

# DOWNBEAT

DEVELOPMENT OF AN ADAPTIVE COMBAT SYSTEM  
BASED ON THE USE OF THE DOWNBEATS AND UPBEATS  
OF A MUSICAL PIECE.

---

**UJI** UNIVERSITAT  
JAUME I

**Pablo Lorente Martínez**

VJ1241 – Bachelor's Thesis

Degree in Video Game Design and Development

Advisor: Vicente Cholvi Juan

July 3, 2020



#####  
#####



# ABSTRACT

---

This document presents the Final Report of a Bachelor's Thesis in the Degree in Video Game Design and Development.

The project consists of the composition, production, implementation and analysis of a **dynamically changing musical piece** and the development of a playable demonstration of a unique, challenging and innovative **combat system** based on three main concepts: **rhythm**, **strategy** and **reactivity**.

This combat system is oriented to boss fights and proposes a symbiotic relationship between music, visuals, narrative and mechanics. Therefore, in addition to mainly dealing with aspects related to the combat demonstration, this document will also cover narrative aspects of *DownBeat* –the complete game to which this combat system would belong– with the intention of facilitating the understanding of both the musical and the non-musical narrative background.

# KEYWORDS

---

- Game developing.
- Combat system.
- Musical composition.
- Adaptive music.
- Artificial intelligence.

# INDEX

---

<b>1. INTRODUCTION</b>	<b>1</b>
1.1. Work motivation	1
1.2. Related subjects	1
1.3. Objectives	2
1.4. Tools	3
<b>2. PLANNING</b>	<b>4</b>
2.1. Tasks and expected duration	4
2.2. Dependencies and expected order	5
<b>3. DESIGN</b>	<b>6</b>
3.1. Synopsis	6
3.2. Narrative background	7
Involved Characters	7
Preceding events	7
3.3. Mechanics	9
Basic concepts	9
Attack	11
Block	11
Special Techniques	12
Cut	13
Movement	13
Statistics and values	14
3.4. User Interface	17
Player status indicators	17
Enemy status and numerical indicators	18
3.5. Battle against Garad	19
Enemy attacks	19
Enemy techniques	19
Phases and behaviour	20

<b>4. SOUNDTRACK</b>	<b>21</b>
4.1. Introduction	21
4.2. Inspiration	21
4.3. Instrumentation	22
4.4. Leitmotifs	23
Fate	23
The Hero	24
The Beast	25
Home	26
Ursa Major	26
4.5. Sections of the soundtrack	27
Introduction: Prophecy	27
Interlude I: Steel Tempest	27
Phase I: Hunt	28
Interlude II: Mediation	28
Phase II: Determination	29
Interlude III: What Could Have Been	29
Phase III: A Terrible Fate	30
Interlude IV: An Empty Vessel	30
Phase IV: Conflagration	31
Coda: And Then Fate Rewinds	31
<b>5. WORK DEVELOPMENT</b>	<b>32</b>
5.1. Rhythm system	32
Rhythm bar	32
Circular design	33
Implementation	33
Auto-synchronization system	34
Scheduling system	34
5.2. Combat system	35
Success rate	35
Movement	35
Attack	36
Health	36
Ammunition	37
Block	37
Special techniques	38
Cut and offbeat	38
5.3. Enemy behaviour	39
Artificial intelligence	39
Counter	40
Attacks and techniques	40

5.4. User interface	42
Dialog box	42
Numerical indicators	43
5.5. Animations and visual effects	44
Combat animations	44
Visual effects	46
Camera behaviour	47
5.6. Dynamic soundtrack	48
Learning stage	48
Soundtrack adaptation	48
Implementation	49
Sound effects	50
5.7. Testing and debug	51
<b>6. RESULTS</b>	<b>52</b>
6.1. Time balance and deviations	52
6.2. Completion of the objectives	56
6.3. Links	57
<b>7. CONCLUSIONS</b>	<b>57</b>
<b>BIBLIOGRAPHY</b>	<b>58</b>



# 1. INTRODUCTION

---

## 1.1. Work motivation

Interaction between the user and the work is the most distinctive feature of the video game as an form of artistic expression. Therefore, designing and implementing a unique, surprising and engaging form of interaction is an essential step for a video game to achieve excellence. Recent examples –such as *Sekiro: Shadows Die Twice* [1], *Doom Eternal* [2], *Undertale* [3] or *Super Hexagon* [4]– achieve this effect through carefully designed mechanics or giving a surprising and effective twist to the mechanical tropes of their genres.

As forms of artistic expression, video games usually aim to transmit certain emotions, sensations or stories to the player through the narrative. However, is frequent to attempt to integrate the narrative into the game by conceiving it as an addition to the mechanics instead of conceiving both things –the mechanics and the narrative– as the same whole. As a consequence, praxis brought from other art forms are used to overcome this problem in a way that is far from optimal and that greatly misuses the video games' potential as interactive audiovisual artworks: long cinematic sequences or texts in which most of the narration of the story is encapsulated separately from the playable part of the game.

This project was born as a proposal to unify both of this aspects towards the development of the video game *DownBeat*, a personal project in which I intend to address interesting and unconventional topics and concerns through a very emotional story displayed by charismatic characters, a rich original leitmotiv-based soundtrack, a simple and clean graphic style and a unique, dynamic and exciting combat system in which music, narrative and mechanics feed each other in a symbiotic relationship.

## 1.2. Related subjects

- [VJ1222](#) - Video Game Conceptual Design.
- [VJ1236](#) - Sound Production Techniques.
- [VJ1227](#) - Game Engines.
- [VJ1231](#) - Artificial Intelligence.
- [VJ1208](#) - Programming II (Computing).

## 1.3. Objectives

- To develop a **unique, dynamic, enjoyable and exciting combat system** that may explore the fight between the player and the enemy as a dance in which every opponent have to choose –in real time and following the beat of the music– the most appropriate techniques to react to the enemy movements.
- To develop this combat system in order to allow **victory at first try** with only a basic knowledge of the moves, mechanical skill and reflexes, but also to be challenging and deep enough to allow **improvement and perfecting** through practice and memorization of patterns, as happens in *Undertale*, *Sekiro: Shadows Die Twice* or in the dance itself.
- To **coordinate the mechanics with the music, the visuals and the rest of the narrative section**, in order to transmit to the player the rhythm, the thrill and the emotional singularities of every phase of the battle.
- To compose and produce an **original soundtrack** that, through programming and middleware [5], **dynamically adapts** to the various phases of the battle and changes depending on aspects as status conditions, remaining health, etc.
- To develop **varied patterns of enemy attacks** that may be consistent with the musical development of the soundtrack in each phase, and an **artificial intelligence** that allows the enemy to assess which patterns to use and to strategically act in response of the player when this patterns are interrupted.

## 1.4. Tools

The game engine and main tool for the development of the demo is [Unity3D](#). Third-party libraries and assets have been used in order to cover those aspects that are necessary for the the playable demo but that do not fall within the objectives of the project, such as character models, environment, animations, etc. For example, [iTween](#) library is used for the most basic animations such as simple translations, rotations and scaling, while most of the models and complex animations are from [Mixamo](#). The rest of the 3D assets have been created in 3DS Max.

The programming tasks have been performed using [C#](#) and [Visual Studio](#) IDE.

[GitHub](#) has been used for version control and backup managing.

[Toggl](#) has been used for keeping track of the hours spent in each task.

The soundtrack has been composed and converted to audio files in [Sibelius](#).

The mix and production of the final track have been performed using [Audacity](#).

The middleware used to implement the dynamic soundtrack is [Wwise](#).

This report and the previous documents have been written using [Google Docs](#). In the same way, the presentation has been composed using [Google Slides](#).

Diagrams have been made using [Draw.io](#).

Graphic design and 2D art tasks have been performed using [Adobe Photoshop CC 2019](#).

The playable demo is compiled for [Windows 10](#) and the input system is configured to play using a [Nintendo Switch Pro Controller](#).

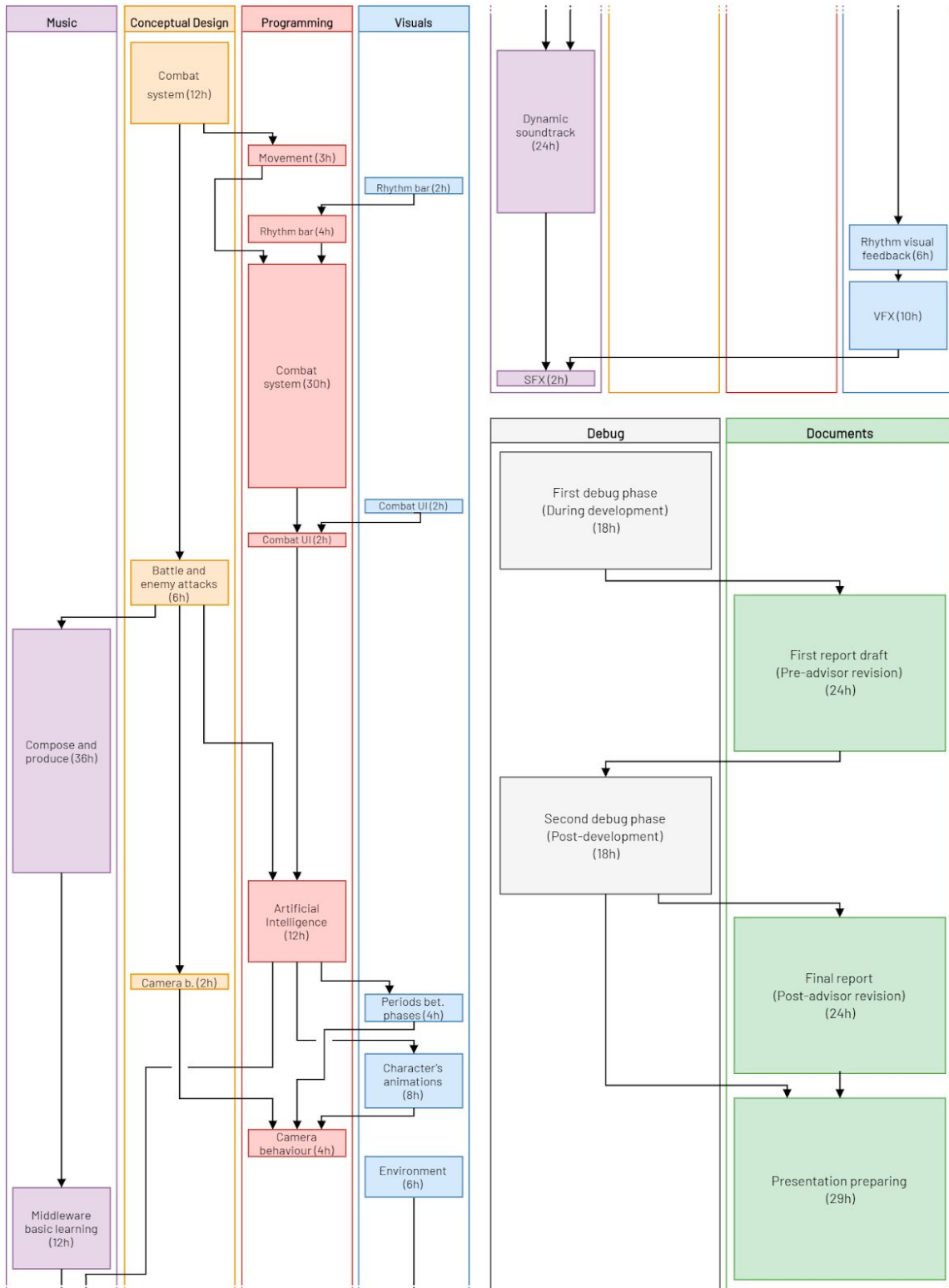
## 2. PLANNING

---

### 2.1. Tasks and expected duration

To design and script the whole battle, including the different enemy attack patterns, the different phases of the battle and the narrative and emotional singularities of each phase.	6h
To compose and produce a musical piece adapted for its use in a battle with different phases of undefined duration.	36h
To learn the basics of a sound middleware like Wwise or Fmod.	12h
To prepare a system for dynamically alter the music depending of the current phase of the battle and various events and variables.	24h
To design and program a rhythm bar system that keeps synchronized with the music and the visuals anytime.	6h
To program the movement of the player on a circular grid centered on the enemy.	3h
To design a combat system structured over the idea of acting on downbeats or upbeats and that presents, at any moment, different interesting options to counter enemy moves.	12h
To program this combat system, with values for damage, block and success chance that depend on the player's accuracy when pressing and releasing the buttons following the beat.	30h
To program the artificial intelligence for the enemy and their attack patterns.	12h
To design and program the combat UI.	4h
To implement the transitions between phases, including programming, UI and animations.	4h
To design and program the camera behaviour.	6h
To search and choose the graphic assets for the scene, design it and set it up.	6h
To develop visual feedback on the scene that helps the player to visualize the rhythm.	6h
To search, choose and implement the characters models and animations.	8h
To design and implement the visual effects and the sound effects.	12h
To test and fix bugs and design errors.	36h
To write the project's report.	48h
To prepare the presentation.	29h
<b>Total expected hours:</b>	<b>300h</b>

## 2.2. Dependencies and expected order



# 3. DESIGN

---

An essential aspect to understand the project is to acknowledge what role would each of its parts play in a complete game and what would be the relationships between them and the rest of the parts of the game. For this reason, this section offers, in addition to information about the design of the combat system and the musical section, a synopsis of *DownBeat* —the complete game to which they would belong— and a brief introduction to the narrative background of the two characters involved in the battle.

## 3.1. Sinopsis

It's getting dark. The protagonist sleeps leaning against the window of an almost empty bus on her way home —or, at least, what was once her home. Four years after her departure, she returns to the small town that saw her grow after finishing her stay at the University. At the bus stop, her grandmother and her two best childhood friends are waiting for her. Despite being happy to meet them again, she cannot ignore the feeling in his stomach that tells her that, after so long away, nothing can be like before.

Her return home will coincide with a great stir in the town, with deep folkloric and religious roots, where stories and prophecies have a pronounced weight in the lives of the inhabitants. A series of strange events will increase the strength to an ancient legend that tells of the arrival of a beast that will bring ruin and death to the land that nourished its nascency. The disappearance of a child will be the last straw and will spread panic among the inhabitants. Determined to find the little one and put an end to the misfortunes that torment the town, several young people from the town, including the protagonist, enter the forest aiming to end the life of the beast.

Nevertheless, the forest trails are wayward and wind their way through a sinister limbo where the barriers between the world of the living and the the dead vanished long ago.

*DownBeat* presents a story of magical realism that, using the great assets of the medium, involves the player in a fascinating and fun world, full of charismatic and familiar characters, overwhelmed by their own circumstances. *DownBeat* explores day to day topics like interpersonal relationships, insecurities, guilt, family bonds, and loss, but it also explores larger themes like life after death, ideals, or how beliefs and expectations sculpt the destiny of people.

## 3.2. Narrative background

### Involved Characters

**Protagonist:** The player decides her name at the beginning of the game and will be moderately free for shaping her behavior. She has just finished her stay in the University and returns home, but she no longer feels that she belongs there. She is full of insecurities and is afraid of not meeting the expectations that people have of her. She often boycotts herself unconsciously seeking acceptance or compassion.

**Grandma:** Fun and tired-minded, perhaps because of her old age or perhaps because she always liked the world that tales and old stories proposed to her more than the world that the war left her. Behind her apparent nonsense there is a lot of wisdom, and the initial rejection of her teachings by the protagonist will give way, throughout the game, to acceptance and reconciliation with others and with herself.

**The Beast:** A mysterious entity that stars in an ancient legend. Little is known about it, but it seems to be the cause of the strange events that flood the inhabitants of the town with fear.

**Garad:** The oldest of the last generation of the Hunter's bloodline, a respected lineage of defenders of the village, ruled by the strict values of loyalty and honor. It is said that several of his ancestors perished before The Beast when trying to hunt it down in his previous appearances. According to family history, his destiny as the firstborn is to stop The Beast by giving his life in return. For this reason, like his predecessors, he has grown keeping always in mind that, when the time comes, he will have to die to rid his people from greater evil. Reserved and determined, this young man assumes his role as defender of the village as soon as the return of The Beast becomes more evident, with the disappearance of his little sister.

### Preceding events

During the climax of the first narrative act, at a critical moment during the rescue of the missing child, Garad reveals his weaknesses and insecurities, leading to a scene where the ties between him and the protagonist strengthen. At the end of this first act, the protagonist and the hero return to the village with the child, but without a trace of The Beast.

Before returning to the hunt, they decide to take a rest night and leave at dawn. That night, Garad, the protagonist and her grandmother gather around the bonfire to talk. At one point in the conversation, when addressing her granddaughter, the grandmother asks her for her name. The protagonist attributes it to old age, and a possible neurodegenerative disease that has been glimpsing for a long time, and becoming increasingly more evident.

Grandma doesn't seem satisfied when the protagonist answers the name that the player decided.

—So, what's your real name, then? Did you have a name before waking up in that bus?

Suddenly, doubt begins to invade the mind of the protagonist. Grandma continues:

—Everyone in the village had our names, our lives, our history and our traditions before you arrived. My granddaughter also had a name. But your name... My daughter did not choose that name of yours. No, no... Everyone in the village seems comfortable calling you that, but... [Name of the protagonist]... That's not my granddaughter's name.

The conversation flows, and Grandma, who seems to be talking nonsense, looks more lucid than ever. Can the protagonist remember the name of her parents? No, since the player has not been given that information at any time. How is it possible that the protagonist does not remember even the name of her own parents?

—I know you do not belong to this world —continues Grandma—, I know that you are traversing by and you need this body to exist in our world. Don't be afraid, it's not a bad thing. We are all like this... just souls that occupy a body in order to exist in this world until our time comes. Still, we all keep memories of our time in this world: A name, a family... You are different. And it worries me.

Confused, the protagonist keeps her gaze fixed on the fire. A torrent of doubts prevents her from articulating a word. The crackling of the fire and the calm song of the crickets is the only thing that dares to break the silence. An idea emerges into her mind, and let she lets her heart speak for her:

—“The arrival of The Beast shall bring ruin and death to the land that nourished its nascency”, that's what the legend says.

—“And they who have the Hunter's blood coursing through their veins shall cease the calamity...”  
—Garad's voice becomes weak and trembling— “...by sacrificing their own life”.

—“And then, fate rewinds” —adds Grandma—. It has always been like this. No one remembers how it started, but this is the terrible fate that the village is doomed to repeat. Until the end of time.

The protagonist breaks down in tears.

—But Grandma, I... I don't... —she can't finish the sentence.

The protagonist's crying fills the calm atmosphere of the night. Soon, a metallic jingle catches the protagonist's attention: Garad picks up his lantern, his halberd and his pike, and leaves this latter on the protagonist's lap.

—Tomorrow at dawn at the entrance to the forest, we will finish this —says Garad, with his voice trembling like never before—. Is the only way. At least we'll do it together.

Garad fakes a smile, but several tears sparkle in the air as he turns to pass through the gate of Grandma's neglected garden for the last time. The next day, waiting for Garad at the agreed site, only his pike is found. The protagonist has entered the forest to hunt down The Beast and prove her innocence. A wayward soul that refuses to accept the destiny that has been imposed on it.

With a lump in his throat, Garad enters the forest to hunt her down. When their destinies cross again, weeks later, the protagonist will have discovered important clues about the true nature of The Beast and the curse of the village. However, Garad's blind obstinacy to fulfill his destiny and protect his loved ones is stronger than anything the protagonist can say. Bound by a terrible fate, they will have to fight one last battle, but, this time, against each other.



## 3.3. Mechanics

### Basic concepts

**Downbeats and upbeats:** Both the combat and the soundtrack are structured around two musical concepts, **downbeats** and **upbeats**. The music, despite having more complex secondary rhythms, has a constant basic rhythm that consists of the alternation between strong and low beats, such as the kick drum (beats), and weak and sharp beats, such as the snare (upbeats). Some movements, such as **attacks**, can only be performed on **downbeats**, while other movements, such as **special techniques**, can only be performed on **upbeats**. For visual clarity, all the actions and UI elements related to downbeats are blue, while the ones related to upbeats are orange.

**Rhythm circle:** The rhythm circle is a tool that tells the player when downbeats and upbeats happen in a very visual way. This circle is located at the bottom of the screen, between the **health bar** and the **ammunition indicator**. The circle grows and shrinks to the rhythm. When the circle reaches its minimum size, it glows blue –indicating the perfect moment for performing an action on downbeat. When the circle reaches its maximum size, it glows orange –indicating the perfect moment to perform an action on upbeat (See Figure 1).

**Success rate:** The precision with which a command is used to the rhythm of the music determines its **success rate**. The success rate not only determines if the activity **fails or not**, but also determines **numerical values** such as the damage that an attack can deal or the amount of damage that a block can protect from, for example. Success rate is shown as a percentage, with 0% being the lowest achievable value, and 100% being the highest.

**Combat grid:** Combats take place on a grid constituted by three concentric circles divided by six radii, resulting in **three rings** (inner, middle and outer) **made up of six squares each**. The enemy is always placed in the center of the grid, and the player is initially placed in one of the squares of the inner ring. The squares light up in advance indicating danger when an enemy attack or technique is going to inflict damage on those squares.

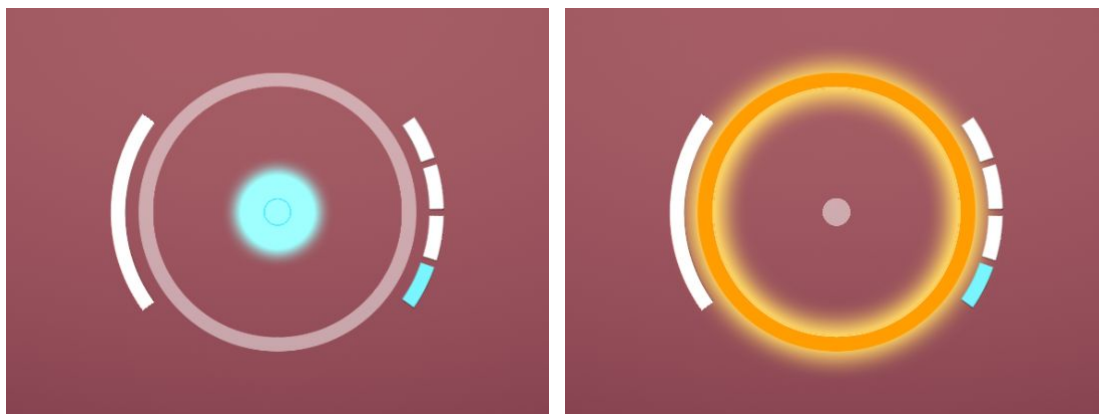


Figure 1: The rhythm circle, indicating a downbeat and an upbeat, respectively.

**Offbeat:** Offbeat is an altered status effect that causes to be **unable to perform any action until achieving a perfect success rate**. While the player is afflicted with the offbeat status effect, both the music and the visual representation of the rhythm fade, increasing the difficulty to achieve a perfect success rate (See Figure 2).

**Phases:** The combat is structured in **phases**. In each phase, the enemy can use various series of attacks, blocks, techniques and cuts to try to reduce the player's life to 0. Likewise, the player can use the mechanics described below in order to survive and reduce the life of the enemy. The enemy's behavior and attack patterns can vary in each phase. During every phase, action is rampant and uninterrupted. Between phases, the action momentarily ceases to give the player time to mentally prepare for the next barrage of enemy moves. In this period of rest between phases, an increasing tension is built through small capsules of written narration, music and different camera shots.

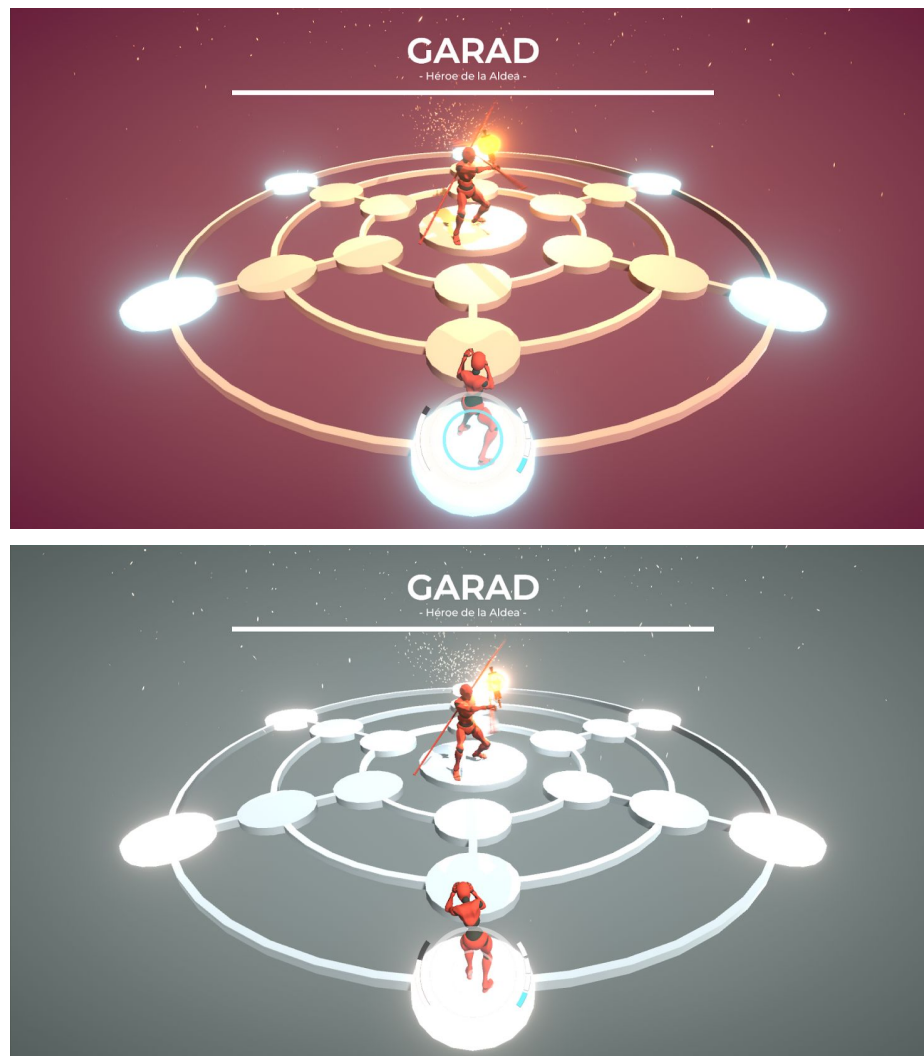


Figure 2: Visual comparison between not being afflicted (up) and being afflicted (down) by the offbeat status effect.

## Attack

Pressing and holding the **A Button**<sup>1</sup> on a **downbeat** starts loading an attack. By releasing the **A Button** after one bar —on the next downbeat—, the attack is performed. The success rate achieved when pressing the button determines whether the attack hits or fails, while the success rate achieved when releasing the button determines the amount of damage to be inflicted. In addition, if the success rate is 100%, the attack is **critical**. Critical attacks deals **150% of the attack damage**.



There are two types of attacks: **melee** and **ranged**. Attacking from the inner ring results in a **melee attack**, while attacking from any other point of the grid results in a **ranged attack**.

**Ranged attacks** require **ammunition**. Each ranged attack consumes a bullet of a cumulative maximum of four bullets. The fourth bullet in each charge is **powered**: by consuming this bullet, the player can perform an attack that deals double damage and heals a percentage of their maximum health.

**Melee attacks** do not require ammunition, but are more risky as they require staying close to the enemy. Critical melee attacks reloads a bullet into the chamber. In this way, the game encourages the player to alternate between attacking in a safer way —from afar, facilitating the evasion of enemy attacks and recovering health through powered attacks— and attacking more aggressively —being more exposed to enemy attacks, but recovering ammunition.

## Block

Pressing and holding the **B Button** on **downbeat** or on **upbeat** —with an appropriate success rate— puts the player in a defensive stance. Releasing the **B Button when the enemy attacks** blocks part of the damage. The blocked damage depends on the success rate when releasing the button. In addition, if the success rate is 100%, the enemy attack is completely **deflected**, applying **offbeat** to the enemy —this perfect block is also known as **parry**.



Moving when an attack is going to be blocked results in a **dodge**. Dodges avoid total damage but cannot apply offbeat to the enemy. Consequently, the game encourages the player to choose between dodging and blocking, depending on whether they prefer to play safe or to opt for dangerous option with a much more profitable risk-reward ratio.

Both **melee** and **ranged** attacks can be blocked, but special techniques cannot be blocked. Receiving a **cut** while blocking applies **offbeat** to the blocker.

---

<sup>1</sup>: Button icons retrieved from [6].

## Special Techniques

Special techniques are powerful but risky moves that are intimately related to **ammunition** and **reloading**. A special technique is loaded by **pressing and holding the Y Button on upbeat**, and is performed by **releasing the Y Button on the next upbeat**. The success rate when pressing the button determines if the technique is executed satisfactorily or if, on the contrary, it fails.

There are three types of special techniques:

**Reload:** It's the most basic technique and can be executed **standing still**. When performing this technique, the player reloads a number of bullets that depends on the success rate when releasing the button. A perfect success rate guarantees a full magazine and, therefore, at least one powered bullet.



**Pirouette:** Can be executed by **moving forward while in the inner ring**. It consists of jumping over the enemy's head to backflip and land in the outer circle. The dealt damage depends on the success rate when releasing the button. With a perfect success rate, a powered bullet is inserted into the chamber if a slot is available.



**Flash:** Can be executed by **moving backwards while in the outer ring**. It consists of charging energy to perform a thrust so fast that the player instantly pierces through the enemy to position right behind their back. The damage caused depends on the success rate when releasing the button. With a perfect success rate, a powered bullet is inserted into the chamber if a slot is available.



Definitely, the special techniques are more powerful than attacks since they cannot be blocked and generally give better ammunition rewards. Nevertheless, techniques are a much riskier option than attacks, as **receiving an attack or a cut while charging a special technique applies offbeat to the user**. In addition, special techniques are a less efficient source of damage than attacks, since, unlike attacks, a perfect success rate does not result in more damage, but in a better reload. For this reason, the special techniques are ideal to alternate between rounds of attacks in order to ensure enough ammunition to attack safely from the outer and middle rings, achieving greater amounts of damage and healing thanks to the powered bullets.

It should be noted that special techniques are not only an efficient source of damage and ammunition, but are also a valid tool for **repositioning**. For example, if the player is attacking from the outer ring and runs out of ammunition, they can use the **Flash** technique to inflict damage on the enemy, load a powered bullet into the chamber, and also reposition themselves in the inner ring, where they can recover ammunition using melee attacks. Likewise, if the player has a full chamber, they can use the **Pirouette** technique to move directly to the outer ring, potentially increasing their damage and healing until they run out of ammunition again.

## Cut

The **cut** is an extremely dangerous tool but of great utility and simplicity. Unlike the rest of the movements, **cuts don't require to be loaded**. Instead, the player can execute a cut instantly by **pressing the X button**. Cuts can only be successful if a success rate of 100% is achieved. Additionally, a cut is considered to fail if is used against an enemy that is not blocking or charging a special technique. If the cut is successful, it interrupts the action that the enemy is performing and applies **offbeat** to them. If the cut fails, that is, if the percentage of success is not perfect or if the enemy is not blocking or loading a special technique, **offbeat** is applied to the user of the movement.



Therefore, all three types of movements —attacks, block and special techniques— form a “Rock, paper or scissors-like” [7] triangular scheme of advantage and disadvantage relationships: Blocks are effective against attacks but useless against special techniques, attacks apply offbeat to combatants who are charging a special technique, but is useless against blocking combatants, and the special techniques are effective against blocks but lose against attacks.

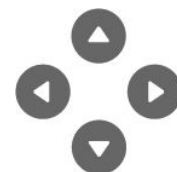
The cut is introduced as a fourth element that gives an additional depth layer to this triangle and that can reverse some disadvantageous situations if it is used correctly, but that in return can bring very negative results to the user if it is used incorrectly.

## Movement

The player can **move** using the **left stick** or the **directional pad**. It can be performed **both on downbeat and upbeat**. As happens with cuts, movement does not require loading time, it is executed instantly when moving the joystick. There are four allowed directions of movement — left, right, forward, and backward — that allow the player to move to each of the adjacent squares — left adjacent square, right adjacent square, adjacent square of the immediately inner circle, and adjacent square of the immediately outer circle, respectively.



The success rate determines the probability of successfully making the move, that is, having a success rate higher than 90% guarantees the player to move in the desired direction, but if the success rate is less than 50% means that the movement will fail. Between those two values, the success chance is equal to the success rate divided by 1.2 —83.33% of the success rate. This conditions are much more exigent than those of the other actions, that guarantee to perform the action with any success rate higher or equal than 50%, and, with lesser rates, the success chance is equal to the totality of the success rate.



Due to these peculiarities, movement is the most suitable way to move quickly on the combat grid, but it is a worse defensive tool than dodging (although it does not require loading time) due to a higher probability of failure. This means that there is a higher chance of receiving the totality of the enemy damage, that is, a devastating blow from which it can be difficult to recover.

## Statistics and values

PLAYER STATISTICS		
Statistic	Value	Description
Maximum health	100	Maximum accumulable health. Starting health.
Damage	20	Base damage value.
Final increase	+1	Damage increment added before every enemy attack during the final phase of the battle.
Maximum blockable ratio	80%	Maximum percentage of damage that can be blocked with an imperfect success rate.
Success rate threshold	+6%	When calculating success rate, the base value is increased by 106%.

ENEMY STATISTICS		
Statistic	Value	Description
Maximum health	600	Maximum accumulable health. Starting health.
Damage	20	Base damage value.
Tempest damage	30	Base damage value during the first 16 seconds of the battle.
Final increase	+2	Damage increment added before every attack during the final phase of the battle.
Maximum blockable ratio	80%	Maximum percentage of damage that can be blocked with an imperfect success rate.
Success rate threshold	+5%	When calculating success rate, the base value is increased by 106%.
Randomness	0.1	Maximum error time when executing an action other than a cut (in seconds).
Cut fix ratio	30%	When executing a cut, the maximum error time is 30% of the general maximum error time (0.033 seconds).

PLAYER ACTIONS		
Action	Success rate (SR)	Effect <sup>2</sup>
Movement	>90%	100% chance of success.
	50%~90%	$0.83 \cdot [\text{SR}]$ % chance of success.
	<50%	0% chance of success.
Dodge, loading any action	>50%	$[\text{SR}]$ % chance of success.
	<50%	0% chance of success.
Melee attack	100%	30 damage points, +1 bullet.
	<100%	$20 \cdot [\text{SR}]$ damage points.
Ranged attack (regular bullet)	100%	30 damage points.
	<100%	$20 \cdot [\text{SR}]$ damage points.
Ranged attack (powered bullet)	100%	60 damage points, +20 health points.
	<100%	$40 \cdot [\text{SR}]$ damage points, +20 health points.
Block	100%	100% damage blocked, apply offbeat to the enemy.
	<100%	$0.8 \cdot [\text{SR}]$ % damage blocked.
Flash, twirl	100%	20 damage points (unblockable), +1 powered bullet.
	<100%	$20 \cdot [\text{SR}]$ damage points (unblockable).
Reload	100%	+3 bullets, +1 powered bullet.
	90%~100%	+3 bullets.
	60%~90%	+2 bullets.
	30%~60%	+1 bullet.
	<30%	No effect
Cut	100%	Apply offbeat to the enemy if the conditions are met. If not, apply offbeat to the player.
	<100%	Apply offbeat to the player.

<sup>2</sup>: Damage values calculated from base damage (20) without taking into account the damage increase during the final phase of battle.

ENEMY ACTIONS		
Action	Success rate (SR)	Effect <sup>3</sup>
Attack	100%	30 damage points.
	<100%	20 · [SR] damage points.
Block	100%	100% damage blocked, apply offbeat to the enemy.
	<100%	0.8 · [SR]% damage blocked.
Offensive special techniques	100%	20 damage points (unblockable).
	<100%	20 · [SR] damage points (unblockable).
Prayer (healing special technique)	Any	+60 health points.
Cut	100%	Apply offbeat to the player if the conditions are met. If not, apply offbeat to the enemy.
	<100%	Apply offbeat to the enemy.

---

<sup>3</sup>: Damage values calculated from base damage (20) without taking into account the damage increase during the final phase of battle nor the increased damage during the beginning of the battle (30).



## 3.4. User Interface

### Player status indicators

Player status indicators are located on the bottom of the screen. Their function is to display the most important information to the player: their health, their ammunition and the rhythm.

**Rhythm circle:** See first subsection *Basic concepts* from section 3.3. *Mechanics* (Also see Figure 3).

**Player health bar:** Is located on the left side of the rhythm circle. Consist of three overlapping bars, each of which displays the player's remaining health, recently lost health, and maximum health respectively (See Figure 3).

**Bullet chamber:** Is located on the right side of the rhythm circle. Each one of the slots of the chamber is represented with an annular section that is displayed in white, blue or semi-transparent black, depending if the slot is filled with a regular bullet, a powered bullet or is empty, respectively (See Figure 3).

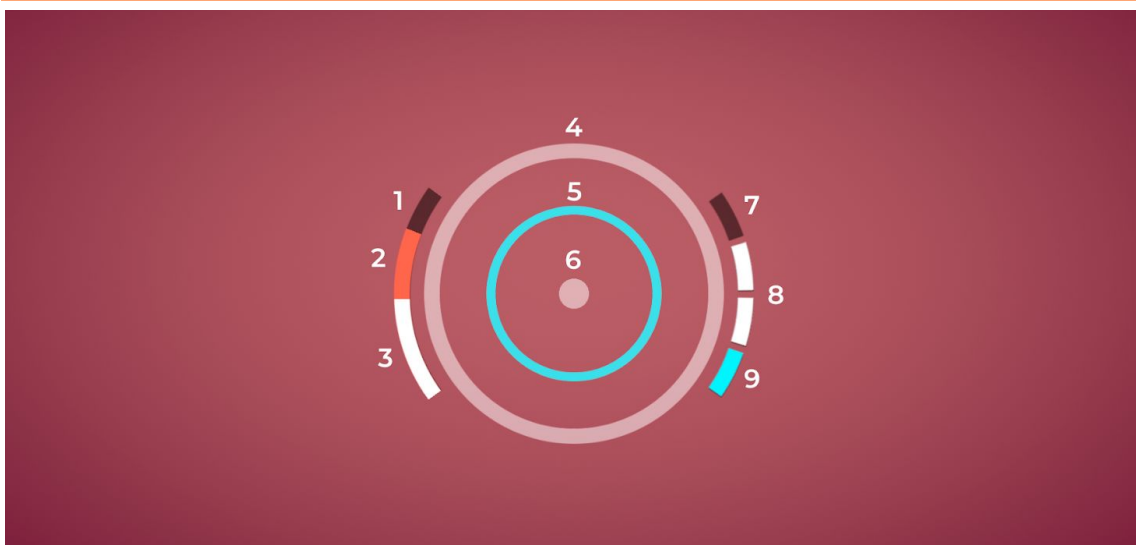


Figure 3: Player's health bar (1, 2 and 3), rhythm circle (4, 5 and 6) and bullet chamber (7, 8 and 9).

- |                         |   |   |
|-------------------------|---|---|
| 1. Maximum health       | 4. Maximum circle size (reached on upbeats)   | 7. First slot of the chamber (empty).                                   |
| 2. Recently lost health | 5. Rhythm circle, shrinking and blue because is transitioning from an upbeat to a downbeat. | 8. Second and third slots of the chamber (filled with regular bullets). |
| 3. Remaining health     | 6. Minimum circle size (reached on downbeats)   | 9. Fourth slot of the chamber (filled with a powered bullet).           |

## Enemy status and numerical indicators

This indicators are also important for the battle, but in a lesser way. The main difference between this indicators and the seen above is that this ones are merely informative, while reacting properly to the ones above represents the main win condition for the player.

**Enemy health bar:** Is displayed on the top of the screen and indicates the enemy's remaining health, recently lost health, and maximum health, in a very similar way to the player health bar. (See Figure 4).

**Numerical indicators:** They are displayed next to the rhythm circle or next to the enemy's head, and show values such as the percentage of success achieved in an action, the damage caused, the damage blocked, etc. Aiming to not overwhelm player's attention, most relevant values are displayed more clearly and with larger characters, while the less relevant values are smaller and disappear faster. Numerical indicators are displayed in white if they indicate blocked damage, success rate or status effects; in red if they indicate received damage, in blue if they indicate dealt attack damage and in orange if they indicate dealt technique damage.



Figure 4: Enemy health bar and numerical indicators.

On the left, the enemy receives a critic attack. The damage received (120) is shown in blue (indicating it was dealt through an attack and not through a special technique) and succeeded by an exclamation mark (indicating it was a critic attack). The white part of the health bar indicates the remaining health, while the orange part indicates the recently lost health.

On the right, the player receives and partially blocks an enemy attack. Under the rhythm circle, success rate is displayed (99%). Both the blocked and unblocked amounts of damage are shown next to the health bar. The damage received (unblocked damage) is shown in red numbers and as a red part of the bar

**The dialog box:** is the interface element where narrative snippets of text are displayed during break periods between phases. It is located on the bottom of the screen, in the center of the lower matte —each one of the black bars that frames the screen between this cinematic periods. Except on rare occasions and despite its name, the dialog box does not show dialogues, but rather small observations about what the protagonist feels, thinks, or draws her attention at a specific moment of the combat.

In addition to these interface elements, the rhythm and the status effects are visually accentuated through post processing effects like chromatic aberration and color correction.

## 3.5. Battle against Garad

In order to keep the battle interesting and to provide the player an experience with an appropriate difficulty curve, the battle against Garad is composed of four different phases. In each phase, Garad may show a different behaviour and movement set. Although the behaviors shown in these phases mainly respond to functional questions, it is also intended to show, through the selection of attacks, special techniques and some other behavioral traits, the narrative peculiarities of each phase of the battle.

### Enemy attacks

For visual clarity, attacks are represented in blue. For this reason, the theme chosen for Garad's attacks is electricity. This choice is also related to the Tomoe [8], an element traditionally used in the emblems of Japanese families –from which Garad's design is inspired– and strongly linked with Raijin [9], the god of storms in Japanese mythology and the Shinto religion.

- **Thrust:** Garad charges a powerful electric thrust with his halberd on the player's current radius. After two beats, the thrust inflicts damage on the three selected squares.
- **Sweep:** Garad charges a powerful electric sweep on the player's current ring. After two beats, the sweep inflicts damage on the six selected squares.
- **Lightning Strike:** Garad casts a lightning on the player's current square. After two beats, the lightning strikes on this square, inflicting damage.
- **Storm:** Garad casts up to seven lightnings on the player's current square and random near squares. After two beats, the lightnings strike on the selected squares, inflicting damage.

### Enemy techniques

Special techniques are represented in orange, and don't follow any specific theme due to their occasional use and the imitative nature of the most important of them.

- **Prayer:** Garad prays during two beats. After that, he recovers a percentage of his maximum health.
- **Backhand Sweep:** Garad charges a powerful halberd sweep on the inner ring. After two beats, the sweep inflicts damage on the six inner squares.
- **Replication:** Garad can charge this technique while charging an attack. This technique replicates the same or other attack with one beat of delay. Thanks to this technique, Garad can execute an attack-technique combo or an attack-technique-attack combo.

## Phases and behaviour

Unlike other enemies, Garad begins this fight by cornering the protagonist and executing an excessive barrage of his most deadly attacks and techniques. This serves as a threat —Garad will show no mercy and will try to end the protagonist's life by any means. Then, the first phase begins.

- **First phase:** The Hero shows a cold and trained mind by defending flawlessly against the player attacks and techniques and only dealing precise and determined attacks when the player is off guard. During this phase, Garad will **counter** every player's action, that is, try to parry every player's attack and to cut every player's technique or block. He will only attack —with **thrusts** and **lightning strikes**— if the player is not executing any action.
- **Second phase:** During this phase, The Hero seems mildly affected by his feelings towards the protagonist, and decides to end the battle as soon as possible to stop the suffering of both. Consequently, Garad keeps attacking uninterruptedly, now adding slightly more desperate attacks and techniques —**sweeps** and **backhand sweeps**— and will only counter player actions occasionally.
- **Third phase:** The Hero's determination is unstoppable. During this phase, Garad will keep attacking whenever he can —now using **thrusts**, **sweeps**, and **storms**—, countering most of the player's movements and using **backhand sweeps** and **prayers** to damage the protagonist and to heal himself. If the player takes advantage of Garad's attempts to heal by countering his technique, Garad will **learn** and modify his behaviour, using the prayer less frequently except in times of extreme need.
- **Fourth phase:** This phase is similar to the barrage of attack and techniques that starts the battle. Consumed by despair and determined to fight to death, The Hero will not counter any of the player's movements nor attempt to heal himself. Instead, he will attack uninterruptedly chaining his most powerful attacks with his **replication** technique. During this phase, Garad will **gradually increase his damage** so much that his own attacks will damage him when the player deflects them, making impossible for the player to extend this phase over its maximum duration —if the player deflects Garad attacks, Garad will die; if not, the protagonist will die. If the player lands a lethal hit on Garad during this phase, he will endure it and receive no damage under ten remaining health points. Instead, **offbeat** will be applied to Garad.

At the end of the fourth phase, a final cutscene is shown. If the player has managed to reduce Garad's health enough, this cutscene will show Garad's death and display the victory screen. Otherwise, the cutscene will show Garad landing a final thrust on the protagonist's chest, and the defeat screen will be displayed.

Due to the limited duration of the previous phases, it is impossible for the player to deliver a lethal blow to Garad before the fourth phase.

# 4. SOUNDTRACK

---

## 4.1. Introduction

*Bound by a terrible fate* is the track that plays during the combat demonstration. The track tries to move away from the classic conceptions of "boss battle music" to more accurately reflect the nature of the fight: it is not about the main hero fighting against a horrifying demon or about the forces of good fighting against the forces of evil: it is about an utterly despairing fight to the death between two old friends who are helpless against their own circumstances. In this fight, the protagonist, exiled from the village and judged as The Beast, faces not only the hero of the village—who is willing to die in order to end her life—, but a forking fate: ¿Is the protagonist willing to save her life if she has to kill the hero and, consequently, confirm the suspicions about her identity and be forever rejected by the society?

## 4.2. Inspiration

*Bound by a terrible fate* is primarily inspired by *Undertale's* soundtrack [10][11][12] because of the similar nature of the fights: Enemies are not evil nor do they want to kill the protagonist—in fact, in most cases, they are their friends or they feel appreciation of some kind towards them—but they have to confront them because the protagonist's objective involve a danger to their world and wellness. Mostly, *Undertale* battles are melancholic encounters in which the protagonist face characters who are just doing what is right for them, aiming to keep alive the hopes and dreams of the rest of the Underworld inhabitants.

*Undertale's* soundtrack also inspires *Bound by a terrible fate* due to its strong leitmotivic nature [13]—of 101 tracks on the official soundtrack, *Game Score Fanfare* on YouTube [14] mentioned that only 12 were isolated, without a trace of any other music used in the game, meaning the remaining 83 are connected through leitmotifs.

In other ways, *Bound by a terrible fate* is also inspired by traditional Japanese music, chosen to represent Garad as a hero of a village with deep traditional and folkloric roots, educated in strong ideals and deeply guided by aspects such as honor and destiny.

It also extracts certain tropes of Spanish-Arabic music, especially from the processional marches. These inspirations are mainly functional, although they serve to reinforce the same traditional and folkloric aspects as traditional Japanese music.

## 4.3. Instrumentation

Each of the instruments in *Bound by a terrible fate* has been chosen either for functional reasons or for reasons of meaning, that is, to represent certain characters, ideas or other narrative elements. The complete instrumentation is made up of the following instruments.

- **Shakuhachi:** A high pitched traditional Japanese woodwind instrument. Represents the voice of the protagonist.
- **Trombone:** A medium-low pitched brass instrument. Represents Garad's voice.
- **Trumpets:** Medium-high pitched brass instruments. They represent the inhabitants of the village, and therefore they usually appear as multiple voices that complement each other to form chords, even when playing the main melody.
- **Piano:** An exceptionally versatile instrument with a very wide playing range. It works in the piece as a cohesive element and to emphasize both the lowest notes—which consolidate harmony—and the high notes that make the melody shine. The piano brings a very special dramatism to extremely low and extremely high notes.
- **Koto:** The national instrument of Japan. It brings a very characteristic classical-feudal japanese sound. It represents fate, and appears iterating the Fate leitmotiv during almost all the work, as an ostinato.
- **Unpitched percussion:** Composed of taiko drums, snare drum, bass drum, cymbals and drum set. Its main function is to clearly communicate the downbeats and upbeats to the player for reasons of gameplay, although, of course, they also play an essential role in constructing the emotional singularities of each of the sections.
- **Timbales:** Pitched percussion instruments. They fulfill a merely functional task, which is to support the bass line provided by the piano. By reinforcing these low notes with percussive beats, a noteworthy strength is given to the track.
- **Choir and tubular bells:** They bring a pronounced dramatic weight to the piece, since it's hard to not relate these instruments to death and funeral rites. They help enormously to build in the player the sensation of a turning point that only can bring despair and loss.

## 4.4. Leitmotifs

A leitmotiv or leitmotif is a short, constantly recurring musical phrase associated with a particular person, place, or idea. Leitmotifs should be clearly identified so as to retain their identity if modified on subsequent appearances, whether such modifications be in terms of rhythm, harmony, orchestration or accompaniment. Leitmotifs may also be combined with other leitmotifs to suggest a new dramatic condition or development [15].

Given the peculiar characteristics of the battle, leitmotifs are a very powerful tool for the dramatic development of the combat, appealing to the emotional ties the player has established with the characters through reminiscences of the music that accompanied past key events of the game, especially the happier ones.

In order to ease the comparisons between different fragments, the following leitmotifs are shown in key of C major or A minor –with no flats or sharps in their key signatures.

### Fate

The Fate leitmotiv represents the homonymous idea and, unlike the rest of the leitmotifs of the soundtrack, it usually appears as an accompaniment instead of as a melody. Nevertheless, this leitmotiv reaches its maximum protagonism in *Bound by a terrible fate*, where it appears as an ostinato that is incessantly maintained throughout the entire piece (See Figure 6).

In previous appearances of the leitmotiv –where it is used to represent an optimistic and friendly fate or to highlight the strong ties between fate and the village– the bracketed notes are kept (See Figure 5). These notes –first degree of the major scale– are responsible for the stability of the melody. Despite that, in this piece, these notes are removed to represent a hostile fate.



Figure 5: Comparison between the original leitmotiv (left) and the hostile version (right).



Figure 6: A bar of the koto ostinato based on this leitmotiv, doubling each note and adding harmonic support.

---

## The Hero

The Hero's leitmotiv (See Figure 7) represents the figure of the hero, who usually coincides with the character of Garad, although it also appears in other contexts denoting other characters' heroic actions –or especially unheroic actions, when the leitmotiv appears modified in a certain way– or symbolizing former heroes of the village. It consists of two sections:

The first one, composed by the first three notes, is the hook of the leitmotiv and uses long notes separated by wide intervals –an ascending perfect fifth and a descending perfect fourth descending on the I, V and II degrees of the scale– in order to achieve simplicity and easy recognition. The first jump, from I to V degree, quickly establishes the key, especially when the leitmotiv appears accompanied by its original harmony.

The second section is made up of the following five notes. The first two act as an ornament that anticipate the first note or chord of the important part of the section, that is, the last three notes. These last three notes form a descending arpeggiated perfect major chord –the I degree chord. The first of these notes –as well as the supporting notes that precede it– play a fundamental role in turning the nature of the leitmotiv. By leaning on the III degree –which is descendingly altered in minor keys– these notes are the ones that most clearly communicate to the listener if a major key is being used to maintain the heroic and triumphant tone of the leitmotiv or, on the contrary, a minor key is being used to communicate melancholy.



Figure 7: The Hero leitmotiv, in its most pure and simplified form.

---

The main way the hero's leitmotiv appears on the soundtrack is through the Hero Theme, which uses the main chords of the major scale –I, IV and V degrees– to highlight the heroic and victorious tone of the melody. Nevertheless, in *Bound by a terrible fate*, this theme appears exclusively in minor mode, becoming the Fallen Hero Theme (See Figure 8).

---

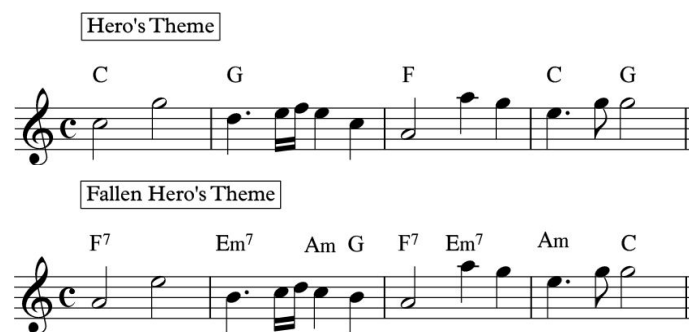


Figure 8: Comparison between the Hero's Theme in a major key (up) and the same theme in a minor key (down).

---



Throughout the entire piece, the Fallen Hero's Theme is intertwined and merged with others to build different melodies. Thus, this theme leads some of the most important melodies, such as the chorus melody or the melody that sounds during the first phase of the battle.

During the chorus, the Fallen Hero's Theme appears solemnly and ominously constructed, unlike his appearance during the first phase, in which its influenced by the Beast's Theme and adopts its harmony, also altering the development of the melody (See Figure 9).

---

**Fallen Hero's Theme (As it appears in *Bound by a terrible fate*)**

F<sup>7</sup>      Em<sup>7</sup>      Am      G      F<sup>7</sup>      Em<sup>7</sup>      Am      C

**Fallen Hero's Theme (With Beast's Theme harmonic progression)**

A      E<sup>7(sus4)</sup>      F      E<sup>7</sup>      A      E<sup>7(sus4)</sup>      F      E<sup>7</sup>

Figure 9: Comparison between the two main iterations of the Fallen Hero's Theme in *Bound by a terrible fate* –chorus melody (up) and first phase melody (down).

---

## The Beast

The Beast's leitmotiv is characterized by its descending chromatic triplet (See Figure 10). The objective when composing this leitmotiv was to achieve a sensation of sinister strangeness, which is achieved through elements that are strange to the piece both in key and in rhythm –accidental alterations and irregular groups. This sensation is reinforced in the Beast's Theme, in which a modified version of the Andalusian cadence [16] is used.

This cadence already provides a shade of exoticism and strangeness by varying the seventh degree of the scale during the chord sequence. Also, in the Beast's Theme, the second chord appears modified to add another layer of strangeness. Substituting the G chord for the E7sus4 chord –which maintains two notes of the previous chord, Am, and two notes of the G chord– the feeling of progress in the harmonic sequence is blurred.

---

**The Beast**

**Beast's Theme**

Am      E<sup>7(sus4)</sup>      F      E<sup>7</sup>

Figure 10: The Beast's leitmotiv (left) and its developing and harmonization in Beast's Theme (right).

---


## Home

The Home leitmotiv mainly represents the village and, sometimes, the idea of “feeling at home”. This leitmotiv is a melody in itself, and appears as such in those parts of the soundtrack associated to the village as a physical place, usually accompanied by a first type compound cadence (I-IV-VI), the most common sequence of major chords, which brings familiarity and joy to the melody.

In *Bound by a terrible fate*, this leitmotiv appears both rhythmically and harmonically distorted in order to adequate its tone to the rest of the piece (See Figure 11). Through this distortion, a new dramatic condition is communicated to the listener, who can easily recognize the leitmotiv –cheerful and lively on its past iterations during the soundtrack– surrounded by a much more melancholic accompaniment.


---

**Home**



The original Home melody is shown in 12/8 time. The notes are: C4 (quarter), D4 (quarter), E4 (quarter), F4 (quarter), G4 (quarter), A4 (quarter), B4 (quarter), C5 (quarter), B4 (quarter), A4 (quarter), G4 (quarter), F4 (quarter), E4 (quarter), D4 (quarter), C4 (quarter). The chords above are: C (under C4), F (under F4), G (under G4), C (under C5), F (under F4), G (under G4).

**Home (As it appears in *Bound by a terrible fate*)**



The distorted Home melody is shown in 6/8 time. The notes are: C4 (quarter), D4 (quarter), E4 (quarter), F4 (quarter), G4 (quarter), A4 (quarter), B4 (quarter), C5 (quarter), B4 (quarter), A4 (quarter), G4 (quarter), F4 (quarter), E4 (quarter), D4 (quarter), C4 (quarter). The chords above are: Am (under C4), F (under F4), Em (under E4), Am (under A4), C (under C5), G (under G4).

Figure 11: Comparison between the original Home melody and harmony (up) and the distorted form of this piece (down).

---

## Ursa Major

The Ursa Major leitmotiv represents the bonds between two characters. Like the Home leitmotiv, it's a melody in itself (See Figure 13), and the shade variations –as well as the representation of different characters and emotions– are achieved by changing the instrumentation and harmonic accompaniment. For example, in the Interlude II: Mediation, this leitmotiv is displayed as a conversation between the koto –which represents the protagonist– and the trombone –which represents Garad. Also, the melody is subtly modified to include the Beast leitmotiv.

---

**Ursa Major**



The Ursa Major leitmotiv is shown in 6/8 time. The notes are: C4 (quarter), D4 (quarter), E4 (quarter), F4 (quarter), G4 (quarter), A4 (quarter), B4 (quarter), C5 (quarter), B4 (quarter), A4 (quarter), G4 (quarter), F4 (quarter), E4 (quarter), D4 (quarter), C4 (quarter).

Figure 13: The Ursa Major leitmotiv.

---

## 4.5. Sections of the soundtrack

### Introduction: Prophecy

This introduction presents the ostinato that is maintained throughout the rest of the work, based on the Fate leitmotiv. It has a harmonic accompaniment performed by the piano, the choir and the tubular bells. As for percussion, there is a procession march rhythm that will later evolve, doubling its time, in the rhythm that accompanies the rest of the work.

This section aims to represent the prophecy that connects the destiny of Garad and the protagonist, intimately linked to death. For this reason, processional rhythms and instrumentation reminiscent of funeral rites have been chosen. This section is intended to be completely abstracted from the characters and their situation and to focus solely on prophecy, therefore the use of shakuhachi, trombone, and trumpets is discarded.

This section lasts 16 seconds and repeats until the player presses the button to start the fight. When this occurs, the music transitions to the closing bars of the intro, adding a short drum fill that begins the next section.

### Interlude I: Steel Tempest

This section lead off the battle and represents the first barrage of attacks dealt by Garad—a quick succession of outrageous strikes that catch both the player and the protagonist by surprise. From a narrative point of view, these attacks represent the fulfillment of the prophecy and how the terrible fate that haunted the characters has already loomed over them.

To represent this, this interlude is a direct variation of Introduction: Prophecy. The timid ostinato of Fate leitmotiv is now an incessant discharge of koto blows, and the voices of the villagers—trumpets— and their hero—trombone— have joined the voices of prophecy—choir— and interpret the same harmonic accompaniment.

For its part, the the processional march rhythm has doubled its speed, and has been merged with a drum set rhythm that, in addition to strongly emphasizing the downbeats and upbeats due to functional and mechanical reasons, presents a noisy subdivision on the cymbal ride, representing the steel tempest that gives its name to the section.

This section lasts 18 seconds and automatically transitions to the next.

## Phase I: Hunt

This section is built around the Beast's Theme and the Fallen Hero's Theme. It maintains the rhythmic section of the previous interlude, but the chorus—which is still interpreting the harmonic accompaniment— goes from playing long notes to playing very distant quavers that replicate the harmonic progression of the Beast's Theme. The piano starts in the first part of this section, interpreting a bass line based on The Beast leitmotiv, which is later alternately joined by the trumpets and the right hand of the piano.

The first phase of the fight is conceived as a hunt in which the hero, after having dealt a first barrage of attacks, is patiently defending himself waiting for the perfect moment to return to the charge and end the life of The Beast. For this reason, the trombone—Garad—is incorporated over the bass line based on The Beast leitmotiv, interpreting the Fallen Hero's Theme, but adapted to the harmony of the Beast's Theme and, consequently, modifying the development of its melody. Thus, it is intended to represent the Hero's blind obstinacy in hunting The Beast, which has led him to decide to kill his friend so as not to fail the inhabitants of the village and himself.

This section lasts 50 seconds and automatically transitions to the next.

## Interlude II: Mediation

The objective of this section is to represent the mediation between the protagonist—who is trying to convince Garad that she doesn't want to kill him and that there is no reason to fight—and Garad—who, convinced that the protagonist is the Beast, is determined to die if necessary in order to end her life.

This is represented by using the Ursa Major leitmotif presented as a conversation between the shakuhachi—the protagonist—and the trombone—the hero. The use of the Ursa Major leitmotif, which represents the ties between these two characters, reinforces the dramatism of the scene for the player, who will recognize the melody and associate it with happier moments of the characters.

This leitmotiv is subtly modified to include The Beast leitmotiv at the end of the first phrase, that is, the protagonist's phrase. This can be interpreted as the protagonist is explicitly trying to make Garad understand that she is not The Beast. In the last part of this section, the trumpets start to accompany the conversation between the shakuhachi and the trombone, representing the hopes of the people of the village inhabitants, that lead Garad to return to the fight with Phase II: Determination.

This section has three subsections of 16 seconds each. At the end of the first phase of combat, the first subsection begins. If the player does not press the button to continue the fight, the second subsection starts when the first subsection is over. The second subsection is looped until the player decides to advance. During both the first and second subsections, when the player presses the button to start the fight, the music transitions smoothly and seamlessly to the analog part of the third subsection, which will automatically link to the second phase.

## Phase II: Determination

The objective of this section is to represent Garad's decision to fulfill his destiny at any price. This section of the work is structured in a ternary way, that is, it has three parts of which the first one exposes the chorus, the second one exposes a different development of some of the previous musical ideas that appear throughout the piece, and the third re-exposes the chorus with some minor changes.

In the chorus, the melody, based on the Fallen Hero's Theme, is performed by trumpets and trombones, although the shakuhachi steals their prominence in the final sections of each phrase. Thus, an attempt is made to represent the determination of an entire people –trumpets– who sing in unison with their hero –trombone– in whom they trust all their hopes, while an allegedly innocent protagonist –shakuhachi– painfully laments to see a terrible fate looming over them all unstoppably –represented by the incessant ostinato of koto executing the leitmotif of fate.

The middle part begins with the trumpets playing the Hero's Theme, and the trombone trying to imitate it, but failing to reach the high notes –this represents Garad's fear of not meeting the expectations and failing to protect the village. Despite this, the trombone finally manages to join the trumpets to perform a more emphasized version of the Hero's Theme, this time in major mode, moving away from the concept of the fallen hero and enunciating his determination to meet his destiny as The Hero and to fulfill the hopes of his people. In addition, the piano plays the Home leitmotiv, to add emotion and meaning to the fragment.

Finally, the chorus is exposed again, this time harmonically accompanied by the choir, introduced in the previous part. This section lasts for 1 minute and 28 seconds, and automatically transitions to the next when finished.

## Interlude III: What Could Have Been

During this section, the processional rhythm of the snare drum, the harmonies of the choir and a waltz accompaniment of the piano serve as the basis for the meeting of the Home and Ursa Major leitmotifs, performed by instruments that are added throughout the three repetitions of the same 8 bar phrase. Through these two leitmotifs, it is intended to represent the characters' last thoughts about the moments lived together and how happy life could have been if they hadn't been involved in the prophecy against their will.

In the first repetition, the trumpets perform the Home leitmotif. In the second repetition, the shakuhachi and the trombone are added playing the Ursa Major leitmotiv. In the third repetition, a harmonic shakuhachi reinforcement is added to this leitmotiv. At the end of this third repetition, the drum kit and the piano –making use of the original version of the Fate leitmotiv as a link– give way to Phase III: A Terrible Fate.

This section ideally lasts 56 seconds. If after this time the player has not yet pressed the button to advance to the next phase, the last 16 seconds are looped until necessary. If, on the other hand, the player presses the button before the section ends, the music transitions seamlessly to its analog point of the last 16-second sentence, linking to the next section when it ends.

## Phase III: A Terrible Fate

This section depicts how each of the three parts involved in the conflict—the protagonist, Garad, and the villagers—face these final moments of indeterminacy about how the prophecy is going to be resolved. As the voices of the villagers unite as the Home leitmotiv, Garad proudly raises the Hero theme, making it clear that he will fight to death to protect his village. Meanwhile, the protagonist continues to use the Ursa Major leitmotiv to value her relationship with Garad, regret having to battle him and self-convince herself that the friend with whom she shared so many moments still lives behind the person who is trying to kill her.

This section reinforces the leitmotivic encounter dynamics of the previous section by maintaining the same melody in the trumpets—Home—and adding a trombone melody based on the Fallen Hero's Theme. In addition, the shakuhachi reinforces the sharpest parts of the trumpet melody.

As for the accompaniment, the koto ostinato—Fate leitmotiv—the drum rhythm, the timpani, the tubular bells and the piano bass exposed in Phase II are reintroduced.

The first sentence of 8 bars is repeated three times, adding new layers with respect to the Ursa Major leitmotif—interpreted by the shakuhachi from the second repetition—and to the accompaniment, increasing the complexity of the rhythm of the snare drum, for example. In total, this section lasts 48 seconds, and links to the next through a shakuhachi solo.

## Interlude IV: An Empty Vessel

Consumed by despair and determined to burn his last energies in ending the protagonist's life, there is no longer any trace of the Garad that the player knew at the beginning of the game. Nothing matters anymore: neither the village, nor the good times lived. It only matters to vanish the Beast.

To represent this, the accompaniment of Phase I: Hunt is retrieved. This accompaniment uses The Beast leitmotiv as a bassline and also includes the harmony of the theme of the Beast's Theme, performed by the choir. Along with the choir and piano, the funerary nature of this section is reinforced through the use of bells, timpani and the processional rhythm of the snare drum.

The musical objective of the second part of this section is to create in the player a feeling of discomfort and strangeness. To accomplish this, the piano interprets the Beast's leitmotiv, which, as explained in *The Beast* subsection of 4.4. *Leitmotifs*, seek this same sensation through the use of irregular groups, accidental alterations, and a harmonic progression that twists the Andalusian cadence, which already sounds exotic in itself. As if that were not enough, the melody of the beast is interpreted erratically, advancing, delaying or lengthening some notes, making them sound out of time and with irregular durations.

The first part of this section lasts 26 seconds, while the second part, which is repeated as many times as necessary, lasts 16 seconds.

## Phase IV: Conflagration

Without a drop of life in his eyes and only moved by despair, the empty vessel keeps fighting until his last breath.

This phase extends the musical ideas exposed in the previous section, adding the koto ostinato, the drum rhythm and reinforcing the Beast's Theme through the trumpets and the trombone. In addition, in the last 16 bars, the trombone retakes for the last time the Fallen Hero's Theme, which depicts that, even on the verge of death, the only thing that matters to him is to fulfill his destiny and protect his town.

This section lasts 48 seconds and automatically links to the last section through a shakuhachi solo, which represents the final blow of the protagonist who definitively ends Garad's life.

## Coda: And Then Fate Rewinds

According to the prophecy, when The Hero sacrifices his life to stop The Beast, fate rewinds. In Grandma's words, *"It has always been like this. No one remembers how it started, but this is the terrible fate that the village is doomed to repeat. Until the end of time"*.

This last section—that only lasts 14 seconds—closes the work, taking up some of the musical ideas exposed in the Introduction: Prophecy, giving full prominence to the leitmotif of fate and alternating between the first two chords of the harmonic succession of The Beast. In this way, it is intended to represent how the curse of the village has not been ended, but rewinded, and how other Heros and other Beasts will be embroiled in this conflict, generation after generation.

# 5. WORK DEVELOPMENT

---

## 5.1. Rhythm system

After the initial conceptual design tasks, the first great task of the project was to implement a rhythm system that kept the music perpetually synchronized with both the gameplay and the visuals despite possible low performance problems or unstable framerate, and that, in addition, was simple and clear enough so the player can stay constantly informed without having to take his attention away from the movements of the enemy.

For this, a brief study has been made of what different types of visual representations of rhythm are usually used in rhythm video games.

### Rhythm bar

The rhythm bar is the key element in most rhythm video games. In this games, a series of notes go through the bar until they reach a key point. The player must press the correct button when the note is centered on the key point.

The problem with this system is that it is designed for games that have different types of notes that correspond to different buttons—usually marked with different colors or shapes—, as in *Guitar Hero* [17] or *Dance Dance Revolution* [18]. In these games, the elongated design of the bar is important because it allows the player to see in advance which buttons they will have to press next.

Nevertheless, in games in which there is only one action button or in which the decision of what button to press remains entirely with the player—as is the case of *DownBeat*— this design loses its meaning, becoming excessively invasive on screen and capturing too much of the player's attention. Games like *Crypt of the Necrodancer* [19] or the more recent *Cadence of Hyrule* [20] propose a horizontal bar at the bottom of the screen, with the key point centered instead of at one end. In this way, the player is allowed to continue to focus his attention on the central part of the screen, where the action happens.

This solution is not applicable to *DownBeat* since two key points are needed—one for the downbeats and one for the upbeats. This means a decision would have to be made in order to keeping the two key points centered and close together—which would give very little response time to the player— or to keep them separate—giving a longer response time to the player but returning to the errors described above. Another solution would be to adapt the *Crypt of the Necrodancer* system and mark the notes with different colors depending on whether they are downbeats or upbeats, but it would be a form of representation that focuses most of its informative efforts on data that the player already know— after a downbeat an upbeat comes, and vice versa.



## Circular design

The circular design chosen for *DownBeat* —inspired by the *Pokémon GO!* [21] capture system— solves these problems in an elegant and functional way: All visual information is collected within a circle that remains centered at the bottom of the screen. Since the enemy and player are always centered horizontally on the screen, the player can refer to the rhythm circle without having to take their attention away from combat.

For more details about the rhythm circle, see subsection *Basic concepts* from section 3.3. *Mechanics*.

## Implementation

The first task of implementing the rhythm circle was to make the circle grow and shrink to the beat, given a specific value of beats per minute (BPM). To do this, I implemented two coroutines, `CircleGrow()` and `CircleShrink()`, to grow and shrink the circle respectively. Also, I implemented a third coroutine, `Loop()`, which perpetually toggles between the execution of the other two.

`CircleGrow()` and `CircleShrink()` also update the `normalized` variable, that is always equal to the current portion of the normalized rhythm, being 0 a downbeat and 1 an upbeat. For example, when the rhythm circle is on the 20% of its way to an upbeat or on the 80% of its way to a downbeat, `normalized` is equal to 0.2, regardless of the BPM value.

When initiating the program, the duration of a beat is calculated from the given BPM value, and the `Loop()` coroutine starts. This beat duration value is the duration that is applied to the other two coroutines, which use methods from the iTween library to properly animate the different circle elements, previously designed.

Testing this system resulted in a rhythm circle that started out in sync with the music but gradually became out of sync. Using the Unity console log, it was determined that, after 50 seconds being active, the system accumulated an error of 0.5 seconds, that is, a cumulative error of 1%.

Although this amount may seem insignificant, an error of 0.5 seconds makes the game completely unplayable. For example, using 60 bpm music, beats last 1 second, which means that, being 0.5 seconds out of sync, a player who presses the buttons to the beat of the music in perfectly accurate timing would get a 0% success ratio in all their actions.

To contextualize these numbers, a delay of more than 100 milliseconds in online games is considered a high ping value and makes difficult to play. Ideally, for a rhythm game that works entirely locally, there should never be a cumulative delay of more than 30 milliseconds, that is, approximately two frames at a rate of 60 frames per second.

## Auto-synchronization system

Using the Unity audio engine time value, I implemented a simple algorithm—which iterates once at the end of the `CircleGrow()` and `CircleShrink()` coroutines—to keep the game perpetually in sync with the music:

- Two new variables of the float type are declared: `lastTimeStamp` and `syncCorrection`, which indicate the time value of the audio engine in the previous iteration and the time correction to be applied to the next coroutine, respectively.
- The expected time value for the next iteration is calculated as the sum of the previous time value, the duration of a beat, and the time correction.
- The current time offset value is calculated as the expected time minus the real time value.
- `lastTimeStamp` is updated with the `audioSource.time` value;
- `syncCorrection` is updated with the time offset value.

To make these time corrections effective, the duration of `CircleGrow()` and `CircleShrink()` is now equal to the duration of the beat plus the time correction value.

The testing of this algorithm resulted in that, after 15 minutes being active, the rhythm circle accumulated an error of less than 10 milliseconds, that is, less than 0.00001% of the elapsed time. Not only that, but also this testing showed that, even in low performance conditions and irregular framerates, the cumulative error is not increasing, but ranges between values of 0 to 10 milliseconds regardless of the time it has been active, unlike from the previous implementation that accumulated error over time. Therefore, this implementation is considered satisfactory.

## Scheduling system

In the same script that controls the rhythm—`Rhythm.cs`—scheduling methods were implemented. These methods are useful for the operation of the combat system and allow to execute a certain method of a certain script after a specific number of beats or after a specific interval of time or to prevent the player from performing actions until they pass a specific number of beats—`locked beats`.

The difference between waiting for a certain number of beats and their time equivalent is very important since, although the player can only perform one action on each beat, they must be allowed to perform one action at the end of a beat and another action at the beginning of the next beat, although the time between these actions is much less than there would be between two actions performed at the beginning and at the end of the same beat, respectively.

## 5.2. Combat system

### Success rate

In the `PlayerController.cs` –the script that controls the player character– the `CheckSuccess()` method was implemented, which checks the `normalized` variable of the rhythm circle to calculate and update the `successRate` variable of `PlayerController.cs`. This method, in addition, returns a boolean that indicates whether the action is a success or a failure depending on the success rate achieved.

A `CheckMovementSuccess()` method was implemented that works in a similar way but following stricter criteria to calculate the success rate.

These methods are normally called from the conditional statements that must be met in order for the main action methods – move, attack, block, technique, and cut – to be executed.

### Movement

The `Node.cs` class was implemented to allow square based movement. Each of the cylinders that represent the squares in the scene has this script. Every `Node` object represents a node of the graph that is the combat grid. Each node has references to its left node, its right node, and its anterior and posterior nodes, if any.

After implementing the input system by binding each of the essential actions of the game to a button or stick on the controller –including two additional debug actions linked to the left and right triggers respectively– the `Move()` method was implemented in `PlayerController.cs`. This method is executed when sufficient input is received from the Move action. Its purpose is to translate the character in the desired direction if there is an adjacent square in that direction and the success rate obtained is sufficient.

To prevent the player from moving multiple times or performing multiple actions on the same beat, the main actions methods check the following statements in order to be executed:

- The current beat is not locked.
- The `CheckSuccess()` –or `CheckMovementSuccess()`– method returns `true`.
- The player is currently not performing any action (`currentAction == Action.None`).

The `Action` type is an enumerated type that includes each of the main and some auxiliary actions: Attack, Block, Tech, Reload, Twirl, Flash, Cut, Move, Wait and None. When the action meets the above requirements and starts, `currentAction` is updated with the new action to be performed. After a given time, `currentAction` is updated to be equal to `Action.None` if the player has not started a different action.

## Attack

For the attack, a system based on two methods was implemented, one for its loading –`LoadAttack()`– and the other for the attack itself –`Attack()`. This same structure and how this method works has been replicated to implement most of the other actions.

The `LoadAttack()` method is executed when pressing the attack button and checks the requirements mentioned in the previous section, in addition to verifying that the rhythm circle is currently on a downbeat using the `IsDownBeat()` method from `Rhythm.cs`. If all of these requirements are met, the next two beats are blocked, `currentAction` is updated to `Action.Attack`, and the `loaded` variable is marked as true, indicating that the attack is loaded. These two variables will automatically return to their initial state after two beats thanks to the `ScheduleUnload()` method from `Rhythm.cs`.

The `Attack()` method is executed when the player releases the attack button. This method checks that the beat is not locked, the current action is an attack and the rhythm is on a downbeat. If these requirements are met, the damage to be inflicted is calculated from a new calculation of the success rate.

Depending on whether the player is in the inner ring or not, damage is calculated differently:

- If the player is in the inner ring, damage is equal to the player's damage stat multiplied by the success rate and divided by 100. Additionally, if the attack is critical, this damage is multiplied by 1.5 and a bullet is reloaded into the chamber using the `FillChamberSlot()` function, which will be covered in the *Ammunition* subsection.
- If the player is not in the inner ring, the same calculation is performed but replacing the player's damage stat with the `Shoot()` method, which goes through the four slots of the chamber and, when it finds a slot filled with a bullet, it marks that slot as empty, plays the animation of spending that bullet in the user interface, heals the player if the fired bullet is a powered bullet and returns the damage value of the fired bullet, which varies depending on whether it is a powered bullet or not. If it does not find a bullet in the chamber, this method returns 0. If this happens, the attack is not executed.

After calculating the damage to be inflicted and taking the corresponding steps, this value is passed to enemy's `GetAttack()` method, which is analogous to the `GetAttack()` method from `PlayerController.cs`, which will be discussed in the *Block* subsection.

## Health

Basic methods have been implemented to control the player's health, including methods to take damage, heal, and animate the life bar accordingly.

These methods will not be discussed due to their triviality, but the link to the source code can be found in the section [6.3. Links](#).

## Ammunition

In order to manage ammunition expense and recovery, various methods and coroutines were implemented in the `PlayerController.cs` script. The most functionally important are the `FillChamberSlot()` method and the `ReloadCoroutine()` coroutine, which is started using the `Reload()` function.

These methods use the `Bullet` enumerated type, which includes the values `Bullet.Empty`, `Bullet.One` and `Bullet.Plus` to indicate if each of the slots in the chamber—which is implemented as a `Bullet` array—is empty, filled with a normal bullet or filled with a powered bullet, respectively.

`FillChamberSlot()` is used to reload a single bullet, which may or may not be powered—this is indicated through a boolean argument. This method goes through each of the chamber's slots and assigns the value of the new bullet to the first empty slot, if any. Also, it plays a small animation in the user interface to indicate that a bullet has been reloaded.

The `ReloadCoroutine()` coroutine works in a similar way but for a specific number of bullets. This function only receives an integer indicating the number of bullets, since it is not necessary to specify whether these bullets are powered or not—that will depend on whether a full charge has been made when the chamber was empty or not. Given the number of bullets to reload and the number of empty slots, this coroutine determines which types of bullets to reload in each slot and plays a small animation for each reloaded bullet in the user interface sequentially.

The two remaining coroutines—`OutOfAmmo()` and `BurnBullet()`—have a purely aesthetic and communicative function, since they manage small animations in the user interface to indicate to the player that they are trying to shoot without ammunition and that they have spent a bullet when shooting, respectively.

## Block

The block implementation follows the same “load and execution” structure discussed in the *Attack* subsection, and uses a very similar logic. The main differences are that the block can be loaded both on a downbeat and on an upbeat, and that, when releasing the button, a coroutine—`WaitForDamage()`—is started in order to handle the possible delay between the enemy attack and the player block.

When the player receives an attack, the `GetAttack()` method is called, which, if the player is blocking, assigns the value of the attack received to the `damageToBlock` variable from `PlayerController.cs`. Otherwise, the `GetDamage()` method—mentioned earlier in the *Health* subsection—is called instead.

The `WaitForDamage()` coroutine waits for the `damageToBlock` variable to be non-zero during a given time. When this happens, it calls the `GetDamage()` function indicating a damage value to receive, which depends on the success rate achieved.

## Special techniques

The special techniques also follow the same logic and structure as the previous two actions, with the peculiarity that, when starting to load the technique, a differentiation must be made between which technique is being loaded, depending on the movement of the stick.

The method for loading techniques `–LoadTech()`– is executed when the button is pressed: Consequently, if the movement of the stick is checked with a simple conditional statement, the technique selection will only work properly if the player moves the stick before or in the exact same frame in which the button is pressed. In other words, if the player moves the stick a few milliseconds after pressing the button, the loaded technique may not be the correct one.

To solve this, the `WaitForTechCorrection()` coroutine was implemented, which, following a logic similar to the `WaitForDamage()` coroutine, waits to receive enough input from the left stick to then correct the current technique to the desired one. This coroutine is called when the player starts to load the reload technique, which is the technique that is performed without any movement input. The maximum duration chosen for this coroutine is 0.2 seconds. After testing, this value was consolidated as an adequate value, as it is an interval that is permissive enough with the player's errors so the game feels responsive and satisfactory, but strict enough so the player cannot exploit this threshold as an advantage and the transition between animations is not compromised.

Regarding everything else, the implementation of the techniques is analogous to the implementation of the attacks, but changing the downbeats for upbeats and adjusting the different damage values and ammunition rewards following the description found in the subsection *Special techniques* from the *3.3. Mechanics* section.

## Cut and offbeat

Given the instantaneousness of the cut, this move does not need a loading method, it only needs a execution method `–Cut()`– that is called when the player presses the button. Therefore, the implementation of the cut is reduced to a conditional statement that checks the conditions described in the subsection *Cut* from the *3.3. Mechanics* section:

- The beat is not locked
- The acquired success rate equals 100%.
- The enemy is charging a block or a technique.

If these conditions are met, the enemy's `OffBeat()` method is called. Otherwise, the player's `OffBeat()` method is called. For the player, this method interrupts any action that is being loaded or performed, sets the variable `offbeat` to true and starts the `WaitForRecover()` coroutine, which, after 0.5 seconds, waits for the player to achieve a 100% success ratio and then sets the variable `offbeat` to false. The `CheckSuccess()` and `CheckMovementSuccess()` methods were modified so that, if the `offbeat` variable is true, they always return false except when the success rate equals to 100%.

## 5.3. Enemy behaviour

Most of the enemy methods related to the main actions, health or success ratio are implemented analogously to the methods explained in the previous section, with the particularity that, in this case, the actions are not executed by the player but by a simple artificial intelligence. Therefore, in this section, only the exclusive implementations from `EnemyController.cs` –the script that controls the behavior of the enemy– and the differences proposed by the non-exclusive methods with respect to their analogs from `PlayerController.cs` will be discussed.

### Artificial intelligence

The enemy exhibits a series of behaviors in each of the different phases of the battle. To achieve this, a method was implemented for each phase –`Phase1()`, `Phase2()`, `Phase3()` and `Phase4()`. One of these methods –depending on what the current phase is– is called in the `Update()` function, which is executed every frame, except if the `offbeat` variable from `EnemyController.cs` is true, in which case the enemy schedules a call to the `RecoverBeat()` function –which sets `offbeat` to false– for three beats later, and stays inactive meanwhile.

The methods that govern the enemy behavior during the different phases are implemented following the descriptions explained in the *Phases and behavior* subsection of section 3.5. *Battle against Garad*, and the link to the source code can be found in the section 6.3. *Links*.

Regarding the accuracy with which the enemy follows the rhythm, one of the conditions required for the enemy to perform a certain action is that the `normalized` variable of `Rhythm.cs` must be less than 0.02 –for downbeat actions– or greater than 0.98 –for upbeat actions. In this way, it is guaranteed that the enemy behaves with a certain intelligence when following the rhythm and does not try to execute actions at arbitrary times.

In addition, for each action the enemy performs, its charge time –the time that elapses between the moment when the action starts to load and the moment when it is executed– is equal to that action's usual charge time plus a random value ranging from -0.1 to 0.1 –although this value is customizable from the Unity editor in order to increase or decrease the difficulty by improving or worsening enemy's accuracy. Thanks to this, the enemy is able to acquire different success rates, resulting in different damage values, occasional critical attacks, etc.

This random value is applied in the same way to all actions except the cut, where this value is multiplied by a ratio that can be customized from the Unity editor. The objective of this ratio is to make the enemy more precise when making a cut than when doing other actions. For example, an enemy that gets a 100% success ratio in one out of every two actions may be frustratingly powerful as the half of their attacks are critical hits, but they may also be excessively dumb because they miss half of their cuts, applying `offbeat` to themselves every time.

## Counter

The `Counter()` method controls the behavior of the enemy when its objective is to counter the player. This method is called from the methods of the different phases when the required conditions are met. This condition may vary in each phase, and are discussed in the

This method checks whether the player is loading an attack, a special technique, or a block, and consequently performs the appropriate move to counter it, using the `ScheduleFunction()` method from `Rhythm.cs` and the corresponding action method `Block()` if the player is loading an attack or `Cut()` if the player is loading a block or a technique.

## Attacks and techniques

A different method has been implemented for each enemy attack and for each enemy technique. In addition, two functions `RandomAttack()` and `RandomTech()` have been implemented to execute a random attack or technique from a given list. These methods receive an array of integers as an argument, randomly choose one of those integers and consequently decide, using a switch statement, which attack or technique to execute. In this way, making the enemy attacks use different attack pools for the different phases is as easy as passing the method a different array in each phase.

To execute the attacks on specific squares, several methods have been implemented in the `Node.cs` script that, using coroutines, increase the emission and color of the squares' material during a given loading time and then inflict damage on the player if they are in that square.

In addition, in this same class, some useful methods have been implemented for the selection of squares and the cancellation of attacks and techniques:

- `GetIndexInAxis()` returns 0 if the node is from the inner ring, 1 if it is from the middle ring and 2 if it is from the outer ring.
- `GetFirstNodeOnThisAxis()` returns the node from the inner ring that is also on this node's axis.
- `GetLastNodeOnThisAxis()` returns the node from the outer ring that is also on this node's axis.
- `GetFirstNodeOnOppositeAxis()` returns the node from the inner ring that is also on this node's opposite axis.
- `Cancel()` stops all coroutines on this node and returns its usual material properties.



Regarding attacks and techniques, they are implemented as follows:

- **Thunder:** Loads an attack on the node the player is on.
- **Storm:** Selects the player's current node and between 4 and 6 additional nodes of that node's axis and adjacent axes, and loads an attack on each of them.
- **Thrust:** Iterates over the player's current axis, loading an attack on each of its nodes.
- **Sweep:** Iterates over the player's current ring, loading an attack on each of its nodes.
- **Backhand sweep:** Iterates over the inner ring, loading a technique on each of its nodes.
- **Prayer:** Uses the `ScheduleFunction()` method from `Rhythm.cs` to schedule a call to the `Heal()` method from `EnemyController.cs` after about two seconds. In order to execute this technique, a random value between 0 and 10 must be lesser than the `retaliate` value. This variable starts with a value of 5, sets to 0 when the enemy uses this technique, increases its value by one (with a maximum of 10) when the random number surpass its value or when the enemy receives damage, and decreases its value by one (with a minimum value of 0) every time the player counter this technique with a cut or an attack. This makes the enemy to use this technique more often as they receive damage, but also to use it only in times of need if the player usually takes advantage of this technique to apply offbeat to the enemy.
- **Storm Combo (Storm + Replication):** All nodes in the combat grid are enqueued in a modified priority queue that allows an integer to be passed to the constructor as an argument. This integer is the `maxRandomVariation` variable of the priority queue. When an item is added to the priority queue, the queue increases its priority by a random value between 0 and `maxRandomVariation`. The initial priority assigned to each node is equal to the Manhattan distance [22] from that node to the player's current node. For the first storm, 5 to 7 nodes are dequeued, and an attack is loaded on each. For the second storm, 5 to 7 more are dequeued, and a technique is loaded on each.
- **Sweep Combo (Sweep + Replication + Sweep):** Randomly, it is decided whether the combo will start on the inner ring or the outer ring. If it starts on the inner ring, the method iterates over it, charging attacks on each of its nodes. This process is repeated after a second for the middle ring, and after two seconds for the outer ring. If the combo starts on the outer ring, the order of the rings is the opposite –outer, middle and inner.
- **Thrust Combo (Thrust + Replication, up to 3 times):** Randomly, it is decided whether the combo will start on the axis that is left adjacent to the player axis or on the axis that is right adjacent to the player's axis. Then, a random number between 3 and 6 thrusts is determined. At 1 second intervals, thrusts begin to be loaded onto the selected axis, iterating over the circle clockwise if started to the player's right or counterclockwise if started to the player's left. The attack stops when the chosen number of thrusts is reached or when one of the thrusts hits the player.

## 5.4. User interface

Developing the user interface was one of the most time consuming tasks compared to initial planning. However, most of these tasks consisted of implementing simple animations using methods from the `iTween` library and testing and adjusting different parameters to obtain a satisfactory visual result.

Since the operation of most of the elements of the user interface have been explained in different subsections from sections [5.1. Rhythm system](#) and [5.2. Combat system](#) —in methods that, in addition to having a function in the playable system, also control part of the user interface—, this section will only cover implementations related to two elements: the `dialog box` and the `numeric indicators`.

### Dialog box

The dialog box makes use of the `CutsceneText.cs` script. This script includes the `LoadText()` method, which loads a text file from one of the subfolders from the `Resources/Text` folder. This method receives a string as parameter, which is the name of the folder to load. Each folder corresponds to one of the cutscenes in the game, and inside it contains different text files with observations or thoughts related to that cutscene. Each time this method is called, it randomly loads one of the text files from the corresponding folder.

In addition, the `CutsceneText.cs` script has three parameters that are customizable from the Unity editor —`letterTime`, `pauseTime` and `stopTime`— that indicate the duration of the pauses that the `ShowText()` method makes after each letter, comma and full stops, respectively.

This method is called at the beginning of every cutscene and everytime the player presses the A button during a cutscene. In order to work properly, this methods also checks the state of the text using the enumerated type `TextState` —which can be `Unloaded`, `Writing`, `Paused`, and `Ended`.

After loading a new text if necessary and if writing process has not ended, the `ShowText()` method displays the text by writing the letters one by one —what is commonly known as *Typewriter effect*— until it finds the character '\$', which is not displayed, and stops the method. When the player calls this method again by pressing the A button, the previous text is erased and the writing is resumed after the last '\$' character reached.

If the player presses the button A and the writing process has ended completely, the text is erased and a timer is displayed on the lower right corner of the screen. This timer indicates the remaining time for the transition to the next combat phase, and its implementation is discussed in the [Implementation](#) subsection from section [5.6. Dynamic soundtrack](#).

## Numerical indicators

To implement the numeric indicators, a text object named `PopUpText` was created, which was configured with the desired text properties and saved as a prefab in the Resources folder so that it could be loaded via script.

The `UIController.cs` script was implemented to manage the color, size, spawn position and animation of each of these indicators. When the `PopUpNumber()` method from this script is called from another script, the following data is passed as arguments:

- The number to display
- The `NumberType` value –an enumerated type which presents various values such as Rate, Attack, Plus, Damage, Block, etc.
- A boolean value indicating whether the number must be displayed as a critical hit or not.

Given these three arguments, the color and size are chosen using the `Setup()` method, which uses a switch statement to determine the color depending on the type and calculates an appropriate size depending on the value of the number, whether it is critical or not, and the `minSize`, `maxSize`, `minValue` and `maxValue` parameters, which establish a linear relationship between minimum and maximum values of size and numerical value.

After this, the distance that the number will translate in its animation is calculated following similar criteria, and the `Animate()` coroutine –which is in charge of varying the position, scale and opacity of the number during its life on screen– begins. The duration of the animation is also chosen depending on the value of the number.

To avoid overlapping numbers, `UIController.cs` makes use of an array of Transform objects that act as pivots on which to spawn the numbers. This array –`pivots`– works in conjunction with an array of the same length named `empty`, which indicates which of the pivots are available and which are occupied. When the `PopUpText()` function is called, it iterates over `empty` to find the first available pivot, and the `PopUpText` prefab is instantiated from the Resources folder as a child of that pivot, in addition to marking that pivot as occupied in the `empty` array.

When the animation ends and the `PopUpText` object disappears, its pivot position in the `empty` array is marked as available again.

This script also has functions analogous to the previous ones but that follow other criteria to decide the size, position or content of what is shown on the screen. For example, the `PopUpRate()` and `PopUpText()` methods follow the same logic as `PopUpNumber()` but use a single-pivot system and call different overloads of the `Setup()` method or the `Animate()` coroutine to satisfy certain peculiarities with respect to the visual representation of the message.

Most of the parameters can be customized from the Unity editor – the position of the pivots, the minimum and maximum values of size and numerical value, the base values of translation distance for the animation, screen time and fade duration, and the different colors for the text.

## 5.5. Animations and visual effects

Unexpectedly, the tasks from this field have been some of the most time consuming tasks in the entire project. This was mainly due to the fact that a large part of the animations used had to be adapted to meet the needs of the project.

Despite the fact that this field may seem of minor importance, a visual section that provides the user with adequate feedback on what is happening in the game is essential for a satisfactory combat system. For example, *Sekiro: Shadows Die Twice*'s combat system [1] feels so responsive, intuitive, and polished largely due to the sparks released by the swords when they collide—which tell the player if attacks are blocked or deflected—and to the fluid and precise animations that allow the player to discern what attack the enemy is going to use, and act accordingly.

In the same way, it is crucial for this combat system that the player has clear and striking visual information about the movements of his own character, the movements of the enemy and even the rhythm.

### Combat animations

The combat animations, both of the enemy and of the player, as well as the models of the characters, have been imported from *Mixamo*. Despite the fact that this page has a wide variety of animations, many of the characters' actions were not properly represented in them, and the most similar ones hindered visual clarity or were incompatible with the character's design. For example, although the main character and the enemy share the majority of the actions, no character animation could be reused for the enemy since the enemy wields a weapon—in fact, a large weapon that easily improperly pierces the geometry of the stage or the enemy himself.

Consequently, many of the animations have been adapted from specific positions of other animations. For example, the posture adopted by the enemy when loading the Flash technique is extracted from a capoeira movement.

Most actions have been implemented following a similar logic: the character can transition to an action loading animation from any other state when the corresponding trigger is activated via script. When the player releases the button to perform the action, a trigger named *release* is activated. This causes the character to transition to the animation in which he performs the action in question. After a certain time, the character returns to the default state. When the same loading animation can result in two different animations—such as loading the block, which can result in a block or a parry—, auxiliary triggers are used.

Also, a trigger called *fail* is used when the player releases the button at half load or, in general, when an interruption of the current animation is necessary. When this trigger is activated, the character returns to its default state.

In short, for the animations of the character the following state machine was implemented using the Unity Animator (See Figure 14):

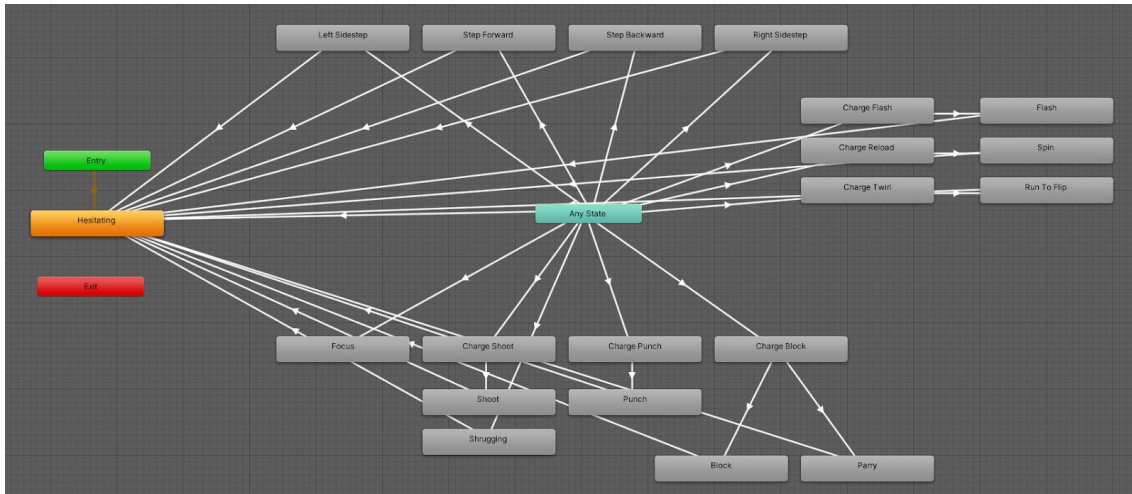


Figure 14: Player animator states and transitions.

For the enemy, in addition to implementing a state machine similar to the previous one, several methods were extended in order to move the halberd. For example, in the `Sweep()` and `Thrust()` methods, an empty object is placed at the player's position (See Figure 15). This empty object acts as a pivot, and the `Effects.cs` script uses it to orientate the halberd to the indicated location. When the animation ends, the halberd returns to its original position relative to the enemy's right arm.

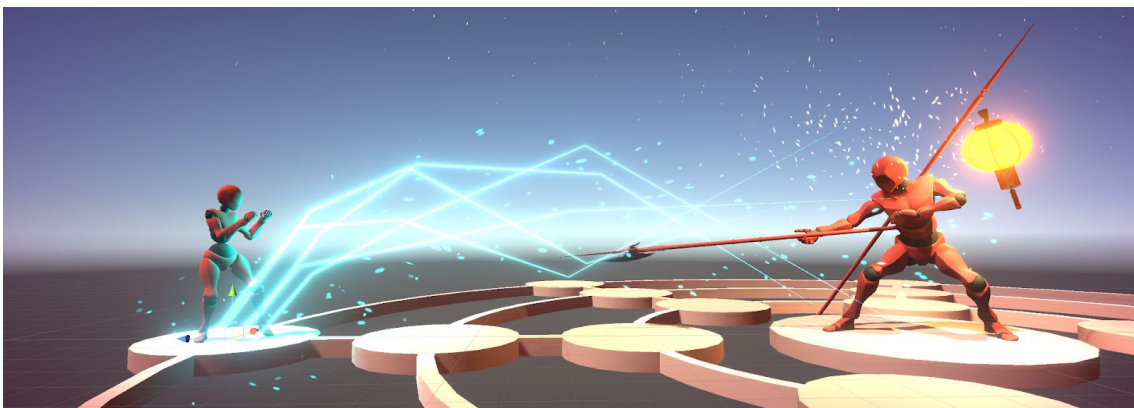


Figure 15: Position of the halberd pivot –right under the player– during an enemy thrust.

Originally, the enemy was going to use the halberd to attack and the pike –which is hanging on his back, holding the lantern– to use techniques. This idea was finally discarded due to the absence of Mixamo animations that held two weapons and that served to faithfully represent the actions of the enemy. The 3D models for the halberd and the pike have been imported from [23], and the model for the lantern has been imported from [24].

In comparison with the previously discussed state machine, the enemy state machine (See Figure 16), presents very similar structure and logic, with the difference that the transition between animations does not depend on the player, which facilitates its implementation. On the other hand, the enemy has a wider variety of attacks than the player, although these animations are reusable for the Replication technique. In addition, the enemy does not have movement animations, since it is always in the center of the stage.

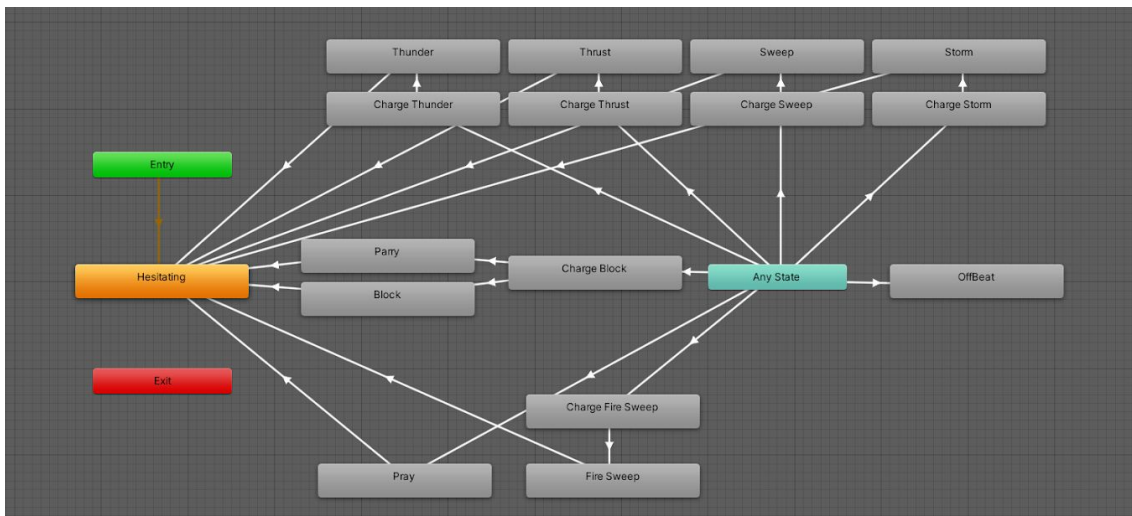


Figure 16: Enemy animator states and transitions.

## Visual effects

Different visual effects have been implemented for both player and enemy attacks and techniques. In general, these effects have been developed using particle systems, trails and lights, mostly derived from the pre-made effects from the Unity Particle Pack [25].

To manage its appearance and some small animations regarding the scale or rotation of the effects –which use methods from the `iTween` library or own scripts– different methods have been implemented in the `Effects.cs` class. Each of these methods is called from the corresponding animation using animation events, and is responsible for instantiating the prefabs necessary to carry out the effect –for example, for an explosion, the script instantiates a shock wave, sparks, a light, etc.– and managing all coroutines that are necessary to properly animate said effect. The Flash technique, in addition, makes use of a plane that uses a custom designed sprite to represent the energy blade that comes out of the character's arm.

For the general environment, three different `post-processing profiles` have been used –one practically empty, one for the general scene and another for when the player is under the `offbeat` effect– and `two cameras` –one that renders the general scene alternating between the two last

post-processing profiles and a superposed camera that only renders, under the first profile, the characters and a few other elements. (See figure 17)

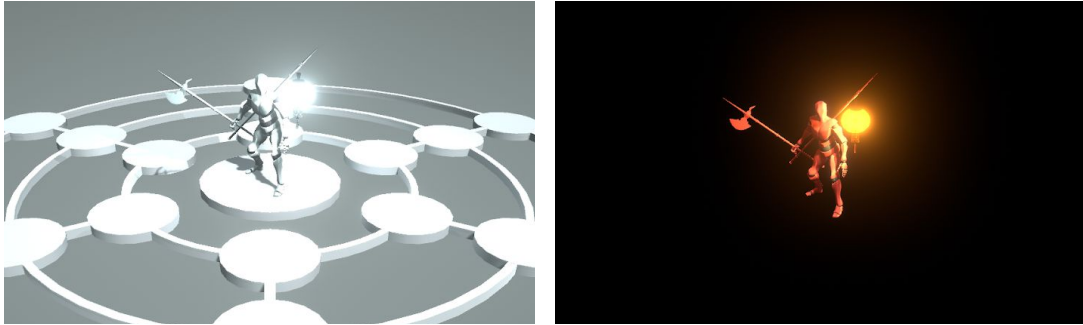


Figure 17: Comparison between the image rendered by the main camera –left– and the image rendered by the superposed camera –right. In the game, these two images are combined in order to show the offbeat post processing effects but without affecting the characters.

## Camera behaviour

The **Cinemachine** package functionalities have been used to create several virtual cameras between which the main camera transitions. For example, the game uses a different virtual camera for each of the three positions that the player can have in his axis of the combat grid, another for when he loads the Flash technique, etc.

Transitions between cameras use a state machine implemented in the Unity Animator, similar to those that control player and enemy animations. (See Figure 18). Triggers that control transitions between cameras are activated when the player performs certain actions, such as moving forward or backward.

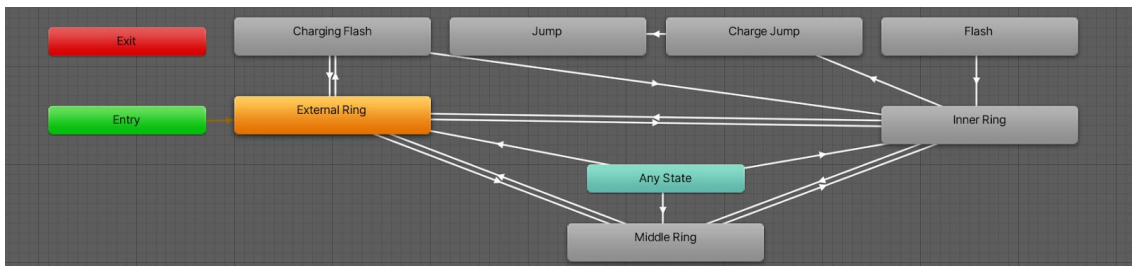


Figure 18: Camera states and transitions.

The main tasks regarding the use of cameras have taken place when implementing the cutscenes, but this section will be discussed in more advanced stages of the development of the project, given that some of these cinematics are not yet implemented.

## 5.6. Dynamic soundtrack

### Learning stage

This stage includes a research about different techniques and practices that may be useful for the implementation of the dynamic soundtrack, as well as the learning of the basic teachings for using sound middleware such as Wwise. A tutorial by Michael Zull [26] has been followed in order to integrate Wwise into Unity, while the official Wwise tutorials [27] have been consulted in order to learn how to use the software.

### Soundtrack adaptation

The implementation of the dynamic soundtrack has been conditioned by some limitations of the audio middleware, such as:

- Wwise offers very limited audio edition options.
- Wwise does not allow jumps within the same audio clip.
- The only useful data from the wwise audio engine that can be queried from Unity is the current time value, that is, the time in milliseconds that has passed since the audio engine was initialized.

Due to the first limitation, the initial plan to attenuate the music directly from the audio middleware when the offbeat effect was applied to the player had to be dropped. Instead, an alternative version of *Bound by a terrible fate* was produced including echo effect, reverb, low-pass filter, and custom equalization to achieve the “underwater attenuation” effect.

In order to achieve the desired implementation despite the second limitation, it has been necessary to cut *Bound by a terrible fate* into 12 audio clips, so that all the desired jumps can be made by transitions between clips. The same process was followed to split the attenuated version of the soundtrack to obtain an attenuated version of each of the clips. Depending on the needs when linking with other clips, many of these clips have pre-entry and post-exit fragments, which overlaps the end of the previous clip or the beginning of the subsequent clip respectively, thus achieving more natural transitions (See *Figure 19*).

Due to this, the cuts had to be made independently for each of the audio tracks—which correspond to each of the instruments—that make the soundtrack. This allows, for example, the pre-entry fragment of the *Phase III: A Terrible Fate* to consist only of the brass anachruses, the snare drum and drum kit introduction and the piano fill that uses the Fate leitmotiv. By leaving the rest of the instruments out of this fragment, it is guaranteed that there will not be duplicate voices and that the transition to this phase can take place from different points of *Interlude III: What Could Have Been*.



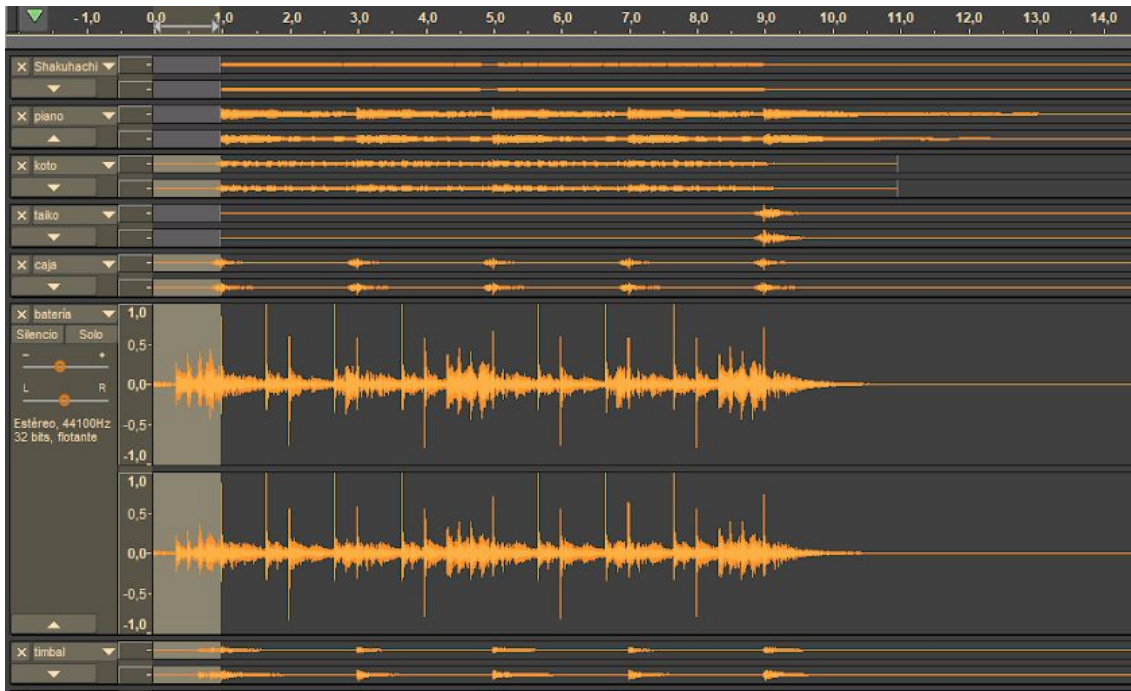


Figure 19: Seven of the eleven audio tracks that compose the Coda: And Then Fate Rewinds audio clip. Highlighted on the left side of the timeline, the pre-entry fragment of this clip.

## Implementation

The first phase of the implementation consisted of preparing in Wwise a whole hierarchy of audio clips —each of which is composed by the original track and attenuated version—, placing cues along each of the clips to mark the start and end of the pre-entry and post-exit fragments and additional transition points, establish the appropriate rules for each of the transitions between clips, and establish the flow between the several clips—as well as the alternation between tracks—using switches and states.

Once this was achieved, the dynamic soundtrack was already working as expected within the audio middleware, but Unity still needed to communicate with Wwise to activate and deactivate the switches and states that govern the flow of the dynamic soundtrack.

This is easily accomplished by using the functions from the `AkSoundEngine` class, which is automatically implemented in the project when integrating Wwise. However, an unexpected problem arose, and that is, after calling that function, there was no way of knowing within how long the transition would happen.

For example, if the player presses the A button on the start menu to start the battle, the transition to the combat –which must match the transition from the Introduction: Prophecy audio clip to the Interlude I: Steel Tempest audio clip– should not happen instantaneously, but should wait for the right moment to ensure a musically correct transition and also to leave the player at least a couple of seconds as a threshold to prepare themselves for the first round of enemy attacks.

To achieve this, several functions were implemented in MenuController.cs to manually calculate the remaining time until the transition occurs. These functions replicate the logic that Wwise follows to decide when to make the transitions depending on the specified rules, and, consulting the time value from the audio engine, saving this value at the beginning of each phase and doing the appropriate mathematical operations, they calculate the remaining time, display it on the screen and schedule the start of the next phase of the combat accordingly.

A major part of the implementation stage consisted of testing if the dynamic soundtrack was working as expected and tweaking parameters until the result was satisfactory. This supposed a large amount of time considering the complete duration of the work and the number of possibilities that had to be tested.

## Sound effects

The last part of the creation and implementation of the dynamic soundtrack consisted of mixing and editing different sounds clips to obtain sound effects that fit the needs of the project, as well as establishing the rules for their correct reproduction.

In addition, to prevent the sound effects from becoming repetitive, some of them were pitch-randomized. Indeed, for the thunder sound effect –which sounds every time the enemy attacks–, three different sound clips were produced. Whenever a thunder sound needs to be played, Wwise to plays randomly one of this three sound effects in order to decrease repeatability.

The links to the original sound effects have been included in the *Bibliography* [29 - 45], although most of the final effects have gone through a deep process of editing and mixing, resulting almost unrecognizable. In total, 19 sound effects were produced for the following events:

- Block
- Confirm (Menu)
- Critical hit
- Cut
- Enemy block
- Flash
- Heal
- Out of ammunition
- Powered shot
- Prayer
- Punch
- Reload
- Shot
- Slash
- Text (Menu)
- Thunder (1)
- Thunder (2)
- Thunder (3)
- Twirl

## 5.7. Testing and debug

The testing stage not only resulted in the discovery of several important bugs and significant conclusions regarding the difficulty balance, but also served as an opportunity to search for exploits of what the player could take advantage of and make corrections in the implementation of the mechanics to get a more satisfying experience.

Most of these bugs and exploits were caused by the problems in the synchronization of the time limits within certain actions can be performed, and were solved by adding time locks using coroutines.

A significant part of the testing phase occurred during the implementation of the dynamic soundtrack and the transitions between combat and cutscenes, since, for example, every time it was necessary to test whether the transition from the fourth phase to the final cinematic—which in the soundtrack corresponds to the transition from *Phase IV: Conflagration* to the *Coda: And Then Fate Rewinds*—was produced correctly, the previous three phases had to be played. This resulted in a short list of bugs that were corrected immediately after their discovery or in the final stage of the project.

After 150 hours dedicated to the development of the demo—without including the composition and production of the soundtrack or the tasks related to the writing of this report— and having obtained a functional build and several final tests without the appearance of bugs, the testing and debug stage was considered as finished, although it new bugs may be discovered in the future.

# 6. RESULTS

## 6.1. Time balance and deviations

Before starting the project, a planning –which can be consulted in section [2. Planning](#)– was made. Despite the fact that this planning has served as a guide for the development of the project in terms of order and dependencies between tasks, various setbacks and minor changes of direction have emerged during the development of the project, resulting in a slightly different final planning.

The main reasons for the most notable changes in planning have been problems derived from the state of alarm, creative decisions and technical difficulties –although, in certain tasks, the opposite effect has occurred and the ease and absence of problems have resulted in a significant saving of time.

The dynamics for measuring the time spent on each task consisted of using the *Toggl* tool to start a timer each time a work session for a specific task started, indicating the name of the task and its classification –design, user interface, combat system, artificial intelligence, composition, etc.– and stopping it when the work session has ended or when the task is finished.

A side-by-side comparison between the expected hours and the real number of hours spent on each task, as well a brief explanation of the difference between them, if any.

MUSIC RELATED TASKS			
Task	Expected hours	Final hours	Deviation reason
Composing and producing	36	58	Lack of inspiration, inability to use Logic Pro X due to LABCOM closure, unexpected Wwise limitations.
Middleware basic learning	12	3.5	Unexpected software intuitiveness.
Dynamic soundtrack implementation	24	8	Rhythm system was expected to stop working when switching from Unity audio engine to Wwise dynamic audio engine, but worked with just a few minor changes.
Sound effects	2	4	The sound effects were mixed, edited and produced instead of just downloaded.
<b>Total</b>	<b>74</b>	<b>73.5</b>	–

CONCEPTUAL DESIGN RELATED TASKS			
Task	Expected hours	Final hours	Deviation reason
Combat system design	12	12	–
Battle scripting and enemy behaviour design	6	3	The battle was originally going to consist of five different phases, but was reduced to four due to time limitations regarding the soundtrack composition.
Camera behaviour design	2	1	Each cutscene were originally going to randomly toggle between at least eight different camera shots. This idea was discarded.
<b>Total</b>	<b>20</b>	<b>16</b>	–

PROGRAMMING RELATED TASKS			
Task	Expected hours	Final hours	Deviation reason
Movement implementation	3	3	–
Rhythm bar implementation	4	4	–
Combat system implementation	30	15.5	Most of the actions follow a similar logic, so the code of one was reused in the others.
User interface implementation	2	13	A lot of time had to be spent adjusting the behavior of some interface elements through trial and error until it was satisfactory.
Artificial Intelligence implementation	12	10	Reducing the number of phases from five to four meant less different behaviors to implement.
Camera behaviour implementation	4	1.5	Discarding the idea of randomized cutscenes considerably reduced the workload.
Menus and cutscenes implementation	–	9	These tasks were erroneously not taken into account when making the initial planning
<b>Total</b>	<b>55</b>	<b>56</b>	–

2D AND 3D ART, ANIMATIONS AND VISUAL EFFECTS RELATED TASKS			
Task	Expected hours	Final hours	Deviation reason
Rhythm bar design and art	2	1	—
User interface design and art	2	2	—
Cutscenes and main menu animations and visuals	4	7.5	Some custom animations were made but they stopped working when changing the hierarchy of character models for technical reasons.
Combat character's animations	8	18.5	Most imported animations had to be adapted to meet the needs of the project. In addition, more animations than expected were necessary.
Environment 3D modeling, lighting and post processing effects	6	3	Opting for a minimalist and symbolic aesthetic reduced the workload when building the scene.
Rhythm visual feedback	6	1	Due to the environment minimalism, it was only necessary to implement the glow of the tiles and some changes to the chromatic aberration.
Visual effects	10	13	Custom particle system and other visual effects were implemented for most player and enemy animations.
<b>Total</b>	<b>38</b>	<b>46</b>	—

DEBUG TASKS			
Task	Expected hours	Final hours	Deviation reason
Dynamic soundtrack debug	6	6	—
Combat system debug	8	8	—
Artificial intelligence debug	4	2.5	The simplification of the enemy AI also resulted in fewer bugs.
Combat animations debug	8	8	—
Cutscenes and menu debug	4	4	—
Build debug	6	0.5	Unlike most of my previous projects, there was only one bug derived from the build and it was easily fixed.
<b>Total</b>	<b>36</b>	<b>29</b>	—

DOCUMENTATION			
Task	Expected hours	Final hours	Deviation reason
Planification	—	6	Erroneously not taken into account in the planning.
Code cleaning	—	5	Erroneously not taken into account in the planning.
Sheet music layout	—	3	Erroneously not taken into account in the planning.
Video demonstration	—	3	Erroneously not taken into account in the planning.
Final report	48	64	Much more time consuming than expected.
Presentation preparing	29	12	—
<b>Total</b>	<b>77</b>	<b>81</b>	—

TOTAL BALANCE		
Task	Expected hours	Final hours
Music	74	73.5
Conceptual design	20	16
Programming	55	56
Visuals	38	46
Debug	36	29
Documentation	77	81
<b>Total</b>	<b>300</b>	<b>320</b>

## 6.2. Completion of the objectives

Regarding the initial objectives –discussed in section 1.3. *Objectives*– and having considered the project development as finished, these are the conclusions regarding their completion:

- A **playable demo of a complete boss battle has been developed**. This demo presents a unique, dynamic, enjoyable and exciting combat system, in which both opponents must respond to the movements of the rival following the beat and using in their favor the downbeats and upbeats of the music, as if it were a dance.
- Once the controls and mechanics are known, this combat system allows the player to achieve **victory at first try** relying solely on reflexes, sense of rhythm and mechanical skill, but it also leaves room for **challenge and improvement** by practicing and memorizing patterns.
- The mechanics, the music, the visuals and the rest of the narrative elements are **intimately coordinated** at any moment of the battle.
- An **original soundtrack** that **dynamically adapts** to player inputs and other in-game changes has been composed, produced and implemented
- The enemy has been provided with an artificial intelligence that allows him to pose an interesting challenge to the player, react to their actions and display different behaviors and attack patterns that remain consistent with the musical and narrative development of each phase.



## 6.3. Links

GitHub repository: <https://github.com/PabloPSK99/DownBeat-TFG>

DownBeat demo executable file:

<https://drive.google.com/file/d/1tu1dMy09SQnP2x64jTEDkCkvHq5Zs0uE/view?usp=sharing>

Video demonstration on YouTube: <https://www.youtube.com/watch?v=yPHhTktPbFY>

Bound by a terrible fate on SoundCloud:

<https://soundcloud.com/pablo-lorente-martinez/downbeat-bound-by-a-terrible-fate>

Bound by a terrible fate sheet music:

<https://drive.google.com/file/d/1iYfmgbd1RaRm000xhNKDOGEKj086i-MF/view?usp=sharing>

## 7. CONCLUSIONS

---

This project is the first step in a long journey to make *DownBeat* a reality. This game existed within me as a still formless idea: I had come up with some fun and unique mechanics for the combat system and had certain concerns in mind that I wanted to address through the game, but there was no common thread between the two nor an appropriate world where this idea could flourish.

Undoubtedly, the need to take this idea to paper in order to use part of it as my Bachelor's Thesis has been the kind misfortune that nourished its nascency.

This project has taken away my sleep more nights than my mind and body could to endure, but it has also given me a goal and the determination to reach it. Thanks to this project, I have been able to fully devote myself to my two passions: video game development and musical composition, and, from them, synthesize a work that I am extremely proud of.

Regarding future plans, *DownBeat* still needs a lot of conceptual and narrative design work before starting with the development of the full game. Having achieved that and finally having a solid vision of what I want *DownBeat* to be, I would like to assemble a small team and try to bring the project to a version advanced enough to be able to obtain financing to fully devote ourselves to its development.

I really hope that these wishes can be fulfilled in the future and I know that I will work hard to make it happen.

# BIBLIOGRAPHY

---

- [1] *Sekiro: Shadows Die Twice* (PC version)[Video game]. (2019). From Software. <https://www.sekirothegame.com/uk/en/home>. Accessed:2020-06-02.
- [2] *Doom Eternal* (PC version)[Video game]. (2020). Bethesda. <https://bethesda.net/en/game/doom>. Accessed:2020-06-02.
- [3] Fox, T. (2015). *Undertale* (PC version)[Video game]. <https://undertale.com/>. Accessed:2020-06-02.
- [4] Cavanagh, T. (2013). *Super Hexagon* (PC version)[Video game]. <https://superhexagon.com/>. Accessed:2020-06-02.
- [5] Wikipedia. Middleware. <https://en.wikipedia.org/wiki/Middleware>. Accessed: 2020-06-02.
- [6] Lyubo. (2018). *Introducing Coherent Console Fonts*, Coherent Labs. <https://coherent-labs.com/posts/introducing-coherent-console-fonts/>. Accessed: 2020-05-28.
- [7] Wikipedia. Rock Paper Scissors. [https://en.wikipedia.org/wiki/Rock\\_paper\\_scissors](https://en.wikipedia.org/wiki/Rock_paper_scissors). Accessed: 2020-06-02.
- [8] Wikipedia. Tomoe. <https://en.wikipedia.org/wiki/Tomoe>. Accessed: 2020-06-07.
- [9] Wikipedia. Raijin. <https://en.wikipedia.org/wiki/Raijin>. Accessed: 2020-06-02.
- [10] Toby Fox. (2015). *Heartache*. On Undertale Soundtrack. [Audio File]. Retrieved from <https://tobyfox.bandcamp.com/track/heartache-2>. Accessed: 2020-06-02.
- [11] Toby Fox. (2015). *ASGORE*. On Undertale Soundtrack. [Audio File]. Retrieved from <https://tobyfox.bandcamp.com/track/asgore>. Accessed: 2020-06-02.
- [12] Toby Fox. (2015). *Battle Against a True Hero*. On Undertale Soundtrack. [Audio File]. Retrieved from <https://www.youtube.com/watch?v=aWBtpBwzzdM>. Accessed: 2020-06-02.
- [13] Jason M. Yu. (2016) *An Examination of Leitmotifs and Their Use to Shape Narrative in UNDERTALE - Part 1 of 2*. <http://jasonyu.me/undertale-part-1/>. Accessed: 2020-05-30.
- [14] Game Score Fanfare. (2017). *How the song "Undertale" Hits Home*. [Video File]. Retrieved from <https://www.youtube.com/watch?v=9xR-x0kKP44>. Accessed: 2020-05-30.
- [15] Wikipedia. Leitmotif. <https://en.wikipedia.org/wiki/Leitmotif>. Accessed: 2020-05-30.
- [16] Wikipedia. Andalusian Cadence. [https://en.wikipedia.org/wiki/Andalusian\\_cadence](https://en.wikipedia.org/wiki/Andalusian_cadence). Accessed: 2020-05-30.

- [17] *Guitar Hero Live* (Xbox One version)[Video game]. (2015). Activision. <https://www.guitarhero.com/game>. Accessed:2020-06-07.
- [18] *Dance Dance Revolution A20* (Arcade version)[Video game]. (2019). Konami. <https://p.eagate.573.jp/game/ddr/ddra20/p/>. Accessed:2020-06-07.
- [19] *Crypt of the Necrodancer* (Nintendo Switch version)[Video game]. (2015). Brace Yourself Games. <https://braceyourselfgames.com/crypt-of-the-necrodancer/>. Accessed:2020-06-07.
- [20] *Cadence of Hyrule – Crypt of the NecroDancer feat. The Legend of Zelda* (Nintendo Switch version)[Video game]. (2019). Brace Yourself Games. <https://braceyourselfgames.com/cadence-of-hyrule/>. Accessed:2020-06-07.
- [21] *Pokémon GO!* (Android version)[Video game]. (2016). Niantic, Inc. <https://www.pokemongo.com/en-gb/>. Accessed:2020-06-07.
- [22] Wikipedia. Taxicab Geometry. [https://en.wikipedia.org/wiki/Taxicab\\_geometry](https://en.wikipedia.org/wiki/Taxicab_geometry). Accessed: 2020-06-07.
- [23] Helsssoo. (2016). *Halberd - Weapons 18th Century*, Unity Asset Store <https://assetstore.unity.com/packages/3d/props/weapons/halberd-weapons-18th-century-61568>. Accessed: 2020-06-08.
- [24] MIBS. (2019). *Free 3D Model : Chinese Paper Lantern*, Behance <https://www.behance.net/gallery/75603975/FREE-3D-MODEL-Chinese-Paper-Lantern>. Accessed: 2020-06-08.
- [25] Unity Technologies. (2020). *Unity Particle Pack*, Unity Asset Store <https://assetstore.unity.com/packages/essentials/tutorial-projects/unity-particle-pack-127325>. Accessed: 2020-06-08.
- [26] West Side Electronic Music. (2016). *Wwise and Unity: Getting Started with Unity & Wwise, Part 1*. [Video File]. Retrieved from <https://www.youtube.com/watch?v=rK96gLoZMEY>. Accessed: 2020-07-03.
- [27] Audiokinetic. (2019). *Wwise-2011/Wwise 2019.1*. [Video Playlist]. Retrieved from [https://www.youtube.com/watch?v=g\\_PAUvbBMS0&list=PLXMeprTk4ORP8xloWRduLTVp3vYHdZ1Qw](https://www.youtube.com/watch?v=g_PAUvbBMS0&list=PLXMeprTk4ORP8xloWRduLTVp3vYHdZ1Qw). Accessed: 2020-07-03.
- [28] Robinhood76. (2010). *01904 Air Swoosh.wav*. [Audio File]. Retrieved from <https://freesound.org/people/Robinhood76/sounds/101432/>. Accessed: 2020-07-03.
- [29] Selector. (2016). *Sleigh Bells Hit*. [Audio File]. Retrieved from <https://freesound.org/people/Selector/sounds/369506/>. Accessed: 2020-07-03.
- [30] Jtn191. (2020). *Regen-healing3.wav*. [Audio File]. Retrieved from <https://freesound.org/people/jtn191/sounds/514271/>. Accessed: 2020-07-03.
- [31] Langerium. (2009). *Air\_cut.wav*. [Audio File]. Retrieved from <https://freesound.org/people/Langerium/sounds/84616/>. Accessed: 2020-07-03.

- [32] Imbubec. (2011). *1\_Knife\_Slash\_A.wav*. [Audio File]. Retrieved from <https://freesound.org/people/Imbubec/sounds/118792/>. Accessed: 2020-07-03.
- [33] Loopsamples.clu. (2019). *Cmaj7 Bell30.ogg*. [Audio File]. Retrieved from <https://freesound.org/people/loopsamples.club/sounds/483383/>. Accessed: 2020-07-03.
- [34] Nekoninja. (2016). *Shield Guard*. [Audio File]. Retrieved from <https://freesound.org/people/nekoninja/sounds/370203/>. Accessed: 2020-07-03.
- [35] Ecfike. (2011). *Computer Error.wav*. [Audio File]. Retrieved from <https://freesound.org/people/ecfike/sounds/135125/>. Accessed: 2020-07-03.
- [36] InspectorJ. (2017). *Cmaj7 Bell30.ogg*. [Audio File]. Retrieved from <https://freesound.org/people/InspectorJ/sounds/339818/>. Accessed: 2020-07-03.
- [37] Bspiller5. (2019). *BPS-Boxing-Punches-Designed-Face Hit-Medium.wav*. [Audio File]. Retrieved from <https://freesound.org/people/bspiller5/sounds/478150/>. Accessed: 2020-07-03.
- [38] Aiwha. (2013). *Hits on wood*. [Audio File]. Retrieved from <https://freesound.org/people/Aiwha/sounds/190027/>. Accessed: 2020-07-03.
- [39] EthanChase7744. (2018). *Epic Sword Clang 2.wav*. [Audio File]. Retrieved from <https://freesound.org/people/ethanchase7744/sounds/439538/>. Accessed: 2020-07-03.
- [40] Goup\_1. (2013). *Boom Kick*. [Audio File]. Retrieved from [https://freesound.org/people/Goup\\_1/sounds/195396/](https://freesound.org/people/Goup_1/sounds/195396/). Accessed: 2020-07-03.
- [41] ElenZack. (2019). *Breaking Glass\_2.wav*. [Audio File]. Retrieved from <https://freesound.org/people/ElenZack/sounds/500604/>. Accessed: 2020-07-03.
- [42] ToneDock. (2018). *UI-134*. On User Interface Sound Effects. [Audio File]. Retrieved from <https://www.tonedock.com/samples/288>. Accessed: 2020-07-03.
- [43] ToneDock. (2018). *UI-135*. On User Interface Sound Effects. [Audio File]. Retrieved from <https://www.tonedock.com/samples/289>. Accessed: 2020-07-03.
- [44] Gabe Nwagbala. (2019). *Pikachu's Thunderbolt Charge Sound Effect Pokémon The Series The Beginning*. [Video File]. Retrieved from <https://www.youtube.com/watch?v=NL-mr7dnGVY>. Accessed: 2020-07-03.
- [45] Movie Man. (2016). *Thunder Clap Sound Effect HD (Best Thunder Quality)*. [Video File]. Retrieved from <https://www.youtube.com/watch?v=q0HuT5njtRA>. Accessed: 2020-07-03.