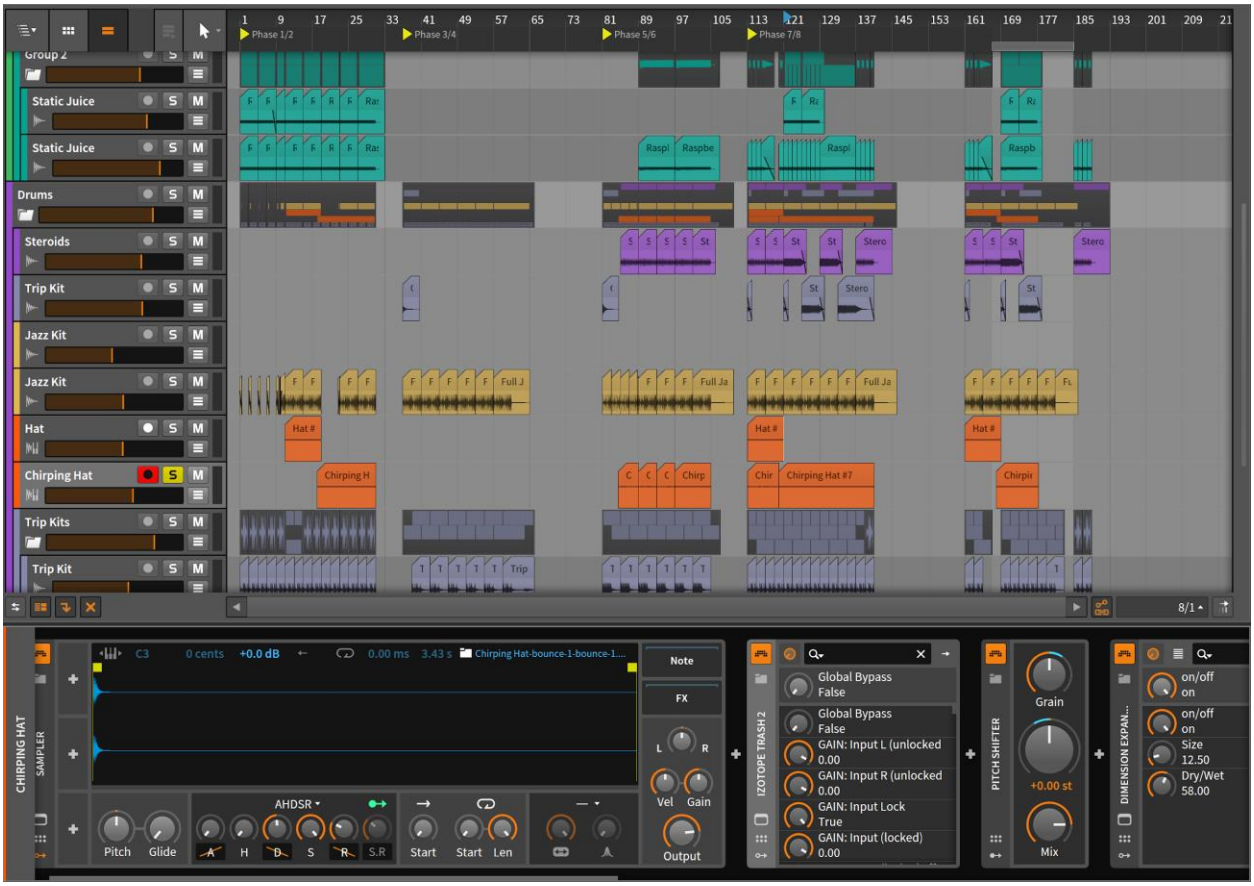


Figure 65: Music arrangement in Bitwig Studio 2

9 PROJECT MANAGEMENT

We used a variety of project management software to facilitate productivity and clarity for the team.

9.1 TRELLO

We chose to use *Trello* as our task list and issue tracking software. It was selected due to ease of use. Additionally, several members of the team already had experience with *Trello* and it was picked up quickly by those who did not. While we had decided to use the software at the beginning of the project, we did not strictly use it until approximately halfway through the project. Once this transition happened, we noticed a significant increase in productivity and a decrease in confusion about objectives.

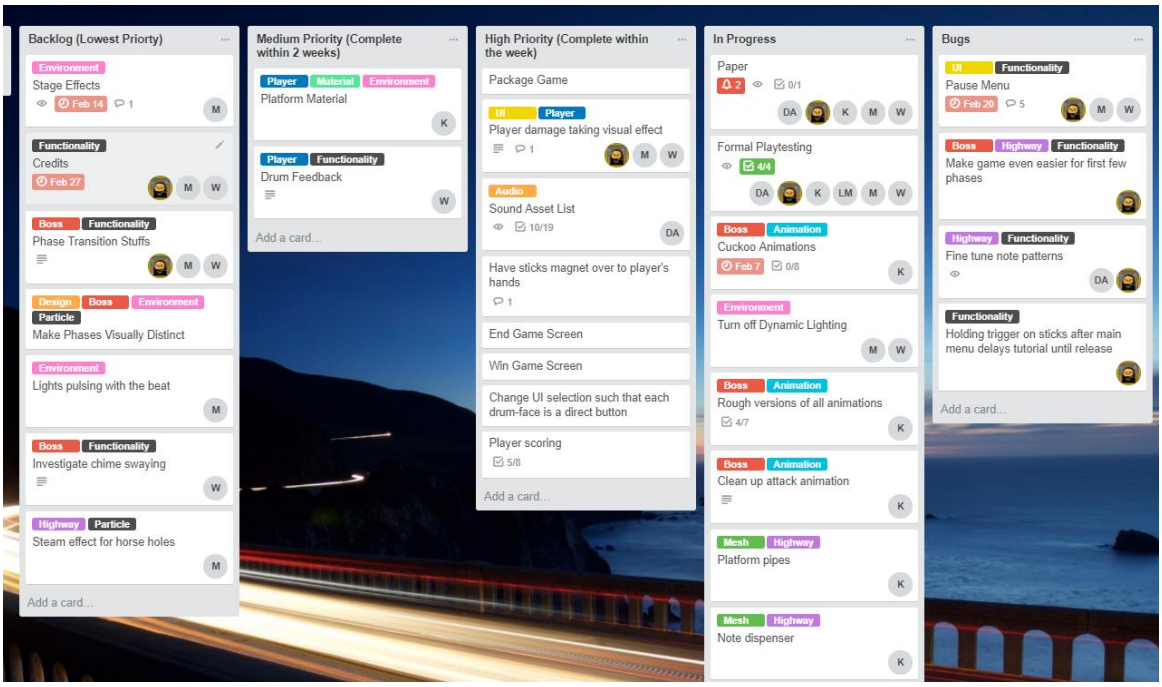


Figure 66: A screenshot of the Doldrum Trello board

Trello is a web-based organization application which imitates the act of moving sticky notes on a whiteboard. Tasks are represented by “cards” which are organized into “lists” which are named columns. A given card can have a variety of attributes including comments, organizational tags, checklists, and an assigned team member. We chose to organize our cards primarily by priority levels but also had lists for “bugs”, “needs implementation”, and “awaiting review”. Cards 'can be freely moved between columns, allowing us to easily move items up and down in priority. Cards were further organized by tags. The tags would both list what component the card was addressing (e.g. “Cuckoo Boss”, “Note Highway”) and

what type of work had to be done (e.g. “Animation”, “Functionality”). With this organization system, it was easy to understand what a card was describing when looking at the title and the tags.

9.2 PERFORCE

Perforce was our chosen source control platform. A private Perforce server was maintained by a team member. Most of the project team had no experience with Perforce. Despite this, Perforce was easy to learn and performed incredibly. The decision to use Perforce over *Git* or *SVN* was due to its robust integration with Unreal Engine 4. There are a few distinct advantages to using Perforce over other source control options with Unreal. One significant advantage is the ability to *diff* between Blueprints when comparing versions. This functionality is built into Unreal and is not present with other source control options due to their inability to handle large binary files (i.e. Blueprints).

The Perforce integration in Unreal is intuitive. Whenever someone edits a file, it is “checked-out” by that member. The checked-out file will be read-only for all other team members and have a small lock icon on the file. This prevents file conflicts and the need to merge changes. Other features include the ability to sync individual files from within the Unreal editor. The use of Perforce made development smooth and we did not encounter any issues with Perforce after initial setup.

9.3 GOOGLE DRIVE

Documents and other non-project files were hosted on *Google Drive*. Google Drive was chosen over similar platforms such as *Dropbox* due to the collaboration features. It was easy to collaboratively write design documents, meeting agendas, and other documents together rather than one person editing a file. This made document creation easy and fast. In addition to documents, many art assets were uploaded to the Drive as a means for file sharing. Art assets were often shared this way instead of through Perforce as it was typically the programmer’s responsibility to implement the assets. Because of this, it was simpler for the programmers to decide where the assets should be located in the file structure of the Unreal project. The other reason this happened was because the artists would often use a variety of campus computers which did not have Perforce installed on them. As Google Drive is web-based, they could easily upload their work from any computer.

9.4 SLACK

Most online communication was done through *Slack*. The ability to create channels allows conversation to be organized and to make context-switching effortless. Most of our channels were categorized by types of work, e.g. “programming”, “visuals”, “audio”, or “paper”. However, we also had channels such as “meetings”, “resources”, and “brainstorming” for other specific needs. Slack was a valuable tool which facilitated fast, organized, and rapid communication.

9.5 DISCORD

Discord was occasionally used for online communication. Discord is set up similarly to Slack but also features voice channels. When working from home, members of the team would occasionally work while in a Discord voice channel, so we could quickly communicate with each other. It also boosted productivity and comradery as it would give the feeling of working beside each other in an office setting. The other use for Discord was to have some impromptu meetings online. If it we wanted to meet and the meeting was either planned last minute or would simply be very brief, we would talk in a voice channel on Discord.

10 TESTING

10.1 METHODOLOGY

10.1.1 Overview of the Experiment/Design

The goal of playtesting was to observe and analyze user behaviors when engaging with *Doldrum*. Users were fitted with an HTC Vive virtual reality headset and a pair of HTC Vive wireless controllers. When the Investigator began the game, they observed the subject from the start of the game until the very end, or whenever the subject decided to opt out. When the playtest was complete, subjects were asked to complete a post-test survey on their thoughts and observations.

Observed behaviors and results from the post-test survey were used to improve the design of our game.

10.1.2 Population

The population consisted of WPI students. The team had considered populations from demonstration events and high-traffic events such as Worcester Game Pile and Penny Arcade Expo East. Neither were chosen as explained in the *Location* section.

10.1.3 Location

Playtesting was conducted on the WPI campus in the Digital Art Studio in Fuller Laboratories. The location was already fitted with the necessary virtual reality equipment needed to conduct this study. We had also considered conducting playtesting at events mentioned in the *Population* section but decided that managing the logistics of formal playtesting for these events would be difficult. These logistical difficulties may have resulted in the data being less detailed compared to playtesting in a private and controlled environment like the Digital Art Studio.

10.1.4 Restrictions

The time limit for a single playtest session was 30 minutes long.

10.1.5 Technique

During the playtest portion of the study, the data collected by the investigators were observable actions, reactions, questions, and comments subjects had while playing the game. Data points that the team considered to be important to how well the game played were the following:

1. Time user takes to complete the game
2. Subject sentiments
3. Number of times subjects played the game

The post-test survey consisted of a series of questions that assessed the subject's physical and emotional attitude during the playtest. The post-survey allowed subjects to provide and anonymize any remaining information they had for the investigator(s) that they did not mention during the playtest. Attitudes were measured using Likert scales so investigators could assess the direction and intensity of a subject's opinions towards a particular event in the game the game overall. When asking for subject moods, a selection of moods from a mood chart were listed as checkboxes for the user to check if they experienced a listed mood. This helped investigators gauge how the game affected players.

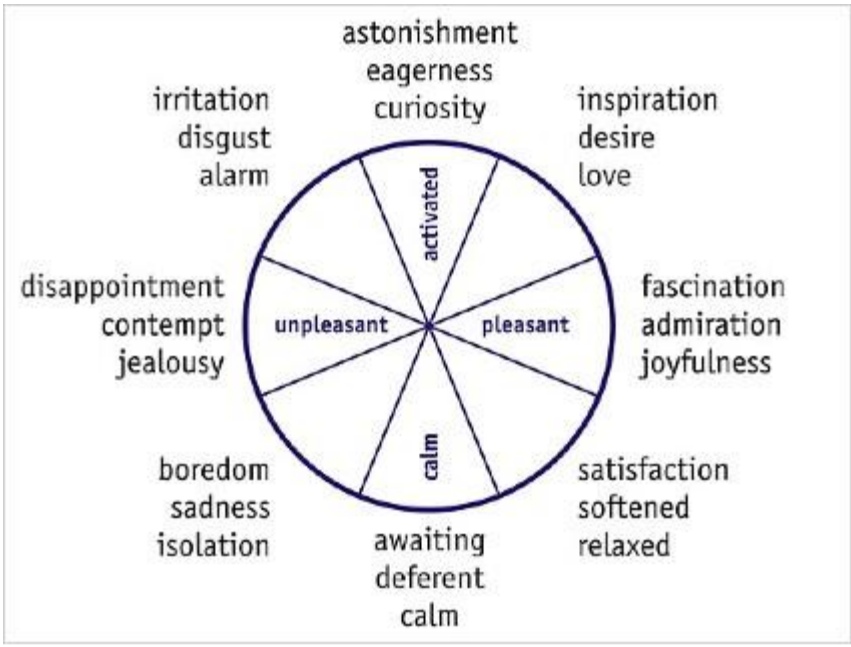


Figure 67: Circle of Affect used to categorize and measure positive and negative words and moods of user [13]

General demographic information such as age was asked in order to provide additional context to the collected data. The survey included questions asking of previous exposure. For example, it was asked if the subject had used virtual reality equipment as well as if they have played rhythm games before may also affect how they behave towards the equipment and the game. The survey provided the subject several text input sections to voice specific thoughts, observations, and recommendations.

10.1.6 Materials

Playtest subjects were given an informed consent form which detailed the process and procedure of the study, as well as potential risks. Playtest subjects were fitted with a cleaned HTC Vive virtual reality headset and a pair of motion controllers. Investigators had note sheets for each individual subject to record observations. Investigators also had a digital post-study survey on a laptop that subjects had the option to take after playtesting.

10.1.7 Entrance and Setup Procedure

When a playtest subject entered the testing premises, they were given an informed consent form to read. If the playtest subject consented and signed, they were directed to the Play Area where the playtest was conducted. The Investigator gave a brief overview of what to expect. Subjects were encouraged to speak out their thoughts to assist investigators with observational data collection. To assist with observational data collection, a portable audio recorder recorded the entire playtest. Subjects were informed of when the recording began and ended. With the assistance of an Investigator, the subject was

fitted with a headset, a pair of controllers, and a pair of headphones. Once the subject was comfortable with the setup, an investigator began the playtest by opening the game for the subject.

10.1.8 Study Procedure

From here, the subject acted and reacted to our game anyway they chose. The investigator noted down these actions and reactions in conjunction to what the subject saw in the headset. The investigator also noted any questions and comments the subject had during playtesting. If a subject directed a question to the investigator regarding the study and/or the setup (i.e. unable to see anything in the headset), the investigator was allowed to intervene.

10.1.9 Ending Procedure

Once the subject finished the game, either through early withdrawal, winning, or defeat, the investigator stopped the game and assisted with the removal of the equipment from the subject. The subject was then asked to complete an anonymous post-test survey. The survey was to collect data on subject attitudes and observations during playtesting. Once the subject completed the survey, they were free to leave.

10.1.10 Variables

The outcomes that we considered important were the following:

From audio data:

1. Subject sentiments

From observational data:

1. Time user takes to complete the game
2. Number of tries user takes to play game before quitting

From data gathered from post-test survey responses:

1. Physical comfort level
2. Usability/Interface difficulty
3. Gameplay difficulty
4. Subject mood

These outcomes gave the team a general understanding of how a player felt towards the game's aesthetics and gameplay on a physical and emotional level. This also allowed us to pinpoint areas that caused subjects to engage and/or disengage in our game in a way that does not align with our goal. We were then able to iterate upon those areas as needed.

10.1.11 Statistical Treatment

Observations seen in more than 50% of subjects was considered as a general finding in our report. Observations seen between 10% and 50% was seen as a unique finding, and any observations seen below 10% was considered as an outlier but was still noted in findings. These percentages were obtained from our desired sample size of 20 subjects. With 10%, we were considering a subject count of less than or equal to two as outliers. With 50% of our sample size, there would be at least 10 subjects.

For post-test survey results, written responses were treated similarly to observation data. Data from Likert scale responses were compiled into charts. Any correlation found between certain attitudes and observed subjects' behavior were noted in the report. Any correlation found between certain attitudes and general demographics were noted in the report. Any correlation found between observed subjects' behavior and general demographics were also noted in the report.

10.2 RESULTS

10.2.1 Playtesting Demographics

Though we had originally intended for 20 subjects to playtest our game, only 11 subjects playtested due to poor planning. On analyzing the demographics, our subject population was not diverse. All of the subjects were in the 18-24 year-old range and a majority were white and male. Our subject population was also familiar with rhythm games before playtesting-- having played either *Guitar Hero* or *Rock Band*, which made the game easier to understand. However, the data is not representative of audiences who are not yet exposed to game mechanics found in *Guitar Hero*.

During playtesting, our subjects encountered a few game-breaking bugs that we fixed between testing sessions. This may have affected the data in terms of assessing usability and subject emotion. Aside from this, overall gameplay and design stayed the same over the course of the playtests. We organized the evaluated data into the following categories: Physical, Usability, Gameplay, and Emotional. A majority of the data evaluated was from the Post-Playtest Survey.

10.2.2 Physical Assessment

On a physical level, a majority of our subjects were comfortable with wearing the HTC Vive headset and holding the controllers. The majority also did not experience simulation sickness after playing. Since 63.6% of our subjects had prior experience with VR before playing our game, and 85.7% of that group had exposure to an HTC Vive before, it is likely they have been well-adjusted to the headset already.

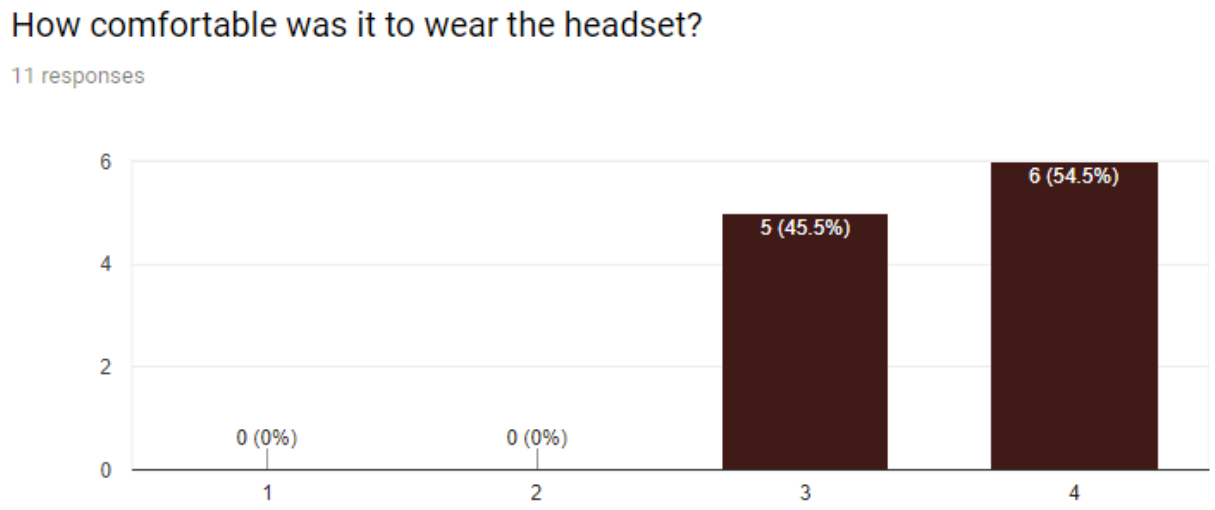


Figure 68: Post-Playtest Survey, Physical Assessment Results: Headset comfort on a scale of 1-4 with 1 as Very Uncomfortable and 4 as Very Comfortable

10.2.3 Usability Assessment

On a usability level, our menus were easy to navigate. A majority of subjects found it neither very easy nor very hard to hit the notes, though some vocalized during gameplay that they were very bad at hitting them. There were mixed responses when it came to difficulty performing actions. For performing attacks, some subjects found it hard, while some found it very easy. On the other hand, more subjects found performing movement actions to be very hard than very easy. In audio recordings, subjects expressed confusion on how they were supposed to move.

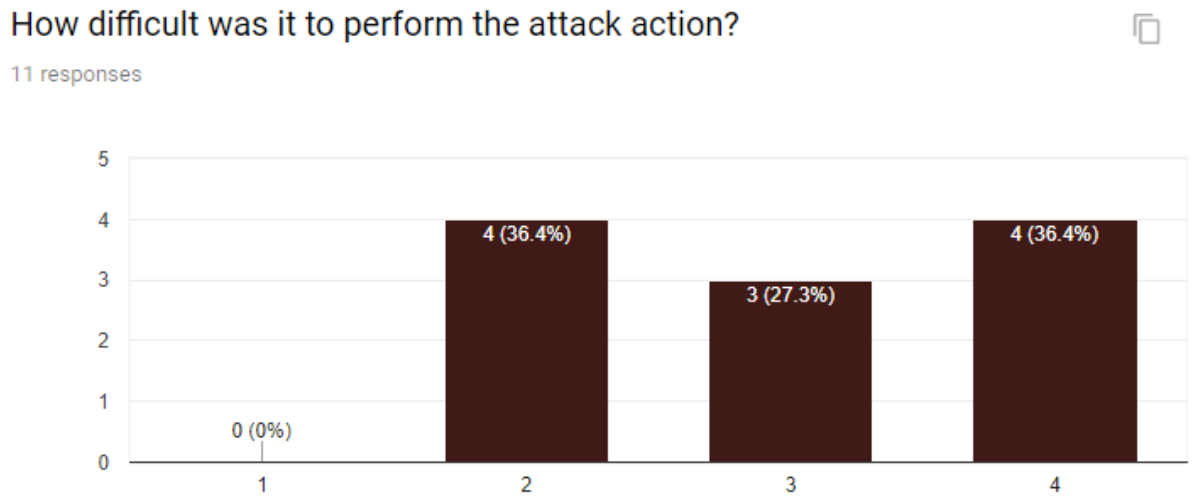


Figure 69: Post-Playtest Survey, Usability Results: Performing attack action on a scale of 1-4 with 1 as Very Hard and 4 as Very Easy

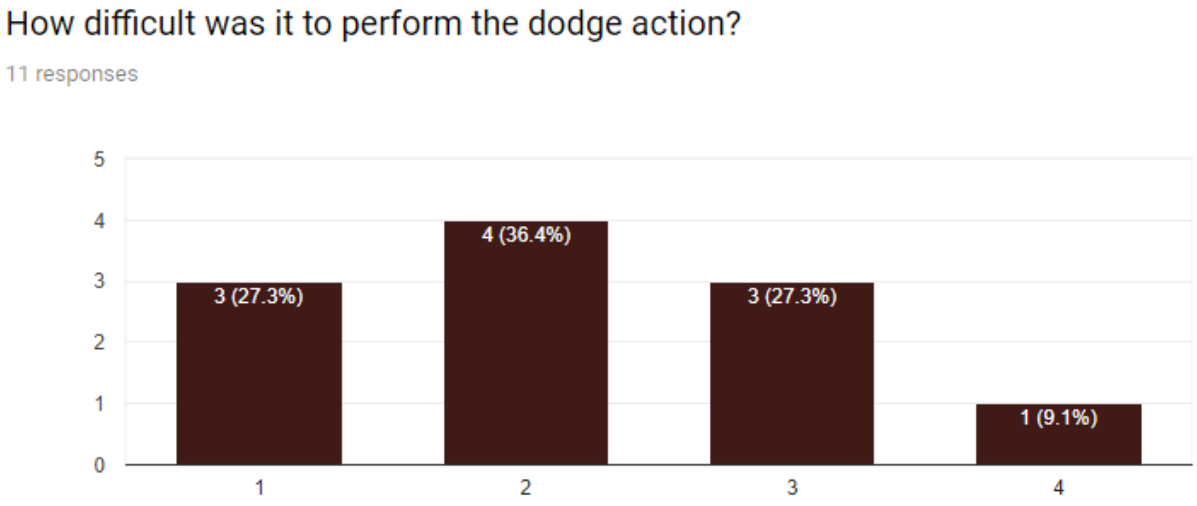


Figure 70: Post-Playtest Survey, Usability Results: Performing dodge action on a scale of 1-4 with 1 as Very Hard and 4 as Very Easy

10.2.4 Gameplay Assessment

On a gameplay level, subjects had mixed reactions toward various gameplay elements. A majority found the xylophone intuitive to use. They also found the boss's actions to be neither hard nor easy to interpret. Subjects felt similarly about interpreting notes coming down the Note Highway. In

recordings, a few subjects vocalized that they were unable to focus on both the notes coming down and the boss. As such, they did not realize that the boss was firing attacks at them. As for the pacing of the game, most subjects found it to be fast. Similar to how subjects vocalized how difficult it was to hit notes, the same subjects mentioned how there were too many notes coming down at the very start of the game. In the iteration that the subjects were playing on, our team did not yet remove a difficult test sequence that played at the very beginning of the game.

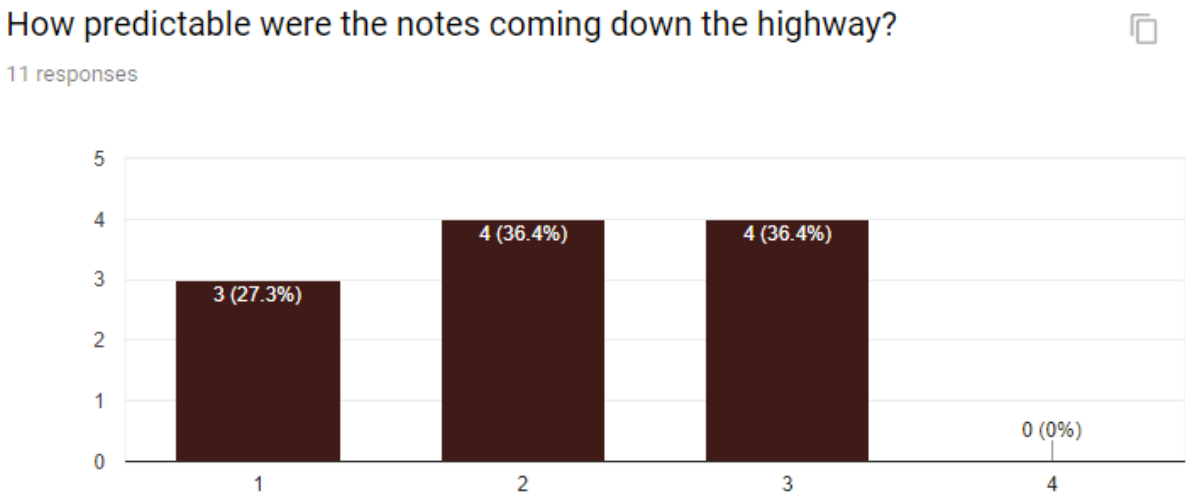


Figure 71:Post-Playtest Survey, Gameplay Results: Note predictability

on a scale of 1-4 with 1 as Not Predictable and 4 as Very Predictable

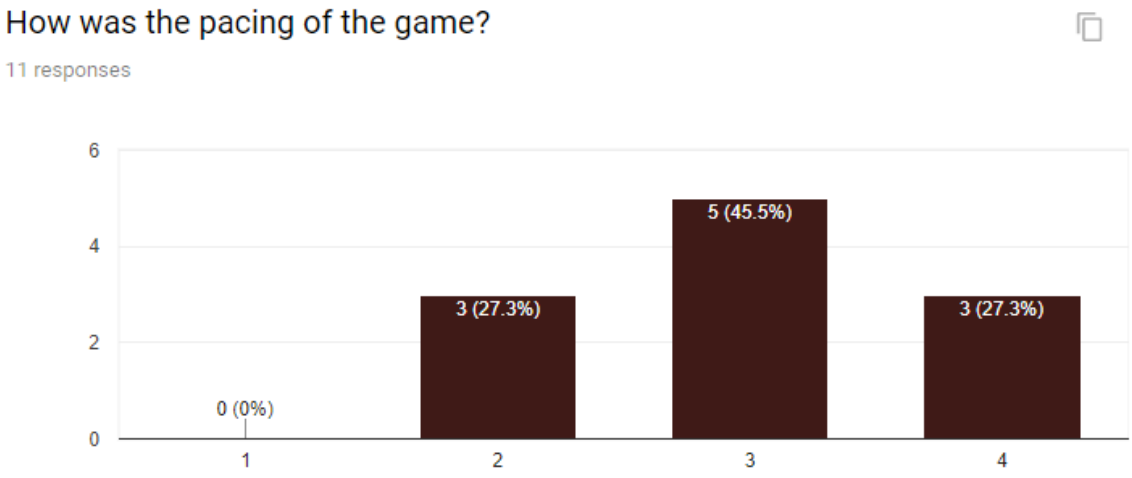


Figure 72: Post-Playtest Survey, Gameplay Results: Pacing

on a scale of 1-4 with 1 as Too Slow and 4 as Too Fast

Despite difficulty of playing, some subjects ended up wanting to play the game more than once. As we took notes on subjects during the playtest, we also tallied down the amount of times they played the game before wanting to quit.

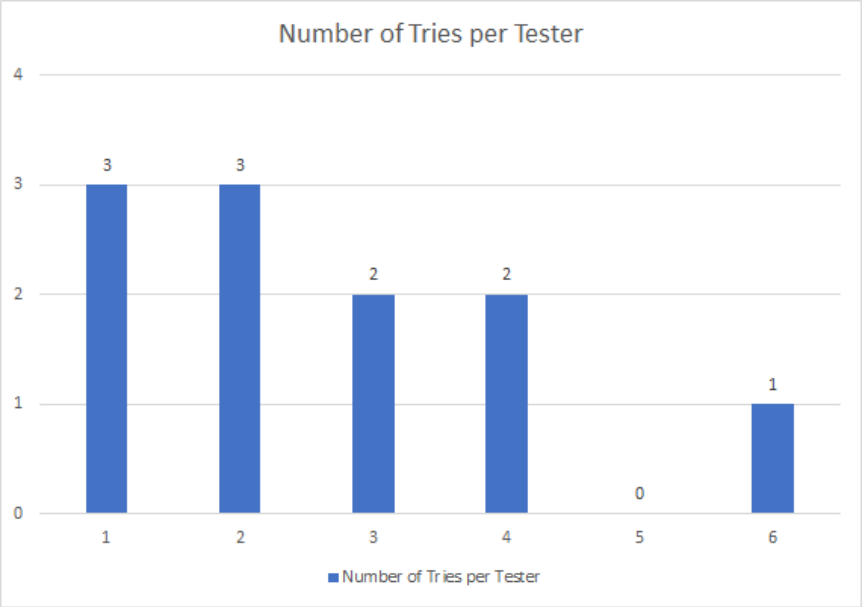


Figure 73: Chart on the number of tries per playtester

Multiple playthroughs demonstrated that our game does have some replayability with some players. During playtesting, most of the subjects that did replay did so because they had a better grasp of how the game worked and wanted to progress to a further point in the game compared to the previous attempt.

As for gameplay predictability, subjects found note predictability and boss behavior to be on the less predictable side. Lack of predictability is part of the goal of the note generation system and boss behavior as we want gameplay to vary in between sessions.

10.2.5 Emotional Assessment

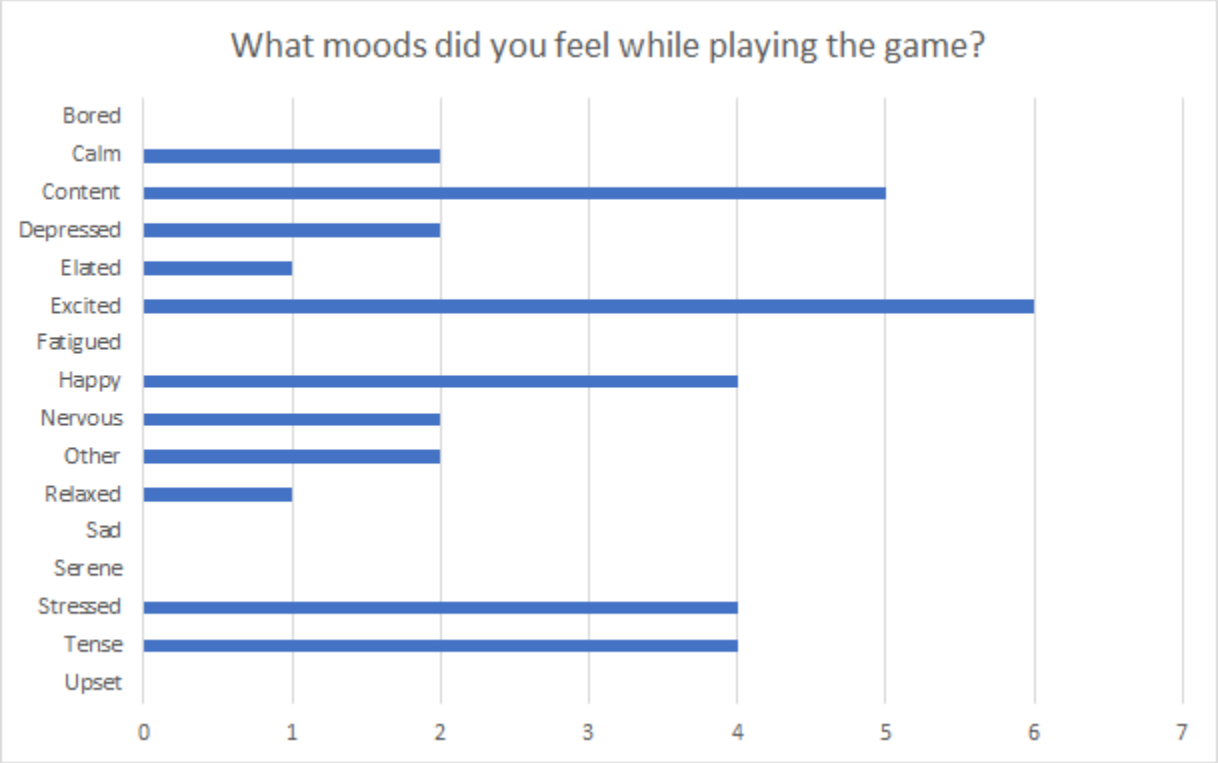


Figure 74: Post-Playtest Survey--Emotional Assessment, chart on moods felt during gameplay

Subjects experienced a wide range of emotions. During the playtest, subjects expressed frustration when trying to learn how to play the game. In the Post-Playtest Survey, many subjects said they were “Excited”, “Happy”, “Content”, “Stressed”, and “Tense”. In asking when the subject experienced the moods they have selected, a majority responded with feeling positive when they were able to hit Notes and negative when missing Notes or when they got attacked by the boss.

10.2.6 Sentiment Analysis

When running sentiment analysis on the audio recordings of the subjects, around half of them outputted a negative sentiment. It is likely that the subject vocally expressing confusion while learning the controls of the game led to an overall negative result. However, when looking at the transcript, many of the negative results point to subjects vocalizing that they have died in-game, thus skewing the data to be overall negative.

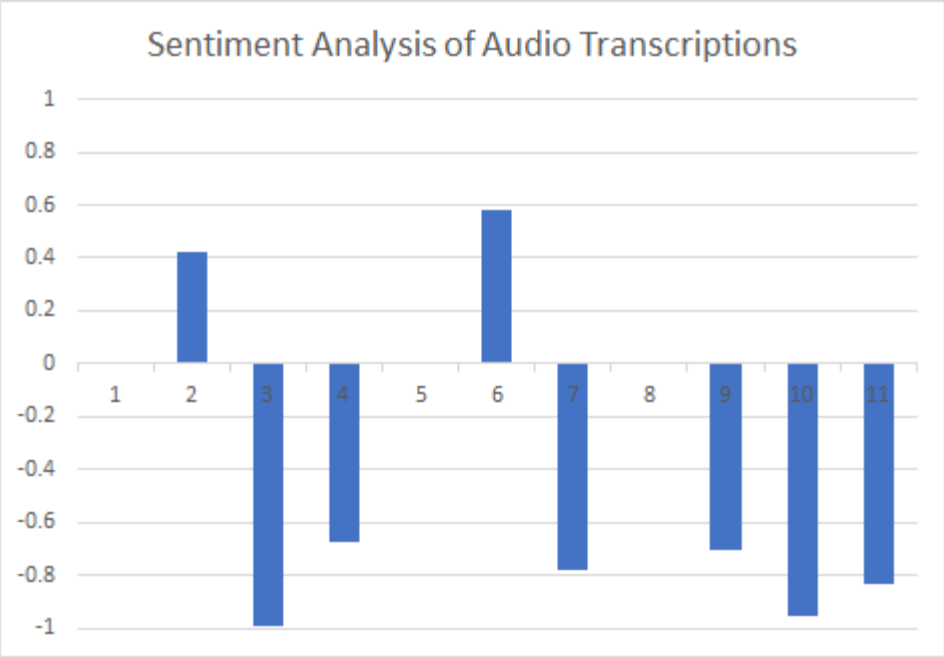


Figure 75: Chart of playtester's sentiment during audio recording

In *Figure 73*, subjects whose sentiment is zero is considered neutral as they had both positive and negative responses to the game.

When looking at subject comments in the post-playtest survey, many of the subjects' comments outputted a positive sentiment. For components of the game such as visuals and music, all subjects responded positively, expressing interest and delight. For gameplay, a majority responded positively, mentioning that it was fun and easy to learn. Subjects who expressed neutrally or negatively mentioned mechanics that they struggled with such as hitting Notes and moving.

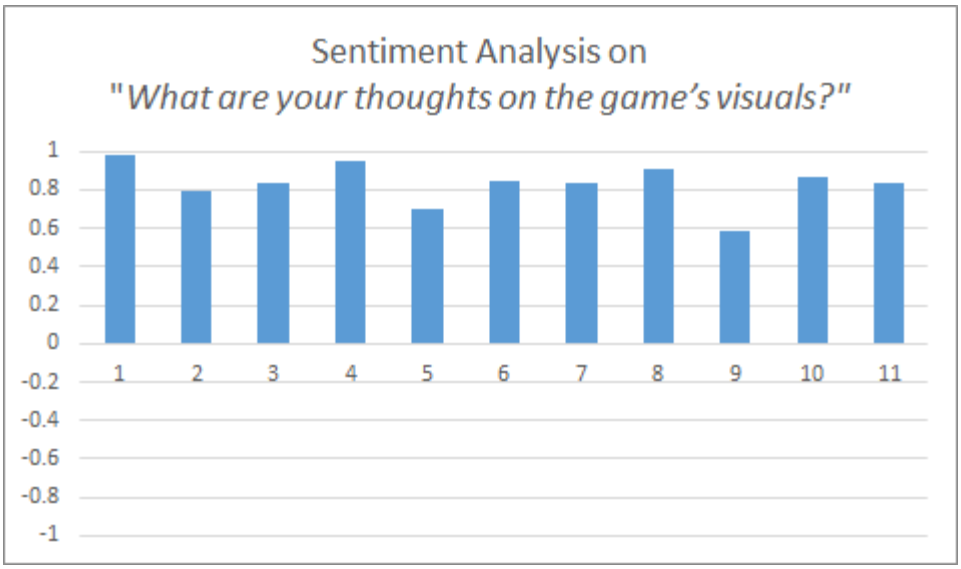


Figure 76: Post-Playtest Survey--Chart of playtester's sentiment on game visuals

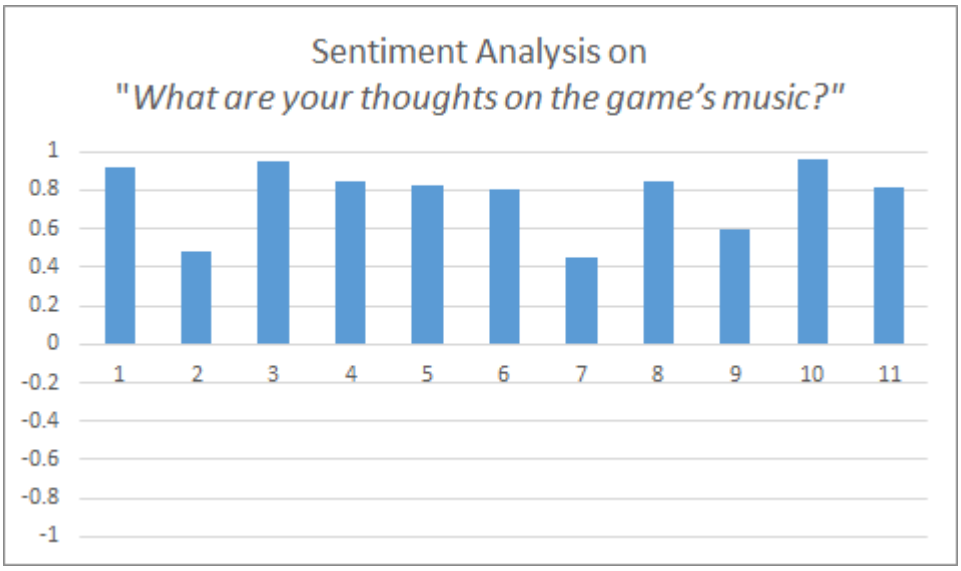


Figure 77: Post-Playtest Survey--Chart of playtester's sentiment on game music

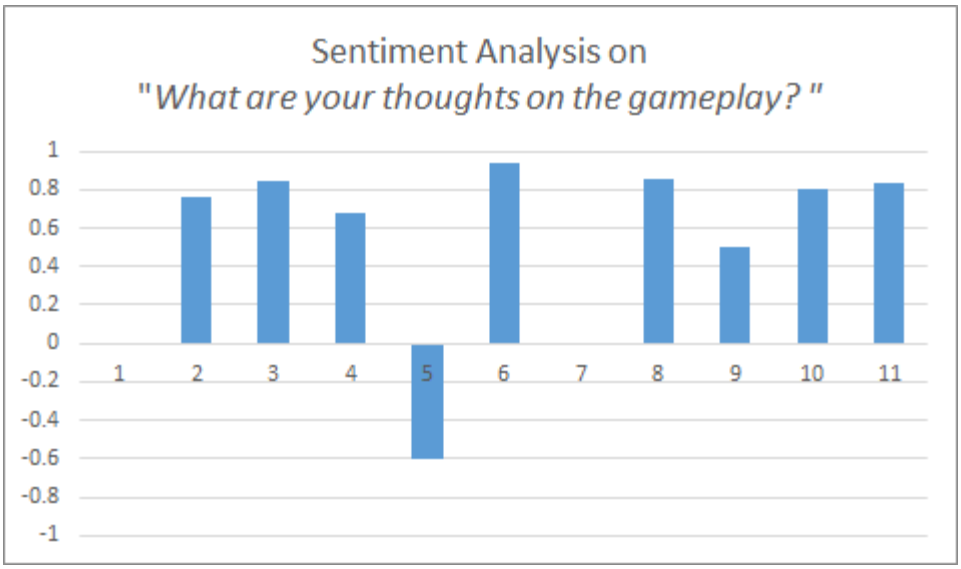


Figure 78: Post-Playtest Survey--Chart of playtester's sentiment on gameplay

10.2.7 Improvements Made

Based on subject observations and feedback, the major takeaways our team received were that the game was too difficult for players and that there needed to be more player feedback when certain events were occurring in-game (e.g. boss attacking). Following the playtests, we made the following improvements and changes:

- Increased the window for which notes can be hit
- Removed sixteenth notes altogether
- Added additional indicators for when the Cuckoo charging an attack (coloring fog orange)
- Added sound effects for player feedback
- Changed intro sequence so that it had predictable patterns
- Removed scripted sequences that were after the intro

10.2.8 PAX East

Doldrum was selected to be showcased at the WPI booth during PAX 2018. The team agreed to not conduct any formal playtesting due to the sheer volume of players and difficulty in setting up surveys for the number of players. Even without formal playtesting at PAX, we did ask players for the feedback on their experience. Many players were unfamiliar with VR and even fewer had played a VR rhythm game. Despite the unfamiliarity surrounding our game, *Doldrum* was received positively by those who played it.

There were a few minor bugs and problems with our game. Navigating the UI, specifically trying to get from main menu to tutorial, was a struggle for many players. However, people understood the game

quickly after being taught. The difficulty of the game was also an issue for many players. Not many players reached the chime phase of the game and many lost without ever utilizing our input bars. There was also a bug where the "damage dealt" would output 0 at the end of the game which discouraged many players who thought they were doing well. Additionally, many players seemed to read the tutorial without understanding what it meant. Usually, players fared better on subsequent attempts. Despite this, two players were able to defeat the Cuckoo Boss. This was achieved only after giving them a hint on how to get past the chime stage. Overall, we learned a lot from the players and discovered a few bugs. PAX was a success.

10.3 RELEASE

Following PAX East, the overall reception from those who played *Doldrum* at PAX was positive. Some asked if there were any plans for releasing the game. There were no plans originally to release on major content distributors such as *Steam*, mostly due to the amount of additional development time required to make the game marketable. *Itch.io* ended up as the favored alternative, where we were able to put up our game for distribution in less than an hour.

On April 15th 2018, we put *Doldrum* up for free download on itch.io. The download is a zip file containing the game's executable. Though playing the game using the HTC Vive is highly recommended, we have provided keyboard instructions for players who do not own the hardware.

Incoming visits from other sites over the past 30 days

Referrer	Visits
https://itch.io/games/free/input-htc-vive	20
https://itch.io/games/tag-rhythm	14
https://itch.io/games/newest/tag-virtual-reality	10
https://itch.io/games/free/tag-rhythm	7
https://itch.io/games/tag-virtual-reality	6
https://itch.io/games/newest/input-htc-vive	5

Figure 79: itch.io Analytics Panel, Visits to Doldrum's itch.io page from the following sites

Using itch.io's analytics page, we are able to view user engagement data from users on the site. Since then, the game page has received a small but steady amount of views as well as downloads. Within a week of release, the game has been downloaded 27 times. As seen in *Figure 78*, most engagement has been from user searches. Particularly, the download page has received the most traffic from people searching for free virtual reality games.

The itch.io game page has also received one user comment since release. Based on the comment, the user played the game without VR equipment. They found the keyboard controls to be unintuitive. However, the user found the art, music, and atmosphere enjoyable. Since the keyboard controls were strictly for debugging purposes, we did not formally playtest on keyboard controls. Therefore, this an expected criticism. We are glad that the response towards the art and music were similar to that in our formal playtesting data.

11 POST-MORTEM

11.1 DESIGN

Ultimately, *Doldrum*'s final iteration was an enjoyable yet fairly derivative rhythm game. Despite its attempts to tie together the action and rhythm game genre, the final design was extremely similar to other popular rhythm games such as *Guitar Hero* or *Rock Band*. While we wanted to experiment with players performing music in a freeform fashion, there was not enough guidance in showing what the player was supposed to do. This lack of structure resulted in a design that we believe would be confusing and unintuitive for players. Because of this, we found it safer to look upon the mechanics of successful rhythm games and iterate upon them. We incorporated a note highway as foundation and then used the action system to expand on the typical rhythm game design. The primary unique element of Doldrum ended up being the fact that it was VR rather than the boss fight. This was primarily due to the player's ability to interact with the boss being limited both in terms of what they could do and when they had the opportunity to make meaningful choices.

The Input Bar system combined with the player's ability to either attack or dodge meant that whenever the player was provided with the opportunity to make an action they only needed to react to what the boss was currently doing. If the boss was readying an attack, the player always had to dodge. If the boss was not readying an attack, the player could always safely make an attack input themselves. Thus, the Input Bar system was fairly simplistic and did not adequately allow for an engaging combat experience.

Despite the ambitious departure from conventional rhythm games in early designs, it was concluded that the conventions exist for a reason. Without the strict requirement to continuously hit notes on beat, there is little incentive for the player to engage with the game's rhythm. While we made many attempts to break the conventions to allow for more player improvisation, we found that the designs were too open-ended to create engaging gameplay.

However, the significant reduction in scope that *Doldrum* experienced allowed us to focus on polish and game feel rather than sheer content.

11.2 Future Work

As important as working on this project was as an experience, our team does not have any plans on continuing work on it after graduating. The members of our team are heading in different directions following graduation as we begin our professional careers. However, there is a great amount of potential in the base mechanics and the scrapped content. Thus, we may return to *Doldrum* in the future.

11.3 What Went Wrong

The most significant problems we encountered were with scoping and communication. The combination of this delay and overscoping led to much of the initial game being cut back or removed. Additionally, communication was an issue for our team in the early terms of the year. While individual team members worked on their respective tasks, there was a lack of cohesion in team-wide goals. This resulted in a disorganized project. Later, we established multiple weekly meeting times to reinforce team cohesion.

An additional problem was with the lab equipment. The computer in the IMGD VR lab was not setup properly to facilitate development in Unreal Engine 4. There were a number of programs that required administrative privileges in order to install or run. This significantly hindered our initial development. As a result, we were forced to develop without a VR system until late B-Term. This meant that all VR testing and adjustments were delayed until the beginning of C-Term.

We encourage teams to build a vertical slice rather than attempt to create a larger game. By taking this approach, we were able to polish the game to our standards. Additionally, we encourage teams to fail fast and cut any content that does not fit with the game's experience goals. We spent too much time iterating on the design in the first half of the academic year. If we had decided on a design earlier, we could have had more time to add more polish and content to Doldrum.

11.4 What Went Right

Despite early difficulties, we ultimately created a fun and compelling rhythm-action game. *Doldrum* is a polished vertical slice of an ambitious concept. All of the challenges we overcame gave us valuable experience that we can bring into future work we take on.

It was very useful for the entire team to play games together. We were able to discuss aspects of gameplay as they were happening and decide whether or not to incorporate elements into *Doldrum.* More importantly, these playthroughs brought the team together, boosting morale and camaraderie.

Through development, the team gained a myriad of technical skills. Programmers gained experience with C++, Unreal Engine 4, and Perforce. David experimented with performance-based composition and learned how to use the Maschine mk3 hardware. Kent gained experience with rigging and 3D animation. The most significant achievement was learning how to develop for VR. Designers had to consider the unique benefits and constraints of VR. Programmers and artists had to optimize to meet the high performance impact of the Vive. Finally, we learned how to develop a game which was larger in scope than any game we had ever made. We accomplished this through frequent meetings and a variety of software. Despite communication problems and a slow start to development, Doldrum is a polished and engaging virtual reality game.

12 REFERENCES

[1] 3D_Guy_2008. 2010. wing Rig Animation Test. Video. (October 2010). Retrieved from https://www.youtube.com/watch?v=YHKQTWI9qh0

[2] David Allen. 2018. Doldrum - Backing Tracks. Retrieved from https://soundcloud.com/fraud-sound

[3] Atlus. 2008. Shin Megami Tensei: Persona 4. Game [PlayStation 2]. (10 July 2008). Played September 2017.

[4] Gary Bailey. 2016. Celebrating 15 years of Silent Hill 2. (September 2016). Retrieved http://www.godisageek.com/2016/09/celebrating-15-years-of-silent-hill-2/

[5] Sean Baron. 2012. Cognitive Flow: The Psychology of Great Game Design. (March 2012). Retrieved from https://www.gamasutra.com/view/feature/166972/cognitive_flow_the_psychology_of_.php

[6] Bass. 2017. Import Review: Taiko no Tatsujin: Drum Session! (November 2017). Retrieved from https://www.destructoid.com/import-review-taiko-no-tatsujin-drum-session--471536.phtml

[7] Team Coco. 2016. YouTube VR Lab Outtake: Conan Plays "Space Pirate Trainer". Video. (November 2016) Retrieved from http://teamcoco.com/video/conan-plays-space-pirates

[8] Craigsapp. 2018. Midifile. Retrieved from http://midifile.sapp.org/

[9] colxi. JSMIDIparser javascript Library: .MIDI file to JS Object. Retrieved from http://www.colxi.info/portfolio/jsmidiparser-midi-binary-file-to-javascript-object-library/

[10] Mihaly Csikszentmihalyi. 2004. Flow, the secret to happiness. Video. (February 2004). Retrieved from https://www.ted.com/talks/mihaly_csikszentmihalyi_on_flow

[11] Dustyn Bush. 2016. Thumper - Stage 6 Final Boss. (October 2016). Retrieved from https://www.youtube.com/watch?v=eBfjIjQFrRw

[12] Kate Compton, Ben Kybartas, and Michael Mateas. 2015. Tracery: An Author-Focused Generative Text Tool. Interactive Storytelling Lecture Notes in Computer Science (December 2015), 154–161. DOI:http://dx.doi.org/10.1007/978-3-319-27036-4_14

[13] Pieter Desmet and Paul Hekkert. 2007. Framework of Product Experience. International Journal of Design 1, 1 (March 2007), 13–23.

[14] Drool. 2016. Thumper. Game [PC]. (10 October 2016). Played September 2017.

[15] Epic Games. 2018. Blueprints Visual Scripting. Retrieved from https://docs.unrealengine.com/en-us/Engine/Blueprints

[16] Moby Francke and Randy Lundeen. 2008. How Valve Connects Art Direction to Gameplay. (2008). Retrieved from http://www.valvesoftware.com/publications/2008/GameFest08_ArtInSource.pdf

[17] Free Lives. 2017. Gorn. Game [PC]. (10 July 2017). Played September 2017.

[18] Gaming with Me. 2017. Ash's pikachu Vs Ho-Oh legendary Pokemon. Video. (December 2017). Retrieved from https://www.youtube.com/watch?v=7ZBrt9N0oTc

[19] GeneralMcBadass. 2011. Patapon 3 Boss: The Great Gigante King w/ Cannogabang [1/3]. Video. (May 2011) Retrieved from https://www.youtube.com/watch?v=DHw54llgZXY

[20] Google. 2018. A new dimension - Designing for Google Cardboard. Retrieved from https://designguidelines.withgoogle.com/cardboard/designing-for-google-cardboard/a-new-dimension.html

[21] Kyle Hanson. 2018. Audioshield Update Adds YouTube Support, Removes Soundcloud. (March 2018). Retrieved from https://attackofthefanboy.com/news/audioshield-update-adds-youtube-support-removes-soundcloud/

[22] I-Illusions. 2017. Space Pirate Trainer. Game [PC]. (12 October 2017). Played September 2017.
[23] JUDOKATANOKA. 2014. Street Fighter sound Hadouken. Video. (January 2014). Retrieved from https://www.youtube.com/watch?v=49jKoMbJyEo
[24] Konami. 2001. Silent Hill 2. Game [PlayStation 2]. (24 September 2001). Konami, Kyoto, Japan. Played September 2017.
[25] Qiufeng Lin, John Rieser, and Bobby Bodenheimer. 2012. Stepping over and ducking under. Proceedings of the ACM Symposium on Applied Perception - SAP 12 (2012). DOI:http://dx.doi.org/10.1145/2338676.2338678
[26] Tom Looman. 2017. VR Template Guide for Unreal Engine 4. (November 2017). Retrieved http://www.tomlooman.com/vrtemplate/
[27] Tony Lynch. 2014. Writing Up Qualitative Research. dissertation.
[28] Moby Games. 2018. Guitar Hero III: Legends of Rock. Retrieved from https://www.mobygames.com/game/windows/guitar-hero-iii-legends-of-rock/
[29] Natalie Neumann. 2017. Puella Magi Madoka Magica Review. (June 2017). Retrieved from https://nigmabox.wordpress.com/2012/11/29/threeweeksinthewaiting/
[30] Oculus. 2018. Introduction to Best Practices. Retrieved from https://developer.oculus.com/design/latest/concepts/book-bp/
[31] PlayStation. 2018. Patapon™ Remastered. Retrieved from https://www.playstation.com/en-us/games/patapon-remastered-ps4/
[32] RUST LTD. 2016. Hot Dogs, Horseshoes & Hand Grenades. Game [PC]. (5 April 2016). Played September 2017.
[33] Melena Ryzik. 2015. Guillermo del Toro's House of Horrors. (October 2015). Retrieved from https://www.nytimes.com/interactive/2015/10/07/movies/11guillermodeltoro-house.html
[34] SHN Survival Horror Network. 2015. Rule of Rose Full HD 1080p/60fps Longplay Walkthrough. Video. (8 November 2015). Retrieved from https://www.youtube.com/watch?v=YZjnaTmhp3I
[35] Barbara Sommer. 2006. Measuring attitudes. Retrieved from http://psc.dss.ucdavis.edu/sommerb/sommerdemo/scaling/attitude.htm
[36] Alex Schwartz and Devin Reimer. 2017. 'Rick and Morty: Virtual Rick-ality' Postmortem: VR Lessons *Burrrp* Learned. Video. (September 2017). Retrieved from https://www.youtube.com/watch?v=7aqIbeQQL8c
[37] Sony Picture Classics. 2014. Whiplash. Film. (16 January 2014). Retrieved April 2017.
[38] SUPERHOT Team. 2017. SUPERHOT VR. Game [PC]. (25 May 2017) Played September 2017.
[39] UEAKCrash. 2016. Audioshield (Vive VR Gameplay). Video. (May 2016). Retrieved from https://www.youtube.com/watch?v=BWEC_MBB1pA
[40] Unity. 2015. User Interfaces for VR. Retrieved from https://unity3d.com/learn/tutorials/topics/virtual-reality/user-interfaces-vr
[41] Gary Wolf. 2016. Measuring Mood – Current Research and New Ideas. (August 2016). Retrieved from http://quantifiedself.com/2009/02/measuring-mood-current-resea/

13 APPENDIX

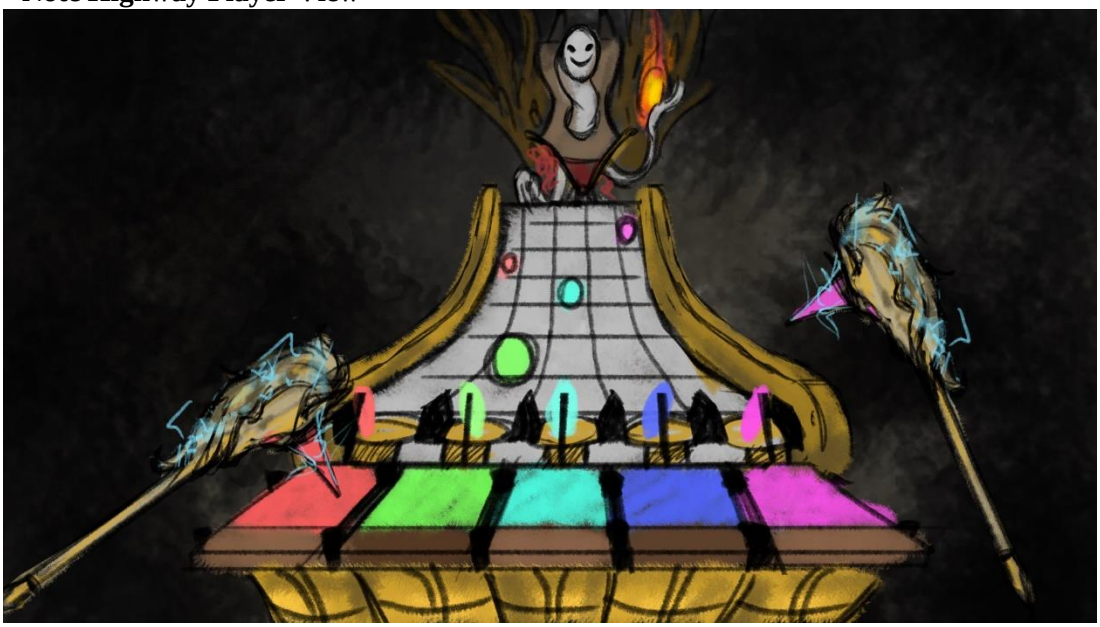
13.1 ADDITIONAL CONCEPT ART

13.1.1 Trumpet Boss

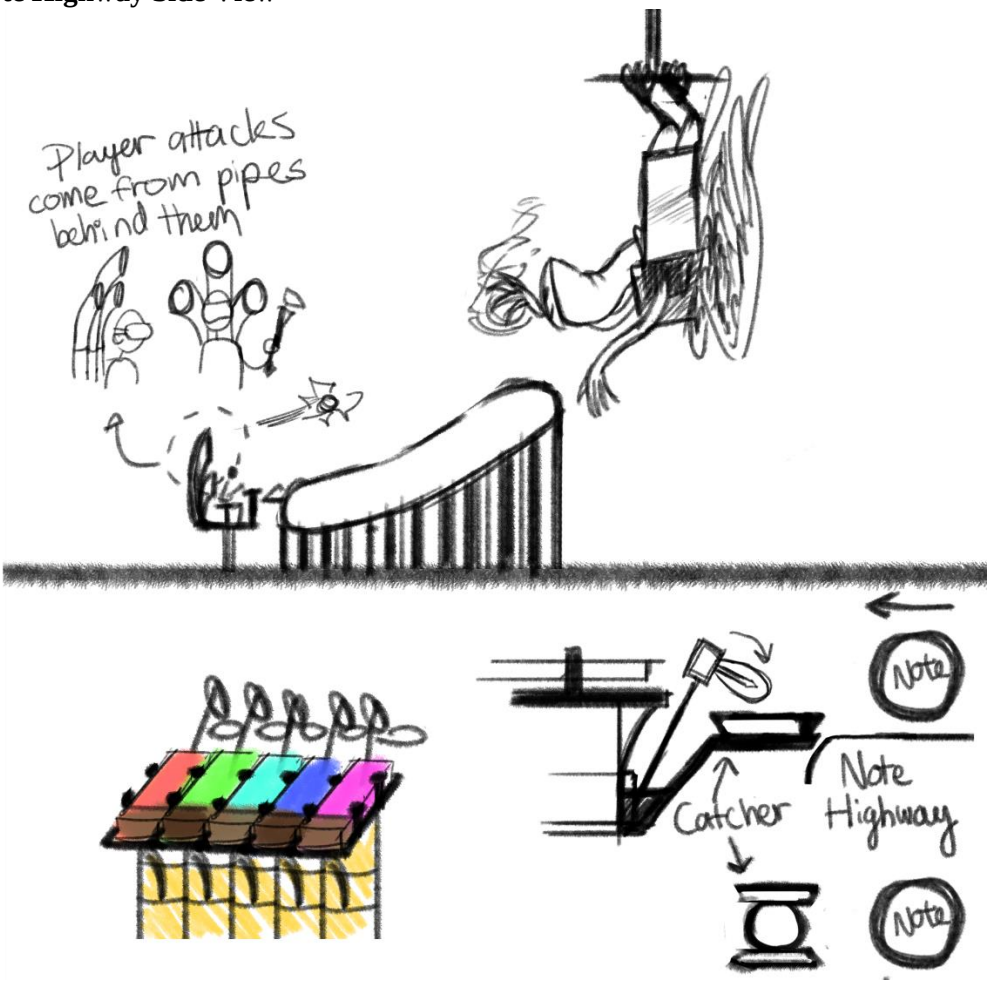
13.1.2 Audience Boss Environment

13.1.3 Owl Mentor Environment

13.1.4 Note Highway Player View

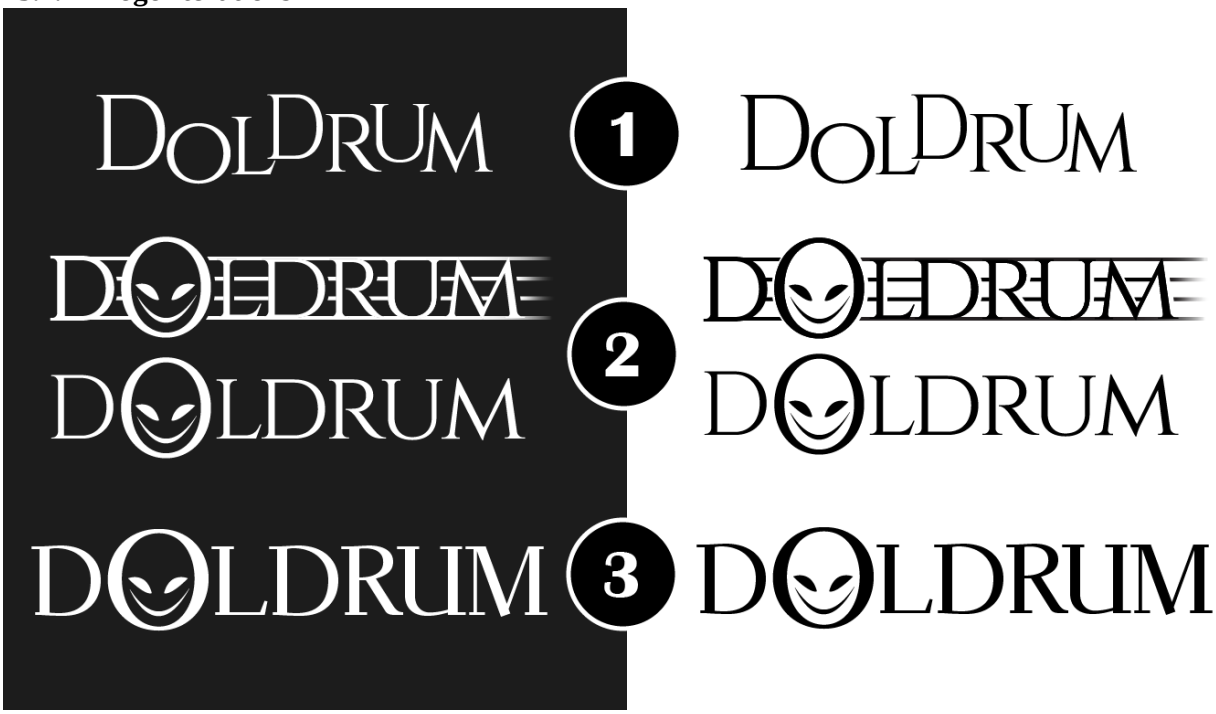


13.1.5 Note Highway Side View



13.1.6 Menu Concept

13.1.7 Logo Iterations



13.2 Playtest Informed Consent Form

Informed Consent Agreement for Participation in a Research Study

Investigator(s): VR Horror MQP Team

- David Allen (dpallen@wpi.edu)
- Kent Fong (kfong@wpi.edu)
- Matt Szpunar (mjszpunar@wpi.edu)
- Henry Wheeler-Mackta (hjwheelermackta@wpi.edu)
- Kelly Zhang (kezhang@wpi.edu)

Contact Information: vrpmqp@wpi.edu

Title of Research Study: DOLDRUM Playtesting

Introduction: You are being asked to participate in a research study. Before you agree, however, you must be fully informed about the purpose of the study, the procedures to be followed, and any benefits, risks or discomfort that you may experience as a result of your participation. This form presents information about the study so that you may make a fully informed decision regarding your participation.

Purpose of the study: The goal is to observe and analyze how players behave and feel during and after playing our game, DOLDRUM, while using virtual reality technology.

About the Game: DOLDRUM is a horror-esque, action, and rhythm virtual reality game. As the player in the virtual world setting, you will be equipped with a set of drums and drumsticks. To survive, Hit the notes as they come towards you by hitting the drums with your drumsticks. Perform actions on the drums with your drumsticks to fight against the enemy in front of you. Fight, drum and survive until the very end to win.

Procedures to be followed: The study is to be conducted within 30 minutes or less. You will be seated during the playtest. You will also be fitted with an HTC Vive headset and controllers. When comfortable, signal to the investigator(s) that you are ready to begin the playtest.

During the playtest, we encourage you to speak out your thoughts and actions. The investigator(s) will be taking notes on your behavior during the procedure.

If at any moment during the playtest you feel uncomfortable and would like to withdraw early, pause the game and tell the investigator(s) to stop the playtest.

Upon finishing the game, either through withdrawing early, or reaching the win/lose condition, you will leave the play area with the virtual reality equipment. You are free to leave at this point. There is a post-test survey you can take after the playtest. You have the right to choose to not complete the survey.

Risks to study participants: Potential and foreseeable risks you may experience following this study include: motion sickness during gameplay, contracting pink eye via headset. The investigator(s) wipe down equipment following each study to mitigate the risk of pink eye.

Benefits to research participants and others: While there are no immediate benefits to you on completing the playtest, your contributions to playtesting will help us, the investigator(s), to understand user behaviors in virtual reality better and create more enjoyable virtual reality experience in DOLDRUM.

Record keeping and confidentiality: Data recorded will be anonymous, with only demographic info preserved, and stored in a Google Drive accessed only by the MQP team. Records of your participation in this study will be held confidential so far as permitted by law. However, the study investigators, the sponsor or it's designee and, under certain circumstances, the Worcester Polytechnic Institute Institutional Review Board (WPI IRB) will be able to inspect and have access to confidential data that identify you by name. Any publication or presentation of the data will not identify you.

Compensation or treatment in the event of injury: In the event of injury during the playtest, the investigator(s) will immediately call WPI EMS for medical assistance, but will be unable to compensate costs regarding said injury. You do not give up any of your legal rights by signing this statement.

For more information about this research or about the rights of research participants, or in case of research-related injury, contact:

VR Horror MQP Team (Email: vrpmqp@wpi.edu)

VR Horror MQP Adviser (Ralph Sutter, Email: rsutter@wpi.edu)

VR Horror MQP Co-Adviser (Gillian Smith, Email: gmsmith@wpi.edu)

IRB Chair (Professor Kent Rissmiller, Tel. 508-831-5019, Email: kjr@wpi.edu)

University Compliance Officer (Jon Bartelson, Tel. 508-831-5725, Email: jonb@wpi.edu)

Your participation in this research is voluntary. Your refusal to participate will not result in any penalty to you or any loss of benefits to which you may otherwise be entitled. You may decide to stop participating in the research at any time without penalty or loss of other benefits. The project investigators retain the right to cancel or postpone the experimental procedures at any time they see fit.

By signing below, you acknowledge that you have been informed about and consent to be a participant in the study described above. Make sure that your questions are answered to your satisfaction before signing. You are entitled to retain a copy of this consent agreement.

_____ Date: _____

Study Participant Signature

_____

Study Participant Name (Please print)

_____ Date: _____

Signature of Person who explained this study

Additional clauses to add to Consent Agreements, as appropriate:

The treatment or procedures used in this research may involve risks to the subject (or to an embryo or fetus, if the subject is or may become pregnant), which are currently unknown or unforeseeable.

Additional costs to the subject that may result from participation in this research include: None.

Significant new findings or information, developed during the research, may alter the subject's willingness to participate in the study. Any such findings will be promptly communicated to all research participants.

Should a participant wish to withdraw from the study after it has begun, the following procedures should be followed:

1. Pause the game OR Press the Vive Menu button on any of the controllers.
2. Vocally state to investigator(s) to withdraw and/or exit the game.

The consequences for early withdrawal for the subject and the research are: The Investigators will still use data from the participant unless explicitly requested to discard data recorded during the playtest.

Special Exceptions: Under certain circumstances**,** an IRB may approve a consent procedure which differs from some of the elements of informed consent set forth above. Before doing so, however, the IRB must make findings regarding the research justification for different procedures (i.e. a waiver of some of the informed consent requirements must be necessary for the research is to be "practicably carried out.") The IRB must also find that the research involves "no more than minimal risk to the subjects." Other requirements are found at 45 C.F.R. §46.116.

13.3 POST-PLAYTEST SURVEY QUESTIONS

1) Attitude Assessment (Likert scales 1-5 unless mentioned otherwise)
    - **a. Physical Assessment**
        - i. How comfortable was it to wear the headset?
          (Very Uncomfortable - Very Comfortable)
        - ii. How comfortable was it to handle the controllers?
          (Very Uncomfortable - Very Comfortable)
        - iii. How motion sick do you feel after playing the game?
          (Very Uncomfortable - Very Comfortable)
    - **b. Usability Assessment**
        - i. How difficult was it to navigate the menu?
          (Very Hard - Very Easy)
        - ii. How difficult was it to hit the notes?
          (Very Hard - Very Easy)
        - iii. How difficult was it to perform the attack action?
          (Very Hard - Very Easy)
        - iv. How difficult was it to perform the dodge action?
          (Very Hard - Very Easy)
    - **c. Gameplay Assessment**
        - i. How difficult was it to understand how the drum worked?
          (Very Hard - Very Easy)
        - ii. How difficult was it to understand what the boss did?
          (Very Hard - Very Easy)
        - iii. How difficult was it to interpret the notes coming at you?
          (Very Hard - Very Easy)
        - iv. How was the pacing of the game?
          (Too slow - Too fast)
        - v. How predictable were the notes coming down the highway?
          (Unpredictable - Very Predictable)
        - vi. How predictable were the boss' actions?
          (Unpredictable - Very Predictable)
    - d. Emotional Assessment
        - i. What moods did you feel while playing the game? (Check all that apply)
            1. Excited
            2. Elated
            3. Happy
            4. Content
            5. Serene
            6. Relaxed
            7. Calm
            8. Fatigued
            9. Bored
            10. Depressed
            11. Sad
            12. Upset
            13. Stressed
            14. Nervous
            15. Tense
        - ii. When did you experience the moods that you have selected?

2) Subject Write-in Questions and Comments
    a. What are your thoughts on the game's visuals?
    b. What are your thoughts on the game's sounds?
    c. What are your thoughts on the gameplay?
    d. If you were to change something in the game, what would it be?
    e. Do you have any additional feedback about the game?
3) General Demographic Information
    a. Age
    b. Have you played in VR before this playtest?
        i. If so, which platform(s)? (HTC Vive, Oculus, etc.)
    c. Have you played a rhythm game before (i.e. Guitar Hero, Rock Band)?
        i. If so, list the title(s)

13.4 HEADER FILES

13.4.1 BossCharacter.h

```
UENUM(BlueprintType)
enum class ENoteLength : uint8
{
	One_Fourth		UMETA(DisplayName = "1/4"),
	One_Eighth		UMETA(DisplayName = "1/8"),
	One_Sixteenth	UMETA(DisplayName = "1/16")
};

UCLASS()
class ORB_MQP_API ABossCharacter : public ACharacter, public IBossStateMachine
{
	GENERATED_BODY()

public:
	// Sets default values for this character's properties
	ABossCharacter();

#pragma region BP Properties
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Boss State: Settings")
		bool startOnBeginPlay;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Boss State: Settings")
		float startDelay;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Boss State: Settings")
		bool showDebug;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Boss State: Settings")
		int debugVerbosity;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Boss State: Settings")
		ENoteLength noteLength;
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Boss State: Settings")
		AMetronomeController *targetController;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Boss State")
		EState RootState;
	UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Boss State")
		EState CurrentState;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Boss State")
		TMap<EState, FState> StateTransitionMap;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Boss State")
		TArray<FPhase> PhaseSequence;
#pragma endregion

#pragma region Actions
	UFUNCTION(BlueprintImplementableEvent, Category = "Boss State|Action")
		void Idle(const ESubStateIdle& subState, const EStateProgress& stage);
	UFUNCTION(BlueprintImplementableEvent, Category = "Boss State|Action")
		void Attack(const ESubStateAttack& subState, const EStateProgress& stage);
#pragma endregion Implement in BP

#pragma region State Machine Events
	virtual void Setup_Implementation() override;
	virtual void Start_Implementation() override;
	virtual void OnStateEnter_Implementation(const EState & from, const EState & to) override;
	virtual void OnState_Implementation(const EState & from, const EState & to) override;
	virtual void OnStateExit_Implementation(const EState & from, const EState & to) override;
	virtual bool isValidTransition_Implementation(const EState& from, const EState& to) override;
#pragma endregion Overridable in BP

#pragma region State Machine Actions
	// ExecuteAction: Runs the appropriate BlueprintImplementableEvent given state
	void ExecuteAction(const EState& state, const EStateProgress& stage = EStateProgress::SE_During);

	// ChangeState: Changes state of state machine given from and to states.
```

```
	//		if shouldInterrupt is true, any states queued up will be emptied and desired
state
	//		to change to will be added to run.
	UFUNCTION(BlueprintCallable, Category = "Boss State")
		EState ChangeState(EState to, bool forceChange = false);
	// ChangeRandomState: Changes state of state machine randomly. Values can be passed in to
influence random result
	//	setting int values to -1 will cause that value to not influence the result
	UFUNCTION(BlueprintCallable, Category = "Boss State")
		EState ChangeRandomState(bool forceChange = false);
	UFUNCTION(BlueprintCallable, Category = "Boss State|Action")
		void StopMachine();
	UFUNCTION(BlueprintCallable)
		bool IsValidTransitionMap();
	UFUNCTION()
		void OnMetronomeBeat();

#pragma endregion

protected:
	// Flag in case we get an external call to the state machine (i.e. player input)
	bool bIsRunningAState;
	// Queued up states to be run by AsyncTask
	TQueue<EState> stateQueue;

	UMetronomeListenerComponent *listener;

	// AActor Functions //
	virtual void BeginPlay() override;
	virtual void BeginDestroy() override;

private:
	bool bIsStopping;
	bool bMachineStarted;
	EState mFromState;
	EState mToState;
	uint8 mSubstate;
	EStateProgress mCurrentStateProgress;
	int mBeatCounter;
	int mSubDivCounter;

	// Helper function to add transitions to StateTransitionMap
	void ADD_TRANSITION(FState t, FStateEntry s);
	void LogToScreen(FString s, FColor c, bool showDebug, int debugLevel);

	FStateEntry GetStateEntry(EState fromState, EState toState);
	EState GetNextState(EState fromState);

};
```

13.4.2 BossState.h

```
UENUM(BlueprintType)
enum class EState : uint8
{
	SE_Any		UMETA(DisplayName = "Any"),
	SE_Idle UMETA(DisplayName = "Idle"),
	SE_Attack	UMETA(DisplayName = "Attack"),
	SE_STOP		UMETA(Hidden)
};

UENUM(BlueprintType)
enum class ESubStateAttack : uint8
{
	SE_Base			UMETA(DisplayName = "Base Attack"),
	SE_Line			UMETA(DisplayName = "Line Attack"),
	SE_Projectile	UMETA(DisplayName = "Projectile Attack"),
	NUM_SUBSTATES	UMETA(Hidden)
};
UENUM(BlueprintType)
enum class ESubStateIdle : uint8
{
	SE_SwapLeft		UMETA(DisplayName = "Swap Left"),
	SE_SwapCenter	UMETA(DisplayName = "Swap Center"),
	SE_SwapRight	UMETA(DisplayName = "Swap Right"),
	NUM_SUBSTATES	UMETA(Hidden)
};

UENUM(BlueprintType)
enum class EStateProgress : uint8
{
	SE_EnterUMETA(DisplayName = "Enter"),
	SE_During	UMETA(DisplayName = "During"),
	SE_Exit		UMETA(DisplayName = "Exit")
};

USTRUCT(BlueprintType)
struct ORB_MQP_API FStateEntry
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
		EState targetState;
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
		int enterFinishDuration;
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
		int duringFinishDuration;
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
		int exitFinishDuration;

	FStateEntry(EState state, float onEnterDuration, float duringDuration, float exitDuration) {
		targetState = state;
		enterFinishDuration = onEnterDuration;
		duringFinishDuration = duringDuration;
		exitFinishDuration = exitDuration;
	}

	FStateEntry(EState state) {
		targetState = state;
		enterFinishDuration = 0;
		duringFinishDuration = 0;
		exitFinishDuration = 0;
	}

	FStateEntry() {}
};

USTRUCT(BlueprintType)
```

```
struct ORB_MQP_API FState
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere, BlueprintReadWrite)
		TArray<FStateEntry> entries;

	//UFUNCTION(BlueprintCallable, Category = "Boss State|Utility")
		static FString EStateToString(EState state);

	//UFUNCTION(BlueprintCallable, Category = "Boss State|Utility")
		static FString EStateProgressToString(EStateProgress state);
};

class ORB_MQP_API FRunStateTask : public FNonAbandonableTask
{
	friend class FAutoDeleteAsyncTask<FRunStateTask>;

public:
	typedef void(*FunctionPtrType)(void);
	FRunStateTask(TFunction<void(void)> function) :
		func(function) {}

protected:
	TFunction<void(void)> func;

	void DoWork();

	FORCEINLINE TStatId GetStatId() const
	{
		RETURN_QUICK_DECLARE_CYCLE_STAT(FMyTaskName, STATGROUP_ThreadPoolAsyncTasks);
	}

};
```

13.4.3 BossStateMachine.h

```
UINTERFACE(BlueprintType, Blueprintable)
class ORB_MQP_API UBossStateMachine : public UInterface
{
	GENERATED_UINTERFACE_BODY()
};

class ORB_MQP_API IBossStateMachine
{
	GENERATED_IINTERFACE_BODY()

public:

	UFUNCTION(BlueprintNativeEvent, Category = "Boss State|Event")
		void Setup();
	UFUNCTION(BlueprintNativeEvent, Category = "Boss State|Action")
		void Start();
	UFUNCTION(BlueprintNativeEvent, Category = "Boss State|Event")
		void OnStateEnter(const EState & from, const EState & to);
	UFUNCTION(BlueprintNativeEvent, Category = "Boss State|Event")
		void OnState(const EState & from, const EState & to);
	UFUNCTION(BlueprintNativeEvent, Category = "Boss State|Event")
		void OnStateExit(const EState & from, const EState & to);
	UFUNCTION(BlueprintNativeEvent, Category = "Boss State")
		bool isValidTransition(const EState & from, const EState & to);

};
```

13.4.4 MetronomeController.h

```
/**
 * MetronomeController.h
 *	Metronome object that derives from the Actor class.
 *	Multiple instances can be held in a level with their own designated BPM (Beats Per Minute)
 */
UCLASS()
class ORB_MQP_API AMetronomeController : public AActor
{
	GENERATED_BODY()

public:
	// Sets default values for this actor's properties
	AMetronomeController();

	// Constants
	const int MAX_BPM = 440;
	const int MIN_BPM = 20;
	const int MAX_LISTENERS = 20;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Metronome")
		bool bStartOnBeginPlay;

	// Starts metronome
	UFUNCTION(BlueprintCallable, Category = "Metronome")
		void Start();
	// Stops metronome
	UFUNCTION(BlueprintCallable, Category = "Metronome")
		void Stop(bool shouldClearListeners);

	// Getter & Setter for BPM
	UFUNCTION(BlueprintCallable, Category = "Metronome")
		int GetBPM() const;
	UFUNCTION(BlueprintCallable, Category = "Metronome")
		bool SetBPM(int newBPM);

	FTimerHandle GetTimerHandle() const;

	// Listener Registration
	//	NOTE: Possible to expose these functions as BlueprintCallable in the future but only if
necessary
	AMetronomeController *RegisterListener(UMetronomeListenerComponent *comp);
	bool UnregisterListener(UMetronomeListenerComponent *comp);
	void ClearListeners();

	int BeatsUntilNextMeasure();

	ENoteType GetCurrentNoteType();
	ENoteType GetNextNoteType();

	// Converts BPM (Beats per Minute) into seconds
	float BpmToSeconds(int bpm);

protected:
	// Called when the game starts or when spawned
	virtual void BeginPlay() override;

	// Objects listening to this metronome
	TArray<UMetronomeListenerComponent*> listeners;

	// Calls all objects listening
	void SendBeatToListeners();

private:

	const int SUBDIVIDE_FOURTH_MAX = 4;
	const int SUBDIVIDE_EIGHTH_MAX = 2;
```

```
        const int SUBDIVIDE_SIXTEENTH_MAX = 1;

        // Internal bpm value
        int _bpm = 60;
        long _internalCount = 0;

        // Handle for timer
        FTimerHandle MemberTimerHandle;
};
```

13.4.5 MetronomeListenerComponent.h

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE(FBeatDelegate);

UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class ORB_MQP_API UMetronomeListenerComponent : public UActorComponent
{
	GENERATED_BODY()

public:
	// Sets default values for this component's properties
	UMetronomeListenerComponent();

	int64 BeatsSinceBeginPlay;

	/* MetronomeController to listen to */
	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Metronome")
		AMetronomeController *targetController;

	/* Blueprints uses this to listen for beats from MetronomeController */
	UPROPERTY(BlueprintAssignable, Category = "Metronome")
		FBeatDelegate OnBeatEvent;

	UFUNCTION(BlueprintCallable, Category = "Metronome")
		float TimeBeforeNextBeat();

	UFUNCTION(BlueprintCallable, Category = "Metronome")
		float TimeSincePreviousBeat();

	UFUNCTION(BlueprintCallable, Category = "Metronome")
		bool InProximityOfBeat(float offset);

	UFUNCTION(BlueprintCallable, Category = "Metronome")
		float GetSecondsPerBeat();

	UFUNCTION(BlueprintCallable, Category = "Metronome")
		int GetBeatsUntilNextMeasure();

	UFUNCTION(BlueprintCallable, Category = "Metronome")
		ENoteType CurrentNoteType();

	UFUNCTION(BlueprintCallable, Category = "Metronome")
		ENoteType NextNoteType();

	/*
		Internal call to trigger OnBeat.
		Broadcasts OnBeatEvent to Blueprints listeners
	*/
	virtual void OnBeat();

protected:
	// Called when the game starts
	virtual void BeginPlay() override;
	virtual void BeginDestroy() override;
};
```

13.4.6 NoteActor.h

```
UCLASS()
class ORB_MQP_API ANoteActor : public AActor
{
	GENERATED_BODY()

public:
	// Sets default values for this actor's properties
	ANoteActor();

protected:
	// Called when the game starts or when spawned
	virtual void BeginPlay() override;

public:
	// Called every frame
	virtual void Tick(float DeltaTime) override;



};

UENUM(BlueprintType)
enum class ETargetFace : uint8 {
	TOP                             UMETA(DisplayName = "Top"),
	LEFT                    UMETA(DisplayName = "Left"),
	RIGHT                   UMETA(DisplayName = "Right"),
	CYMBAL_LEFT             UMETA(DisplayName = "CymbalLeft"),
	CYMBAL_RIGHT    UMETA(DisplayName = "CymbalRight"),
	NONE                    UMETA(DisplayName = "None")
};

UENUM(BlueprintType)
enum class ENoteType : uint8 {
	FOURTH                  UMETA(DisplayName = "1/4"),
	EIGHTH                  UMETA(DisplayName = "1/8"),
	SIXTEENTH               UMETA(DisplayName = "1/16"),
	ACTION                  UMETA(DisplayName = "Action"),
	PRESCRIPTED_END UMETA(DisplayName = "Prescripted End"),
	EMPTY                   UMETA(DisplayName = "Empty")
};

USTRUCT(BlueprintType)
struct ORB_MQP_API FNoteStruct {

	GENERATED_BODY()

	UPROPERTY(BlueprintReadWrite)
		ETargetFace target;
	UPROPERTY(BlueprintReadWrite)
		ENoteType noteType;
	bool isSpreadNote;

	FNoteStruct() {
		target = ETargetFace::NONE;
		noteType = ENoteType::EMPTY;
		isSpreadNote = false;
	};
	FNoteStruct(ENoteType t) {
		target = static_cast<ETargetFace>(rand() % ((int) ETargetFace::NONE));
		noteType = t;
		isSpreadNote = false;
	};
	FNoteStruct(ETargetFace d, ENoteType t) {
		target = d;
		noteType = t;
		isSpreadNote = false;
```

```
	};
};

USTRUCT(BlueprintType)
struct ORB_MQP_API FNoteSet {

	GENERATED_BODY()

	UPROPERTY(BlueprintReadWrite, DisplayName = "Target A")
		FNoteStruct Target_A;
	UPROPERTY(BlueprintReadWrite, DisplayName = "Target B")
		FNoteStruct Target_B;

	FNoteSet() {
		Target_A = FNoteStruct();
		Target_B = FNoteStruct();
	}

	FNoteSet(FNoteStruct targetA) {
		Target_A = targetA;
		Target_B = FNoteStruct();
	}

	FNoteSet(FNoteStruct targetA, FNoteStruct targetB) {
		Target_A = targetA;
		Target_B = targetB;
	}

	void AddToSet(FNoteStruct &note, bool isTargetA = true) {
		if (isTargetA)
			Target_A = note;
		else
			Target_B = note;
	}
};
```

13.4.7 NoteGeneratorFunctionLibrary.h

```
UCLASS()
class ORB_MQP_API UNoteGeneratorFunctionLibrary : public UBlueprintFunctionLibrary
{
	GENERATED_BODY()

	UFUNCTION(BlueprintPure)
		static FNoteSet GetNextNoteSet();
	UFUNCTION(BlueprintPure)
		static FNoteSet PeekNextNoteSet();
	UFUNCTION(BlueprintCallable)
		static void RegenerateNotes();

	UFUNCTION(BlueprintPure)
		static bool IsOkToPop();
	UFUNCTION(BlueprintPure)
		static bool IsNextBeatAction();
	UFUNCTION(BlueprintCallable)
		static void SetDoubleNoteProbability(float Probability);
	UFUNCTION(BlueprintCallable)
		static float GetDoubleNoteProbability();
};
```

13.4.8 NoteGrammar.h

```
DECLARE_LOG_CATEGORY_EXTERN(NoteGrammarLog, Log, All);

enum NoteType {
	FOURTH,
	EIGHTH,
	SIXTEENTH
};

enum GrammarType {
	Test,
	MeasureA,
	MeasureB,
	Four,
	Three,
	Two,
	One,
	Fourth,
	Eighth,
	Sixteenth,
	Action,
	NONE
};

enum NoteDensity {
	LOW,
	MED,
	HIGH
};

/**
	NoteGrammarRule - Class containing info about a grammar rule
*/
class NoteGrammarRule
{
public:
	NoteGrammarRule() {};
	TArray<FString> Tokens;
	float Probability;
	TArray<int> Density;
};

/**
	NoteGrammarTerm - Class containing info about a grammar terminator
*/
class NoteGrammarTerm : public NoteGrammarRule
{
public:
	NoteType noteType;
	NoteGrammarTerm() {}
	NoteGrammarTerm(NoteType type) {
		noteType = type;
	}
};

/**
	FNoteGenTask - Task to run generation on separate thread
*/
class ORB_MQP_API FNoteGenTask : public FNonAbandonableTask
{
	friend class FAsyncTask<FNoteGenTask>;
public:
	FNoteGenTask(int count) :
		notesToGenerate(count)
	{}
```

```
        void SetNotesToGenerate(int count) {
                notesToGenerate = count;
        }

protected:
        int notesToGenerate;
        void DoWork();

        // This next section of code needs to be here.  Not important as to why.

        FORCEINLINE TStatId GetStatId() const
        {
                RETURN_QUICK_DECLARE_CYCLE_STAT(FNoteGenTask, STATGROUP_ThreadPoolAsyncTasks);
        }
};

/**
        UE4 WRAPPER for NoteGenerator
*/
UCLASS(BlueprintType)
class ORB_MQP_API UNoteGenerator : public UObject {

        GENERATED_BODY()

public:
        UNoteGenerator(const FObjectInitializer & ObjectInitializer) {}

};

/**
        FNoteGrammarParams - Additional parameters that affect note generation
*/
struct ORB_MQP_API FNoteGrammarParams {

public:
        bool shouldGenerate;
        NoteDensity noteDensity;
        bool useDoubleNotes;
        TArray<ETargetFace> *currentTargetPattern;
        FNoteGrammarParams() {};
};

/**
        NoteGrammarGenerator - Singleton that generates notes
*/
class NoteGrammarGenerator {

        friend class FNoteGenTask;

public:

        static NoteGrammarGenerator &getInstance() {
                static NoteGrammarGenerator instance;
                return instance;
        }

        const int MAX_NOTES_IN_QUEUE =   500;    // Maximum notes in queue at once
        const int MAX_LANES =            5;      // Maximum number of lanes

        TMap<GrammarType, TArray<NoteGrammarRule*>*> Grammars;   // Ruleset
        TArray<TArray<ETargetFace> *> TargetPatterns;
        bool mIsGraphBuilt;     // Doesn't get destroyed in editor when end play so check for this
        bool mSetForClear;
        FNoteGrammarParams mParams;      // Generator parameters

        bool BuildRuleGraph();           // Builds ruleset based on JSON file
        bool LoadNotesetFromFile(FString filename, TArray<FNoteSet> &notesResult);
        void AsyncGenerate(int count);
```

```
        void StopGenerate();
        FNoteSet PopNextNoteSet();       // Pops off the oldest noteset out of the queue
        FNoteSet PeekNextNoteSet() const;
        void SetPrescripted(TArray<FNoteSet> notes);
        void UpdateParameters(FNoteGrammarParams newParams);     // Updates parameters for when we
generate more notes
        void ClearQueue(bool shouldRegenerate = true);
        bool IsOkToPop();
        void SetDoubleNoteProbability(float newValue);
        float GetDoubleNoteProbability() const;

private:

        /* Tokens to parse thru ruleset */
        const FString PATTERNS_TOKEN = "#patterns";
        const FString NOTE_PATTERNS_TOKEN = "#notePatterns";
        const FString TARGET_PATTERNS_TOKEN = "#targetPatterns";
        const FString ORIGIN_TOKEN = "@origin";
        const FString M_ONE_TOKEN = "@measureOne";
        const FString M_TWO_TOKEN = "@measureTwo";
        const FString FOUR_TOKEN = "@four";
        const FString THREE_TOKEN = "@three";
        const FString TWO_TOKEN = "@two";
        const FString ONE_TOKEN = "@one";
        const FString ACTION_TOKEN = "@action";
        const FString FOURTH_TOKEN = "@fourth";
        const FString EIGHTH_TOKEN = "@eighth";
        const FString SIXTEENTH_TOKEN = "@sixteenth";
        const FString FOURTH_TERM_TOKEN = "*fourth";
        const FString EIGHTH_TERM_TOKEN = "*eighth";
        const FString SIXTEENTH_TERM_TOKEN = "*sixteenth";
        const FString FOURTH_REST_TOKEN = "*fourthRest";
        const FString EIGHTH_REST_TOKEN = "*eighthRest";
        const FString SIXTEENTH_REST_TOKEN = "*sixteenthRest";
        const FString SPREAD_THREE_TOKEN = "*spreadThree";
        const FString SPREAD_FOUR_TOKEN = "*spreadFour";
        const FString ACTION_TERM_TOKEN = "*action";


        const int NOTES_TO_GENERATE_WHEN_LOW = 5;        / How many notes to generate when we run low

        const float DOUBLE_NOTE_PROBABILITY = 0.15f;     // How often double notes show up
        const int MAX_NOTE_DENSITY = 28;                 // Measure 1: 16 (1/16th) notes
                                                         // + Measure 2: 12 (1/16th) notes
        const int NOTE_DENSITY_HIGH_THRESHOLD = 8;
        const int NOTE_DENSITY_LOW_THRESHOLD = 1;
        const float NOTE_DENSITY_HEURISTIC = 2.0;
        const float NOTE_PROBABILITY_HEURISTIC = 0.5;

        FString mJsonPath;                       // JSON File path
        bool mShouldGenerateMoreNotes;           // Should generator make more notes when running out
        TCircularQueue<FNoteSet> *mNoteQueue;    // Notes that get generated go here
        float mDoubleNoteProbability;
        bool mIsReadyToPop;

        FAsyncTask<FNoteGenTask> *genTask;


        NoteGrammarGenerator();
        NoteGrammarGenerator(NoteGrammarGenerator const&);       // Don't Implement
        void operator=(NoteGrammarGenerator const&);             // Don't implement

        void Generate(int count);                                // Builds the noteset for the queue

        void GenerateNoteSet();
        /* HELPER: Starts generating a note set based on given grammar rules */
        TArray<FNoteStruct*> * GenerateGrammar();
        /* HELPER: Recurses down the ruleset and creates the note structure and patterns */
```

```
        TArray<FNoteStruct*> * GenerateGrammarHelper(NoteGrammarRule *rule, int noteCount);
        TArray<FNoteStruct*> * GenerateSpread(TArray<ENoteType> possibleNotes, int spreadCount);
        /* HEURISTIC FUNCTION: Gets a random index from a ruleset based on probability and current note
density rules */
        int GetNoteIndex(GrammarType t, int noteCount) const;
        /* HELPER: Gets a random target for a double note */
        FNoteStruct GetDoubleNote(FNoteStruct firstNote) const;
        /* HELPER: Gets note object from a given token */
        FNoteStruct GetNoteFromTermToken(FString token) const;
        /* HELPER: Gets grammar type from a given token */
        GrammarType GetGrammarTypeFromToken(FString token) const;
        /* HELPER: Gets target face from a given token/string */
        ETargetFace GetTargetFaceFromToken(FString token) const;
        /* HELPER: Determines if token is a rest note */
        bool IsRestToken(FString token) const;
        /* HELPER: Checks that the set generated has valid measures */
        bool isValidSet(int expectedMeasures, TArray<FNoteStruct*> notes);
        /* HELPER: Gets a random neighboring face */
        ETargetFace GetRandomNeighborFace(ETargetFace currentFace);
        /* HELPER: Prints out the set in readable format */
        void PrintLinearSet(TArray<FNoteStruct*> set);
        FString FormatNoteStruct(FNoteStruct note) const;
};
```

13.4.9 PhaseChangeListenerComponent.h

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FPhaseChangeDelegate, int, NewPhase);

UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class ORB_MQP_API UPhaseChangeListenerComponent : public UActorComponent
{
	GENERATED_BODY()

public:
	// Sets default values for this component's properties
	UPhaseChangeListenerComponent();

	/* Blueprints uses this to listen for phase change from Phase System */
	UPROPERTY(BlueprintAssignable, Category = "Phase Change")
		FPhaseChangeDelegate OnPhaseStartEvent;
	/* Blueprints uses this to listen for phase change from Phase System */
	UPROPERTY(BlueprintAssignable, Category = "Phase Change")
		FPhaseChangeDelegate OnPhaseEndEvent;

	/* Tells PhaseSystem that the Actor holding this component is ready to change phases */
	UFUNCTION(BlueprintCallable)
		void ReadyForNextPhase();

	/*
	Internal call to trigger OnPhaseChange.
	Broadcasts OnPhaseChangeEvent to Blueprints listeners
	*/
	virtual void OnPhaseStart(int phaseToStart);
	virtual void OnPhaseEnd(int phaseToEnd);

protected:
	// Called when the game starts
	virtual void BeginPlay() override;
	virtual void BeginDestroy() override;
};
```

a. PhaseSystem.h

```
DECLARE_LOG_CATEGORY_EXTERN(PhaseSystemLog, Log, All);

USTRUCT(BlueprintType)
struct ORB_MQP_API FPhase {

	GENERATED_BODY()

public:
	FPhase() {
		dexTestCountThreshold = -1.0f;
		healthThreshold = -1.0f;
	}

	/* If set to true, Note Generator will make generated notes. If false, PhaseSystem will load the
provided ScriptedDexTest into NoteGenerator */
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
		bool useGeneratedDexTest;
	UPROPERTY(EditAnywhere, BlueprintReadWrite, meta=(DisplayName = "Pre-Generated file name"))
		FString fileName;
	/* Whether or not this phase will utilize chimes */
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
		bool useChime;
	/* If set to true, PhaseControllerComponent will check against the thresholds */
	UPROPERTY(EditAnywhere, BlueprintReadWrite)
		bool doesPhaseLoop;
	/* If phase advances by how many dex tests have passed by, set to non-negative. */
	UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ClampMin = "-1", ClampMax = "100.0", UIMin =
"0.0", UIMax = "100.0"))
		float dexTestCountThreshold;
	/* If phase advances by passing an HP threshold, set to non-negative. */
```

```
        UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ClampMin = "-1", ClampMax = "100.0", UIMin =
"0.0", UIMax = "100.0"))
                float healthThreshold;
        /* How frantic/intense this phase should be */
        UPROPERTY(EditAnywhere, BlueprintReadWrite, meta = (ClampMin = "1", ClampMax = "5", UIMin = '1',
UIMax = '5'))
                int phaseIntensity;
};

class UPhaseChangeListenerComponent; //forward declaration

/**
 *
 */
class ORB_MQP_API PhaseSystem
{
public:

        static PhaseSystem &getInstance() {
                static PhaseSystem instance;
                return instance;
        }

        bool LoadPhaseSequence(TArray<FPhase> sequence);
        bool UnloadPhaseSequence();
        int GetCurrentPhase() const;
        int GetPhaseCount() const;
        TArray<FPhase> GetCurrentPhaseSequence();

        bool RegisterListener(UPhaseChangeListenerComponent *comp);
        bool UnregisterListener(UPhaseChangeListenerComponent *comp);

        void StartPhaseSequence();
        void NextPhase();
        void ReadyForNextPhase(UPhaseChangeListenerComponent *comp);
        void ResetPhaseSequence();

private:
        PhaseSystem();
        PhaseSystem(PhaseSystem const&);
        void operator=(PhaseSystem const&);
        ~PhaseSystem();

        int mCurrentPhase;
        int mReadyCount;
        TMap<int, TArray<FNoteSet>*> mCustomNoteMaps;
        TMap<UPhaseChangeListenerComponent*, bool> mListenerReadyMap;
        TArray<UPhaseChangeListenerComponent*> mListeners;
        TArray<FPhase> mCurrentPhaseSequence;

        void internal_NextPhase(int nextPhase);
};
```

b. PhaseSystemFunctionLibrary.h

```
UCLASS()
class ORB_MQP_API UPhaseSystemFunctionLibrary : public UBlueprintFunctionLibrary
{
        GENERATED_BODY()
        /*
                Gets the current phase the game is in
        */
        UFUNCTION(BlueprintCallable)
                static int GetCurrentPhaseIndex();

        UFUNCTION(BlueprintCallable)
                static FPhase GetCurrentPhase();

        UFUNCTION(BlueprintCallable)
                static void StartPhaseSequence();
```

```
	UFUNCTION(BlueprintCallable)
		static void GoToNextPhase();

	UFUNCTION(BlueprintCallable)
		static int GetPhaseCount();

	UFUNCTION(BlueprintCallable)
		static bool IsLastPhase();

};
```