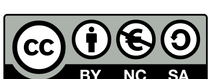
K-Punch!
Implementation of music and its rhythm in order to improve players' coordination, reflexes and focus skills.
Final Degree Project's Report

Jorge Trens Roda

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

July 3, 2023

Supervised by: Cristina Rebollo Santamaría.



To my parents for supporting me during all these years.

Acknowledgments

Thanks to my past self for not giving up and carrying on until the end, to my parents for staying with me and to my friends for helping me to cope with the difficult times.

Thanks to Dreamcatcher for their music, that has inspired mostly of this game and has encouraged me to continue through this project and all previous years.

I would also like to thanks the professors Víctor Jiménez, José Ribelles and Miguel Chover for transmitting their passion for their work through their lessons.

And thanks to Sergio Barrachina Mir and José Vte. Martí Avilés for their LaTeX template for writing the Final Degree Work report that helped so much to get this final document completed.

Abstract

This document contains the report of the Video Games Design and Development Degree's Final Project developed by Jorge Trens Roda. The idea of this project was born due to the student's love for rhythm games and listening to his favourite songs on repeat. The desire of playing them and not being able to find any existent game that has them available came together to create this project. "K-Punch" is a rhythm game where the player has to click the proper button on beat in order to hit a training dummy that will give them points.

Contents

Contents v

1 Introduction 1
1.1 Work Motivation . . . 1
1.2 Objectives . . . 2
1.3 Environment and Initial State . . . 3

2 Planning and resources evaluation 5
2.1 Planning . . . 5
2.2 Resource Evaluation . . . 6

3 Game Desing Document 9
3.1 Introduction and General description . . . 9
3.2 Gameplay Mechanics . . . 11
3.3 Game States . . . 11
3.4 Graphic style and interface . . . 12
3.5 Weapons . . . 14
3.6 Planning . . . 15
3.7 Flowchart . . . 16
3.8 Tools . . . 16
3.9 Arts and Concepts . . . 17

4 System Analysis and Design 19
4.1 Requirement Analysis . . . 19
4.2 System Design . . . 21
4.3 System Architecture . . . 30
4.4 Interface Design . . . 30

5 Work Development and Results 33
5.1 Work Development . . . 33
5.2 Results . . . 55

6 Conclusions and Future Work 57
6.1 Conclusions . . . 57

6.2 Future work 58

Bibliography **59**

A Source code **61**

A.1 Conductor class 61
A.2 Note Manager class (left) 65
A.3 Song Select Manager class 67
A.4 Album Displayer class 72
A.5 InGame HUD Manager class 74

B Discarded Art **81**

B.1 Countdown numbers 81
B.2 Menu buttons 82
B.3 HUD elements 83

CHAPTER 1

INTRODUCTION

Contents

1.1 Work Motivation 1
1.2 Objectives 2
1.3 Environment and Initial State 3

This chapter must clearly reflect what is going to be done during the development of the work. Although the fundamental point is to state the objectives of the presented work, it is also interesting to comment on the need, idea, etc., that motivates it, and the state from which it was started [?].

1.1 Work Motivation

This section must describe the gestation of the work, that is, the circumstances that motivated its development. This introduction can be generic, commenting on everything that makes the work useful and suitable.

In short, this section should make clear why the work has been done.

Rhythm games is a very wide genre, making it very versatile. It is like a malleable core for a game that can cover tons of possibilities. A rhythm game can just be a piano simulator, a platform game or even a fighting game.
That's where the idea of this game was created, what about fusing rhythm and fighting games? Creating a game like this would let me explore and play with a lot of game development features that I am interested in (how does rhythm games manage to synchronize the music with the gameplay, implement in-game VFX and design a smooth

and pleasant UI).
Adding all that gamedev interests to my music ones define the concept and motivation to carry out this project.
Furthermore, lately there has been kind of a trend around some "gamers" who consider that a game has to be life changing in order to be a good game, so I want to make an statement against this, and remember that arcade games with an entertainment target only can also be good and funny games that can help people to disconnect or challenge themselves trying to beat their own record scores.
Apart from those initial inspirations and motivation, lately I have been interested in neurodivergence and how neurodivergent people perceive the world in a different way (specially people with ADHD) [6] [7]; and considering that games help people to develop their reflexes, focusing skills and psychomotor aptitudes, I would like to try to develop something that can not only amuse but also help people.

1.2 Objectives

This section is essential so that the reader can understand what has to be accomplished by the work. Everything that was intended to be achieved must be described with objectivity and precision.
Also, it should be indicated in this section if the work is part of a larger job, if it is complemented by others, etc.

In short, the readers of this report, after reading the work objectives, should clearly know what the author aims to achieve on his own.
The main objectives I stated at the beginning are very simple ones but I think that are a solid foundation for a game to start building it and adding layers in order to get an enjoyable experience. On the other hand, some objectives are kind of new for me since I have never tried something like this during these previous years.

- Achieving a gameplay where the audio system and the inputs are synchronized: The most important feature in a rhythm game is obviously the music and how to follow its rhythm, trying to make it happen in the most organic way possible.
- Adding some VFX: I have always been interested in visual effects, and trying to add some to my game to add a little bit of punch, vibrancy and atmosphere.
- Getting a complex and coherent menu system: Gameplay is the most important part of a game obviously, but having a coherently connected and intuitive menu in order to get to that gameplay in a satisfying way is something that I consider is really interesting and relevant too.
- Getting all the elements and programming done by myself: This might sound pretty obvious but having everything done from scratch and without needing any asset for such an important project is something that makes me feel realized.

1.3 Environment and Initial State

Every work is developed under a set of given circumstances (the way of working, the technical and human team, etc.) and start from a given initial state. This section should reflect both. Furthermore, all decisions made before starting the work or externally imposed (those that have not been made by the author of the work) must be clearly stated.

The idea of making a rhythm game has been roaming around in my head for quite a few years, I remember starting playing games as osu long ago and trying to make my own beatmaps, but quitting after realizing it required too much time, but as time passed I wanted to play the songs I liked after being released and depending on mapmakers to get them done was a little frustrating, so I started to think about making a rhythm game just for being able to play them.
Taking advantage of having some experience developing games now I thought that it would be a great project for my FDP, and taking it as a starting point maybe I can continue to develop it in the future.

Chapter 2

Planning and resources evaluation

Contents

2.1 Planning 5
2.2 Resource Evaluation 6

This chapter is the most technical part of the work. All engineering work must be understandable and assessed from objective information that should follow a given pattern. Such information should appear in this chapter.

2.1 Planning

In this section, the detailed time planning of the work, including all its tasks and subtasks, the dependencies between them and the possible dependencies with other jobs or external events, should be shown.

This project workflow is clearly divided into two blocks. This occurred because around christmas a little break of about half a month was taken to disconnect.
The first block was essentially the creation and beginning of the project, so it's a more conceptual part.
The scope of this project was focused on getting functional code at first and continuing adding layers to refine the results over and over.

Rhythm game programming and research (20 hours)
Notes spawn and music synchronization (30 hours)
- Song Mapping (20 hours/ song)

Pre-Start programming (10 hours)
Song Selection ver. 1 Design and Programming (50 hours):
- Learning about Scriptable Objects (10 hours)
- Creating MVs clips and 2D assets (15 hours)
- Creating the Songs (5 hours)
- Programming Song Selection (20 hours)
GDD writing (10 hours)
Character creation(10 hours)
Character animations creation (10 hours)
Character implementation (15 hours)
Song selection ver. 2 Design and Programming (25 hours):
- 3D Album Covers (10 hours)
- Skybox learning and creation (10 hours)
- Redesign and programming Song Selection ver. 1 with new components (5 hours)
Menus navigation design and programming (70 hours):
- Dotween extension learning (10 hours)
- Main menu design and programming (10 hours)
- Settings and info menu design and programming (15 hours)
- Weapon selection menu design and programming (10 hours)
- Pause menu programming (5 hours)
- Menu navigation programming (10 hours)
- Redesigning 2D assets (10 hours)
Weapons design and programming (10 hours)
Environment and Atmosphere Design (20 hours):
- Terrain creation (5 hours)
- Shaders and particles programming (15 hours)
Documentation (50 hours)
Total aprox: 350 h.

This distribution will be more visually displayed in the Gantt Diagram. (Fig 2.1)

2.2 Resource Evaluation

The costs of the work that is going to be undertaken must be estimated in advance. The human and equipment costs must be quantified so that the work can be assessed and so that, in a real case, the economic viability of the work could be evaluated.

In this section, the resources evaluation (i.e., the human resources and the equipment necessary to develop and implement the work) should be described, as well as the estimated cost of this resources.

This point will display an approximation of the monetary resources invested in order to develop this project: The hardware used in this project was around 1400€ when it was originally bought, being the most relevant parts of the computer:
- CPU: Intel i7-8700

- RAM: 16GB DDR4
- GPU: Nvidia GTX 1060 6GB

On the other hand, this is the software used:
- Unity: Free. This is the game engine chosen.
- Visual Studio Code: Free. The IDE chosen for programming the game code.
- Blender: Free. The 3D models used in the game are created and texturized with this tool.
- Mixamo: Free. The tool when the character animations were created.
- Github: Free. Software needed to have a backup in case of emergency.
- Adobe Photoshop 2022: 24,19 €/month. The edition software for 2D art chosen. I have been using Photoshop for about 15 years and it is the 2D editor software I am most comfortable with.
- Adobe Premiere Pro 2020: 19,66€/month (Student's version). Video editor used to crop the previews and helped me in the process of creating the beatmaps.
- Google Sheets: Free. The beat calculator is made in a google sheet document.

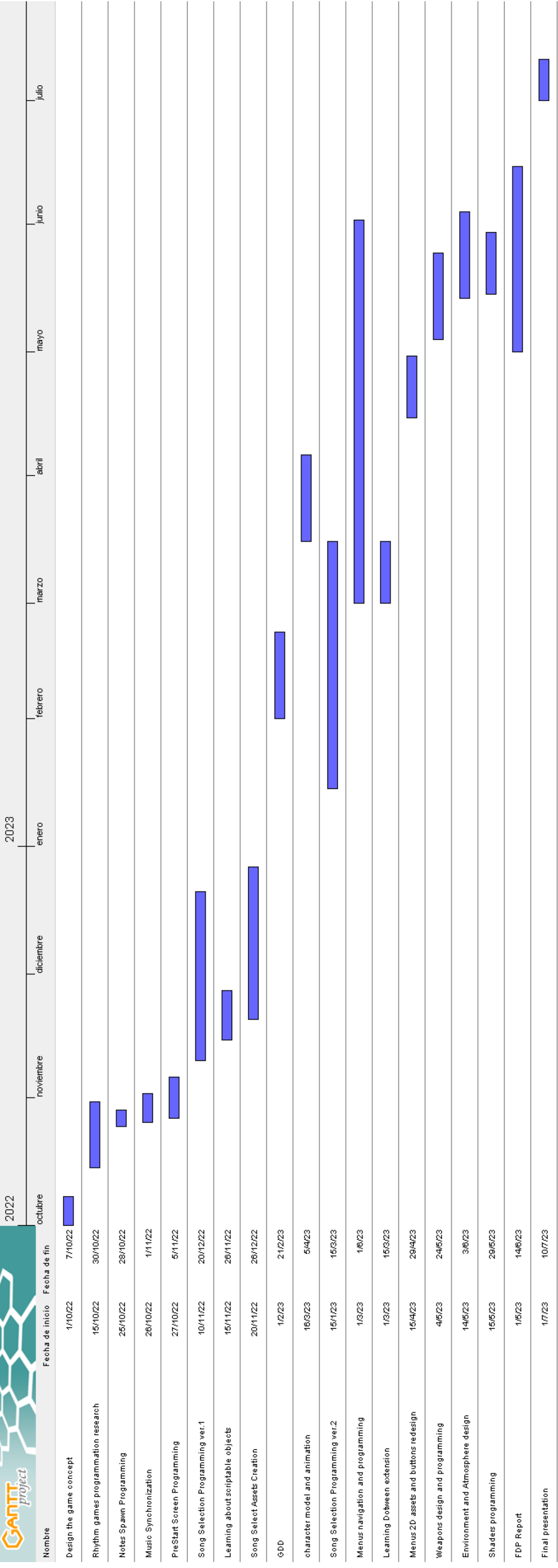


Figure 2.1: Gantt diagram of the full project.

Chapter 3

Game Desing Document

Contents

3.1 Introduction and General description 9
3.2 Gameplay Mechanics 11
3.3 Game States 11
3.4 Graphic style and interface 12
3.5 Weapons 14
3.6 Planning 15
3.7 Flowchart 16
3.8 Tools 16
3.9 Arts and Concepts 17

In this chapter, the conclusions of the work, as well as its future extensions are shown.

3.1 Introduction and General description

Title: “K-Punch!” / “Left, Right, Kick, Punch” (WIP)
Developed by: Jorge Trens Roda
Platform: PC
Genre: Rhythm game
Age Rating: +3 Yrs

This is an arcade rhythm game where the player has to click the proper mouse button following the song beat. The notes will approach from both sides of the screen scrolling to its center. If the player doesn’t miss any beat they will enable a combo bonus, stacking extra points. The goal is to score the maximum “Rhythm Performance

Points" in order to get the best grade possible after finishing the level.
The song will have been chosen by the player previously at the "song selection screen".

Most mainstream and simple rhythm games (based on mechanics and peripherals needed to play) usually consist of a scrolling track, or beatmap, approaching downwards (or in any other direction) until a determined checkpoint when the player has to click, tap or perform any other action on time. By doing this the player will stack points that are displayed on the screen and get a final score when the song ends. Some famous games following these patterns are *Piano Tiles*, *SuperStar* and *Beatstar* for smartphones and *Guitar Hero* for several platforms (Fig 3.1).

Figure 3.1: Beatstar Screenshot.

In this project this feature will have a little twist inspired by the game *One Finger Death Punch*, a high accuracy clicking game where the player must defeat enemies approaching from both sides of the screen by clicking the proper mouse button once they are close enough. The character will use different kung-fu styles mixed with some weapons in order to create a hectic and dynamic combat gameplay. In "K-Punch!" all these features will be combined so it has a different vibe when playing, being not so simple but not too difficult to get used to.

This project covers two different but not subtractive purposes: developing a funny rhythm game for people who enjoy them and also like to listen to Korean music, and trying to create a tool that can help people with focusing skills issues (such as ADHD). This is why the age rating covers from children to any age adult that wants to try it, being an intuitive and user-friendly game.

3.2 Gameplay Mechanics

The main mechanic of this game is very simple, as it was explained before, the player has to click the proper mouse button when a note hits the checkpoint displayed; depending on the side that it approaches from. By doing this continuously without missing, a combo will be started and the player will be able to get a higher score. To make this feel better, the character will execute different chained moves as if a fighting game was. This is the core mechanic of the game, and this will be seasoned with multiple layers of items to make it more visually appealing and epic.

3.3 Game States

Main Menu screen: The game will always start at this screen, and will display it after finishing by default. It contains three buttons: the "Play" button that leads to the "Song selection screen", the "Options" button which opens the "Options and Settings" menu and the "Exit" button that shuts the game down.

Options and Settings screen: A menu when the game volume can be adjusted, and a game info panel will be displayed.

Song Selection screen: This screen will display the songs that are available to play one by one in a carousel with a preview panel that contains the name of the song, the group that sings it and a snippet of the song. It also contains a button to return to the "Main Menu screen" and a button which opens the "Weapon" menu (WIP) (Fig 3.2).

Weapon Selection screen: This is a feature that will be added once most of the game is finished in order to optimize the rest of the game elements. This song will display an example of the weapon that is selected, the rest of available weapons and a "Play" button that starts the game and leads to the in-game screen.

Game screen: The gameplay takes place in this screen and will display the HUD and all the elements concerning the gameplay. The "Pause" menu can be opened by pressing the ESC key on the keyboard.

Pause screen: This tab will contain a button to resume the game, an options one which opens the "Options and Settings" menu and another one to return to the "Main Menu screen".

Post Song screen: This will be displayed after finishing a song. It will contain a player's performance summary of the song and a "Replay Song" button and a "Main menu" button.

Figure 3.2: Song Selection screen.

3.4 Graphic style and interface

The graphic aspect of this game can be summed up with three keywords: Asian, retro and vaporwave (Fig 3.3).
Due to both of the main inspirations for this project (musical and mechanical) having relationship with the Asian culture, some environmental game elements will inherit from it (such as Toriis from Japanese culture, Sawon from Korean culture, etc.). This will be mixed with a vaporwave aesthetic in order to have kind of a retro-futuristic appearance with a vibrant color palette.

Figure 3.3: Main menu screenshot.

The game interface will be as simple as possible in order to not distract the player from the action that is taking place (Fig 3.4). It will just have a “combo meter” over the character and the score displayed on the right top corner of the screen. Some visual feedback will be given to the player when they hit or miss a note too, popping on the

screen over the proper dummy.

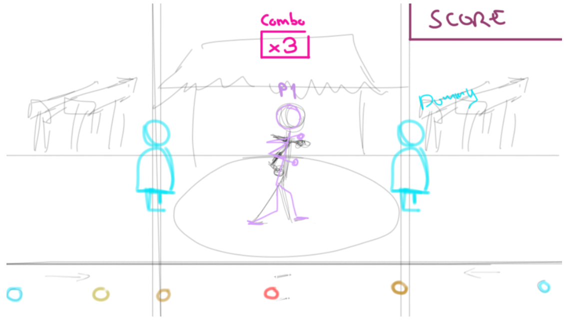


Figure 3.4: In-Game HUD sketch.

In addition, to make the game more dynamic when playing correctly some combos and animations will be displayed as a visual reward. To get that, using the Visual Effect Graph and Shader Graph tools that unity provides, some visual effects (Fig 3.5) will be added depending on the weapon that the player is using at that moment.

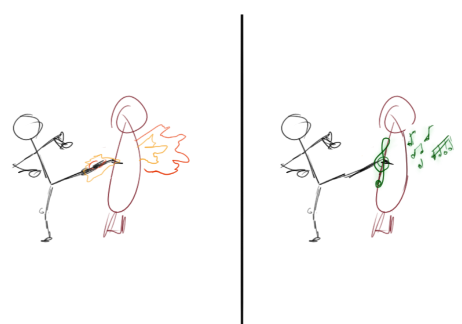


Figure 3.5: Effects example sketch.

3.5 Weapons

As this game does not have a story with a deep background, intense lore or charismatic characters, the elements that can bring some more variability to the gameplay style are the different weapons that will be available to be chosen (Fig 3.6). They will not interfere with the way it is played mechanically, just in a more visual way.
The planned weapons to be introduced are the following ones:

Unarmed: The default way, the character will attack by kicking and punching.

Blinks Passion Hammer: A heavy weapon similar to a warhammer that will be wielded with both hands.

Insomnias Perseverance Staff: A long ranged melee weapon similar to a scepter that will be also dual handed.

Orbits Strength Magic Wand: A magic ranged weapon that will throw magic spells.

Lullets Hope Laser Guns: Dual laser guns that will shoot different laser attacks depending on the gun used.

Figure 3.6: Weapons concepts.

3.6 Planning

The expected time spent on each activity (Fig 3.7).

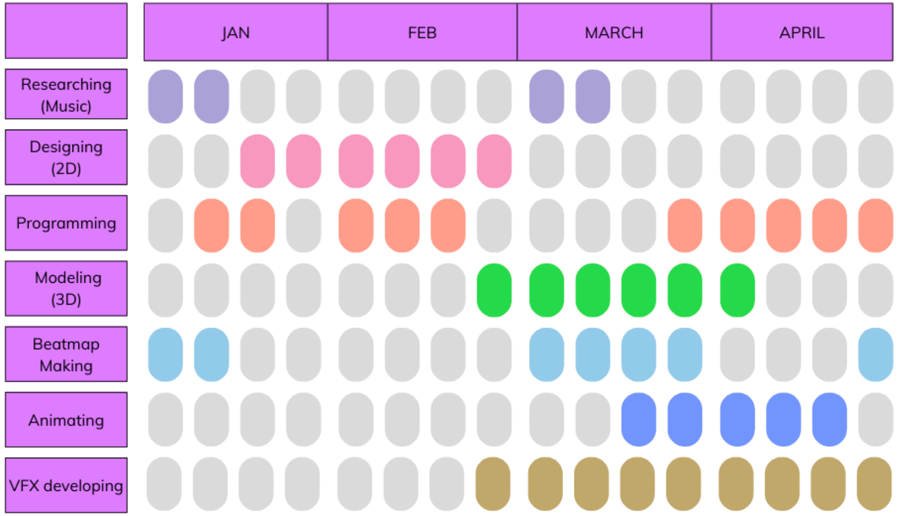


Figure 3.7: Gantt diagram.

3.7 Flowchart

The flowchart showing the possibilities the player has when playing the game (Fig. 3.8).

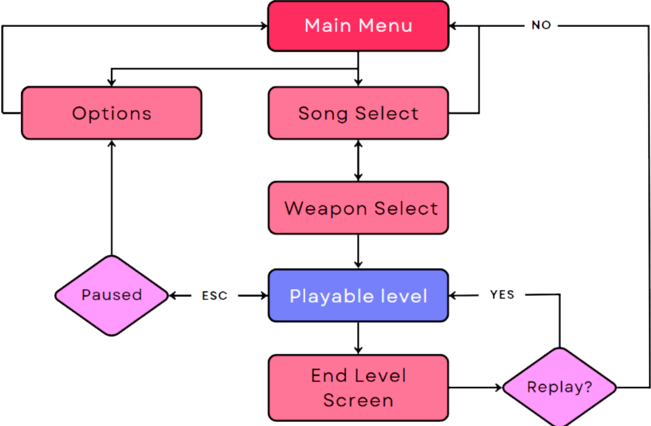


Figure 3.8: Game Flowchart.

3.8 Tools

The main tool used for this project will be Unity3D (ver. 2020.3.27f) as the game engine where it's going to be developed. The 2D assets and art will be designed using Adobe Photoshop and the 3D assets and elements will be modeled using Blender.
Adobe Premiere Pro will also be used to help with the creation of the beatmaps and clips.

3.9 Arts and Concepts

Some elements and discarded concepts (Figs 3.9, 3.10, 3.11).

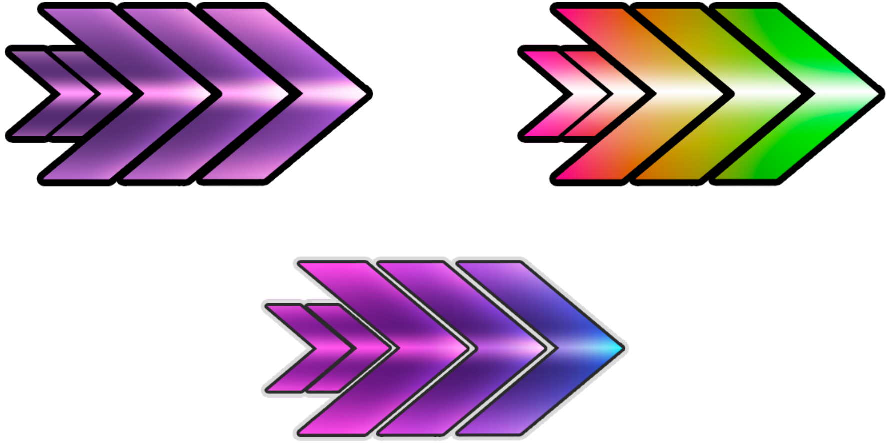


Figure 3.9: Menu arrows.

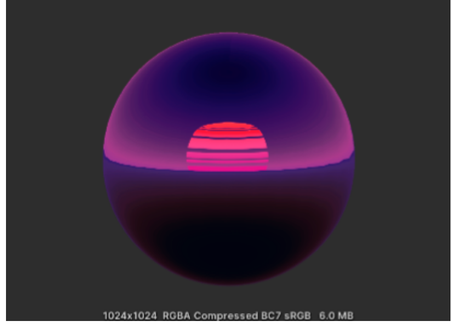


Figure 3.10: Skybox.

Current progress of the game screen (Fig. 3.12).

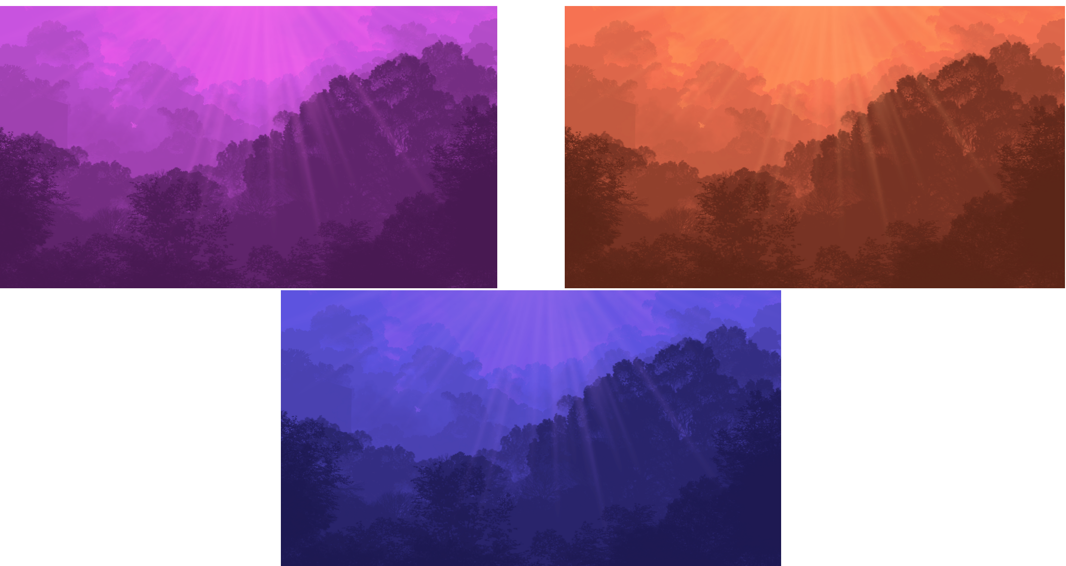


Figure 3.11: Background images.

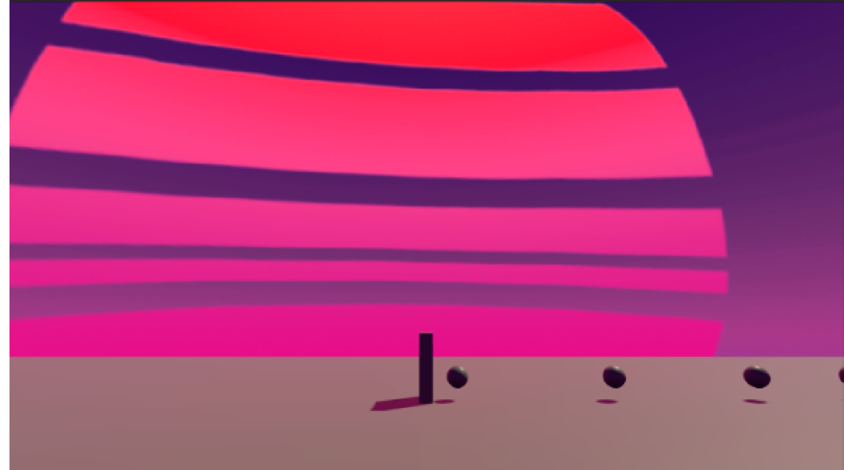


Figure 3.12: In-game screen.

Chapter 4

System Analysis and Design

Contents

4.1 Requirement Analysis **19**
4.2 System Design **21**
4.3 System Architecture **30**
4.4 Interface Design **30**

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

4.1 Requirement Analysis

To carry out a job, it is necessary to perform a preliminary analysis of its requirements. In this section you should detail the functional and non-functional requirements of the presented work.

In you can see an example in which given a certain problem, the creation of an on-line music store, some of its functional and non-functional requirements are obtained. This example also allows to clearly see the difference between the work objectives, which appear as part of the description of the problem to be solved, and the functional and non-functional requirements.

“K-Punch” is a rhythm game where the player controls a character which is training martial arts on the mountain. They have to hit the training dummies following the rhythm of a song in order to get the maximum score.

The first screen loaded when launching the game is the main menu, containing three buttons. The first one is the “Play” button which takes the player to the song selection screen, the second one is “Settings and Info” which opens the settings tab where the player can set the music volume and check how to play the game and the different buffs that weapons supply (the player can navigate through these info slides by pressing the arrow keys on the keyboard); and the last one is the “Quit” button which shuts the game down.

Once the player is in the song selection screen a carousel with the song will be displayed when the player can change the song chosen by pressing the arrow buttons on screen or the arrow keys on the keyboard. There are two more buttons in this menu: the “Settings” button which has the same functionality that the main menu one had, and the “Weapon Select” button which opens the menu selection tab. If they press the Escape key the game will go back to the main menu. In this tab the player can select a weapon to get a buff, or anyone if they prefer to not have buffs and click on the “Play” button to start the game or the cross to close it and return to the song selection.

Once the game is started, a Pre Start screen is the first that appears and will remain on screen until the player press the Return key and the countdown starts, once it reaches 0, the song will start and the player will have to click the left or right mouse button accordingly to the notes to hit the dummies on time. During the gameplay the player can also pause the game with the Tab key, which will open the settings tab which works the same way as in the previous scenes.

When the song ends, the Post song is launched and the player’s score is shown along with the current record for that song. There are two buttons in this screen: the “Main menu” button, which returns to the main menu, and the “Retry” button, that restarts the song.

4.1.1 Functional Requirements

R1) The player can Start the game
R2) The player can Quit the game
R3) The player can change the sound volume
R4) The player can check the controls
R5) The player can preview the songs before choosing any
R6) The player can choose the song to play
R7) The player can choose a weapon
R8) The player can pause the game
R9) The player can return to the main menu
R10) The player can Retry a song when ended
R11) The player can attack the left dummy pressing LMB
R12) The player can attack the right dummy pressing RMB

R13) The player can build up combo following the beat
R14) The player can lose the combo missing the beat
R15) The player can check their score and current record after a song

4.1.2 Non-functional Requirements

R1) The game aesthetics will be vaporwave style with asian elements
R2) The mechanics must be simple to learn
R3) The learning curve must be very accessible
R4) The navigation through menus will be smooth and pleasant
R5) The UI must be coherent and not disturbing for the player
R6) The atmosphere in-game must not be intrusive in the gameplay
R7) The game assets will be made from scratch by hand

4.2 System Design

The case use diagram and its analysis will be explained (Fig 4.1). [18]

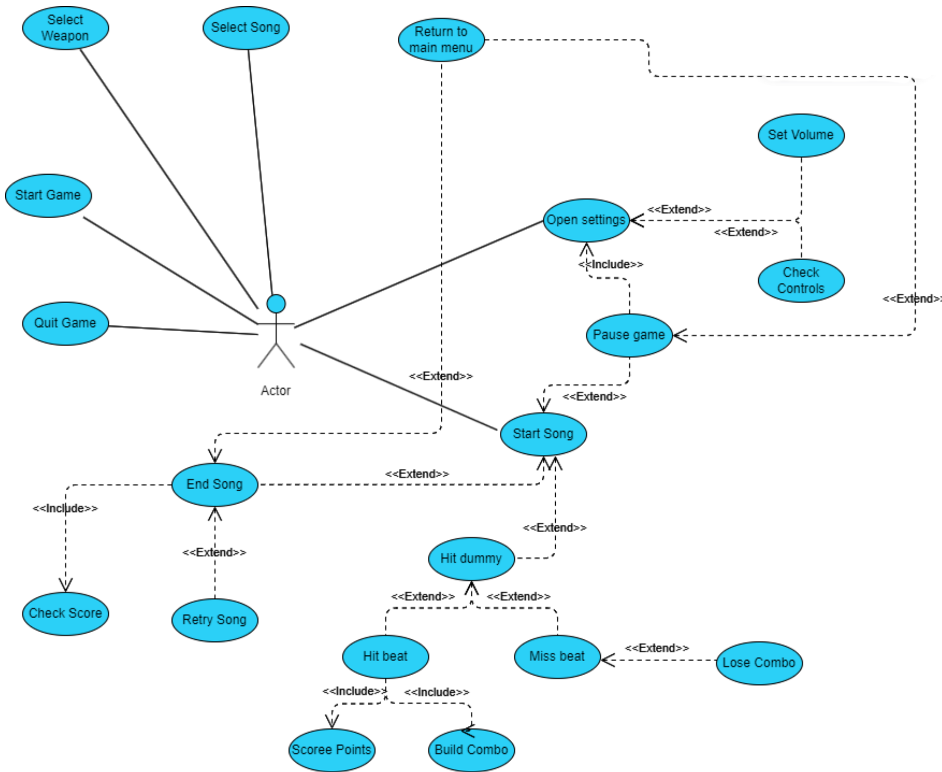


Figure 4.1: Case use diagram.

Requeriment:	R1
Actor:	Player
Description:	The player starts the game and gets to the main menu
Preconditions:	The game must be closed.
Normal Sequence:	1. The player presses the app icon twice. 2. The system launches the game and the player gets to the main menu.
Alternative Sequence:	None.

Table 4.1: Case use «1. Start Game»

Requeriment:	R2
Actor:	Player
Description:	The player closes the game
Preconditions:	The player must be in the main menu.
Normal Sequence:	1. The player presses the button “Quit”. 2. The system shuts down the game.
Alternative Sequence:	None.

Table 4.2: Case use «2. Close Game»

Requeriment:	R3, R4
Actor:	Player
Description:	The player opens the “Settings and Info” screen and can check the controls and set the volume
Preconditions:	The player must be in the main menu, song selection menu or in game.
Normal Sequence:	1. The player presses the button “Settings and Info” 2. The system opens the “Settings and Info” tab
Alternative Sequence:	• 1.1) The player presses the gear button on the top right corner of the Song Selection screen • 2.1.1) The player move the volume slider to choose a volume value • 2.1.2) The system sets the game volume according to the chosen value • 2.2.1) The player checks the controls • 2.2.2) The player presses the right or left arrow key to check the next or previous controls slide • 2.2.3) The system changes the slide to the next or previous one • 2.2.4) If the player presses the left arrow key being on the first slide or the right arrow key being on the last one, the system will not change the slide.

Table 4.3: Case use «3. Open Settings»

Requeriment:	R5, R6
Actor:	Player
Description:	The player previews the songs and chooses one
Preconditions:	The player must be in the main menu
Normal Sequence:	1. The player presses the button “Play” 2. The system loads the Song Selection screen and generate an album displayer with every song stored in the game 3. The player can check the preview panel for the current selected song 4. The song displayed on the panel will be the chosen song
Alternative Sequence:	3.1) The player can press the left or right arrow key on the keyboard or the arrow buttons on screen to navigate to the next or previous song

Table 4.4: Case use «4. Select Song»

Requeriment:	R7
Actor:	Player
Description:	The player can choose a weapon that will give them a buff
Preconditions:	The player must be in the Song Selection menu
Normal Sequence:	1. The player clicks on the button Weapon Select 2. The system loads the weapon selection tab and generate a layout with the available weapons 3. The player clicks the sprite of the weapon they want to use 4. The system indicates the selected weapon to the player displaying a frame around the selected weapon
Alternative Sequence:	3.1) The player clicks the Unarmed sprite button if they want to play with no buffs

Table 4.5: Case use «5. Select Weapon»

Requeriment:	R11, R12, R13, R14
Actor:	Player
Description:	The player can attack the dummies and get points depending on their performance building up or losing a combo.
Preconditions:	1. The player must have chosen a song 2. The player must be in the Pre-Start screen
Normal Sequence:	1. The player must press the Enter key to start a countdown that will start the song 2. The system starts the song and spawns the notes 3. The player presses left or right mouse button to hit the dummies 4. The player Hit the note and scores some points and builds up combo
Alternative Sequence:	• 4.1) The player misses the beat • 4.1.1) The player loses the combo • 4.1.2) The player does not lose the combo due to a weapon buff

Table 4.6: Case use «6. Start Song»

Requeriment:	R8
Actor:	Player
Description:	The player can pause the game by pressing the Esc key
Preconditions:	The player must be playing a song
Normal Sequence:	1. The player presses the Esc key during a song 2. The system pauses the music and notes and opens the pause tab
Alternative Sequence:	None

Table 4.7: Case use «7. Pause Game»

<table>
<tr><td>Requeriment:</td><td>R9</td></tr>
<tr><td>Actor:</td><td>Player</td></tr>
<tr><td>Description:</td><td>The player can return to the main menu if they have finished a song, paused the game or selecting a song.</td></tr>
<tr><td>Preconditions:</td><td>The player must be in the Pause menu, Post song screen or Song selection screen</td></tr>
<tr><td>Normal Sequence:</td><td>1. The player is playing a song
2. The player presses the Esc key to open the pause menu
3. The system pauses the music and notes and opens the pause tab
4. The player clicks on the button “Main menu”
5. The system loads the main menu screen</td></tr>
<tr><td>Alternative Sequence:</td><td>• 1.1.1) The player finishes a song
• 1.1.2) The system opens the Post Song screen
• 1.1.3) The player Clicks on the button “Main menu”
• 1.1.4) The system loads the main menu screen
• 1.2.1) The player is in the Song Selection screen
• 1.2.2) The player presses the Esc key on the keyboard
• 1.2.3) The system loads the main menu screen</td></tr>
</table>

Table 4.8: Case use «8. Return to Main menu»

Requeriment:	R9, R10, R15
Actor:	Player
Description:	The player can check their score and current record, return to the main menu or retry a song after finishing it.
Preconditions:	The player must have played a song until the end
Normal Sequence:	1. The player finishes a song 2. The system launches the Post Song screen after a few seconds 3. The player can check their score on the song, the current record and which weapon was used to get that record
Alternative Sequence:	• 3.1.1) The player presses the button “Retry” • 3.1.2) The system closes the Post Song screen and restarts the song • 3.1.3) The game is at the Pre Start screen • 3.2.1) The player Clicks on the button “Main menu” • 3.2.3) The system loads the main menu screen

Table 4.9: Case use «9. Check Score»

4.3 System Architecture

This section should describe the architecture of the projected system. Aspects that can be included are, for example, hardware and software requirements, components interconnection, minimum requirements for operation —RAM, hard disk, etc.—, provided documentation, etc.

This game is developed with Unity 3D, and using the 2020.3.27f1 version. According to the Unity's manual documentation these are the system requirements (only Windows will be the analyzed OS):

OS: Windows 7(SP1)
CPU: a x64 architecture with SSE2 instruction set support
GPU: A graphic card with DirectX10

Additionally a monitor, a keyboard and a mouse would be necessary.

4.4 Interface Design

If the developed work provides a user interface designed by the student, this section should describe its interface design. The interface mock-ups should be provided in this section.

The GUI in this game is designed in order to not interfere in the gameplay and not distract the player from it but help them instead.
There are three main screens in the game with different GUI distribution: The main menu, the Song Selection screen and the In-game screen.

The main menu is the most simple one, it has just three buttons on the left side of the screen that the player can click on to the menu they choose. If they click on the "Play" button, the game will move to the Song Selection screen; if they click on the "Settings and Info" button the settings tab will open (this menu can be opened on every screen in a different way), and finally if the "Quit" button is pressed, the app will be closed.

The GUI of the Song Selection menu is mostly distributed on the corners of the screen. On the top left corner the player can find the "Weapon Select" button, which opens the weapon selection tab. The settings button is located on the top right corner, this button will open the settings tab as in the main menu. Finally, on the bottom part of the screen, the arrows that let the player scroll through the songs will be found.
If the player clicks on the "Weapon Select" button, a tab with all the available weapons

will be displayed. On the top left corner of this tab there will be a button that closes it and on the right bottom corner, the “Start Game” button will be found. The weapons will be displayed in an horizontal layout and will be selectable. Below, there will be a button that deselect all of them; if a weapon is selected a border will be shown around its respective button and its name will be displayed above it.

Finally, in the In-Game scene the GUI is designed to be the least intrusive possible, some information will be displayed on the top part leaving half of the screen free with only just one element that will help the player to know when to click the proper button. On the top left corner an image of the current weapon will be displayed, on the top right corner there will be an indicator of the score the player has at the moment. On the center of the screen there will be two elements, the combo meter which is constantly displayed and shows the combo multiplier the player has accumulated and a text that will appear when the player hits a note depending on their timing.
The last element remaining is a square-shaped frame placed where the notes converge.
If the player presses the Tab key during a song, the music and notes will be paused and the settings tab will be displayed.

Additionally, before starting playing the song, a PreStart Screen will be displayed until the player presses the Return key on the keyboard, after this happens a countdown will be executed and the gameplay will start.

After the song has ended, the Post Song screen will be launched. It has some information for the player such as their score after the song and the current record that song has, if it has been beaten a message will appear too. Finally on the bottom part of the screen a “Retry” and a “Main Menu” button can be found.

Chapter 5

Work Development and Results

Contents

5.1 Work Development **33**
5.2 Results **55**

The developed work and the obtained results should be made explicit in this chapter. All possible deviations from the initial planning should be detailed and justified. In this way, readers of the memory must be able to understand the possible reasons for discrepancies between the objectives of the work, the planning that supposedly allowed to obtain it, and the final results achieved.

5.1 Work Development

In this section you should explain the most relevant aspects of the developed work. You can follow a chronological order, according to the planning, an order according to the work tasks, or according to the importance of the achieved milestones. It is essential that this section lists the milestones that have been achieved, and the problems that have caused the initial objectives or work planning to be changed.

Only the necessary technical information to clearly understand the work should be provided in this section. This information, in addition, must be presented in such a way that it should be easily understandable by someone who is not specially versed in the topic of the work. This implies that most of the work technical information, which may not be understandable or interesting for the members of the evaluation committee, should be in the appropriate appendices, and be referenced from here when necessary.

In this section all the working effort put in this project will be explained and shredded part by part to get the closest possible to the development experience. This will cover from the very early stages of programming to the last visual touches with shaders and decoration, including the little elements and techniques that helped with the development and were learnt thanks to this project.

The first problem faced was to discover how a rhythm game works and how it can be implemented in Unity due to its own time management and game loop. Thanks to two very useful articles found about music syncing and coding according to the beats of a song, instead of the own Unity method (Time.deltatime) usually implemented when talking about time management, the solution found for this was to use the AudioSettings.dspTime feature, which takes samples from the actual audio system so it does not get desynchronized. [1] [2]

This was a solid base to start building the system, now it was time to get a form of tracking the song along its duration to synchronize it with the future spawning notes. In order to get this achieved, some variables that helped with this issue were created: one for the song position and another one that registers when the player decides to start the song. The song position is calculated by subtracting the second variable to the current AudioSettings time.

However, as it was commented several times before in this document, what was needed to keep a more accurate tracking of the song was to convert everything to beats (since songs are measured in beats). To calculate this, some new variables were needed; the song BPM (beats per minute), information that has to be included manually for each different song, which is used to get the second variable that was necessary: the amount of seconds per beat. Dividing the song position by the seconds per beat will get the result desired, the song position in beats (Fig 5.1). [3]

Following these steps, the tracking method for the song was completed, except for one little detail, as mentioned earlier there is a countdown for the player to get ready when they press the starter button, so a new variable had to be added to the formula: a beat offset to align the song properly (Fig 5.2).

Now that the tracking method was completed, the next step is to get the notes spawned and moving accordingly to the beat. This was relatively easy thanks to scriptable objects [**?**], a very powerful tool that helped a lot in this project development and I wish I had known about them before. Scriptable objects are a class of Unity that can create items which help to store a lot of data in order to save memory and get to make modular designed systems for games.

For the mentioned goal a scriptable object called “Song” was created (Fig 5.3). It contains a lot of important data for each song in the game that is used in the Song

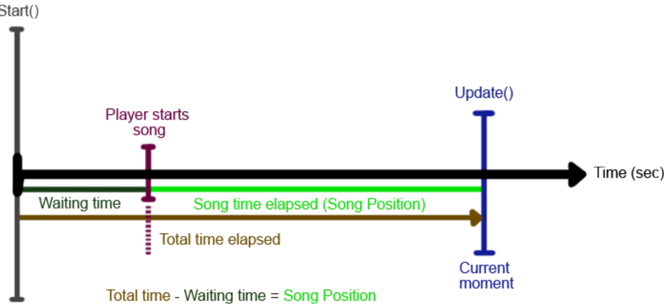


Figure 5.1: Song tracking method.

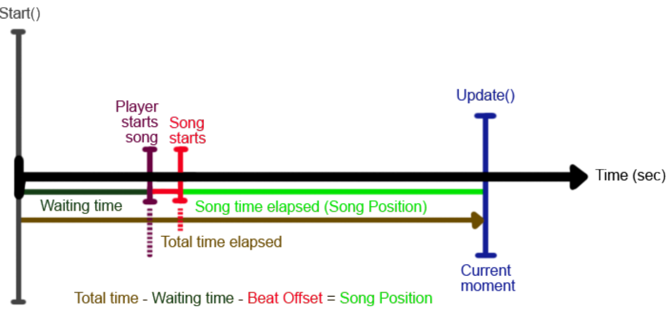


Figure 5.2: Song tracking method with a beat offset added.

Selection screen, which will be addressed later, and In-Game. This object contains the bpm, the audio in a .mp3 file that sounds while playing and two arrays of floats that indicate on which beat will the notes spawn, one for the notes on the right and the other for the left ones.

All the methods that were previously described are contained in the same script, called “Conductor” as it was an orchestra conductor, that controls and has the information of everything happening during the gameplay. This script contains an array of Songs and with the help of an index that indicates which song has chosen the player, getting access to both of these arrays of the song, the notes can be spawned on the beat according to the floats saved in them.

This way, checking if there are still floats in each array and if the song position beat is below the next beat contained, the note will spawn.

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4   using UnityEngine;
5   using UnityEngine.Video;
6
    ...
7   [CreateAssetMenu(fileName = "NewSong", menuName = "Song")]
    1 reference
8   public class Song : ScriptableObject
9   {
        1 reference | 0 references
10      public string songName, group;
        1 reference
11      public float songBpm;
        1 reference | 0 references
12      public AudioClip musicSource, clipSource;
        0 references
13      public VideoClip MvClip;
        0 references
14      [SerializeField] public GameObject album;
        0 references
15      [SerializeField] public Sprite songCover;
        0 references
16      [SerializeField] public int recordScore;
        0 references
17      [SerializeField] public int recordWeapon;
        3 references
18      [SerializeField] public float[] songNotesRight;
        3 references
19      [SerializeField] public float[] songNotesLeft;
20  }       You, 1 second ago • Uncommitted changes
```

Figure 5.3: Song scriptable object code.

The spawning system was created and the notes themselves were what was needed. The solution was pretty obvious, prefabs. After designing and implementing a note object it was transformed into a prefab with a variant prefab (one for the notes on the right and one for the ones on the left). This element controls the movement of the notes spawned and the score the player gets when they hit the dummy.

The movement of each note is managed with the Unity lerp function, it takes the position where it's spawned and the ending position on the center of the screen and interpolates it according to the beats. This depends on a custom variable which determines how many beats in advance are the notes spawned, the more beats in advance the slower is

the movement.

The score system decided for the game was simple, everything was measured in beats, so that was what would be used to decide the amount of points would be given. The target of this project is to get a relatively easy game, at least at the moment, so taking that into account the different score thresholds were created.

With the note object created and a method to instantiate it whenever it was needed the next step was to create the beat map of a song.
At the beginning it was a bit chaotic, to select where every note would spawn during the song, the audio would be imported in Adobe Premiere and a marker would be placed on every frame that was considered to follow the rhythm. As mentioned at the beginning of the chapter, everything in the game is measured in beats, so there was a need to transform those frames into beats. By getting the timestamp where the marker was (expressed in min:sec:frames), it was moved into a frame calculator which would calculate the amount of total frames [4], then by taking the song bpm and getting the frames per beat only dividing the total frames by this last figure was left. This was calculated with a calculator and written down on a paper sheet and was very inefficient.
Furthermore, after trying twice and failing with the synchronization due to the amount of calculations made one by one that took to some mistakes a new way was designed (Fig. 5.4).

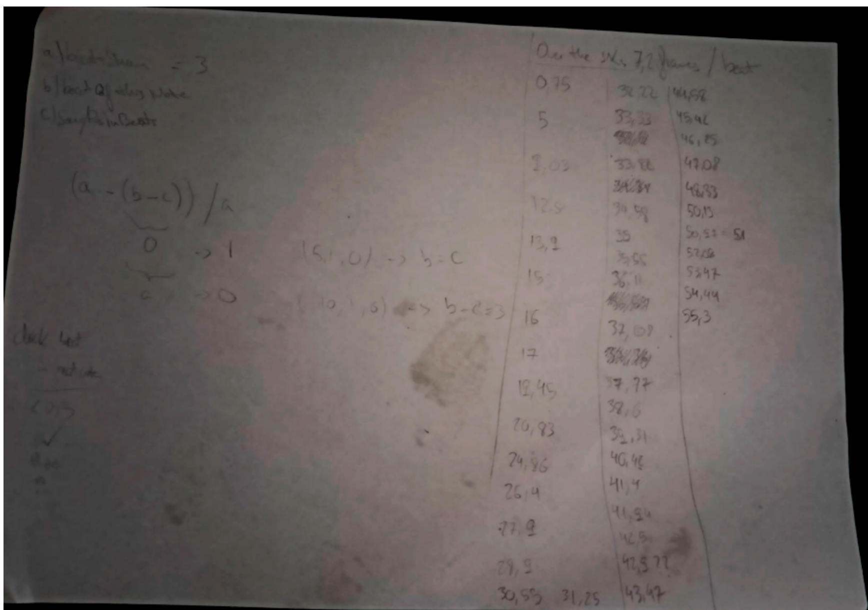


Figure 5.4: Beats calculated by hand.

It was easier to create a new way to automate all the calculations and make it modular so it can be expanded the amount of times is desired. This was called the Beat Sheet and just by writing the song bpm on the proper cell the Beats per Frame were calculated, then the total frames obtained by pasting the timestamp of the marker in Premiere were multiplied by those BPF and the exact beat when each note should spawn was achieved. This was a bit more tedious work but less than all the previous calculations made by hand that could easily lead to more errors (Fig. 5.5).

	A	B	C	D	E	F	G	H
1	BPM		Minute	Second	Frame	FRAMES		BEAT
2	169		0	0	6	6		0,7041666667
3			0	1	0	24		2,816666667
4	BPF (Beats Per Frame)		0	1	12	36		4,225
5	0,1173611111		0	30	0	720		84,5
6			1	0	0	1440		169
7			1	0	15	1455		170,7604167
8			1	35	0	2280		267,5833333
9			1	40	20	2420		284,0138889

Figure 5.5: Beats calculated with the Beatsheet created.

After implementing all those elements, the skeleton of the gameplay was finally constructed, the machine was working, it was time to build the way to get there for the player, starting from the Song Selection menu.
The design of this screen menu was inspired by the "typical" character selection or song selection of an arcade game where the chosen option is in the center of the screen and another option on each side. The main idea was to place the cover of the album that contains the song and its song name and artist below. To get this "carousel" effect, a function that changes an index was created, with this index the correct song from the array was selected, and with two more auxiliary indexes the surrounding songs were updated (Fig 5.6).

Figure 5.6: Initial version of the Song Selection screen.

This was the very first version of the song selection screen but, in this project, the goal was to achieve the most polished and attractive game as possible, and this seemed very poor looking. With a lot of curiosity about GUI elements and the decision of transforming this into a 3D selector, the beginning of the Song selection ver 2. started (Fig 5.8).

The first step was to learn about DoTween [5], a very useful plugin for Unity that is used in UI elements to create animations and effects that give the game a better flow and visual appealing.
Using this plugin a little effect was added to some objects such as the selection arrows and the countdown numbers from the In-Game scene in order to be a little more dynamic.
The next action made was transforming the boring 2D selection screen into a 3D carousel that was generated automatically depending on the amount of songs stored in a “song pool” array. The use of the scriptable objects was a really powerful tool to store all the necessary information for the script that created all the songs in-game.
Now, instead of a simple image of the album cover and the name of the song, there was a display with a preview of the song and a 3D album rotating that was so much more eye-catching. Each time the arrows were pressed all the albums rotated around the center and the one coming to the front instantly faced the user (Fig. 5.7).

Figure 5.7: Album case in 3D.

From those Song objects mentioned before, the album object, the name of the song displayed, and both the audio and the video clips, previously cropped with Adobe Premiere, would be easily stored and accessed whenever the song was changed just with the song index with the song pool array position.

After implementing all the necessary code that made this work, it was time to face a new issue: the communication between the song selection menu scene and the In-Game scene.

Figure 5.8: Second version of the Song Selection screen.

“PlayerPrefs”, a very interesting tool of Unity, was used for this. Since this powerful class not only was useful to store data from one scene to get it from another but it was also stored after the game is closed, meaning that the variables set and stored with this tool would remain the same when it is relaunched.
Using the PlayerPrefs class the index of the selected song in the song selection menu could be easily fetched in the In-Game scene in order to load the proper song.
This class will be used again later with some more menus.

Once those two scenes were created and working properly one of the most important parts of the game still was left: the main menu. The first screen that appears when the game is launched was the next goal to achieve.
This was more a 2D design focused task due to all the buttons and background art that had to be created. Since the chosen aesthetic style was “Vaporwave” some images were used as reference to elaborate the background image.
After making it, it was decided that the design of the buttons distribution was going to be like a strip over the background that also contained the game logo. So that’s how the logo for the game was created. Since the game was named “K-Punch”, the logo was simple and to the point (Fig 5.9).

After placing all the buttons and visual elements on the screen, it was time for coding.
Out of the three buttons in this scene, two were just in charge of changing to the proper scene of the game when clicked, but there was a button that would need to open the settings menu, so there was the new task.
The settings tab contained a slider that was meant to control the volume of the music and some slides that explained with some basic drawings and text how to play the game.
Following a tutorial [19] that showed how to control the volume of the game in Unity using its own audio mixer the volume slider was implemented.

At this point most of the functionality of the game was ready so it was time to start

Figure 5.9: K-Punch! logo.

focusing on the visual section.
The first aspect that was upgraded was the skybox, it was interesting to learn how to make a skybox from scratch, designing it in Photoshop and editing it to adapt to the deformation of the skybox shape was a bit challenging but funny in the end (Fig 5.10 & Table 5.1).

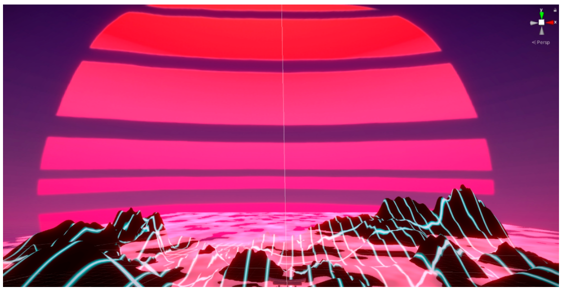


Figure 5.10: Landscape with the skybox.

With a better background for the scenes the next step was to start modeling and customizing the elements In-Game.
The first one was the character since at that moment it was just a cylinder which was used to indicate where the notes should go. As mentioned at the beginning of this document, this game has inspiration from "One finger Death Punch" (Fig 5.11), a game where all the characters are stickmen. So the idea was to make a 3D stickman.
After watching a couple tutorials of simple character modeling in blender the character was modeled and implemented (Fig 5.12). [20] [21]

Once it was implemented in the project the next step was to get some movement, the animations were to come. At first, there were a few tries to animate the character with

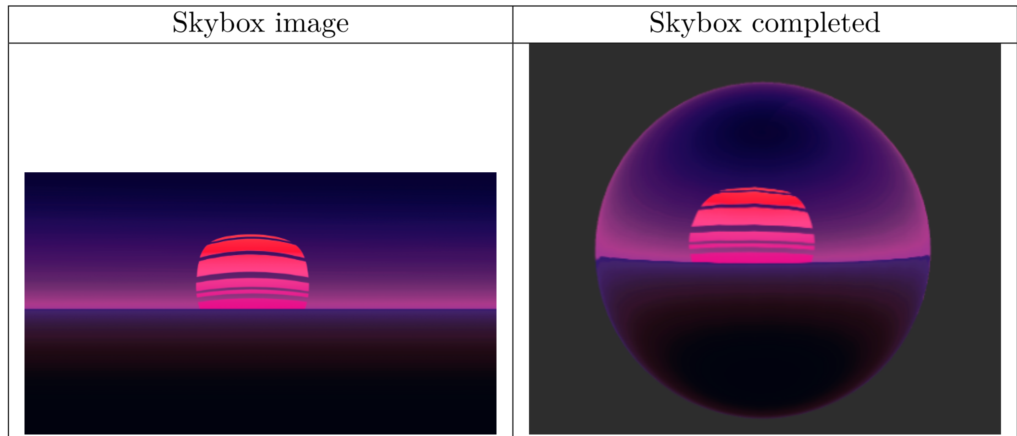
Skybox image	Skybox completed
	

Table 5.1: Skyboxes

Figure 5.11: One Finger Deathpunch screenshot

the blender animation tool [15] but after failing very hard several times and wasting too much time a new decision was taken, get the animations from Mixamo. [22]
The idea was to make a loop of animations to combine them with some VFX so it seemed that the character was making some combos if the same button was pressed repeatedly. So the sequence of 4 movements chosen was starting with a left punch followed by a right punch, then a left punch again and finishing with a kick. so some animations of these strikes were needed in addition to the idle animation of the character (Fig 5.13).

After getting all the animations done and implemented properly in the project, programming the sequence was left. After making some changes to the animation depending on which side the character was looking at they were ready to work (Fig 5.14).
Due to the fast pace of the game it was necessary to adjust the time of those animations in order to fit in the short amount of time between notes.

Now that the animations were in the game, a visual effect after performing the kick

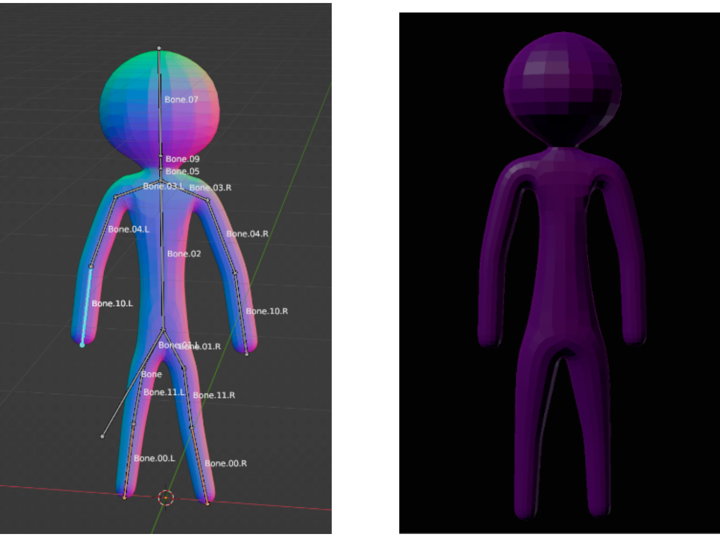


Figure 5.12: Character model in blender and Unity.

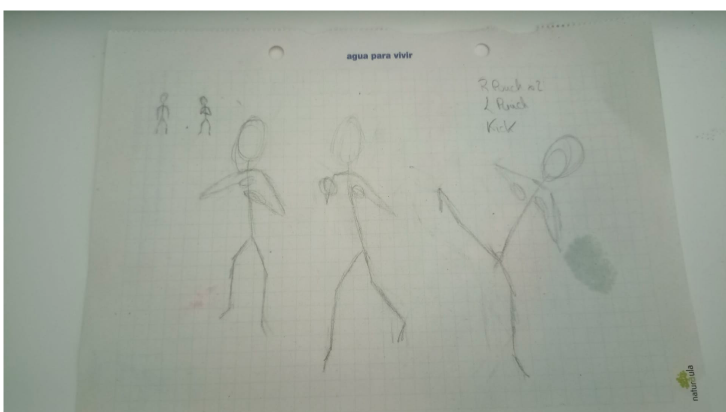


Figure 5.13: Animation sketches

was implemented. It was a flame that came from the dummy when kicked and it was made by following a tutorial from a VFX youtube channel (Fig 5.15) [13]. Sadly, it had to be removed because unknowingly it was causing the PC to crash and causing a lot of BSoDs.

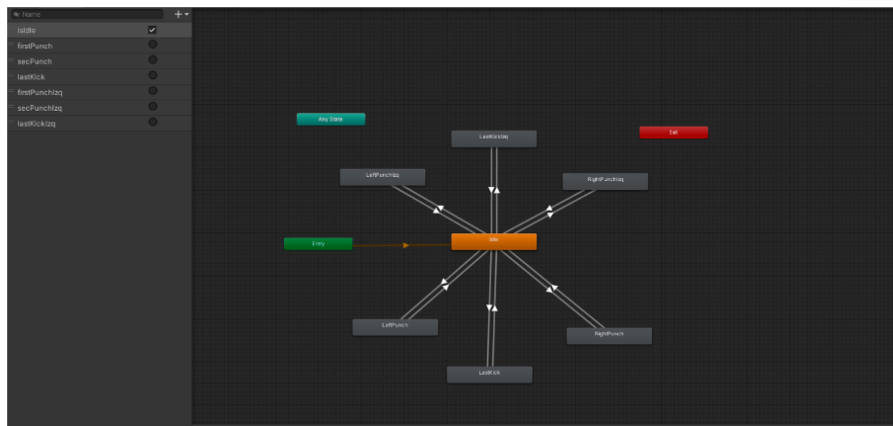


Figure 5.14: Unity Animator of the character

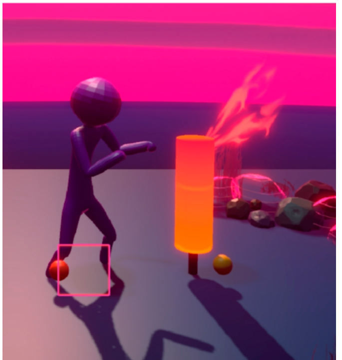


Figure 5.15: Flame VFX made.

On a second try, another visual effect was developed (Fig 5.16), a lightning striking the floor (Fig 5.17), but it caused several problems too so it could not be included either [23]. (It seems that the VFX Graph has some problems with this PC, so to avoid any more BSoDs it was not used anymore)

After having a character that could hit the dummies it was necessary to have maybe the most important part that a game must have, the motivation element. There must be something in a game that encourages people to play it or that can be used to create motivation.

Usually rhythm games are arcade games, so the main motivation to play them is just the fun element, existing games with such diverse mechanics some players prefer one over another, but to add something more challenging they use to have an internal (even global if it is an online game) ranking, so it was time to add the scoring system.

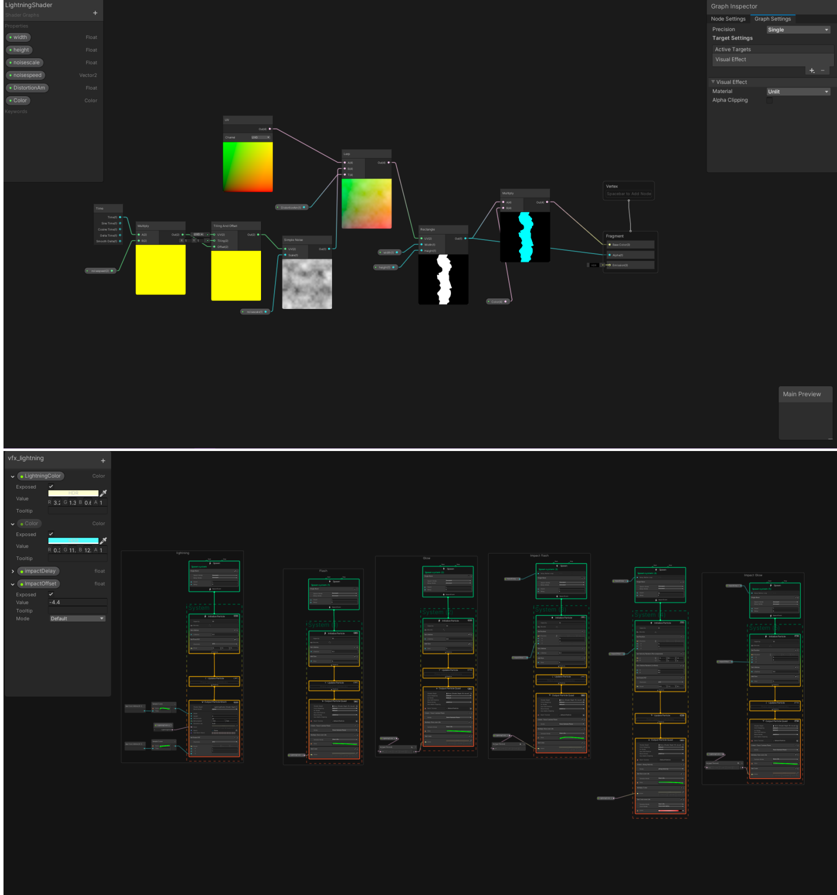


Figure 5.16: Lightning Shader and VFX.

Since the score had to be displayed on the screen the rest of the HUD was set up too.

Implementing the score was not very complicated thanks to the note prefab that was previously created which controlled the timing of the player, only adding the amount of points for each possibility, and coding all the communication between the HUD element In-Game and the notes was needed. After that, the combo mechanic was added.

The combo mechanic is a very typical part of rhythm games, so to implement it some variables were added to control the amount of hits the player could get in a row, the multiplier score and the HUD element. The more notes were hit without missing the more score is given to the player for each note.

Adding these two elements made the game already playable, but there was another element that could add a bit of spice to the game style. The weapons were a proposed feature that was time depending. Due to the restrictions and deadlines they could not

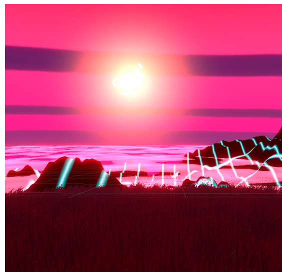
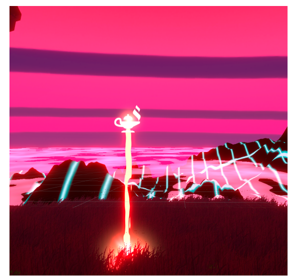
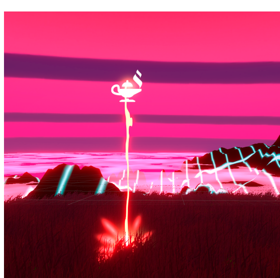




Figure 5.17: Lightning visual effect animation.

be included as they were first designed, because it required an insane amount of work for the modeling, animating and finally implementation, but they could be added in a different and more adapting way: as buffs.

With this in mind, a weapon select tab was added to the song selection menu, opening a tab that generated a layout with all the available weapons in the game (Fig **??**). This contained a button for each weapon that could be clicked with its sprite and the name of the weapon on top of the menu. Everything was stored in a Lightstick object made with the scriptable object class again.

Figure 5.18: Weapon Selector.

Now adding an array and some variables to the Conductor script the effects of each weapon could take effect. Using the PlayerPrefs tool the selected weapon can be stored so the Conductor can know which lightstick was chosen and with that information an indicator could be placed on a corner showing which weapon was being used (Fig 5.19). Then the proper changes to the scripts taking into account the buffs of each weapon were done and, with all this working, new playstyles were unlocked.

These are the weapons and their different buffs (Fig 5.2)

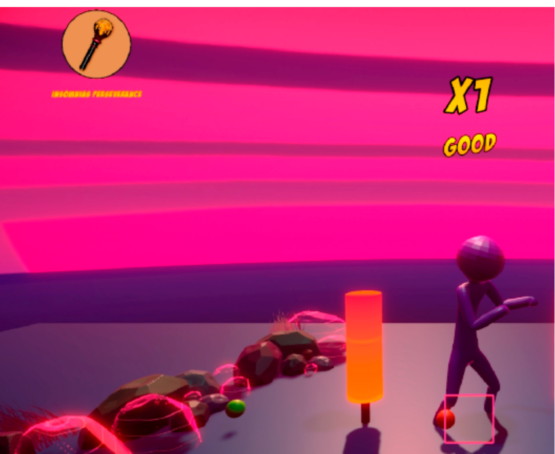


Figure 5.19: In-Game weapon indicator.

Now that the player could have different buffs that would affect their score in several ways, a screen where the player could check their final score and even compare it with the local record was decided to be implemented.
In this "Post Song" screen the player would be able to replay the song or just exit to the main menu (Fig 5.20), which are two very simple functions. The interesting task here was to add the record score feature and update it if beaten.
At first it was implemented by using the PlayerPrefs with a record variable, but not much later it was discovered that there was an issue by doing it that way, it could only store one record, which was enough for the demo, but if the player chose a different song, the record score for all the previous ones were completely deleted, so there had to be a solution. This was once again achieved using the song objects and storing inside them the record score and the lightstick used when it was won.

Figure 5.20: Post Song screen.

Weapon	Effect
Blinks Passion	X2 combo multiplier
Insomnias Perseverance	Double the combo limit
Lullets Hope	Can´t lose combo
Orbits Strength	X2 note score

Table 5.2: Weapons

Before starting with the final 3D part a redesign of the UI elements was made, changing most of the buttons and tabs and adding some new ones so everything was more coherent (Fig 5.21).

Finally the gameplay and the functionality were completed, only the graphic aspect was left and it was time to learn about shaders.
Shaders are one of the most interesting parts in game development, fusing art and coding.
Here, some of the ones created for the project will be explained.

Figure 5.21: Final version of the Song Selection screen.

First of all, the idea for the game was to be placed on a training spot on the mountains, logically surrounded by more mountains.
To achieve this, the Terrain tool of Unity was used [17]. It was relatively simple to understand how to use it so a decent result was obtained after some time practicing with it (Fig 5.23).
The next step was to add a material, and following the vaporwave aesthetic, a dark shade with some kind of technologic lines is the typical one used in that style (Fig 5.22).

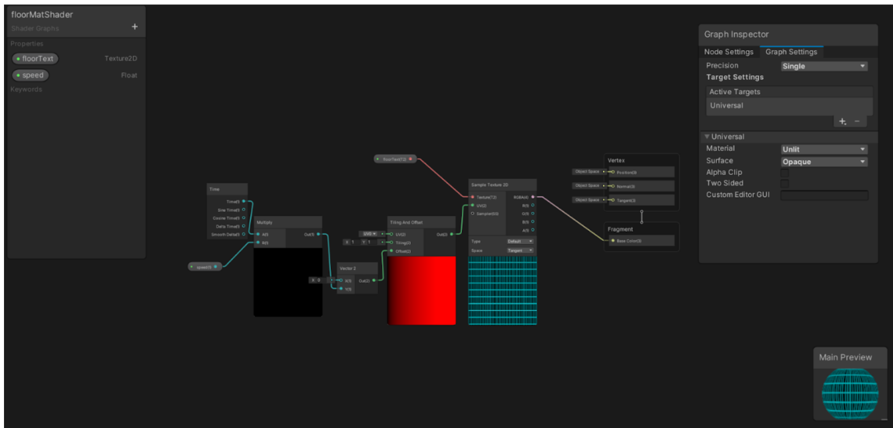


Figure 5.22: Background Shader.

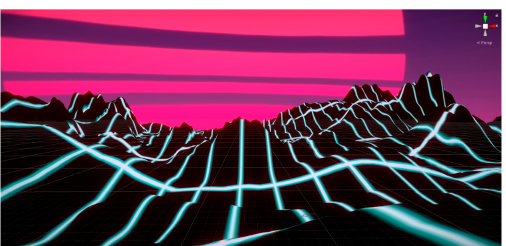


Figure 5.23: Background Shader result.

To add some dynamism to the landscape, some clouds were added in order to simulate that the place was on a very high mountain. This shader was created basically adding some offset on X and Y over time to a noise texture with some color to get the albedo of the shader and the colorless result to the alpha clip to get the clouds effect (Figs 5.24 and 5.25). [8]

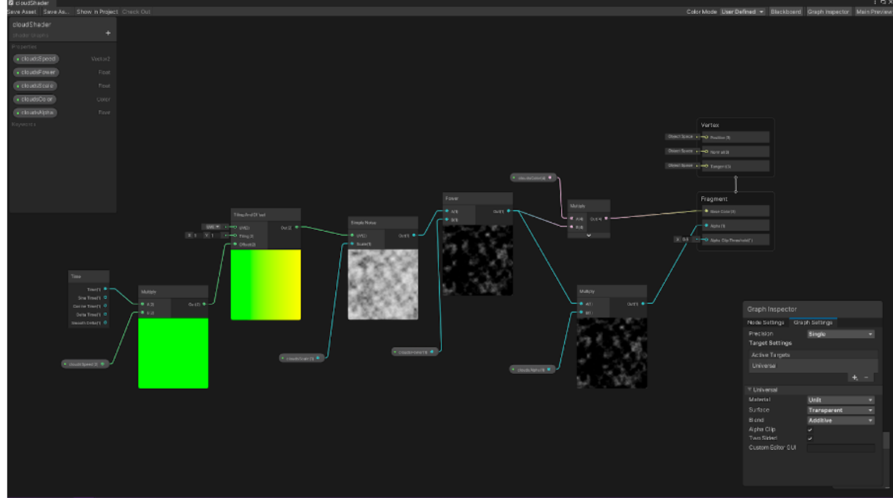


Figure 5.24: Clouds Shader.

The background was finally completed, so it was time to decorate the main scenario where the action took place. Following the retro-futuristic style of the vaporwave aesthetic a shader that seemed like a technologic / holographic theme was something cool to incorporate (Fig 5.26).
Adding the knowledge acquired by doing the mountains shader to a new gradient shader, the obtained shader was perfect to add it to the dummies (Fig 5.27). [?]

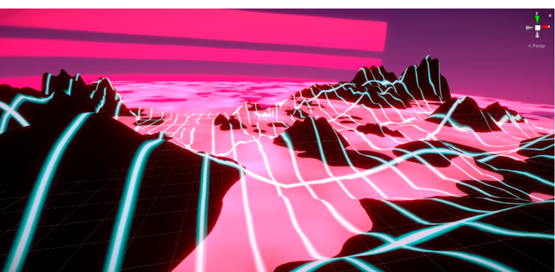


Figure 5.25: Clouds Shader result.

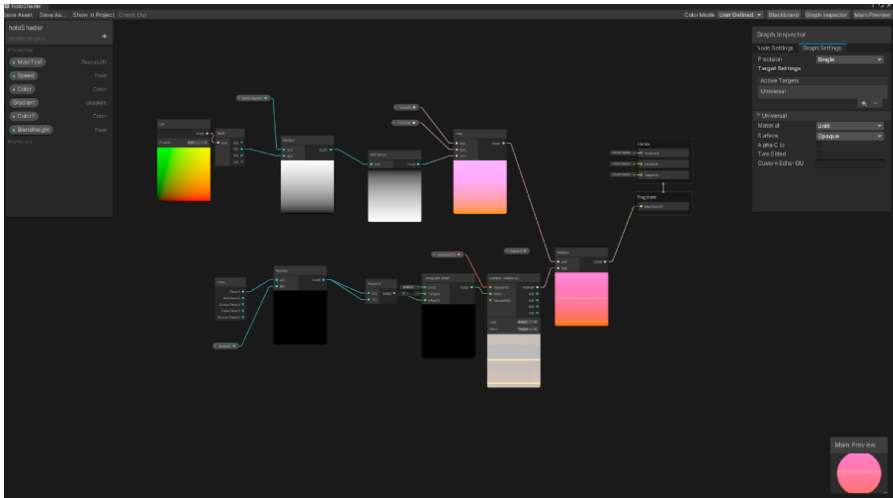


Figure 5.26: Dummies Holo Shader.

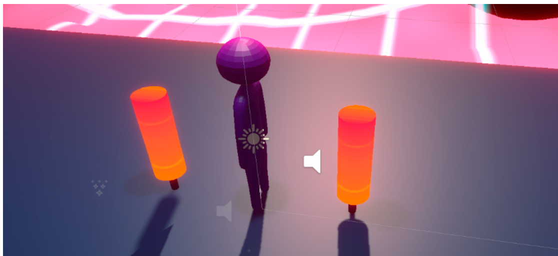


Figure 5.27: Dummies Shader result.

The next step was to decorate the training place, and a plain with grass that inspires peace and serenity was a good choice to replicate, like a lost place on the mountains where a wise monk retired to meditate.

After searching how to make it, a tutorial that mixed shaders with the Unity terrain

tool seemed a perfect choice. The shader for the grass patches was made with a texture and a 3D model imported from blender [9]. It added some movement to simulate the wind blowing and got a nice result (Fig 5.28).

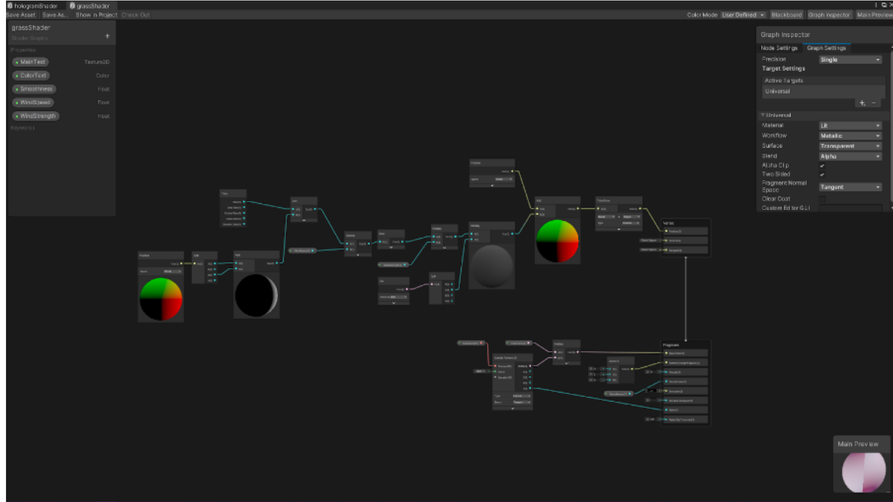


Figure 5.28: Grass Shader.

Sadly, when trying to mix it with the terrain tool as a tree to paint, the result was not as expected so instead of doing that, the terrain would have grass painted with some detail brushes and random patches of grass would be added over the place using some grass brushes to get different textures and density too (Fig 5.29).

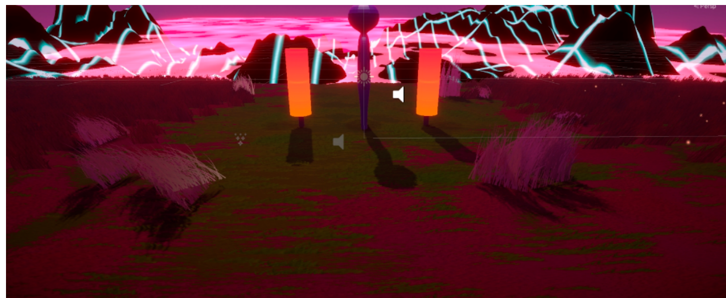


Figure 5.29: Grass final result.

After that, some assets and props were added to the scene in order to make it more comfortable so, a torii door and some rocks were modeled in blender following some tutorials [14] [16] and implemented later to the project.

Finally, to maintain that holo vibe a last shader was made to give some elements the aspect of some glitched holograms.
The shader was made by adding the same nodes from the gradient one created before to the base color of the object. Then, by adding some simple noise again with an offset over time and subtracting it to invert the colors and, adding a tiling with a texture to give that holographic vibe and connecting it to the emission and alpha channels being multiplied by a fresnel effect node to add it some color the shader was almost finished. Some movement to the position of the vertex part of the shader was added to achieve that glitchy effect (Figs 5.30 and 5.31). [11]

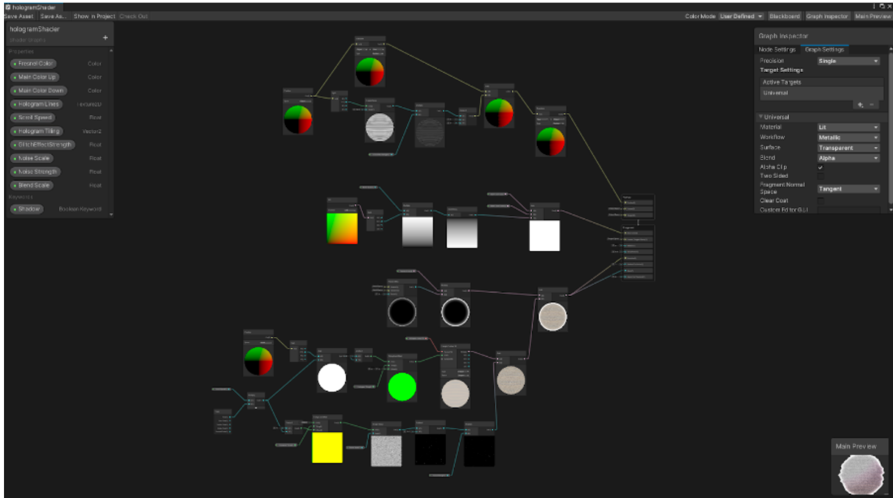


Figure 5.30: Rocks Holo Shader.

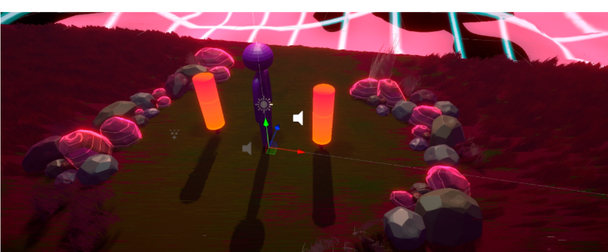


Figure 5.31: Rocks Shader result.

To conclude with the project, some particle systems and a Global Volume element were added in order to get an oneiric like atmosphere and the demo of this game was finished (Fig 5.32).

Figure 5.32: Final scene atmosphere.

5.2 Results

In this section the results of the work are described, always referenced to the initial objectives. The student should focus on the milestones achieved and, again, avoid an excess of technical details.

He/she should also comment on this section the actual or possibles applications or uses of the developed work, as well as all the releases that has been made or will be made of it (software repositories, web pages, etc.).

The final product obtained fulfills all the requirements and objectives described in the introduction section. After having some problems during the development and having been able to carry on even with them is very satisfying. Creating all these elements and assets from scratch, including the 3D ones, learning how to make shaders and how to apply them, and most importantly, being able to create a rhythm game that actually works as intended is a result more than successful. There were some elements that could not be added due to the time, but it had already been planned that it was a bit out of the scope and, only if there was a lot of time available in the end could be implemented.

Chapter 6

Conclusions and Future Work

Contents

6.1 Conclusions **57**
6.2 Future work **58**

In this chapter, the conclusions of the work, as well as its future extensions are shown.

6.1 Conclusions

In this section the student can freely comment on the experiences, professional or personal, that he/she had during the development of the work, the relation between the work and the coursed degree, etc. This section should be considered as an opinion section, whose content is not going to be assessed —although its formal aspects can be evaluated.

By doing this game I have learnt that maybe I am more capable of doing things that I thought I was not. During these last two years I have become much more confident, with coding overall, I always thought that programming a video game was like some kind of magic that was very far away from my level, but doing all this work by myself has made me realize that in the end practice makes perfect (being so much far away from perfection obviously).
There have been ups and downs during the development but something really important I have learnt from it is that time management is crucial for long lasting projects, at the beginning I started working nonstop which led to a big burnout that translated into an obvious decrease in productivity at late stages of the project.
In the end being able to create a game that was in my mind for so long, with the music

that I love and learning a lot of things from it is something unreal. Furthermore, while working on it I discovered parts of a game development process that I find very interesting and fun, such as GUI design and shaders that have definitely made me want to deepen them and maybe specialize in them.

6.2 Future work

If it seems that the work should or could be continued, it is convenient to include this section to indicate how and in which ways. It should also be indicated if the student itself plans to do so in the future.

As I told before this is a project I wanted to try for a long time and, even being proud of what I got for this "subject", this is something that I want to keep working on, adding more songs, maybe the weapons feature that could not get included because of the time limitation and much more features that I have been writing down when they popped in my mind while working such as a difficulty selector for the songs, maybe a blind-color mode for the notes or even a port to mobile devices.
Thinking bigger, if I even feel like launching it, a lot of changes should be done, starting with the songs which have copyright and the weapons that are inspired by real life groups's lightsticks, learning how to develop VFX and shaders for 2D and UI elements to make them cooler or even add an online ranking. But for now, this will continue being my personal project.

Bibliography

[1] Coding to the Beat - Under the Hood of a Rhythm Game in Unity: *https://www.gamedeveloper.com/audio/coding-to-the-beat—under-the-hood-of-a-rhythm-game-in-unity*

[2] Music Syncing in Rhythm Games: *https://www.gamedeveloper.com/programming/music-syncing-in-rhythm-games*

[3] Song BPM: *https://songbpm.com/*

[4] Timecode to Frames Calculator: *https://www.omnicalculator.com/other/timecode-to-frame*

[5] DoTween documentation: *http://dotween.demigiant.com/documentation.php*

[6] Music Games: Potential Application and Considerations for Rhythmic Training - PMC: *https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5447290/*

[7] A 3D Rhythm-based Serious Game for Collaboration Improvement of Children with Attention Deficit Hyperactivity Disorder (ADHD): *https://ieeexplore.ieee.org/document/9453999*

[8] Unity Shader Graph - Clouds Tutorial: *https://www.youtube.com/watch?v=xxhvUyvIH6s&t*

[9] Creating Simple Grass Shader in Unity URP: *https://www.youtube.com/watch?v=jyIukFoDfu0*

[10] How to create a gradient with Unity's Shader Graph?: *https://abbabon.github.io/2020-07-22-shader-graph-gradient/*

[11] Holograms in Unity Shader Graph: *https://www.youtube.com/watch?v=wtZ5WcrV-9A*

[12] Unity documentation: *https://docs.unity3d.com/2020.3/Documentation/Manual/system-requirements.html*

[13] Unity VFX Graph - Fire Attack Effect Tutorial: *https://www.youtube.com/watch?v=k7G9hZEgnOk*

[14] Blender Tutorial | Make Stylized Rocks Fast: *https://www.youtube.com/watch?v=5k119fhSuRA*

[15] Animate a Character in 15 Minutes in Blender: *https://www.youtube.com/watch?v=imbIsNAvUpM&t*

[16] Blender 3.0 Modeling - Japan Shrine Gate Part 1: *https://www.youtube.com/watch?v=mhQTBDiJlc4&t*

[17] Create Mountains in Unity in 5 Minutes: *https://www.youtube.com/watch?v=GbzGbFda9Xc&t*

[18] Visual Paradigm Online: *https://online.visual-paradigm.com/*

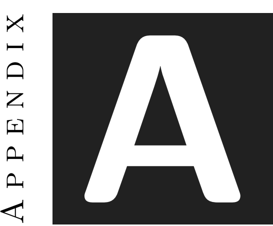
[19] Audio/Volume Slider in Unity Done RIGHT | Unity tutorial: *https://www.youtube.com/watch?v=V_Bf___ynKLE*

[20] Fast Character Modeling with the Skin Modifier in Blender: *https://www.youtube.com/watch?v=DAAwy_l4jw4&t*

[21] How I Create 3D Animated Characters with Blender in 15 Minutes: *https://www.youtube.com/watch?v=hXd4KEqrYEE*

[22] Mixamo: *https://www.mixamo.com/*

[23] Unity VFX Graph - Lightning Shader Effect Tutorial: *https://www.youtube.com/watch?v=40m_HUENh3E*



Source code

This appendix contains the most relevant scripts of the game.

A.1 Conductor class

The conductor class manages the data and lends it to the rest of the scripts that specialize in each action. It has the control of when to start the songs and ends and when to spawn the notes.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using UnityEngine.SceneManagement;

public class Conductor : MonoBehaviour
{
    //public float songBpm;
    public int songIndex;
    private float secPerBeat;
    public float firstBeatOffset;
    public float songPosition;
    public float songPositionInBeats;
    public float dspSongTime;
```

```
18      public int nextIndexLeft = 0, nextIndexRight = 0;
19      [SerializeField] public int beatsShownInAdvance;
20      private AudioSource musicSource;
21      public bool hasStarted = false;
22      public GameObject noteRight, noteLeft;
23      public static Conductor Instance;
24      public bool countdownStart = false;
25      public float countdownTimer = 3f;
26      public bool launchEndScreen = false, pauseSound = false;
27      public String lightstickPower;
28      [SerializeField] public int playerScore = 0, comboMult = 1, comboCount = 0, weaponSel;
29      [SerializeField] public Song[] songPool;
30      [SerializeField] public LightStick[] lightSticks;
31
32
33      private void Awake()
34      {
35          if(Conductor.Instance == null)
36          {
37              Conductor.Instance = this;
38          }
39          else
40          {
41              Destroy(gameObject);
42          }
43
44      }
45
46      // Start is called before the first frame update
47      void Start()
48      {
49          musicSource = GetComponent<AudioSource>();
50          songIndex = PlayerPrefs.GetInt("songIndex");
51          var songSelected = songPool[songIndex];
52          musicSource.clip = songSelected.musicSource;
53          weaponSel = PlayerPrefs.GetInt("Weapon");
54          if(weaponSel != 5){
55              lightstickPower = lightSticks[weaponSel].power;
56          } else{
57              lightstickPower = "Unarmed";
58          }
59          secPerBeat = 60f / songSelected.songBpm;
60      }
61
62      // Update is called once per frame
63      void Update()
64      {
65              if(!hasStarted){
66                  if(Input.GetKeyDown(KeyCode.Return)){
67                      hasStarted = true;
68                      countdownStart = true;
69                      dspSongTime = (float)AudioSettings.dspTime;
70                  }
71              }
```

```
72              else if(hasStarted && countdownTimer <= 0f){
73                  musicSource.Play();
74                  countdownStart = false;
75                  countdownTimer = 3f;
76              }
77              else{
78                  songPosition = (float)(AudioSettings.dspTime - dspSongTime - firstBeatOffset);
79
80                  songPositionInBeats = songPosition / secPerBeat;
81
82                  if(nextIndexRight < songPool[songIndex].songNotesRight.Length - 1 &&
83                   songPool[songIndex].songNotesRight[nextIndexRight] < songPositionInBeats +
84                   beatsShownInAdvance){
85                      Instantiate(noteRight);
86                      nextIndexRight ++;
87                  }
88                  if(nextIndexLeft < songPool[songIndex].songNotesLeft.Length - 1 &&
89                   songPool[songIndex].songNotesLeft[nextIndexLeft] < songPositionInBeats +
90                   beatsShownInAdvance){
91                      Instantiate(noteLeft);
92
93                      nextIndexLeft ++;
94                  }
95
96
97              }
98
99              if(countdownStart){
100                 countdownTimer -= Time.deltaTime;
101             }
102
103             switch(comboCount)
104             {
105                 case 0:
106                 comboMult = 1;
107                 break;
108
109                 case 5:
110                 comboMult = 2;
111                 break;
112
113                 case 12:
114                 comboMult = 4;
115                 break;
116
117                 case 26:
118                 comboMult = 8;
119                 break;
120
121                 case 35:
122                 if(lightstickPower == "deukae"){
123                     comboMult = 16;
124                 }else{
125                     comboMult = 8;
```

```
126                 }
127                 break;
128 
129                 default:
130                 break;
131             }
132 
133             if((hasStarted) && (songPosition > musicSource.clip.length + 3)){
134                 launchEndScreen = true;
135             }
136 
137             if(hasStarted){
138                 AudioListener.pause = pauseSound;
139             }
140     }
141 
142 }
```

A.2 Note Manager class (left)

The Note Manager, as it name says, manages the behaviour of the notes. There is one for the left notes and one for the right ones. It manages their movement, checks the acccuracy of the player and adds the score according to it.

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  public class MusicNoteManagerLeft : MonoBehaviour
7  {
8      private int songIndex;
9      public int beatsShownInAdvance;
10     public float beatOfThisNote;
11     public float songPosInBeats;
12     public String lightstickPower;
13     [SerializeField] Canvas canvas;
14
15     private Vector3 spawn;
16     private Vector3 end = new Vector3(0 , 1 , -1);
17
18     private Conductor conductor;
19
20     [SerializeField] MeshRenderer meshRenderer;
21     [SerializeField] Material startingMaterial, endingMaterial;
22
23     // Start is called before the first frame update
24     void Start()
25     {
26         songIndex = PlayerPrefs.GetInt("SongIndex");
27         conductor = Conductor.Instance;
28         songPosInBeats = conductor.songPositionInBeats;
29         beatsShownInAdvance = conductor.beatsShownInAdvance;
30         beatOfThisNote = conductor.songPool[songIndex].songNotesLeft[conductor.nextIndexLeft];
31         lightstickPower = conductor.lightstickPower;
32         meshRenderer = this.gameObject.GetComponent<MeshRenderer>();
33         startingMaterial = new Material(meshRenderer.material);
34         spawn = this.transform.position;
35
36
37     }
38
39     // Update is called once per frame
40     void Update()
41     {
42         songPosInBeats = conductor.songPositionInBeats;
43
44         transform.position = Vector3.Lerp(spawn, end,
45         (beatsShownInAdvance - (beatOfThisNote - songPosInBeats)) / beatsShownInAdvance);
```

```
46
47          meshRenderer.material.Lerp(startingMaterial,endingMaterial,
48          (beatsShownInAdvance - (beatOfThisNote - songPosInBeats)) / beatsShownInAdvance);
49
50          if(Input.GetMouseButtonDown(0) && Mathf.Abs(songPosInBeats - beatOfThisNote) <= 1.0){
51              if(Mathf.Abs(songPosInBeats - beatOfThisNote) <= 0.25){ // score += 100 * comboMult;
52                  if(lightstickPower == "blackpink"){
53                      conductor.comboCount += 2;
54                  }else{
55                      conductor.comboCount += 1;
56                  }
57
58                  if(lightstickPower == "loona"){
59                      conductor.playerScore += 200 * conductor.comboMult;
60                  }else{
61                      conductor.playerScore += 100 * conductor.comboMult;
62                  }
63                  Instantiate(canvas);
64                  Destroy(gameObject);
65              }
66              else if(songPosInBeats - beatOfThisNote <=
67               0.75 && songPosInBeats - beatOfThisNote >= 0){ // score += 50 * comboMult;
68
69                  if(lightstickPower == "blackpink"){
70                      conductor.comboCount += 2;
71                  }else{
72                      conductor.comboCount += 1;
73                  }
74
75                  if(lightstickPower == "loona"){
76                      conductor.playerScore += 100 * conductor.comboMult;
77                  }else{
78                      conductor.playerScore += 50 * conductor.comboMult;
79                  }
80
81                  Instantiate(canvas);
82
83                  Destroy(gameObject);
84              }
85          }else if(songPosInBeats - beatOfThisNote > 1.0){ // comboMult = 1;
86              if(lightstickPower == "chebul"){
87                  return;
88              }else{
89                  conductor.comboCount = 0;
90              }
91              Instantiate(canvas);
92
93              Destroy(gameObject);
94          }
95
96      }
97  }
```

A.3 Song Select Manager class

This is one of the most complex scripts, manages almost everything that is happening every moment in the Song Selection screen. The weapons and settings menus, the songs information and the data transferred to the In-Game scene.

```
1  using System;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.UI;
5  using UnityEngine.SceneManagement;
6  using UnityEngine.Video;
7  using TMPro;
8  using DG.Tweening;
9  using UnityEngine.Audio;
10
11 public class SongSelectManager : MonoBehaviour
12 {
13     private int songIndex, weaponSel;
14     private float masterVolume;
15     private AudioSource songPreview;
16     [SerializeField] AudioMixer mixer;
17     [SerializeField] Slider volumeSlider;
18     [SerializeField] private TMP_Text songName, volumeText, weaponSelectedText;
19     [SerializeField] private Image nextArrow, prevArrow, songNameTextBox,
20     wpnSelBtn, frame, wpnMenuBG, settingsMenuBtn;
21     private VideoPlayer videoPlayer;
22     [SerializeField] public AlbumDisplayer albumDisplayer;
23     [SerializeField] private GameObject wpnMenu, settingsMenu;
24     [SerializeField]private Button plainBtn;
25     [SerializeField] private Transform layout;
26     private bool weaponMenuActive = false, settingsMenuActive = false;
27     [SerializeField] private Song[] songPool;
28     [SerializeField] private LightStick[] lightSticks;
29     [SerializeField] public List<Button> lightstickList = new List<Button>();
30
31     // Start is called before the first frame update
32     void Start()
33     {
34         weaponSel = PlayerPrefs.GetInt("Weapon", 5);
35         masterVolume = PlayerPrefs.GetFloat("Volume", 0.5f);
36         mixer.SetFloat("Volume", Mathf.Log10(masterVolume) * 20);
37         volumeSlider.value = masterVolume;
38         songPreview = GetComponent<AudioSource>();
39         videoPlayer = GetComponent<VideoPlayer>();
40         songIndex = PlayerPrefs.GetInt("songIndex");
41
42         if(songIndex > songPool.Length - 1){
43             songIndex = 0;
44         }
45         ChangeSong();
```

```
46
47          foreach (var lightstick in lightSticks){
48              Button button = Instantiate(plainBtn, layout) as Button;
49              lightstickList.Add(button);
50              button.image.sprite = lightstick.lightstickSprite;
51              button.onClick.AddListener(() => ClickedWpnBtn(lightstick.numberSel));
52          }
53
54
55          if(weaponSel != 5){
56              LightstickSelected();
57              weaponSelectedText.text = "Selected␣Weapon:␣" + lightSticks[weaponSel].name;
58          } else{
59              UnarmedSelected();
60              weaponSelectedText.text = "Selected␣Weapon:␣Unarmed";
61          }
62      }
63
64      // Update is called once per frame
65      void Update()
66      {
67          if (Input.GetKeyDown(KeyCode.RightArrow) && !settingsMenuActive)
68          {
69              nextArrowAnim();
70              NextSong();
71              albumDisplayer.NextSong();
72
73          }
74          else if(Input.GetKeyDown(KeyCode.LeftArrow) && !settingsMenuActive){
75              prevArrowAnim();
76              PreviousSong();
77              albumDisplayer.PreviousSong();
78
79          }
80          else if(Input.GetKeyDown(KeyCode.A) && !settingsMenuActive){
81              LoadGameScene();
82          }
83
84          if (Input.GetKeyDown(KeyCode.Escape) && !settingsMenuActive && !weaponMenuActive){
85              SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex - 1);
86          }
87
88      }
89
90      private void ChangeSong()
91      {
92          PlayerPrefs.SetInt("songIndex", songIndex);
93          songPreview.clip = songPool[songIndex].clipSource;
94          videoPlayer.clip = songPool[songIndex].MvClip;
95          songPreview.Play();
96          videoPlayer.Play();
97          songName.text = songPool[songIndex].songName + "␣-␣" + songPool[songIndex].group;
98      }
99
```

```
100     public void NextSong(){
101         songIndex = (songIndex +1) % songPool.Length;
102         Debug.Log("AlbumDisplayer␣songIndex:␣" + songIndex);
103 
104         ChangeSong();
105     }
106 
107     public void PreviousSong(){
108         if(songIndex == 0){
109             songIndex = songPool.Length - 1;
110         }
111         else{
112             songIndex--;
113         }
114 
115 
116         ChangeSong();
117     }
118 
119     public void prevArrowAnim()
120     {
121         var sequence = DOTween.Sequence();
122         sequence.Insert(0f, prevArrow.DOFade(0, 0.25f)).SetEase(Ease.InSine);
123         sequence.Insert(0.25f, prevArrow.DOFade(1, 0.25f)).SetEase(Ease.OutSine);
124         sequence.Insert(0f, songNameTextBox.DOFade(0, 0.25f)).SetEase(Ease.InSine);
125         sequence.Insert(0.25f, songNameTextBox.DOFade(1, 1f)).SetEase(Ease.OutSine);
126         sequence.Insert(0.0f, songName.DOFade(0f, 0.0f));
127         sequence.Insert(0.2f, songName.DOFade(1f, 1f)).SetEase(Ease.OutSine);
128     }
129 
130     public void nextArrowAnim()
131     {
132         var sequence = DOTween.Sequence();
133         sequence.Insert(0f, nextArrow.DOFade(0, 0.25f)).SetEase(Ease.InSine);
134         sequence.Insert(0.25f, nextArrow.DOFade(1, 0.25f)).SetEase(Ease.OutSine);
135         sequence.Insert(0f, songNameTextBox.DOFade(0, 0.25f)).SetEase(Ease.InSine);
136         sequence.Insert(0.25f, songNameTextBox.DOFade(1, 1f)).SetEase(Ease.OutSine);
137         sequence.Insert(0.0f, songName.DOFade(0f, 0.0f));
138         sequence.Insert(0.2f, songName.DOFade(1f, 1f)).SetEase(Ease.OutSine);
139     }
140 
141     public void wpnSelectAnim()
142     {
143         var sequence = DOTween.Sequence();
144         sequence.Insert(0f, wpnSelBtn.DOFade(0, 0.25f)).SetEase(Ease.InSine);
145         sequence.Insert(0.25f, wpnSelBtn.DOFade(1, 0.25f)).SetEase(Ease.OutSine);
146     }
147 
148     public void LoadGameScene()
149     {
150         SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
151     }
152 
153     public void ActiveWeaponMenu()
```

```
154     {
155         weaponMenuActive = !weaponMenuActive;
156         wpnMenu.SetActive(weaponMenuActive);
157         wpnSelBtn.gameObject.SetActive(!weaponMenuActive);
158         settingsMenuBtn.gameObject.SetActive(!weaponMenuActive);
159         nextArrow.gameObject.SetActive(!weaponMenuActive);
160         prevArrow.gameObject.SetActive(!weaponMenuActive);
161 
162     }
163 
164     public void ActiveSettingsMenu(){
165         settingsMenuActive = !settingsMenuActive;
166         settingsMenu.SetActive(settingsMenuActive);
167         settingsMenuBtn.gameObject.SetActive(!settingsMenuActive);
168         wpnSelBtn.gameObject.SetActive(!settingsMenuActive);
169         nextArrow.gameObject.SetActive(!settingsMenuActive);
170         prevArrow.gameObject.SetActive(!settingsMenuActive);
171     }
172 
173     public void OnChangeSlider(float VolumeValue)
174     {
175         volumeText.text = ((Int32)(VolumeValue * 100f)).ToString();
176         if(VolumeValue < 0.01){
177             mixer.SetFloat("Volume", -80f);
178         }
179         else{
180             mixer.SetFloat("Volume", Mathf.Log10(VolumeValue) * 20);
181         }
182         PlayerPrefs.SetFloat("Volume", VolumeValue);
183         PlayerPrefs.Save();
184     }
185 
186     public void ClickedWpnBtn(int number){
187         weaponSel = number;
188         weaponSelectedText.text = "Selected␣Weapon:␣" + lightSticks[number].name;
189         PlayerPrefs.SetInt("Weapon", lightSticks[number].numberSel);
190         LightstickSelected();
191     }
192 
193     public void Unarmed(){
194         weaponSel = 5;
195         weaponSelectedText.text = "Selected␣Weapon:␣Unarmed";
196         UnarmedSelected();
197         PlayerPrefs.SetInt("Weapon", 5);
198     }
199 
200     public void LightstickSelected(){
201         for(int i = 0; i < lightSticks.Length; i++){
202             if(i != weaponSel){
203                 lightstickList[i].image.sprite = lightSticks[i].lightstickSprite;
204             }
205             else{
206                 lightstickList[i].image.sprite = lightSticks[i].lightstickSelectedSprite;
207             }
```

```
208            }
209        }
210
211        public void UnarmedSelected(){
212            for(int i = 0; i < lightSticks.Length; i++){
213                lightstickList[i].image.sprite = lightSticks[i].lightstickSprite;
214            }
215        }
216    }
```

A.4 Album Displayer class

Alongside the previous script, this complements the part of the caroussel for the Song Selection screen. It creates each of the album covers and controls the rotation each time the player scrolls through them.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using DG.Tweening;

public class AlbumDisplayer : MonoBehaviour
{
    private int songIndex;
    private float angle;
    private AudioSource beep;

    [SerializeField] public Transform albumDisplayer, targetCam;
    [SerializeField] private Song[] songPool;
    [SerializeField] public List<Transform> albumsList = new List<Transform>();
    // Start is called before the first frame update
    void Start()
    {
        beep = GetComponent<AudioSource>();
        songIndex = PlayerPrefs.GetInt("songIndex", 0);
        angle = 360 / songPool.Length;

        foreach (var song in songPool){
            var album2 = Instantiate(song.album, new Vector3(0, 0, 5),
            Quaternion.identity, albumDisplayer) as GameObject;

            albumsList.Add(album2.transform);
            this.transform.Rotate(new Vector3(0, angle, 0));
        }

        this.transform.Rotate(new Vector3(0, angle * songIndex, 0));

    }

    // Update is called once per frame
    void Update()
    {
        foreach (var album in albumsList)
        {
           album.transform.Rotate(Vector3.up * (25f * Time.deltaTime));
        }

        if(Input.GetKeyDown(KeyCode.RightArrow)){
            beep.Play();

        }
```

```
46          else if(Input.GetKeyDown(KeyCode.LeftArrow)){
47              beep.Play();
48          }
49
50      }
51
52      public void NextSong(){
53          songIndex = (songIndex +1) % songPool.Length;
54          this.transform.DORotate(new Vector3(0, angle * songIndex, 0), 1f).SetEase(Ease.OutSine);
55          albumsList[songIndex].DODynamicLookAt(targetCam.position, 1f);
56
57      }
58
59      public void PreviousSong(){
60          if(songIndex == 0){
61              songIndex = songPool.Length - 1;
62          }
63          else{
64              songIndex--;
65          }
66          this.transform.DORotate(new Vector3(0, (angle) * songIndex, 0), 1f).SetEase(Ease.OutSine);
67          albumsList[songIndex].DODynamicLookAt(targetCam.position, 1f);
68
69      }
70
71  }
```

A.5 InGame HUD Manager class

This script manages every element related to the HUD and the Canvas that takes place In-Game (Score, info, menu, Post-Song screen).

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using DG.Tweening;
using UnityEngine.SceneManagement;
using UnityEngine.Audio;

public class HUDManager : MonoBehaviour
{
    [SerializeField] TMP_Text timerText, playerScoreText, comboMultText, finalScoretext,
    volumeText, weaponText, recordScoreText, congratsText;
    public float fadeTimer = 1.5f, endingTimer = 3f, masterVolume;
    private int weaponSel, recordScore, recordWeapon;
    private Conductor conductor;
    public Transform numbersTransform, mouseTransform;
    public Image countDownSprite, activeWpnSprite, recordWeaponImg;
    public AudioSource beep;
    public Image mouseSprite;
    private bool changed = false, settingsMenuActive = false;
    [SerializeField] Sprite logoUnarmed;
    [SerializeField] GameObject preStartScreen, inGameHUD, endScreen, settingsMenu;
    [SerializeField] Slider volumeSlider;
    [SerializeField] AudioMixer mixer;
    public List<Sprite> numbers = new List<Sprite>();
    public List<Sprite> mouseSprites = new List<Sprite>();


    // Start is called before the first frame update
    void Start()
    {
        conductor = Conductor.Instance;
        DOTween.Init(true, true, LogBehaviour.Verbose).SetCapacity(200, 10);
        timerText.color = new Color(0.4f,0.0f,1.0f,1.0f);
        timerText.text = ((int)conductor.countdownTimer).ToString();
        playerScoreText.text = conductor.playerScore.ToString();
        comboMultText.text = conductor.comboMult.ToString();
        masterVolume = PlayerPrefs.GetFloat("Volume", 0.5f);
        mixer.SetFloat("Volume", Mathf.Log10(masterVolume) * 20);
        volumeSlider.value = masterVolume;
        MouseBlink();
        InvokeRepeating("ChangeMouseSprite", 0.5f, 0.5f);
        weaponSel = conductor.weaponSel;
        if(weaponSel != 5){
```

```
47              weaponText.text = conductor.lightSticks[weaponSel].name;
48              activeWpnSprite.sprite = conductor.lightSticks[weaponSel].ingameSprite;
49          } else{
50              weaponText.text = "Unarmed";
51              activeWpnSprite.sprite = logoUnarmed;
52          }
53
54
55      }
56
57      // Update is called once per frame
58      void Update()
59      {
60
61          if(conductor.countdownStart){
62              countDownSprite.sprite = numbers[1 + (int)conductor.countdownTimer]; // +1 al index
63              if(conductor.countdownTimer <= 0.4f){
64                  beep.pitch = 1.5f;
65                  beep.volume = 0.5f;
66              }
67          }
68
69          if(Input.GetKeyDown(KeyCode.Return) && conductor.countdownStart){
70              preStartScreen.SetActive(false);
71              CountDownAnim();
72              CancelInvoke();
73              SoundBeep();
74          }
75
76          if(playerScoreText.text != conductor.playerScore.ToString()){
77              AddPoints();
78              playerScoreText.text = conductor.playerScore.ToString();
79          }
80
81          comboMultText.text = "x" + conductor.comboMult.ToString();
82
83          if(conductor.launchEndScreen){
84              LaunchEndScreen();
85          }
86
87
88          if(!settingsMenuActive && Input.GetKeyDown(KeyCode.Tab)){
89              ActiveSettingsMenu();
90          }
91      }
92
93      private void CountDownAnim()
94      {
95          var sequence = DOTween.Sequence();
96          sequence.SetLoops(3);
97          sequence.Insert(0f, numbersTransform.DOScale(new Vector3(1f,1f,1f),
98           0.5f).SetEase(Ease.OutSine));
99
100         sequence.Insert(0.5f, numbersTransform.DOScale(new Vector3(0f,0f,0f), 0.5f));
```

```
101     }
102
103     private void MouseBlink()
104     {
105         var mouseSequence = DOTween.Sequence();
106         mouseSequence.SetLoops(-1);
107         mouseSequence.Insert(0.25f, mouseTransform.DOScale(new Vector3(1f,1f,1f),
108          0.25f).SetEase(Ease.InSine));
109
110         mouseSequence.Insert(0.75f, mouseTransform.DOScale(new Vector3(0.75f,0.75f,0.75f),
111          0.25f).SetEase(Ease.OutSine));
112     }
113
114     private void SoundBeep()
115     {
116         beep.Play();
117     }
118
119     private void ChangeMouseSprite()
120     {
121         if(changed){
122             mouseSprite.sprite = mouseSprites[0];
123         }
124         else{
125             mouseSprite.sprite = mouseSprites[1];
126         }
127         changed = !changed;
128     }
129
130     private void AddPoints()
131     {
132         var sequence = DOTween.Sequence();
133         sequence.Insert(0f, playerScoreText.transform.DOScale(2f, 0.1f).SetEase(Ease.OutSine));
134         sequence.Insert(0.1f, playerScoreText.transform.DOScale(1f, 0.25f).SetEase(Ease.InSine));
135     }
136
137     private void LaunchEndScreen(){
138         finalScoretext.text = "You␣got␣" + conductor.playerScore.ToString() + "␣points!!";
139         recordScore = conductor.songPool[conductor.songIndex].recordScore;
140         recordScoreText.text = recordScore.ToString();
141         recordWeapon = conductor.songPool[conductor.songIndex].recordWeapon;
142         if(recordWeapon != 5){
143             recordWeaponImg.sprite = conductor.lightSticks[recordWeapon].ingameSprite;
144         } else{
145             recordWeaponImg.sprite = logoUnarmed;
146         }
147         endScreen.SetActive(true);
148         inGameHUD.SetActive(false);
149         if(conductor.playerScore > recordScore){
150             conductor.songPool[conductor.songIndex].recordScore = conductor.playerScore;
151             conductor.songPool[conductor.songIndex].recordWeapon = weaponSel;
152             congratsText.text = "NEW␣RECORD";
153         }
154     }
```

```
155
156     public void RetryLvl(){
157         SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
158     }
159     public void MainMenu(){
160         SceneManager.LoadScene(0);
161     }
162
163
164
165     public void onChangeSlider(float VolumeValue){
166         Debug.Log(VolumeValue);
167         volumeText.text = ((Int32)(VolumeValue * 100f)).ToString();
168         if(VolumeValue == 0){
169             mixer.SetFloat("Volume", -80f);
170         }
171         else{
172             mixer.SetFloat("Volume", Mathf.Log10(VolumeValue) * 20);
173         }
174         PlayerPrefs.SetFloat("Volume", VolumeValue);
175         PlayerPrefs.Save();
176     }
177
178     public void ActiveSettingsMenu(){
179         settingsMenuActive = !settingsMenuActive;
180         settingsMenu.SetActive(settingsMenuActive);
181         conductor.pauseSound = settingsMenuActive;
182     }
183 }
```

Main Menu Manager class

The Main menu of the game is managed through this script.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.Audio;
using TMPro;
using UnityEngine.UI;

public class MainMenuManager : MonoBehaviour
{
    private float masterVolume;
    [SerializeField] Slider volumeSlider;
    [SerializeField] private AudioMixer mixer;
    [SerializeField] private TMP_Text volumeText;
    [SerializeField] public GameObject settingsScreen, mainButtons;
    private bool settingsOn = false;
    private AudioSource BGM;


    void Start()
    {
        BGM = GetComponent<AudioSource>();
        masterVolume = PlayerPrefs.GetFloat("Volume", 0.5f);
        mixer.SetFloat("Volume", Mathf.Log10(masterVolume) * 20);
        volumeSlider.value = masterVolume;
        BGM.Play();

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void OnChangeSlider(float VolumeValue)
    {
        volumeText.text = ((Int32)(VolumeValue * 100f)).ToString();
        if(VolumeValue == 0){
            mixer.SetFloat("Volume", -80f);
        }
        else{
            mixer.SetFloat("Volume", Mathf.Log10(VolumeValue) * 20);
        }
        PlayerPrefs.SetFloat("Volume", VolumeValue);
        PlayerPrefs.Save();
    }
```

```
49
50      public void playScene(){
51          SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
52      }
53
54      public void OpenCloseSettings(){
55          settingsOn = !settingsOn;
56          settingsScreen.SetActive(settingsOn);
57          mainButtons.SetActive(!settingsOn);
58      }
59
60       public void QuitGame(){
61          Application.Quit();
62      }
63  }
```

Discarded Art

As it was mentioned before in the report, this project was an iterative process so some assets that were designed at first were replaced on late stages of the development.
In this appendix all these discarded assets will be shown.

B.1 Countdown numbers

Some art for the countdown numbers of the PreStart screen that were evolving through the development period (Figs B.1, B.3, B.2).

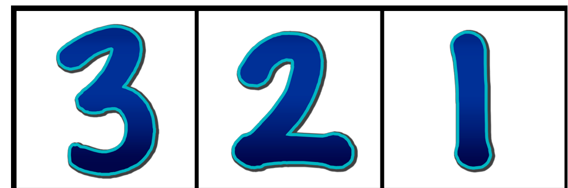


Figure B.1: First countdown numbers designed.

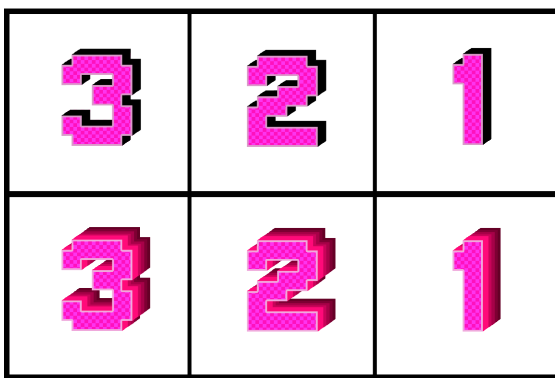


Figure B.2: Second countdown numbers designed.

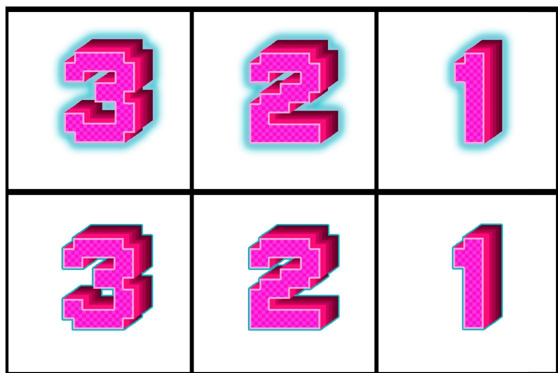


Figure B.3: More extra countdown numbers designed.

B.2 Menu buttons

Buttons for some of the menus that were changed when the remake of the UI occurred (Figs B.4).

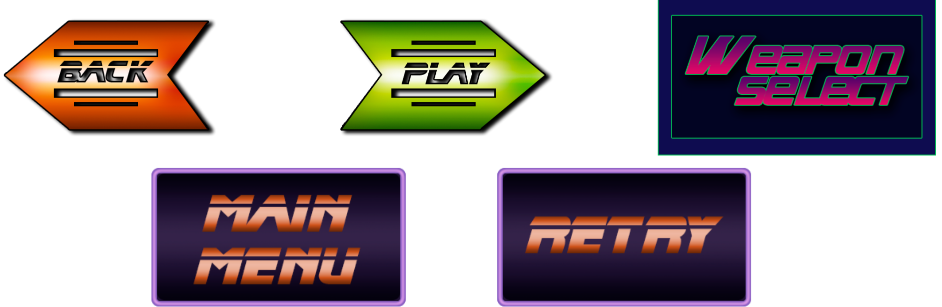


Figure B.4: Some menu buttons.

B.3 HUD elements

PreStart screen elements that were also changed for the ones that were more visullay coherent (Figs B.5).

Figure B.5: PreStart screen HUD elements.