Video Games Design and Development Degree
Final Degree Project Report

RELABEAT

a procedural game based on procedural music

Author: José María Pizana García

Tutor: José Martínez Sotoca

Summary

In this document two different aspects will be presented: the first one is the ideation and programming of two complex algorithms, one that generates music semi-procedurally and other that generates a 3D environment based on the previous procedural song; and the second one is the development of the demonstrator of a game based on the former algorithms in Unity: Relabeat. Relabeat is a fast paced, frenetic speed and reflexes 3D game, in which players will game through a straight environment plagued with dangers that are placed to the beat.

Keywords: Music Generation, Arcade, Unity, Video Game, Ludomusical Interaction

Index

1. - Introduction **6**
1.1. - Project Overview 7
1.1.1. State of the Art in Procedural Music 7
1.1.2. - Game Design 11
1.1.3. - Technical Design 11
1.1.4. - Artistic Design 12
1.2. - Related Subjects 12
1.3. - Tools 13
1.4. - Motivation 13
1.5. - Objectives and Expected Results 14
2. - Initial Planning **15**
2.1. - Planification 15
2.2. - Future Plans 16
2.3. - Risks and Contingency Plans 16
3. - Design **17**
3.1. - Technical Design 17
3.1.1. - Music Generation 17
3.1.2. - Track Generation 19
3.2. - Game Design 20
3.2.1. - Game Flow 21
3.2.2. Mechanics 21
3.2.3. - Resources 21
3.2.4. Dangers 22
3.3. - Artistic Design 22
4. - Development **25**
4.1. Music Generation 25
4.1.1. Sample Loading Tool 25

4.1.2. Generation Algorithm 26
4.1.3. Playing the song 27
4.2. Track Generation 28
4.2.1. Track Data Generation 29
4.2.2. Physical Track Generation 30
4.3. Game 31
4.3.1. Character 31
4.3.1.1. Controls 31
4.3.1.2. Finite State Machine 32
4.3.1.3. Character Movement State Machine 33
4.3.1.4. Resources 36
4.3.2. Dangers 37
4.4. Art 39
4.4.1. Track 40
4.4.2. Environment 40
4.4.3. Dangers 43
4.4.4. Sample Production and Musical Collaboration 44
4.4.5. User Interface 44
4.4.6. Visual Effects 46
5. - Results 48
6. - Final planning 49
7. - Conclusions 51
8. - Bibliography 52

Figure Index

1. **Singing Ringing Tree** 7
2. **Collatz Hailstone Numbers** 8
3. **Markov Chain Example** 9
4. **Wolfram's Automata** 10
5. **Procedural Music Generator interface** 10
6. **Mirror's Edge Catalyst screenshot** 23
7. **Audiosurf 2 screenshot** 23
8. **Redout screenshot** 24
9. **Sample loader prompt and folder structure** 26
10. **Asynchronous musical segment loading** 28
11. **Danger generation threshold splines** 30
12. **Default game controls** 31
13. **Generic Finite State Machine update flow** 33
14. **Character movement state machine** 33
15. **Strafe parameters and acceleration curve** 34
16. **Jumping distance diagram** 35
17. **Energy recovery curve** 36
18. **Tile measures** 37
19. **Laser tile measures** 38
20. **Goalkeeper drone and timer** 39
21. **In-game capture** 40
22. **Track versions** 40
23. **Building models** 41
24. **Example building layout from top** 42
25. **Black Hole effect** 42
26. **In-game laser depiction** 43
27. **Final laser turret model** 43
28. **Final drones model** 44
29. **Menus interface** 45
30. **Relabeat title and logo** 45
31. **Main menu** 46
32. **Damaged effect appearance and animation curve** 47
33. **Distortion effect** 47

1. - Introduction

Usually, when music and games interact, it is the first the one that has to give in part of its nature for the sake of the overall experience, giving what we know as *adaptive music*, reacting to ludical elements. Whereas, when a music game is developed, it is usually thought that one or several musicians would have spent many hours composing themes with a fixed span and that the game has been designed around those, growing from them. What is yet to be tried until this project is combining the flexibility of procedural music with a game that reacts to it mechanically, and not only visually.

This project consists of two separated and well defined parts:

- Generating two algorithms:
    - A music generation algorithm that produces sound that can be recognised as music as faithfully as possible. As the time for this project is limited and a good final result is sought for, musical fidelity is given priority over granularity of the system.
    - An environment generation algorithm that creates the track through which a character can run by. For the version intended in this project, the track is straight and the amount of dangers inside depends on the musical intensity.
- Generating a fast game that is both fun and coherent with the chosen music style. The objective is to survive until the end of the song on the track scoring as many points as possible.

Every component of this project can be iterated almost indefinitely. Because of this, the philosophy of development is generating a first demonstrator of the game, in order to have a solid base for future improvement and work towards a commercial product.

The general graphic design revolves around simple layers and neon lights that can be easily readable. Also, many graphical systems can be bound to different music elements (i.e.: scene lightning with the melody layer).

1.1. - Project Overview

1.1.1. State of the Art in Procedural Music

In the XVII century, Bach already interpreted music in an algorithmical way. Those were not his original ideas, but the result of an iteration that spanned centuries, developing techniques that could be considered algorithmical like counterpoint and canon. With the advent of computer generation, some of this philosophy was taken to create methods that autonomously generate music or sound.

This project is framed inside a category know as *generative, expert based music,* that is produced with little to no input. This category is classified by Farnell [FaAn-InPrAud] dividing generative music in stochastic, algorithmic and Artificial Intelligence based sound:

- **Stochastic** sound is generated through random or chaotic data, that is selectively chosen through mathematical or algorithmical methods to synthesize sound. Usually the results are not musical, and just when the perceived tempo is very low they render understandable results. These systems are not necessarily computer-related, as inputs usually pertain to real-world elements like wind in the case of Liu and Tonkin's *Singing Ringing Tree* [LiuTo-SiRiTr]. This statue consists of several aluminum pipes placed one over another facing different orientations, so when the wind passes through it generates sound that can be understood as music.

Fig 1: the Singing Ringing Tree

- **Algorithmic** sound subverts somewhat the meaning of *algorithm*, referring to iterative, recursive methods that give step-by-step results, not minding about the general composition or states different to the current one. A couple of example are iterative methods like Collatz *hailstone numbers* and Markov chains.

  Hailstone numbers are a set of numbers defined recursively, that are defined as following:

$$a_n = \begin{cases} a_{n-1}/2 & \text{if } a_{n-1} \text{ is even} \\ 3a_{n-1}+1 & \text{if } a_{n-1} \text{ is odd} \end{cases}$$

  Where n is a natural number, this definition cycles until reaching one and restarting the procedure (this has not yet been proven, but there are experimental and probabilistic evidence of its certainty), going up and down (the reason why they are called 'hailstone numbers'). Each number of the series can be mapped to a sound frequency, thus resulting in notes that could be interpreted as abstract music.

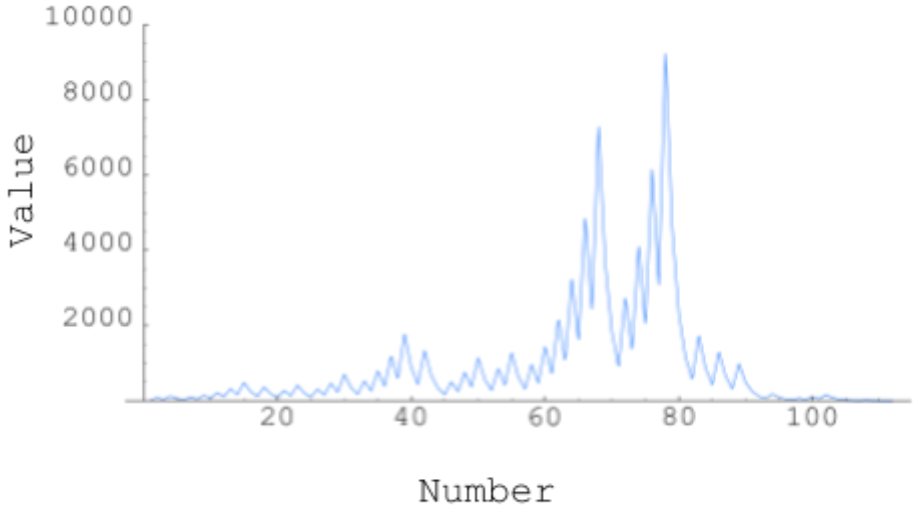

Fig 2: Collatz Hailstone Numbers progression for the first hundred

  Next, Markov chains are discrete statistical state machines whose principle is that the next state of the machine is chosen according to a statistical distribution between other states, whose chances depend only on the current one. Translated to music, the different states could be understood as notes of the pentagram, whose chances would be set by a musician or programmer to suit their specific needs. This process requires more human tuning, choosing which states to connect and with which weights, but gives more human results. Several Markov chains can be used at once: one for pitch, other for note duration…

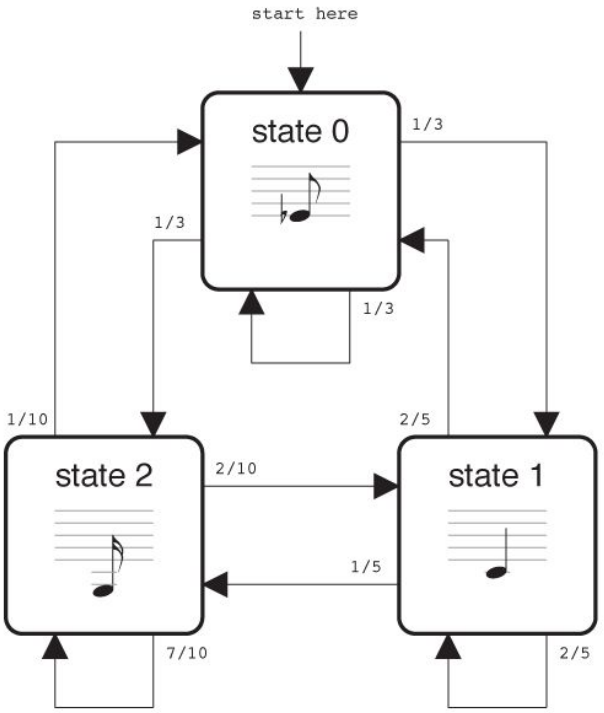

Fig 3: Markov Chain between three different notes

- **Artificial Intelligence** based sound use complex and varied techniques to give results of varied fidelity. This is where more variety exists and where the results are understood as more *musical* to the human hearing, so they will be explained in more depth than their previous counterparts. In general, it can be considered that Artificial Intelligence musical systems contain more previously stored information for them to function, usually, but not always, separated between pure data and program.

  An example is *Wolfram's Automata* [WoTo-HIW, Reiners-CAM], encased in the Cellular Automata category. It uses specific rules of this technique to consider the horizontal automata progression as if it was on a pentagram to generate different sensations. Still as it can be observed in its own homepage demonstrator [WoTo-Gen] the generated music sounds random and not structured in most cases.

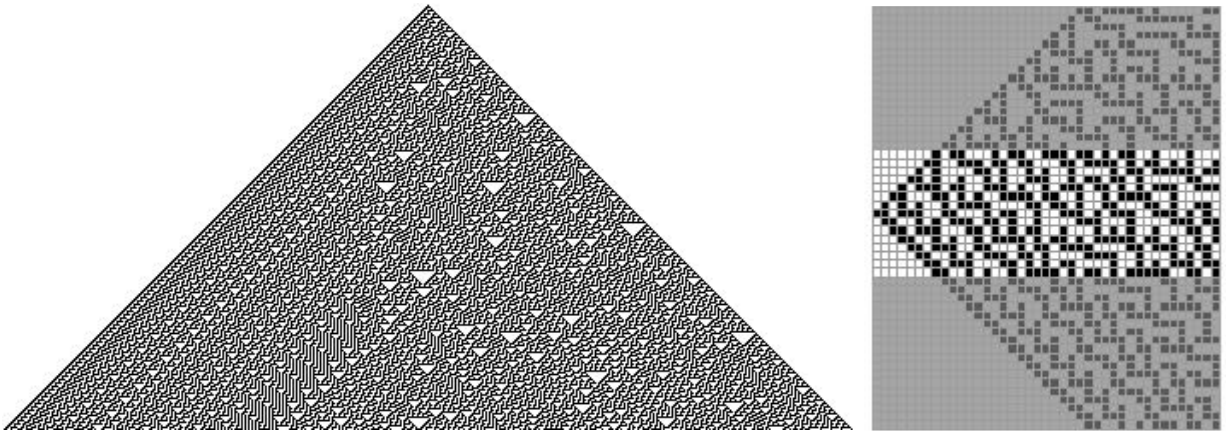

Fig 4: Left: Unidimensional Cellular Automaton generated using rule 30
Right: An instance of Wolfram's Automata sliced to purpose as pentagram for an instrument.

Other kind of Artificial Intelligence Musical systems are called Expert systems. These contain rules that they can look at to make decisions, and they can revise this rules to better suit their goals. People like Tyran Daley have achieved a great sensation of structure and *humanness* keeping all the flexibility of a procedural system using knowledge-based systems like in his Procedural Music Generator [DaTy-PMG]. This means that it has been necessary for its consecution to accrue a comprehensive understanding of occidental musical rules for later transforming it into instructions and algorithms. Despite this, in this case the generated music does not have a clear structure with parts, and thus rhymes and inter-segment communication.

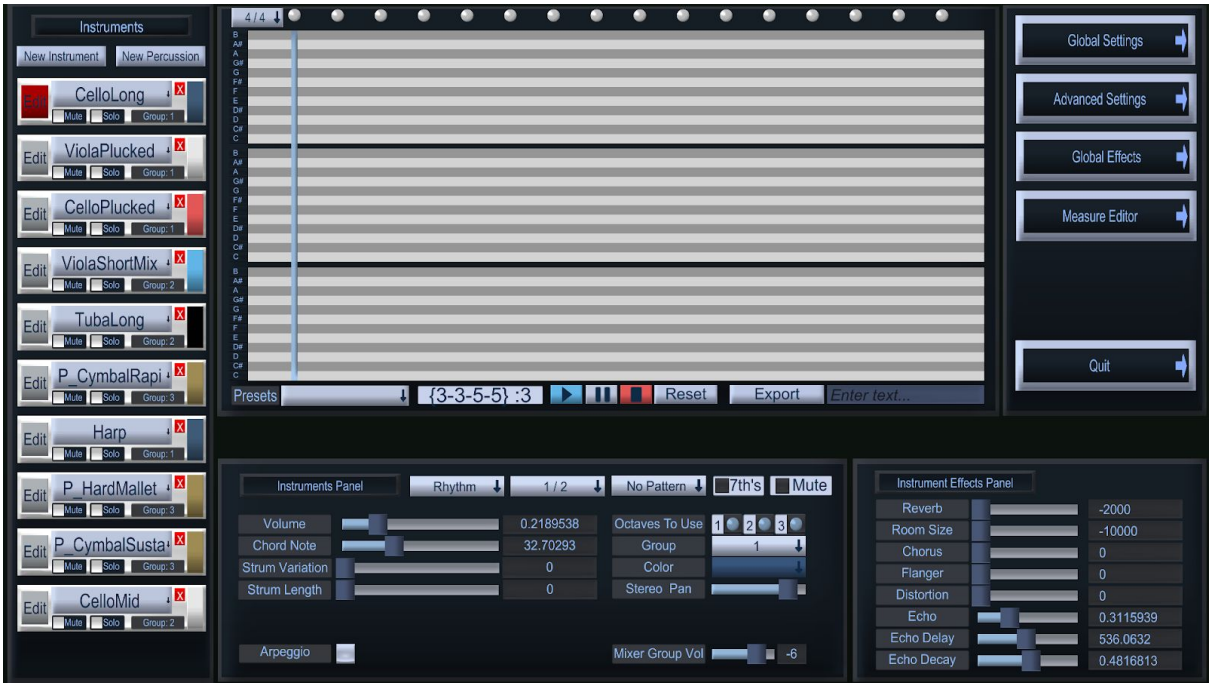

Fig 5: Tyran Daley's Procedural Music Generator interface

Lastly, Genetic and Neural Network based Artificial Intelligences can be pointed out, giving results that very closely seem those of a human like with Aiva technology [Aiva] or Jukedeck [JD], but are narrowed to the styles in which they are trained, so some take the solution of training a great quantity of AIs. Other flaw of these systems is that they can take a long span of time to train, or great computing capabilities, but they are compensated by their fidelity and quality.

1.1.2. - Game Design

In Relabeat, players travel through linear tracks that always accompany the beat of the music, that is generated procedurally from a pool of different rhythms and voices.

The main design directive is to convey the sensations of speed and rhythm. For this, many unusual game elements have been added, and some classical ones have been taken away, all in order to make a coherent and tightly linked game system.

Being a game about speed, and ludo-musical synchronization, the character moves forward automatically, and the actions that are available to him are moving from side to side, dashing sidewards or forward, accompanied with a space warp that makes spaces smaller, jumping and wall running.

The goal of the game is to end the tracks linked to the songs one at a time, scoring as much as possible without dying to dangers as lasers or gaps. This score also grants access to new skills and capacities to make the following stages easier.

This game is inspired by rhythmic games like Beat Saber [HM-BS], and Audiosurf [AS-AS], as well as from their aesthetics, and from fast, speed-based games like Mirror's Edge [EA-ME] and Redout [34BT -RO].

1.1.3. - Technical Design

The technical design is divided in the three main aspects of the game:

Song Generation

Each time a player enters in the game, a song has to be generated. This is the seed to the rest of the game, so it has to be built in a way that can be stored and read easily. For this process, a knowledge-based algorithm with stochastic elements is implemented, for which the collaboration of a musician has been sought after.

The game demonstrator version of the algorithm uses the sample as generator quantum: an audio clip that spans a bar of the song (given a beats per minute measure and bar definition). Many samples are combined stochastically and synchronized in different musical semantic layers (beat, harmony and melody), that in turn generate a song.

Even when the music style chosen for the game is electronic, the system is thought to be independent from styles, as those are set by the samples used (giving the chance to also mix different ones and experiment).

Track Generation

Taking the generated song, this algorithm creates a linear track equating a bar to a fixed distance (that the character travels at a fixed speed), and certain distance thresholds to beats and bars. This allows to insert dangers at certain moments in the song, where the general intensity is higher, or to not to do it if the intensity of some intervals is less than the average, for example.

The designed algorithm places a danger for each interval depending on a pondered combination of the average dangers per second that have to be placed and the normalized intensity of the music.

Game

The character always advances automatically, having the player manage its health and energy in skills and split second decisions that put the player's reflexes to test. Dangers will reach the player at the same time the beats hit, so a synchronic sensation is achieved. Also, graphic elements are synchronized with the music to generate a stronger sense of immersion.

1.1.4. - Artistic Design

Graphical design is not meant to be the focus of this project. The main concerns in this field are for the graphics not to be too much information-packed, as the player is already loaded with a lot of other data and challenges.

It is intended for the game to have a clean neon-like appearance with dark backgrounds that is easy to read, highlighting borders of the elements and interactable objects. Some art inspirations can be found on Mirror's Edge [EA-ME], Audiosurf [AS-AS] or Redout [34BT -RO]. Also, a previous game demo, Ball3 [Pi-B3], of the same author of this project has been rendered with a similar, more simple appearance that the one that is intended to be represented here.

In regard to the samples necessary to kickstart music generation and as said before, a musician has been contacted to do a collaboration.

1.2. - Related Subjects

VJ1205 - Algoritmos y Estructuras de Datos · VJ1231 - Inteligencia Artificial
VJ1206 - Informática Gráfica · VJ1227 - Motores de Juegos
VJ1236 - Técnicas de producción y realización sonora

1.3. - Tools

The tools that used in this project are:

- **Unity 2017.4 [Unity]**
- **Visual Studio 2017 [MS-VS]**
- **Git [Git], powered by Gitlab [GL]**
- **3ds Max 2019 [AD-3DS]**
- **Google Docs [G-GD]**
- **Unity Packages**

Despite the difficulty of specifying which precise Unity packages will be used in the final version, the next list is most likely to be included:

- DOTween, by Demigiant [DG-DT]
- Post Processing Stack, by Unity [Unity-PPS]
- Standard Assets, by Unity [Unity-SA]
- White City Building Models, by Aliyeredon [Aliy-WC]

1.4. - Motivation

Almost every musical game that is currently on the market is based on actual songs, usually from existing composers from which separate licenses are acquired per-theme *à-la* Guitar Hero [AV-GH].

Whereas, musical generators (not necessarily and not often attached to a game) usually stick just to a very narrow range of styles, or otherwise do not convey the sensation of a coherent song. The song generation that is tried to be achieved in this project tries to mix both a knowledge-based system that still relies on a musician to generate its building bricks, to conserve coherence. Furthermore, this intention relies on a very old interpretation of music as the weaving of fixed structures in a grammatical way, as composers like the aforementioned Bach used.

The presence of a musician in this context is more than necessary, as the musical generation relies on a deep knowledge of its processes and rules, so iterations are needed with a professional to tune the generation process.

The system above is used as a seed for an Audiosurf-like[AS-AS] game track, that reacts to the generated song. Having this theme separated in layers and components also enables the connection of different voices with different systems.

To sum up, the intention of the project is to bound more deeply ludical elements with music, giving a fast experience, like a trance, that absorbs players for the duration of the songs in its universe of sound.

1.5. - Objectives and Expected Results

01. To develop a song generation algorithm that is satisfactory to human hearing. This means that a person should not distinguish between a typical song and one composed by this system. This means that the generated music should hold well defined segments (intro, verse, chorus, bridge...). It is expected to develop electronic music at an stable tempo for this project, but the implications of the algorithm make it unbound to tempo or style.

02. To develop a track generation algorithm based on a given procedural song that can be played. This track has to be played by the character in the game, so it has to be coherent with its rules and not have errors that prevent a normal gameplay. This track has to hold dangers in positions according to the beat and the musical intensity.

03. To generate a real time sound connection with game elements such as dangers or lighting. In other words, having the ludical and visual elements respond to the sound events typical of a song.

04. To develop a game based on the previous systems that is fun and gives a frenetic sensation. This game should have a full gameplay loop (enter, play, exit) and perform smoothly and be free of bugs that impede the normal enjoyment of it. The specifics of this game are defined in the Design section.

2. - Initial Planning

2.1. - Planification

At the start of the project, there were assigned tasks for each week, taking into account that the estimated weekly dedication would be between 10 and 20 hours (depending on the week), with an average of 15. Weeks after the final delivery (28/5) are meant for small debugging and showcase-oriented polishing, and not the same work density is expected.

In order to generate the planning table, times have been measured by ordinal week number instead of real dates, as they serve to represent a fixed amount of development time that does not match natural weeks, as flexibility is needed due to the work practices in Wildframe Media that will occur at the same time.

Week	Tasks
1	Game definition
2	Movement Experimenting
2	Visual Effects Experimenting
3	Skills Experimenting
3	Art experimenting (technologies, models, materials...)
3	Final game design and algorithms' first versions.
4	Character Movement
5	1st Music Generation draft (just one looped segment)
6	Dangers programming: lasers, gap structure
7	1st Track Generation draft
8	Special character movement skills and movement tweaks
9	Dangers programming: drones, silences, obstacles
10	2nd Music generation draft (added structure: intro, ending, bridge)
11	2nd Track Generation draft (adding other dangers, tweaks)
12	Game special effects system (post processing, fades...)
13	Menus and complete game flow streamline.
14	Ludomusical synchronization
15	3rd Music Generation Draft (added piece variation for structure blocks, sound effects)
16	Report finishing
17	3rd Track Generation Draft (loops, forks)
18	Dangers Modelling & Materials
19	Definite delivery, debugging & tweaking
20	Definite delivery, debugging & tweaking

2.2. - Future Plans

The project is designed from scratch to be the first step towards a bigger, commercial development that could be potentially released on the different video game markets. Plans for the different systems include:

- **Music Generation:** increase the granularity of the system, starting to generate music per-sound adding concepts like Wolfram's cellular automatons [WoTo-05].
- **Track Generation:** add track deviations, turns, tilts and other structures like loopings.
- **Game:** include more and different dangers and make the difficulty system more complex. Include external music analyzing and interpretation that the player can load.

2.3. - Risks and Contingency Plans

The main risks that can occur due to the nature of the project are:

1. The generation algorithms could take too long to be completed due to difficulties or changes.
2. The search for a musician could delay the music part.

In both cases there is always a buffer given the possibility of delivering the project in the second term. In the case time is yet short, some tasks can be marked as the least priority ones and thuis delayed to the commercial development cycle, as difficulty levels and song and track storing.

In the first case, given than structural elements are referred, the solution would be cutting down the number of dangers or visual variety of the game, as those are the elements of least priority. Following these is skill purchasing, as they are not vital to the development of the project.

In the second case, the song generation part would have to be avoided, falling into analyzing typically composed songs to compose the tracks. This could be a valid and useful clutch as licensed products can be used freely as a means of study and investigation, but they should be removed in the final version of the game if it becomes commercial.

If the environment is too heavy to run at an acceptable speed (less than 0.016s, or 60 frames per second), a simple dynamic occlusion system will have to be implemented, deactivating elements that are behind the player or too far away.

3. - Design

Relabeat is a game about speed, music and the frenetic sensation of being constantly at the brink of victory and defeat. Thus, its mechanics have to convey those same sensations as faithfully as possible, centering themselves not in managing a great amount of resources and taking thoughtful decisions, but in instant reflexes and split second actions.

In the game, the player runs, jumps and warps spacetime in order to complete musical levels called “tracks”. There are two main objectives: to complete all the levels and to set the score for each as high as you can. The players count with a varied set of actions and skills to choose from, that shape the way they affront tracks, from dashing quickly to the sides, to wall run while a great leap comes in front, having to time a final decisive jump.

The game relies in two procedural systems: Song generation, that composes music formed on small, one-layered musical samples, and that serve as a building base; and Track generation, that analyzes said generated music to place different elements on the game course.

3.1. - Technical Design

3.1.1. - Music Generation

Music is one of the center points of the project, thus it has to receive a great amount of attention. In order to improve the musical scope of the project, the collaboration with a musician has been looked for, in order to gather the general techniques and rules that conform musical composition, and to generate the samples that are necessary for the system.

The smallest building element of the system is the sample: a bar of the song (the collection of n beats that are enclosed together in musical notation). Given this algorithm, the musician has to give as many samples as they want as input.

The algorithm selects a random musical mode, compatible rhythms, and playback speed for the samples that it picks in order for them to sound coherent when played together. These samples are sorted in three different layers of musical nature, widely known in music composition: beat, harmony and melody.

With the picked elements, the system generates different segments that cater to typical parts of occidental musical structure: intro, outro, bridge, chorus, verse... Given the time the song has to last, these blocks built with the aforementioned samples are placed.

The fact that the musical generation is built this way assures that this method is style-agnostic, making it depend on the musician’s input regarding the samples, leaving the charge of the decision to the person creating or gathering the samples.

At any specific moment, Relabeat has 3 different layers, not all of them having to be active at a given time. These are:

- **Beat/Rhythm layer:** the layer that holds the tempo. Usually it is filled with percussion-like instruments, but others can take their place, as bass or low pitched synthesizers.
- **Melody layer:** this layer has the main instruments that gives identity to the song.
- **Harmony layer:** other voices that can flesh out the song. Usually at least one is playing, because they conform the backbone of the theme with elements as bass.

Segments are be ordered in this three categories, having the possibility of being arranged in families: groups of similar samples that should never play together, as they are very similar sounds. Also the concept of essential sample groups needs to be introduced, referring to those samples or families of samples for which at least one has to be chosen for a given song, and to be played in almost all segments. One example of this would be the beat drums, that in electronic music are a common resource from which the song develops and there always have to be one kind of.

Of course, for an entire track that can span as long as 3 minutes, generating a n second sample selection and looping it many times is not a valid decision, because it becomes too repetitive quickly. Because of this some different parts have to be selected in some kind of structure. Electronic and Pop composing structures are known commonplace and they can be taken profit of, in order to generate a valid song. The following song segments are defined by usual composition rules [SA12]:

- **Intro**: the prologue of the song. Usually in this segment there are introduced just the base layer and some accompaniment in the secondary voices. It raises up *questions* that have their *answer* at the rest of the song.
- **Verse:** the first main part of the melody. It can repeat itself before the chorus, and usually does well with ending with tension that is released at the aforementioned chorus.
- **Chorus:** the main part of the song. Its hook and energy. It should be as long as the verse. It can be understood as a second verse in the same mode than the first.
- **Solo:** preferably after a round or two of verse and chorus. In it, the melody creates new patterns that break the monotony of the previous parts.
- **Break/Bridge**: usually used to contrapose the rest of the song. Often it stops the beat layer and/or adds new instruments or tonalities. In the game it could be used as a rest between intense parts.

- **Riser**: just like a break, but having a sort of build-up or crescendo preparing tension to be released at the next part, that should be the climax of the song, usually a chorus, and often with extra samples in its layers.

- **Outro:** used to resolve the song and to end it It's an optional part and some songs do better just with dry-cutting their beat. Other times they mirror the intro, or do a diminuendo of the verse or chorus.

Given all this building segments, we could devise the design of a song, but there are just two more previous steps that need to be defined for all samples to generate coherence in it. First is the *tempo*: all samples in a given segment need to have a compatible beat frequency (i.e.: one tempo is a multiple of the other). The second one is the tonality or mode: samples need to have compatible tonalities in order for them to sound harmonious (this does not apply to percussions, of course). Furthermore inside the tonality, tonal samples (those who belong to harmony and melody), have to share their chord rotation to sound well, so that distinction can replace tonality, as a chord rotation implies a specific tonality. Depending on the song, different segments can change chord rotation.

More so, segments can be varied furthermore by adding effects to the played samples so that one segment can be used more times without sounding repetitive. Some of these sound effects, typical in musical productions, are reverb, echo, sidechain compression, distortion, phaser...

Those two categories can be matched for all samples with the same system: tags. Samples need the same tempo and mode tags in order for them to appear together on a segment. We could add a third tag category called mood, in order to differentiate targets for the music (for example, a *relaxed* mood could be set for menus, and an *active* one for the game levels).

For the sake of resource management, for this project all samples are compatible, but future ones could be added following this system, making them independent. The project centers on electronic music as it's more easily mashable and mixable, as the strangement between parts is one of its strengths, but the system is going to be style-agnostic once done.

3.1.2. - Track Generation

Once the song is generated, a collection of dangers and rests has to be created accordingly in order to present them to the player in the shape of a game. This collection is what is referred as 'track'. The selected technique uses a combination of the song's sound and the concept that dangers should be placed in a more or less stable density throughout the song.

This generation algorithm can be divided in two different parts that will be defined next.

Track Data Generation

First dangers are chosen and placed on their respective virtual slots on the tiles. Even when each danger has a different nature, the placement process is the same for everyone.

While the track is being created, the generation of a dangers is decided through a random number that has to be lesser than certain threshold. This threshold is where the complexity of the process relies, as it is calculated by combining with weights two different values.

On the one hand, the number of dangers in the last second is being stored, so it is compared against the desired average number of dangers. This results in a normalized value that is evaluated through a spline function to get the final value.

On the other hand, the intensity of the song in the studied interval has to be analyzed to determine if it enhances the chance of having a danger, so it is compared to the maximum intensity of the track (or normalized) and evaluated in a different curve from the previous one.

The aforementioned intensity analysis includes retrieving the intensity for each layer, but also increasing or diminishing it artificially in a post-process (sharpen, blur...) like way in certain spots. This intensity analysis takes the song as if it was a single audio clip, so it is easily interchangeable with a normal song.

Physical Track Generation

Finally the physical environment can be generated. The algorithm takes the data generated in the previous step and instantiates a track tile for each virtual one. These tiles manage for themselves the necessary changes due to the dangers they contain, for example a gap tile should not instantiate a floor for itself, as that it's the danger in itself.

3.2. - Game Design

This design revolves around the idea of using music (and its associated track) as a structural element. Furthermore, it is decided to convey a strong sensation of speed because it matches with the music style that has been decided, that is new electronic.

In an overall perspective, the game is divided in levels, sorted by difficulty in ascending order, that include new dangers and increase in action density. The objective is to finish all levels and get the maximum score possible, that also grants access to new skills and capacities to make the following stages easier.

3.2.1. - Game Flow

Inter-Games Mode

The player browses from the main menu, where they can find the different difficulty options understood as levels, and browse stored songs and tracks. From there, the only access is to the game mode.

Game Mode

In this part of the game, the player has to control the ever forward advancing character dodging deadly lasers, gaps, drones and more. When the track is completed, the health of the character is reduced to 0, or the character falls through a gap or the sides, the player moves on to a summary screen that accounts the achievements, scores and overall performance in the track, and then redirected to another game or the inter-games menus.

The character advances forward all the time, usually at an stable speed, but the player is able to look around approximately in a 180º vertical span and 200-210º horizontal span (slightly back as the human fisionomy permits) so as to examine its surroundings. This is due to the common user accommodation to First Person controls from other games, even when screen turn is not necessary for Relabeat.

Even when the game can render open spaces, the ludic action happens in a corridor-like space with turns, falls and climbs to make the system seem more organic.

3.2.2. Mechanics

The player interacts with the game through a character. This character has two main points of control: mechanics and resources:

Even when the character moves constantly forward at the same speed, it is able to jump, crouch, dash, wall run and strafe to avoid the different dangers that are presented. Many of this movement skills are limited by a resource called energy that is defined in the next section, but their exact amounts and restrictions will be experimented to find a good combination of energy consuming and not consuming skills.

3.2.3. - Resources

The players manage two main resources on-game: health and energy. Health is depleted when taking damage from different dangers like lasers, and when it reaches 0, the player will have lost the game. Secondly, energy is necessary to use skills such as dashing or crouching, and it regenerates over time.

In order to give a positive incentive to perform as well as possible to the player, a Sync metric is included in the game. It goes up as long as the player performs well, does not do mistakes and maybe if stunts are performed, as passing as close as possible to the dangers, or successfully doing other difficult tasks. The Sync is communicated to the

player visually, aberrating the camera (as in a relativity visualization) and including other effects as chromatic aberration.

3.2.4. Dangers

There are several difficulties that the player has to overcome to emerge victorious in a track. Some take physical form, as lasers or drones, and others add rules for the tile, as gaps or silence zones. Anyhow all dangers span one or more tiles, and a tile is able to harbor more than one danger in high difficulty levels. Here is a list of the designed dangers:

- **Lasers**: lasers represent a line that, in the case the character crosses it, will remove a health point from them, thus they have to be dodged. There are three types:
    - **Static:** They keep their position.
    - **Rotating:** They change angle over time, holding in their virtual center.
    - **Dynamic:** They both change their angle and virtual center over time.
- **Gaps:** gaps are tiles without floor, so they have to be crossed using methods like jumping, wall running or dashing. The should not span more than the character can jump in the case no other traversal methods (as walls) are generated
- **Silence Zones:** silence zones are areas that can span a portion or all of the width of one or more tiles. This areas do not impose a direct danger to the character, but no skill that consumes the energy resource can be used inside them, or otherwise the zone inflicts one point of damage to their health.
- **Drones:** drones are both obstacles and attackers. Drones can be of three different types, but independently of its type, they inflict one point of damage on contact.
    - **Static:** This drones stay in place indefinitely.
    - **Goalkeeper**: When the character passes through them, they travel to the position the character was some seconds ago.
    - **Bouncing:** This last kind bounces with a regular speed like a ball, keeping its depth that sets the time in which will be met.

3.3. - Artistic Design

Graphical design is not meant to be the focus of this project. The main concerns in this field are for the graphics not to be too much information-packed, as the player is already loaded with a lot of other data and challenges.

It is intended for the game to have a clean neon-like appearance with dark backgrounds that is easy to read, highlighting borders of the elements and interactable objects. Some art inspirations can be found on Mirror's Edge [EA-ME], Audiosurf [AS-AS] or Redout [34BT -RO]. Also, visual inspiration can be taken from Ball3 [Pi-B3], a game developed by the same author as Relabeat.

Fig 6: Mirror's Edge Catalyst In-game screenshot

Fig 7: Audiosurf 2 In-game screenshot

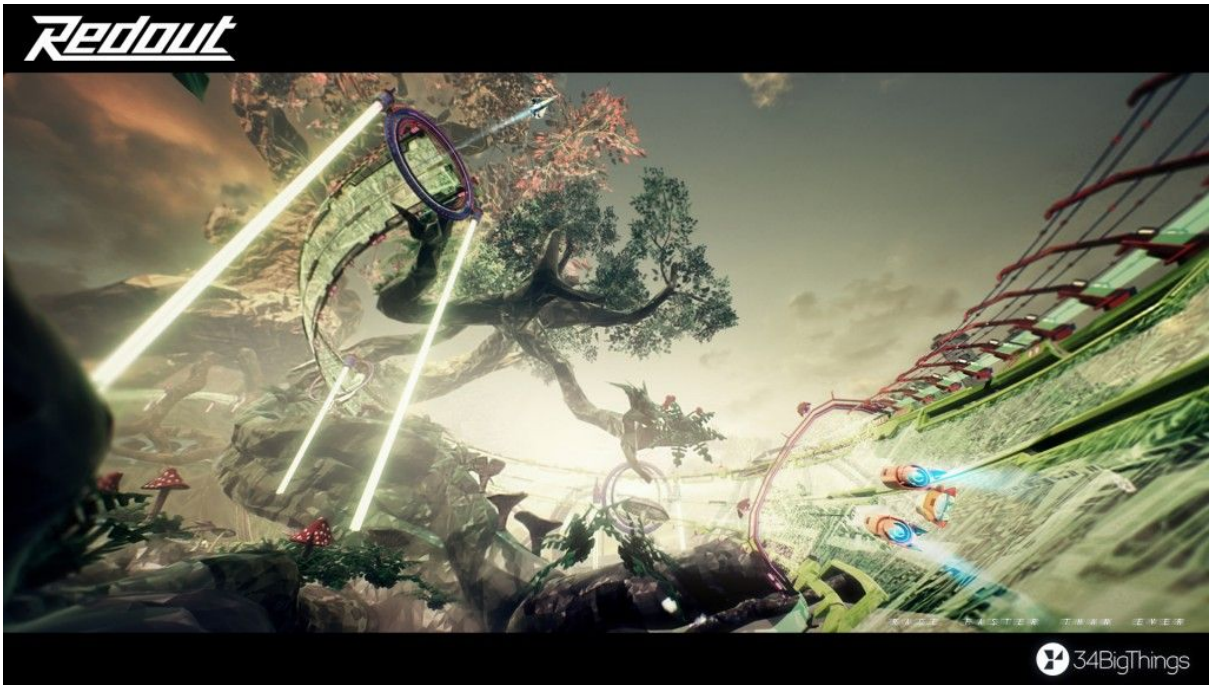

Fig 8: Redout In-game screenshot

4. - Development

4.1. Music Generation

Being one of the center pieces of the project, song creation has been iterated to try to reach as much variation as possible.

When the level is about to start, a song has to be generated, which forces the method to be run at real time in a very short span (smaller than 2 seconds) taking into account that a loading screen can be used, but waiting times are detrimental to the enjoyment of the game.

Since March, the collaboration with the musician, Pablo Caballero [CaPA], has given insight in the rules that are needed to compose music in an structured way. This work has been structured in three iterations with one month of separation, in which results were reviewed, discussed and goals were set. Furthermore, during the development a procedural music asset programmer was contacted, Tyran Daley [DaTy], to compare the techniques that he used in his *Procedural Music Generation* [DaTy-PMG] asset. These knowledge is commented in the next sections together with the music generation algorithm.

4.1.1. Sample Loading Tool

In order to agilize the process of loading samples' data into the system, it was necessary to develop an engine tool that enabled the fast loading of sample clips and data, and that could be easily understood in the eyes of a non-programmer. The only work that the music puts to the system is generating the samples and placing them in a folder hierarchy.

Because of this reasons it was decided to read the different samples in the way of a folder tree inside the hierarchy of the working assets folder, as this kind of manipulation is a basic skill of computer software that requires a small amount of effort. In fact, if the user of the tool decides to load an incorrect hierarchy, an option to generate the correct one is given. This is an *offline* (not in the game's runtime) method that takes around one minute for around 50 different samples (some limitations are present as the engine's input-output functionality needs to be used in order to serialize data in a way that can be modified in the editor).

A database system could have been chosen to input the classification data for each audio clip, but having this hierarchy permitted storing the hierarchical data in fast-access data structures as dictionaries.

This folder structure obeys to an arbitrarily chosen parameter hierarchy, commented previously, that is: Layer -> Mode/Chord Rotation -> BPM (Beats Per Minute) -> Sample Essentiality -> Sample Family.

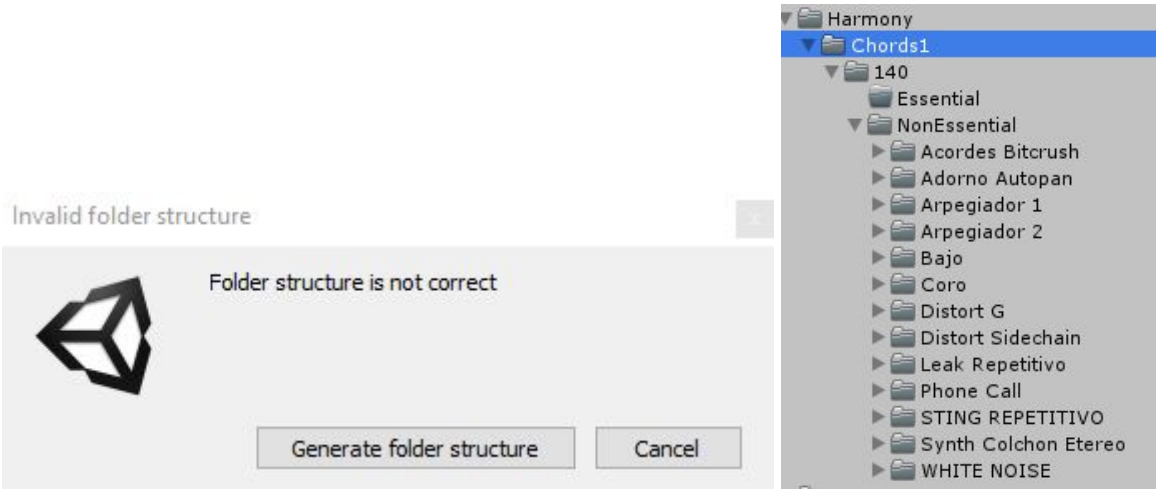

Fig 9: Left: Prompt that pops up when the input clips folder structure is not correct. Right: Folder structure in the editor

When successfully loading the clips, the system generates an equally structured hierarchy in the selected output path that contains the respective data at each level, and an object called *Library* that serves as parent for all nested references.

4.1.2. Generation Algorithm

Choice of parameters

This is the first of the processes that take place at runtime, in which mode and bpm are set for the rest of the operation. A decision has been to select both parameters by getting a random element of their respective lists in order to set the general data by its properties. The logic behind is that more populated parameters should repeat more as they have a greater amount of combinations and thus it is more difficult for them to sound repetitive.

Selection of samples

Samples are chosen by random for each layer, but without discriminating between segments yet. There are several amounts defined for choosing a correct amount of quantities for each segment, that in turn tell the total samples for each layer that have to be gathered. These are: **minimum samples in layer, maximum samples in layer and minimum samples in segment.** With this restrictions, if by chance the total selected samples for a sample group were greater than the total of that group, the final assigned samples would be the necessary multiple of each different one plus a random non-repeated pick of the remainder.

Generation of segments

Once the different segments for the song have been chosen, they have to be arranged for the different segments. Each part has a different policy:

- For the Base layer, sounds can be repeated between segments. Also, when the essential samples start to play they cannot stop to without a good reason. That means that all verse and chorus part have all the indispensable samples.
- For the Harmony layer, samples can be shared between segments, but without any special restriction.
- For the Melody layer, samples that belong to certain segments cannot go into others. For example, a melody sample selected for the chorus should not be repeated in any other segment that is not a chorus.

Placement of segments

Finally, the segments are initialized and placed throughout the song until its duration is fulfilled. The composing follows a set of rules:

- If the song duration is lesser than 3 segments, they will only be segments and choruses.
- If the song duration is greater than 4 segments, it will have an intro in its first slot.
- If the song duration is greater than 5 segments, it will have an outro in its last slot.
- If the song duration is greater than 6 segments, a bridge can appear after any chorus but the last one.
- Verses are followed by choruses.

4.1.3. Playing the song

After the song has to be generated, it has to be prepared to be played for the system and to connect seamlessly between segments. This problem has been solved integrating a system of segment pooling and preparation.

When the song is commanded to start, a pre fixed time buffer is given before it does. Previously, the first segment of the song has been loaded asynchronously in the shape of audio sources and the second one has begun loading in the same fashion. Then when the time comes after the buffer, those audio sources are prepared to be played at the same time.

Then, when a segment changing phase is approaching, the next segment sources are prepared to play and the following segment to the next loads asynchronously. But here is where the pooling enters in place. The segment that is finished is not removed if the pool does not have yet an instance of it. That enables it for use in future cases without having to instantiate another identical segment.

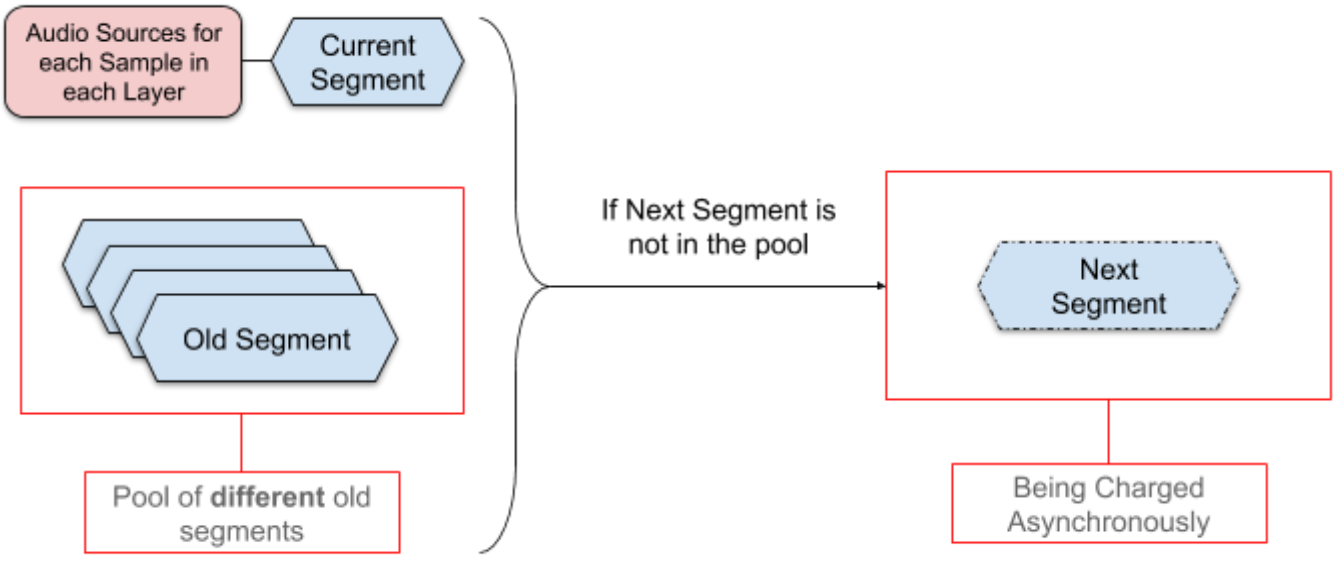

Fig 10: Asynchronous segment loading.

4.2. Track Generation

As the character has to pass through an environment that matches synchronically with the music, a track has to be generated, harboring the information of the dangers that it holds. Two different phases are distinguished: data and physical, so the process is simpler and also a given environment can be stored only by its relevant data.

It is relevant to remark that the track generation algorithm presented in the following sections is the last version of many iterations and changes. Some elements like average dangers per second were kept since the beginning, but the first stub generated dangers at the same time it analyzed the intensities of the song, and used a complex (and thus more difficulty maintainable) formula to combine the distribution of dangers with the song intensity. Throughout the development it has been decided to use enclosed bezier curves instead of mathematical functions where possible for a better maintainability and customization.

An important thing to outline is that danger incompatibilities has not been implemented as ultimately generating one danger per interval as a maximum is enough (and too much for an average player to overcome), and danger duration is also one interval for the developed dangers.

As the track moves instead of the player, in order to avoid potential floating point overflows or lack of precision given big numbers that can come out of a very long song, the environment is moved instead of the character, in an inverse motion of the player's virtual one.

4.2.1. Track Data Generation

Song Intensity Collection and Post-Process

Firstly, an array with the intensities of the song is collected, divided in intervals of which a full beat is a whole multiplier. This intensities are calculated summing up all of their parts, and only when all of them are added in their slots, normalizing using the highest intensity found.

After this first part, post process techniques can be used to give emphasis to different aspects of the song. The ones that have been used here are:

- **Pseudo-sharpening filter**: for each interval that surpasses a predefined normalized intensity threshold, its intensity is multiplied by a value to increase its number.
- **Beat booster**: for each interval that corresponds to the start of a beat, its intensity is multiplied by a value to increase its number.

Before certain post-process techniques, the array has to be normalized to ensure correct results. In this case the normalization has only been necessary at the end to return the array, as the execution order has been as presented.

In order to determine whether a danger is created, a random number in the [0,1] span is tested against a threshold that is calculated through a pondered mix of two methods.

Danger Distribution Threshold Calculation

Dangers have to keep a certain constance in their distribution to not be heavily concentrated in intense parts. This way an average number of dangers per second can be defined and the current amount proportion against it can be evaluated in a curve that gives a normalized result to be used.

Intensity Threshold Calculation

Also, dangers should be placed proportionally to the intensity of the music on the studied interval of the track, so another curve has been defined for the relation between the current and maximum (1 because it is normalized) intensity to be evaluated in.

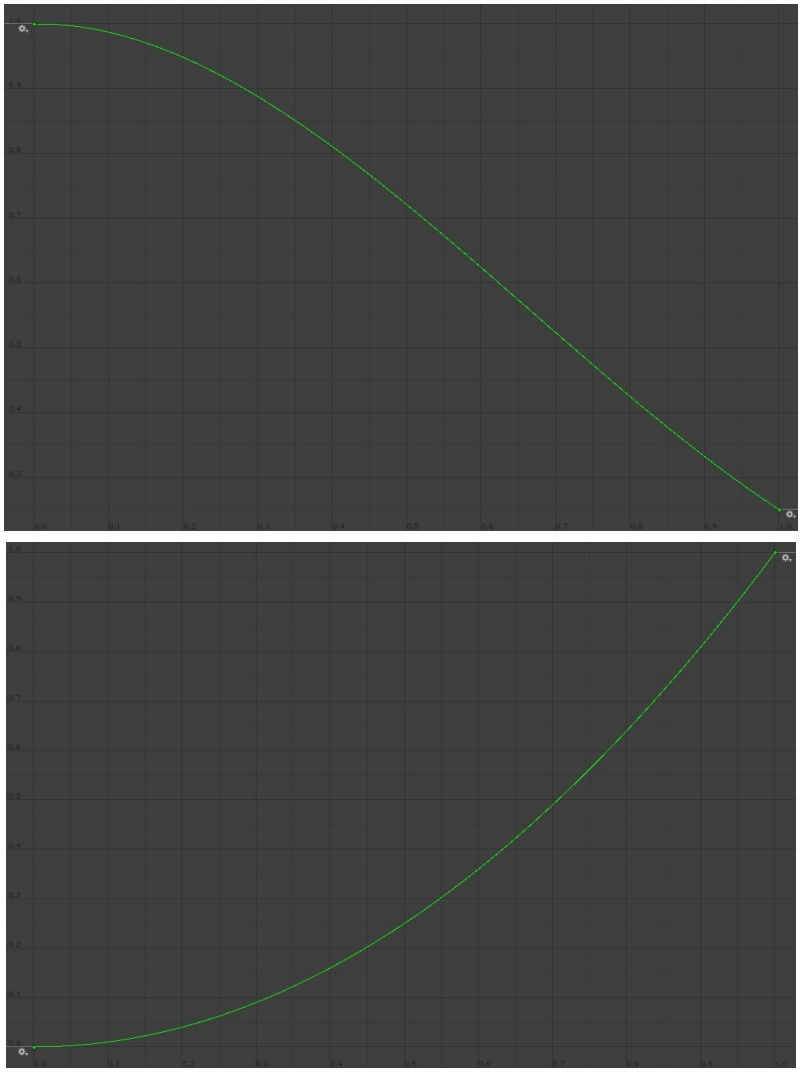

Fig 11: Left: Danger generation threshold to danger amount spline.
Right: Danger generation threshold to intensity spline.

4.2.2. Physical Track Generation

In a separate time than the track data generation, a given track data collection can be instantiated into a *physical* track, meaning that can be traversed and seen by a player. The logic followed has been generating self-filling tiles corresponding to the music and track's time intervals, and passing the full tile data to it so it can decide which elements to spawn and which ones not to.

A normal tile typically spawns a floor (unless a gap danger is placed in it), and all the necessary dangers. In turn, if it corresponds, dangers place and initialize themselves using the necessary data. Danger genericity allows for the logic to be encapsulated in each different element and for this to be a simple process that takes no more work.

Something to comment is that at the start of the level some preparation seconds before the song are given. Because of that, a corresponding preparation track segment, that does not contain dangers, is generated at the start to be synchronized.

4.3. Game

When the game starts, the character starts running and reaches the first beat of the song at the same time it begins playing, so the synchronization is assured as the player moves at a constant speed depending on the song's Beat per Minute and Meters per Beat relationship, established as parameters.

4.3.1. Character

4.3.1.1. Controls

The project uses Unity's functionality to allow for users to change controls as they like, but the defaults have been set for keyboard and controller so they are quick to understand as they mimic the standard fps controls of many other games.

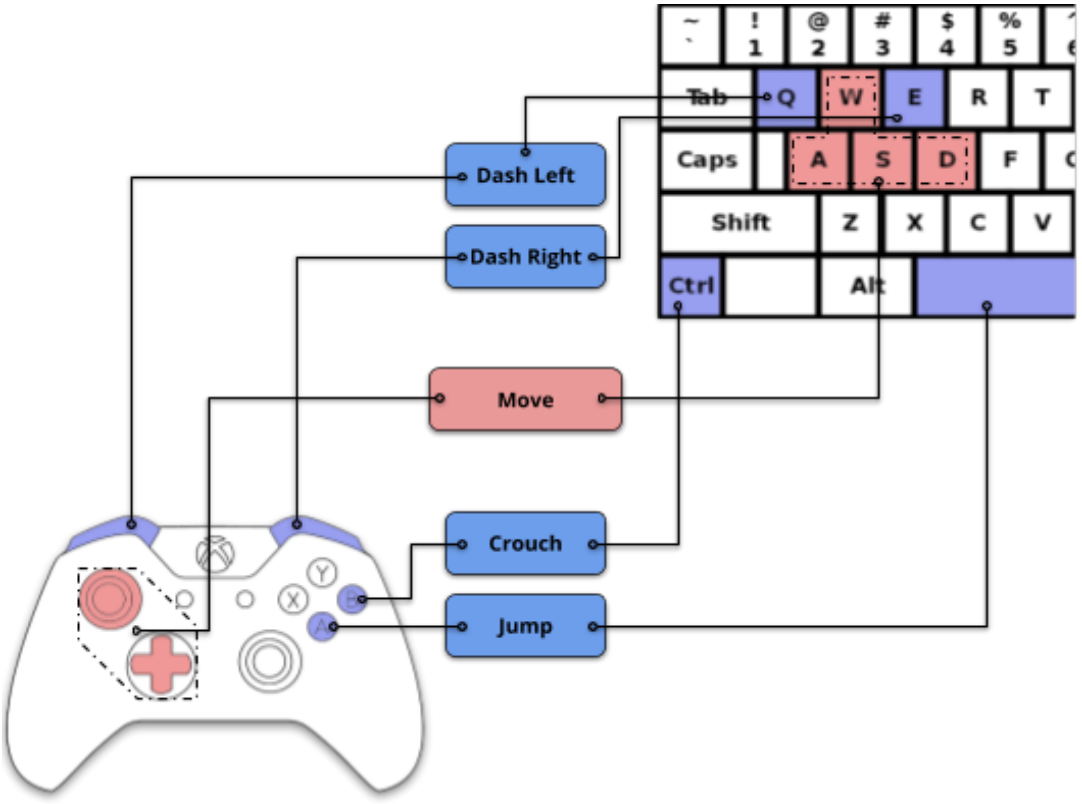

Fig 12: Default game controls

4.3.1.2. Finite State Machine

A Finite State Machine (FSM) has been implemented for the character controls. This allows a wide variety of techniques, situations and skills to be firmly defined whilst reducing complexity.

An option for this system could have been to re-purpose Unity's Animation FSM for this case, but it is widely known for its low performance, so a custom generic FSM was implemented, and a specific instance for the character was created, with its various components.

Firstly, the generic Finite State Machine components will be explained. Any instance of it can be manipulated through the editor instead of having to code specific arrangements, so a direct manipulation is offered to non-programmer users in the case of translating this functionality to future projects.

- **Controller:** the general controller that contains and changes ultimately the current state and fires the update events for it. It also has general ongoing actions that are executed always while the machine is on.
- **Action:** the place of a given state machine where the functionality resided. They hold the system changes and manipulations that are ultimately sought for the system independently of its implementation. A user of the generic state machine has to create as many actions as desired. In this project it has been decided to separate actions in two kinds to ensure more safety when manipulating them in the editor.
    - **Instant Action:** these actions are executed once when transitioning (both from the transition or from the exiting or entering state).
    - **Ongoing Action:** these actions are executed continuously inside a state, or in general actions held by the controller.
- **State:** the main hubs of the others kinds of data except the controller, through which it moves executing actions and checking transitions using their decisions. In this way states could be understood as an abstraction layer between controller and the rest of elements.
- **Transition:** conditional unidirectional translations from a state to another held by the origin state. Each transition held by a state is checked in each update cycle for its validity. The conditions of transitions are represented by decisions, of which one or multiple can be had. All decisions of a transition must be in a valid (true) state for the transition to fire.

- **Decision:** conditions that transitions hold. These can be understood as the counterpart to actions, in the way that both depend on the specific system. A user of the generic state machine has to create as many decisions as desired.

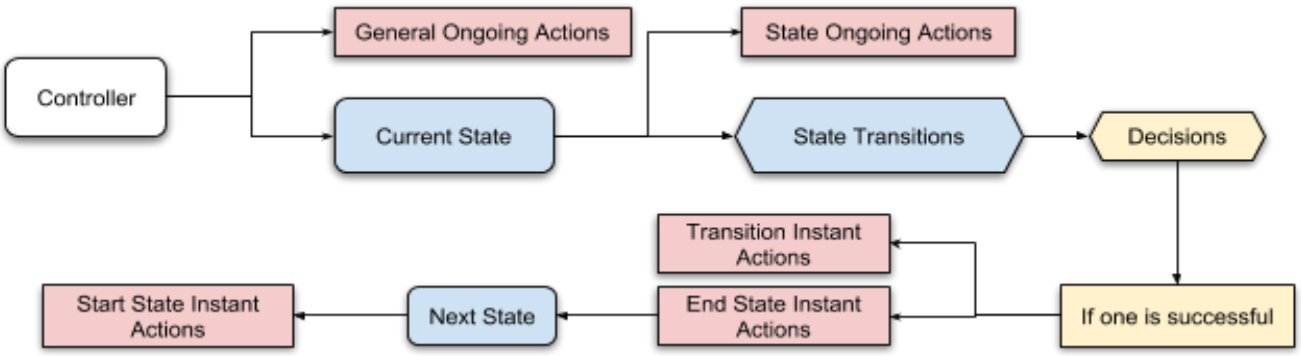

Fig 13: Generic Finite State Machine update flow

4.3.1.3. Character Movement State Machine

This specific instance of the generic Finite State Machine is intended to control the movement and player interaction of the character. All given skills also have their shape here (each one different), as they often set out special situations that can are mutually exclusive (and hence can be represented as states.

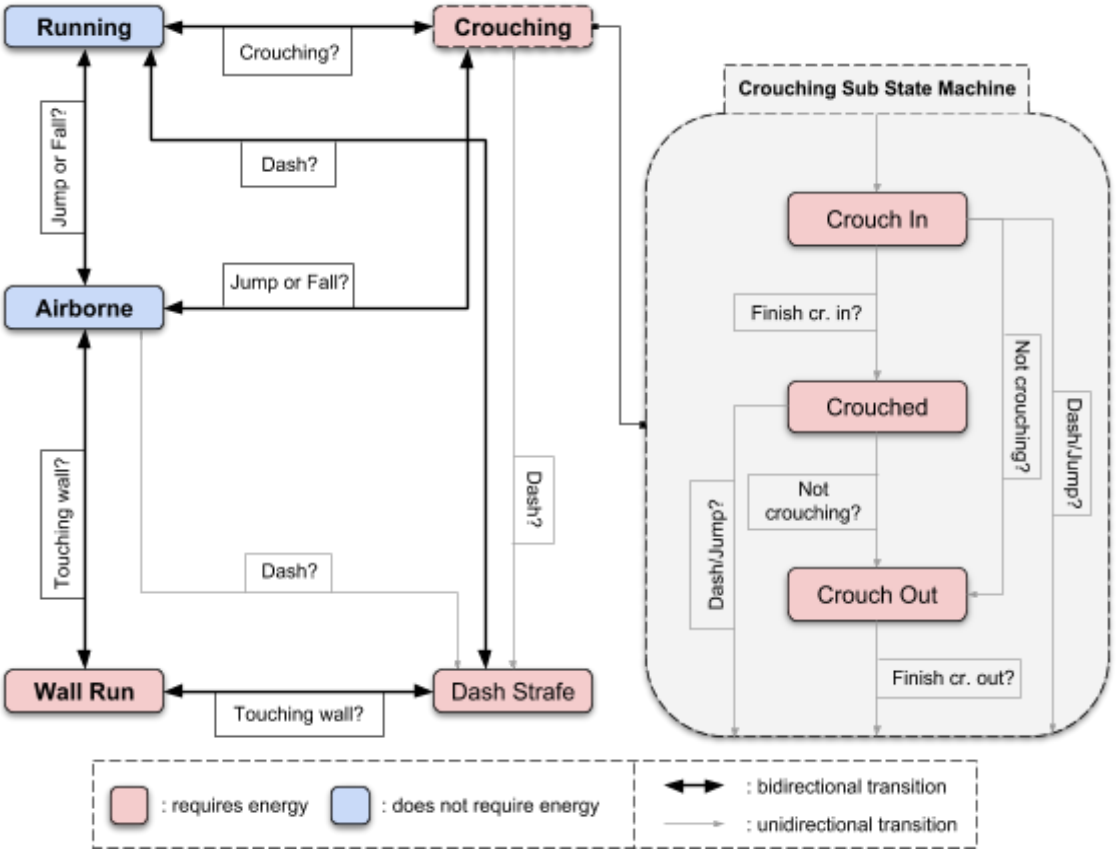

Fig 14: Character movement state machine.

All states of the state machine affect in different ways to the movement. In any case, what ends up being altered is character's virtual speed or virtual position, which are later converted into the environment's current translation.

Movement has been embedded inside the different actions of the state machine, but it can be broken down in:

- **Forward movement:** forward speed that is determined by the song's beats per minute and the track's meters per beat in a simple conversion that assures that it's speed is equal to that of the song. This movement is executed in all states and is constant.
- **Lateral movement:** also known as strafing, it is controlled by a simple physical system that takes in account acceleration, drag and recovery strength. This last element is relevant even when it's not realistic, as it allows to change movement orientation quickly which leads to a better control experience. Different states have different strafing instances controlled by different amounts of the aforementioned parameters giving, for example, a much better control while running compared to being airborne.

  Also, a maximum strafe speed and an acceleration curve are used. This last parameter determines how much the characters accelerate in the strafe direction given the difference between their speed and their maximum allowed speed (being possible for it to be negative if the character was going in the other orientation).

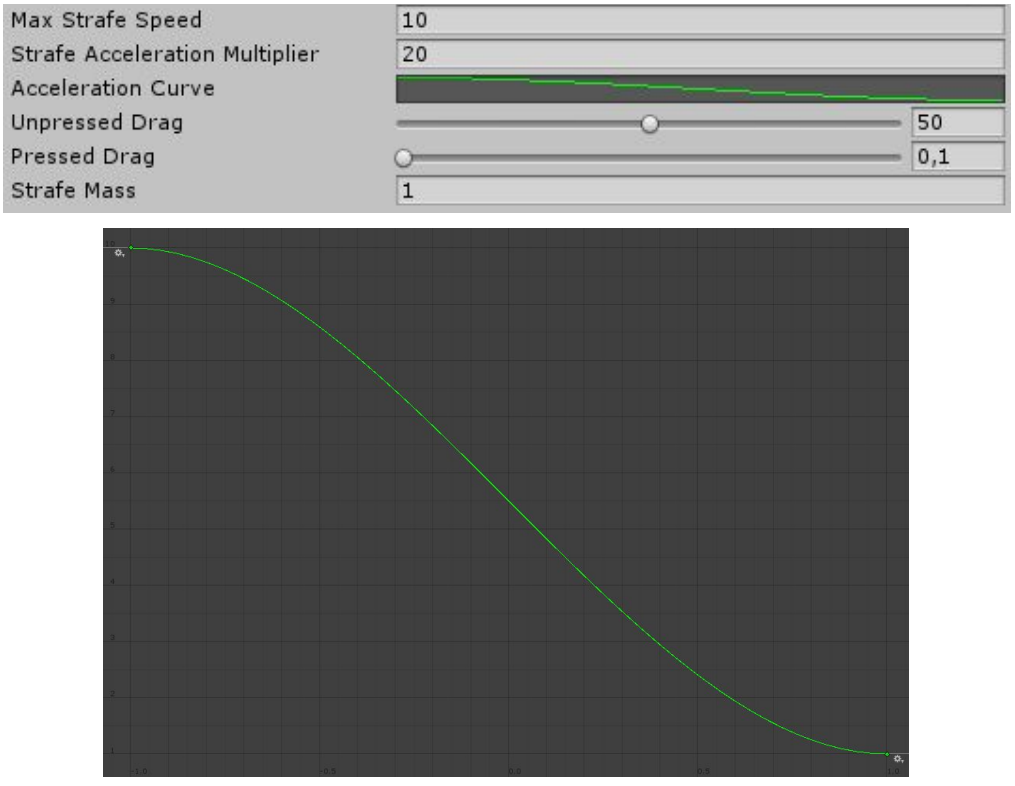

Fig 15: Up: strafing parameters in their running instance. Down: Strafe acceleration curve for the running state.

- **Jumping**: as strafing speed, vertical speed is controlled in its separate axis. In this case, jumping is understood as an instant action inside a transition that gives a great speed push in the vertical direction and sets the state of the character to airborne.

  In this state, vertical speed is varied each frame according to defined gravity's acceleration pressing the button if it's yet ascending or the default gravity acceleration. This lets the player decide how long and high they want their jump to be, inside a certain bracket. This has been taken as an inspiration from platformer games considered canonic like Super Mario Bros [Ni-SMB].

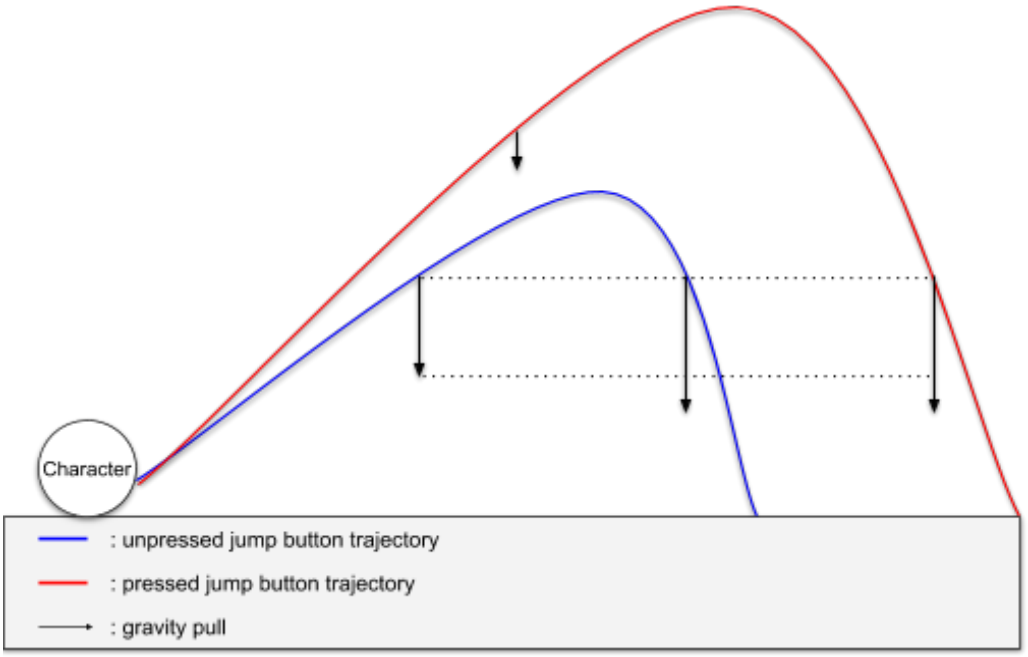

Fig 16: Different jumping trajectories depending on keeping the jump button pressed.

- **Lateral Dashing**: as an state by itself, lateral dashing follows its own defined rules and transitions. Firstly, as in the running state, gravity is not applied, also dashing requires a unit of energy, so the energy pool has to be charged enough. The logic of this kind of movement is applying a quick, linear speed to the selected side, that lasts a fixed amount of time so every dash covers the same distance when activated.

- **Crouching**: for this state it has been sought to give freedom to the player to switch from this state in any of its phases. Usually it follows 3 subphases: crouching in, being crouched and crouching out. This state has its own strafe speed but the most important part is that it decreases the size of the character's collider so it can go through spaces that would not be safe in the normal running state. Currently it is not possible to crouch while in air as a deliberate decision, because there is not any character visual representation so the player wouldn't be able do distinguish its volume in mid-air.

- **Wall Running**: wall running is an state analogous to running that can only be accessed when sticking to a wall while airborne. If the wall has been runned on previously without first touching the ground (reaching the running state), the transition will not be executed to avoid exploiting a single wall.

  A necessary remark is that Wall Running has been implemented and tested but it does not have place in the final version of the game as it added complexity without offering anything differential from dashing or long jumping.

4.3.1.4. Resources

Health and energy have been implemented, leaving Sync to a future iteration of the game, as it was not a high priority.

Health has a quantized functionality, starting a game with 3 points as a default, and also an invulnerability time has been implemented so, after taking damage, the player has the chance to react and stabilize.

On the side of energy, it has been implemented as a continuous value, that regenerates each second. Also, as the player loses health, the energy regenerates faster as a recovery mechanic that intensifies gameplay. This recovery is set by a curve that evaluates the relation between current health and its maximum.

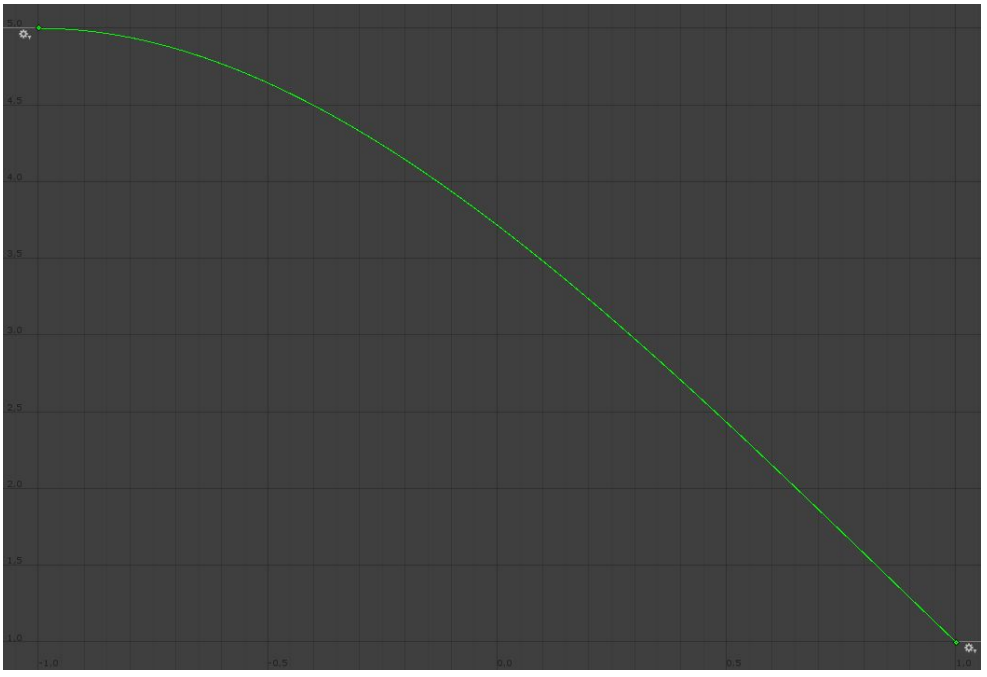

Fig 17: Energy recovery curve from current health.

4.3.2. Dangers

Developed dangers have very different functionality between each other, but all respond to a general functionality: all can be placed inside a tile, initialized and updated, even when this functions can have nothing in common inside.

All dangers in a way or another make reference to the tile that holds them, whose dimensions have been defined in the track generation parameters. This tile can be understood in two different ways: seen from above the relevant measures are its width, manually defined and its length, that is defined by how much a beat spans in meters, and how many intervals are in a beat. Secondly, a tile can be observed in its front section, having said width and a height defined the same way. Thus we can understand a tile as a volume, that will define the placement of dangers. The dimensions of the tile for dangers are taken in a local space that spans 0.5 to each side from the center.

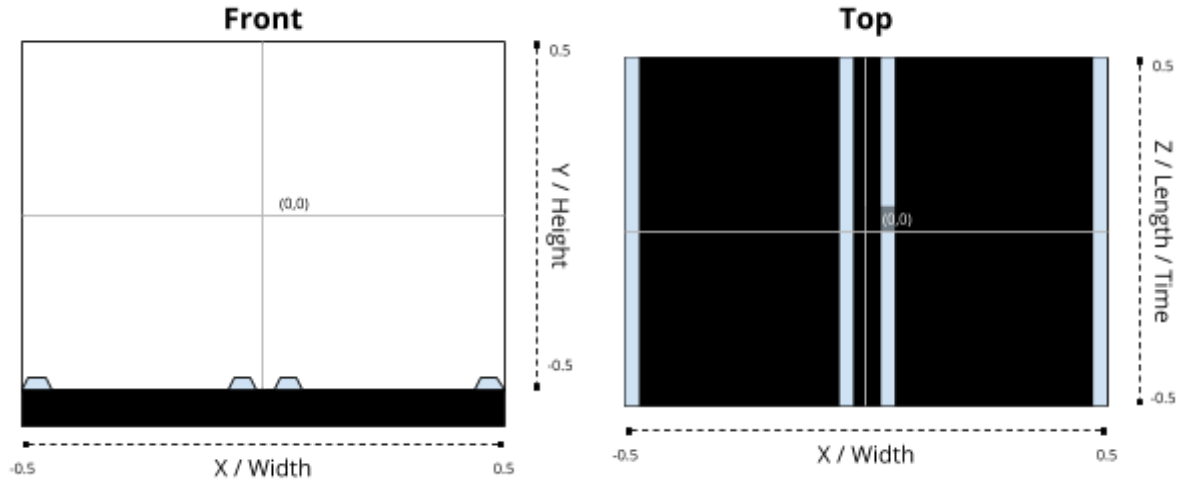

Fig 18: Tile measures

Laser Dangers

Despite their specific kind, lasers are composed of two equal emitters and a line renderer as a beam. Their positions are calculated on runtime, but the emitters are always looking at each other and the beam connects them. The color is set by code, and serves to differentiate between them, being all shades of red.

Lasers have been developed from the interpretation that they can be reduced to a virtual center inside the section of a tile, and an angle between 0 and 180º that sets the tilt of the rect that crosses said center. This center has also been given a restriction in order to no have *aberrant* lasers that just cross a side of the tile and do not represent any kind of danger.

With this numbers calculated, the intersection between this rect and the box that represents the section surface of the tile can be found in both extremes so emitters can be placed.

With this perspective, mobile laser and dynamic ones only differentiate from the static one in that in their update functions they vary center or center and angle, respectively.

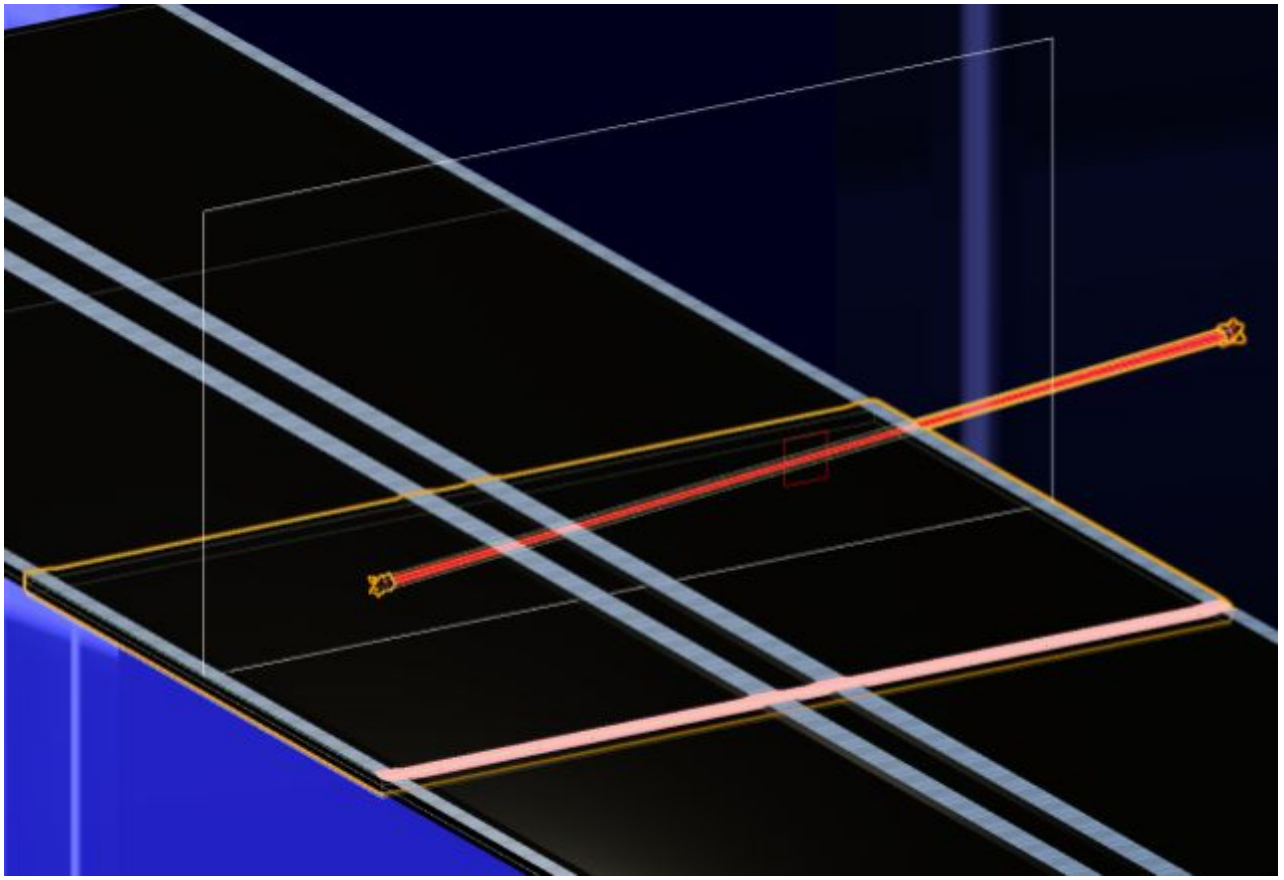

Fig 19: Tile with laser seen from a side in an orthographic perspective. White rectangle is the virtual space of the tile. Red rectangle is the place in which the center can exist.

Gaps

Gaps have been very simple dangers to develop as their only function is to be checked by their parent tile at the time of physical instantiation in order to not generate a floor.

Drones

Drones follow the same logic as lasers, as they get a position local in the section coordinates of the tile. In the case of static ones that is enough, but when referring to goalkeeper drones, they have to sample the position of the player n beats before each time they move so they get to the correct position. This movement is hinted in the timer that accompanies the latter kind of drones, that tells the player when and where they are going to move to.

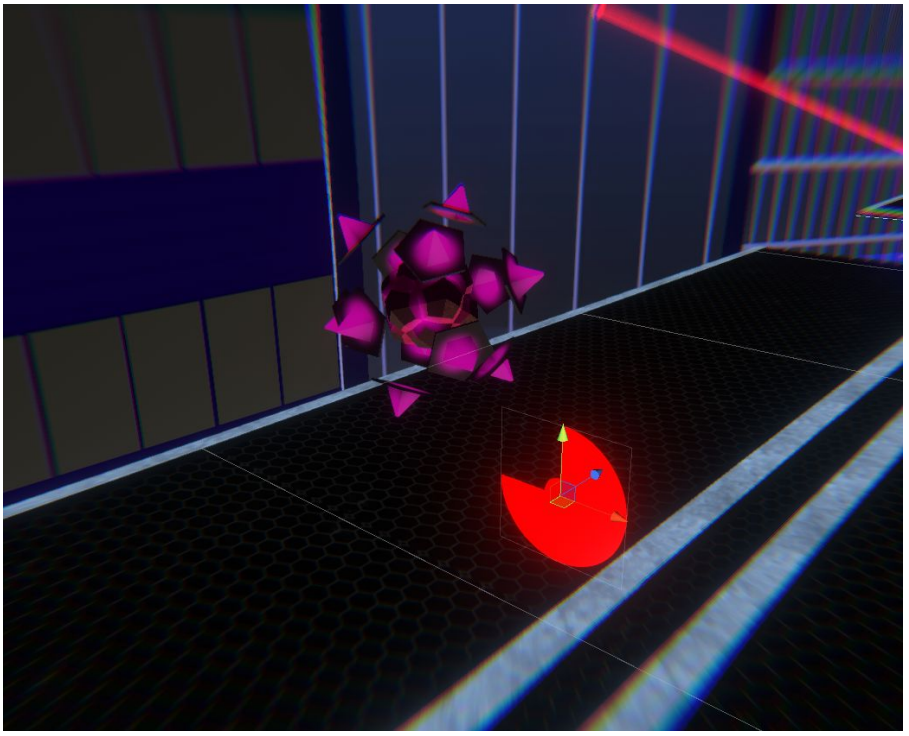

Fig 20: Goalkeeper Drone next to its timer three quarters to completion.

4.4. Art

Artistic style has been developed sticking to the directive of simplicity, as it has never been the center of the project. Also, generating a not confusing art style helps to the readability of the game in a short span of time so reaction times are decreased.

The final aspect of the game conveys two different sensations: the chaotic placed buildings, that mark the “outsides” of the track, and the track in itself, with the neon contrast between very bright and dark colors, using very saturated palettes for interactable items.

Some of the materials for the objects of the game were created using pre-existent PBR (*Physically-Based Rendering*) textures taken using photogrammetry by Dorian Zgraggen and uploaded to his personal repository, *CG Bookcase* [ZgDo-CGB]. These textures were reinterpreted to give effects to the different elements, as dangers or track tiles.

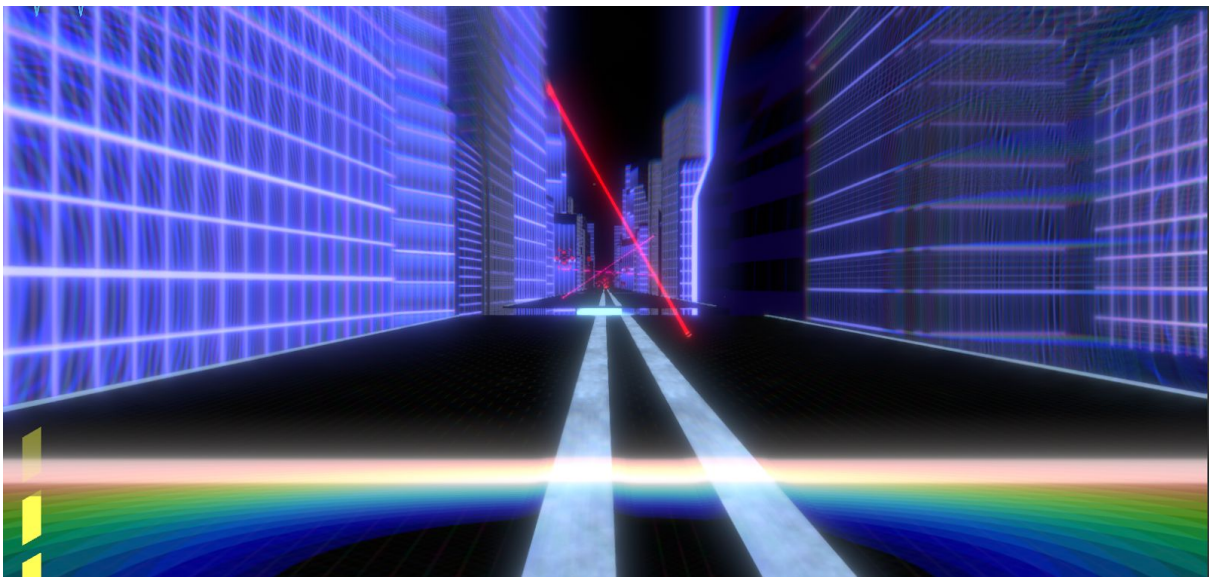

Fig 21: In-Game Capture

4.4.1. Track

There have been developed two kinds of track tiles, whose only difference is the *bar mark*, that as its name tells are used in a periodical fashion to indicate the change of bar. This tiles have a low polygonal value (64 and 92 triangles) to minimize their performance impact.

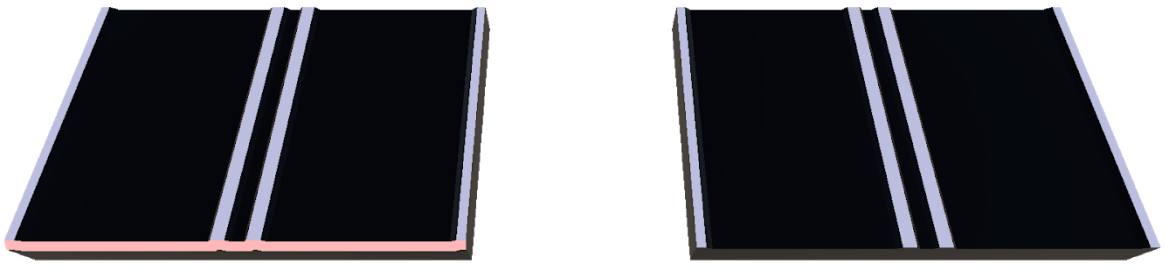

Fig 22: The two track versions: left with a bar marker and right without it.

4.4.2. Environment

For the environment, a pack of pre-made buildings was downloaded from the Unity Asset Store [Aliy-WC], and their materials were altered to adapt to the neon style of the

game, extensively using emission and grid textures further from their original appearance.

To ensure a low processing weight for the buildings, GPU instancing has been used for their materials to reduce draw calls.

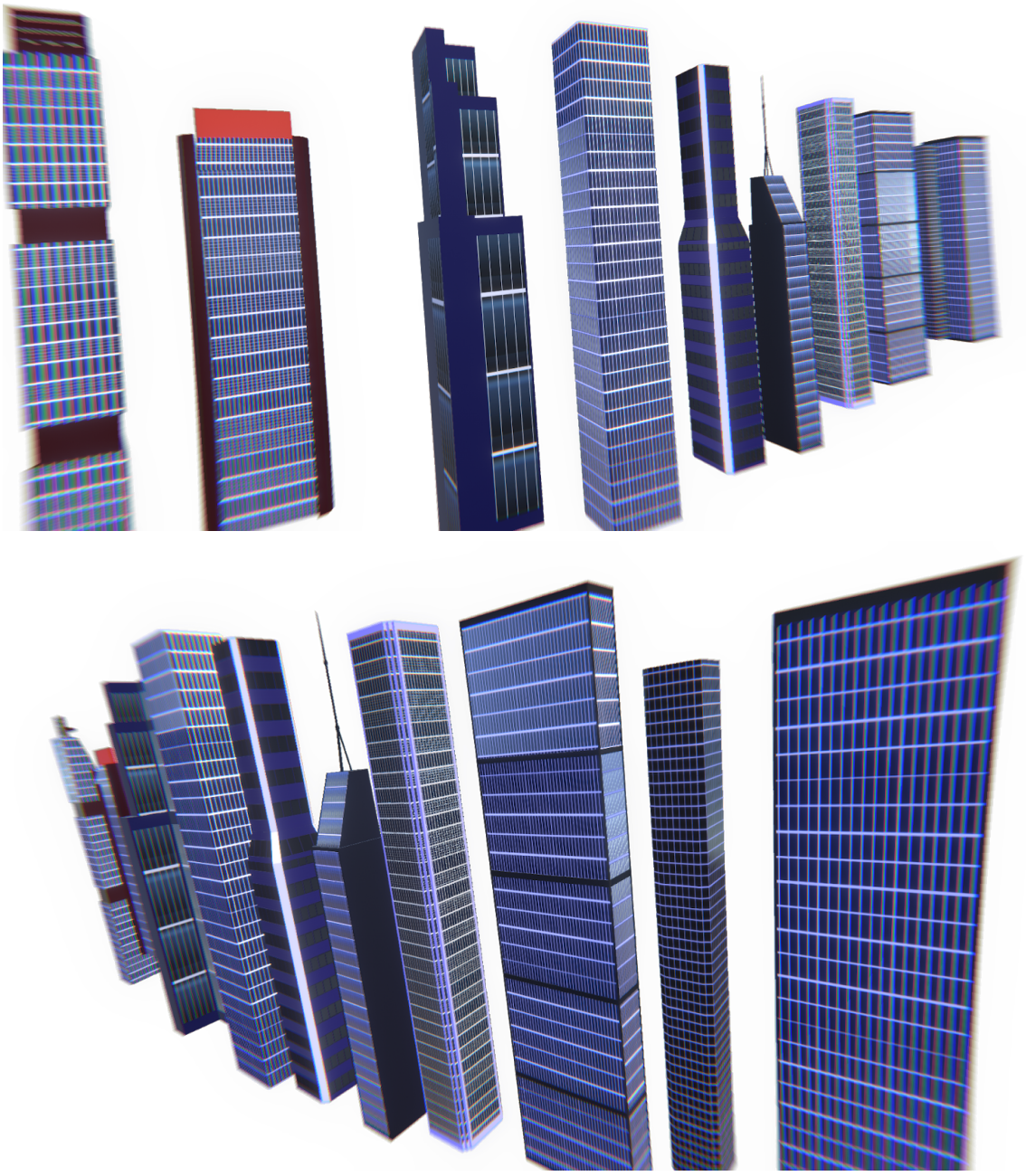

Fig 23: Finished building models from different perspectives

This buildings have been placed using a script that sets layers of buildings with random rotations and separations at the same height to give the sensation of a surrounding metropolis.

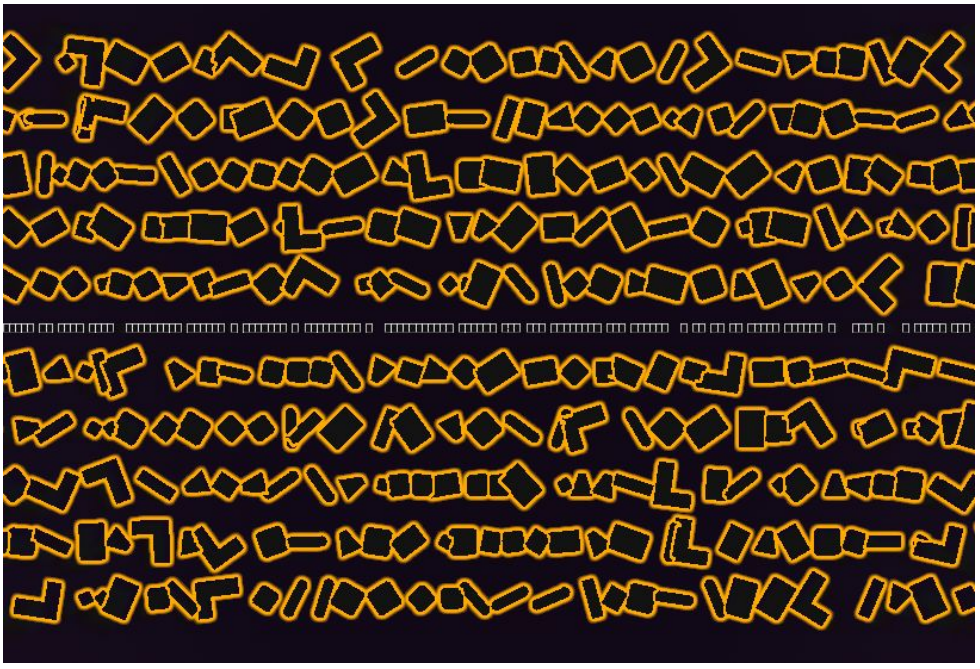

Fig 24: Building layout from top around the track.

Also, the final of the track has been represented as a black hole, that warps the elements seen through. This element has been created to give a clear sense of closure and destruction that meets the musical style and futuristic environment.

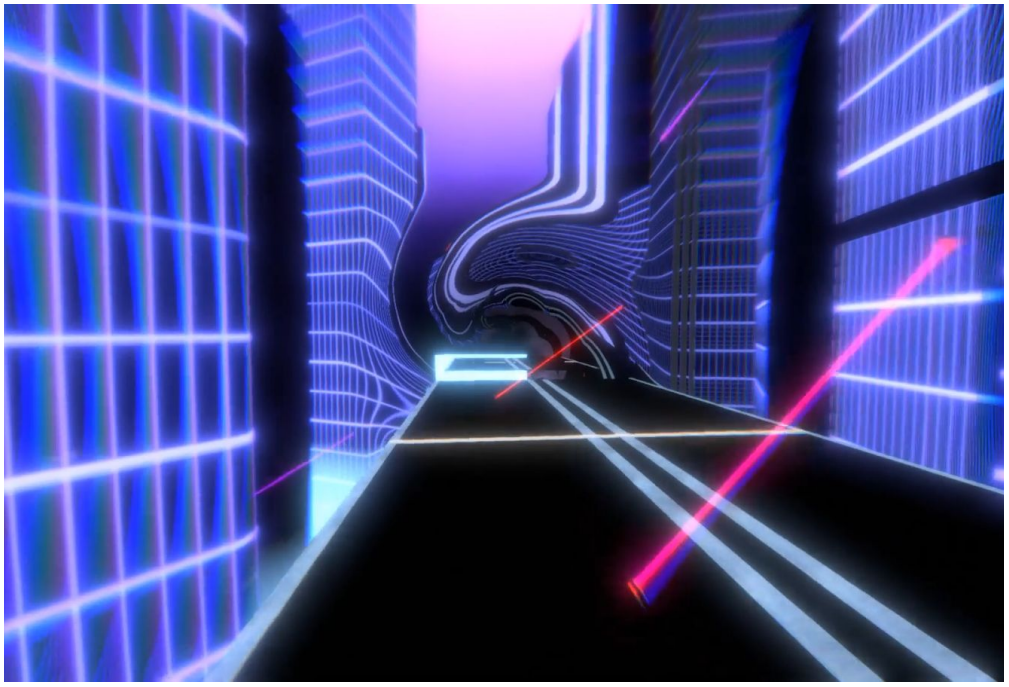

Fig 25: Black Hole effect seen from the track

4.4.3. Dangers

Dangers have been generated in the red shade of colors, having the most contrast between them and the background. Regarding poligonality, they have been designed to be as lightweight as possible, as many instances of each would be active at any given moment, using the technical capabilities of 3DS Max [AD-3DS]. As buildings, GPU instancing has been used for a better performance.

All models have been generated following a technological aesthetic that allows for a low polygonal count so more models can be represented without having to apply more advanced culling techniques than necessary.

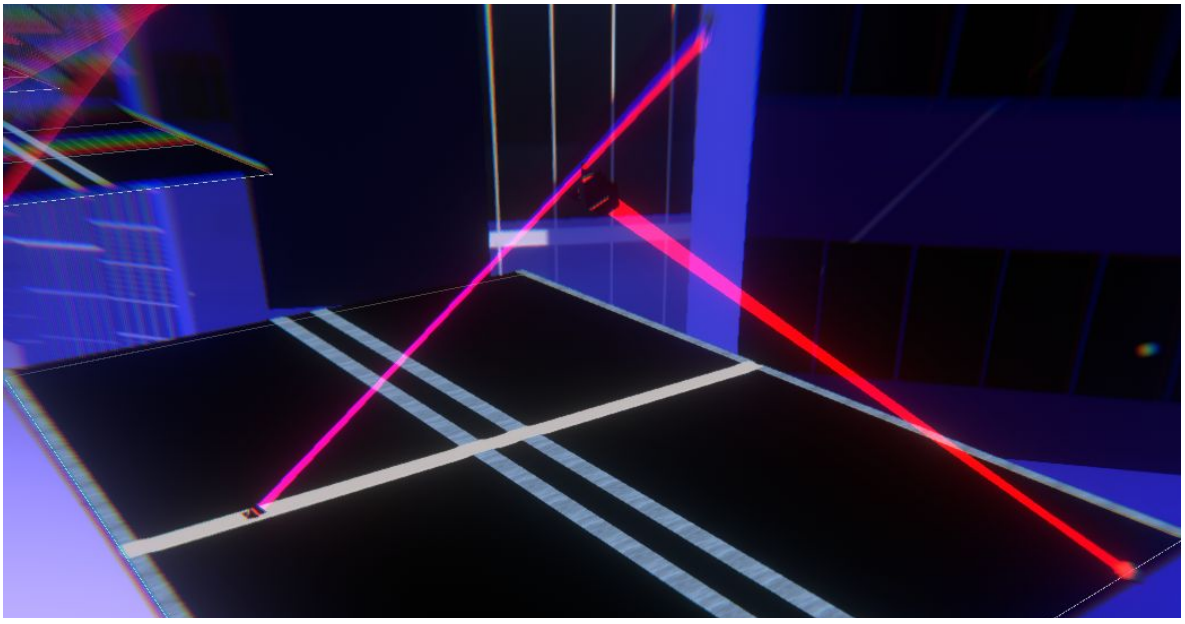

Fig 26: Example of two lasers in-game.

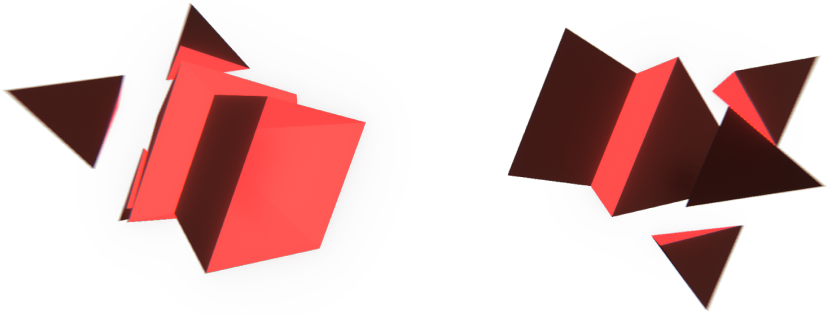

Fig 27: Final laser turret model

Fig 28: Final drones model

4.4.4. Sample Production and Musical Collaboration

Sample production was taken by Pablo Caballero [CaPa], taking in account the technicalities of the project and references like *Alan Parson's Project*, with its album *Eye In The Sky*, and *Justice*'s † album [Ju-†] ranging in the early electronic-disco style range, to *Carpenter Brut*'s *Trilogy* [CaBr-Tri] and *Dan Terminus*' *Wrath of Code* [DaTe-WOC].

Three different iterations were set with the musician for improving the music and discussing the development of the music generation algorithm, with a period of up to one month in between. In these the different sound were polished, and also the algorithm changed greatly adding the information mentioned in the music generation section.

4.4.5. User Interface

User Interface has been designed to be clean and responsive, keeping the visual style of the game. The font selected has been Quantum [OrGr-Qu], being decided to keep the same font for title and text bodies, differentiating by the level of decoration on it.

Some decorators have been placed on top and bottom of menus, and animations have been developed to give a better game feel for this system.

Fig 29: Left: Win Panel. Center: Loss Panel. Right: Pause Panel

Furthermore, a logo and icon have been developed to set the visual brand of the game, using the same font used in the rest of the project.

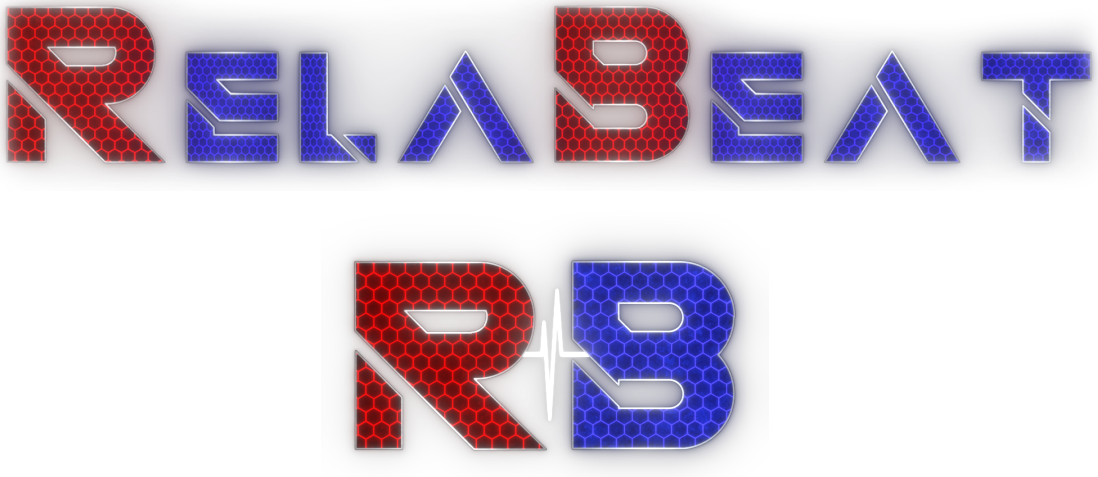

Fig 30: Up: Relabeat title. Down: Relabeat logo

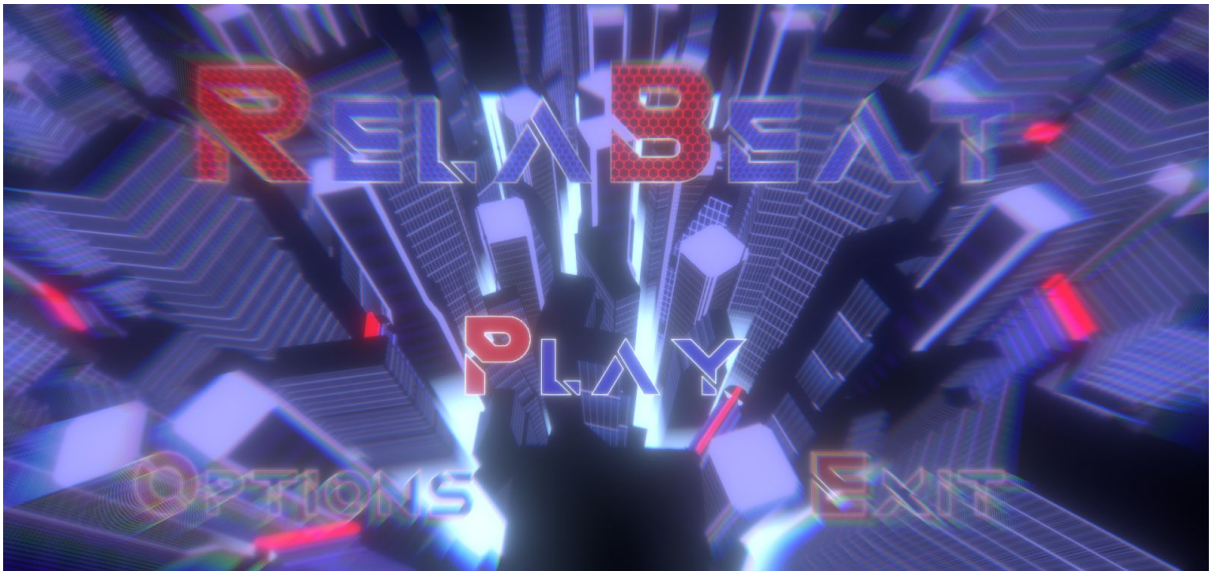

Fig 31: Main Menu final appearance with music-dependent effects

4.4.6. Visual Effects

Post processing has been divided in general effects, that are applied at each frame, and specific effects, that accompany game mechanics or game events.

General effects are bloom and chromatic aberration, that vary to the rhythm of the music in a *dampened bar* style, each with different specific parameters. That means each musical push increases its intensity and it slowly recedes to 0 until the next push. A similar version with different parameters is used for the main menu.

Specific effects are applied to different game states:

- **Game enter-exit**: a black color grading is applied to the screen to generate fade in and out effects.
- **Pause**: the whole game screen is blurred and the pause overlay is placed on top so it is not affected.
- **Being damaged**: a combination of a slight red color grading and dark red vignette is used following an animation curve to communicate to the player that their character has been damaged. Also, a camera shake is used to further convey this sensation.
- **Distortion:** a hyperbolical screen deformation that takes place when the character gets close to the black hole to reinforce its effect.

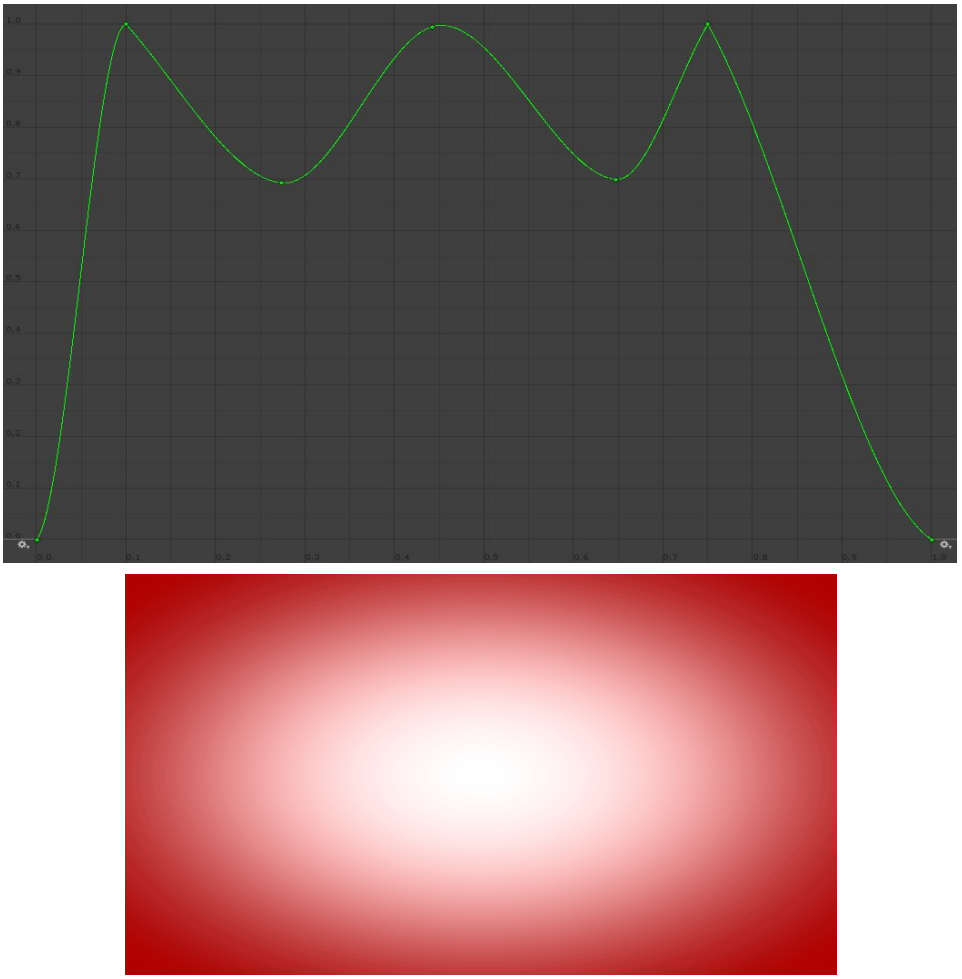

Fig 32: Left: Damaged effect animation curve. Right: damaged effect against a white screen.

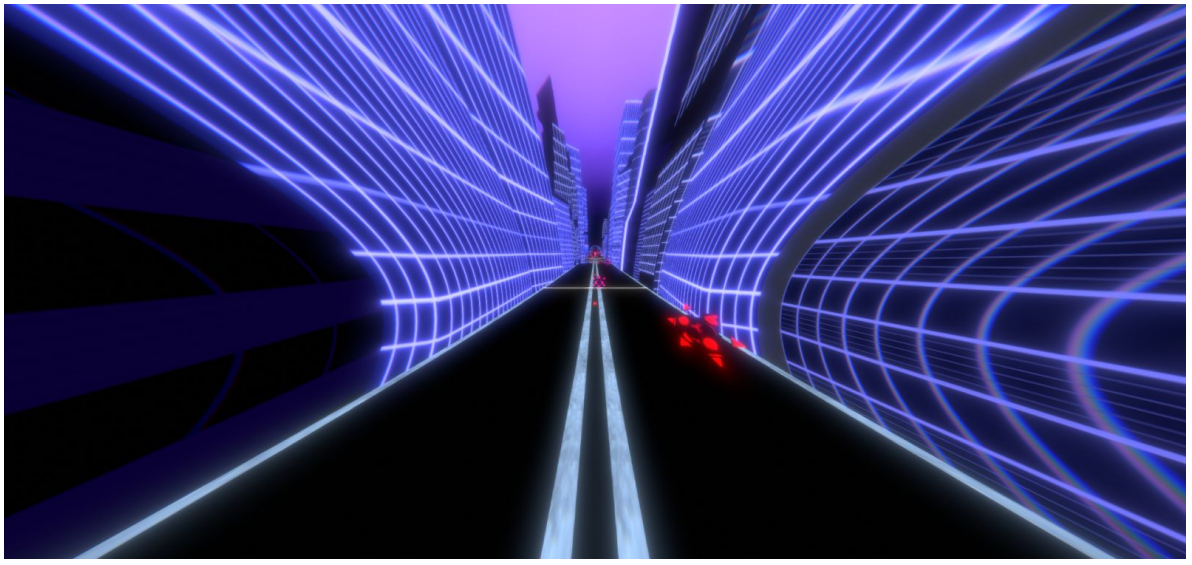

Fig 33: Distortion effect without black hole proximity for better visualization

5. - Results

Code repository: https://gitlab.com/jmpizanagarcia/Relabeat_Open.git
Game build, videos and musical examples:
https://drive.google.com/open?id=1wrTa2b1mnmpdG-RHmobI3FQAujF_h_zP

To start with, Song Generation has resulted in tracks that do not differ much from what we understand as human-composed music. This is in part because the composing parts, samples, are long so the greater source of dissonance (note concatenation) is avoided, but also because the composition algorithm has been guided with musical knowledge. Some testers have got to the extent of pointing out that *'the music didn't seem really procedural*'. The system pipeline is responsive and has allowed for the quick integration of the samples generated in each musical iteration, so in the long run it has saved up time.

Comparing our method with Tyran Daley's *Procedural Music Generator* [DaTy-PMG], it is observed than our method can compose in the classical occidental style, assured by the knowledge contributed by Caballero [CaPa], creating songs that fit said structures (intro, verse…) with an start and an end that do not require a lot of parameter tuning, but an investment in generating samples. This faces with Daley's method and other algorithmical and knowledge-based ones, that can generate music with the deepest kind of flexibility, but without a sense of structure or general order. In the other hand we can compare our method with the mentioned Artificial Intelligences like *Aiva* [Aiva], determining that the scalability of the latter is more effective, as samples are not required, but just input songs to train the neural network; but with a great loss in flexibility that our system keeps because it does not have to stick to a specific genre.

Also relating musical samples, not as many have been developed because of material and economic reasons, so it is more difficult to prove the music generation algorithm functionality, but still a good performance remains for the material and personal resources spent.

Secondly, Track Generation has also given results inside the expected quality. The track is stable, does not present any bugs and satisfies the requirements that were presented, while being fully controllable through parameters to better customize to specific needs like difficulties, dimensions and danger distribution.

This all leads to the product that the previous parts compose: the game. The game has become an interesting challenge for the player, that offers instant decisions and gives a synchronical sensation that was sought from the beginning, supported by elements like the musical response of the effects.

In retrospective, a great amount of musical knowledge has been acquired, and many good practice elements, like decoupling and specifics of Unity like Scriptable Objects have been solidified to serve as the base for this project and others that are to come. The grade of satisfaction is high because of the results exposed, and I look forward to keep developing this product into a fully-fleshed game putting in practice the lessons learnt.

6. - Final planning

The overall first two months of development were less work intensive because of incompatibilities with practices at a company. Despite this, all lost time has been more than compensated from the second third of April onwards, dedicating an average of 6-7 hours from Monday to Saturday.

Despite having to leave some things for future development (which was fully expected from the beginning of the project, as many of the tasks were understood to have a great amount of uncertainty), the majority of them have been correctly implemented, not only for this version, but in preparation for future ones.

New Tasks

The first thing that was not in the initial planification was the tool to allow for a quicker sample loading. This was a necessary development for the agility of iterations, and also a necessity for users of the system that don't necessarily know how to code.

Expanded Tasks

At first, the character movement controllers were thought to be much more simple, with some skills that could be used and generalistic functionality. When the time came to develop it, writing a Finite State Machine rendered more proper as the complexities of movement could be more easily represented, developed and edited in this way, as all the different functionalities coexist in small packets only accessible through its semantics on the editor, and not by function or by code.

Another thing that took more time than expected was the second iteration of the Music Generation system, as the feedback that provided Pablo Caballero [CaPa] needed for the system to be more developed and fleshed out.

The Track Generation algorithm changed considerably from its first stub, as the final approach seemed much more effective, so time was spent redesigning it.

As happened with the character, for the dangers it was decided to finally invest a greater initial time for generating an inheritance system that Tiles could manage with easiness, but from that point dangers were much more easier to develop and the real difficulties of them was in their specific functionalities.

Tasks Gone

Some character skills have not seen the final game, as wall running, that was implemented for a version of the character previous to the Finite State Machine, but that never saw it because of time constraints and the little it contributed to the overall movement possibilities. Furthermore, character Sync because the same low ludic contribution reason.

Also, Silence danger was discarded because it fought against the philosophy of the game, interrupting the movement flow of the player, and resulting in frustration.

Time Balance

To sum up, more time was invested to the project than the initially planned 300 hours. As said previously, the main motivator for this was setting the development base for a game whose span will last more than this project's duration. The final time spent on the project easily surpassed this time.

Even in this situation, the report's writing took less than expected (around 30-40 hours), so not many of other features had to be set to future development.

Roughly, the final layout of tasks can be determined as written on the following table.

Week	Tasks
1	Game definition
2	Movement Experimenting Visual Effects Experimenting
3	Skills Experimenting Art experimenting (technologies, models, materials...) Final game design and algorithms' first versions
4	Design Document
5	Character Movement
6	Sample Loading Tool
6-7	1st Music Generation draft (just one looped segment)
8	Dangers programming: tile structure, and lasers
9	1st Track Generation draft
10	Character Finite State Machine
11	Dangers programming: drones, and gaps
12	2nd Music generation draft (added structure: intro, ending, bridge)
13	2nd Music generation draft (added structure: intro, ending, bridge)
14	2nd Track Generation draft (adding other dangers, tweaks)
15	2nd Track Generation draft (adding other dangers, tweaks)
16	Dangers Modelling & Materials
17	Menus and complete game flow streamline.
18	Game special effects system (post processing, fades...)
19	Sound Effects Game Feedback
20	Report finishing
21	Report finishing
22	Artistic and musical polishing
23	Definite delivery, debugging & tweaking
24	Debugging and tweaking

7. - Conclusions

As a general last remark of the project, it can be said that Relabeat has been a satisfying and instructive experience, ending inside the expected ranges of results. Music generation has been by far the most complex aspect of the project. Many musical concepts have been learnt and implemented, but many other more are yet to be used, as music is a very wide subject with a lot of rules. Regarding future work, musical generation could evolve in the future reducing generatorial elements from samples to more granular items, like instruments or directly numerical seeds from which instrumental lines are generated. Also, a comparative between our method and the different generative methods mentioned throughout the document could be performed in depth to really assure its usability and validity.

For what matters to track generation, many elements can be introduced as side walls, more dangers or loops, but the base is correctly set and works as expected.

Also, the game has resulted in a fun experience and the general feedback obtained during the development has been good. Still, the design space of the current game seems small, and some other elements should be included to give more ludical freedom as path forks, or pseudo-open world, in a style similar to Electronic Art's *SSX* franchise [EA-SSX].

Visual aspect has become more appealing than initially intended, as some simple techniques have been used to enhance it, like post processing behaviours and building placement, so this aspect is interpreted as a full success.

Finally, returning to the musical system, it remains to be tested with a greater amount of samples that span different musical styles, but the first results have been satisfactory and testify as the necessary proof of concept for deeper development. Also, for what matters to the sample loading section, the method used is suitable for a small amount of parameters, but as it scales up, the discarded database method will become the right one to use.

8. – Bibliography

Video Games:

- [34BT -RO] 34 Big Things, 2016, *Redout*, https://34bigthings.com/portfolio/redout/
- [AS-AS] Audiosurf LCC, 2008, *Audiosurf*, http://audiosurf2.com/
- [AV-GH] Activision, 2005, Guitar Hero, https://www.guitarhero.com/
- [EA-ME] Electronic Arts, 2009, *Mirror's Edge*, http://www.mirrorsedge.com/
- [EA-SSX] Electronic Arts, various years, *SSX, SSX Tricky, SSX 3, SSX On Tour*
- [HM-BS] Hyperbolic Magnetism, unreleased, *Beat Saber*, http://www.beatsaber.com/
- [Pi-B3] José María Pizana, 2017, *Ball3*, https://jmpizana.com/portfolio/ball3/
- [Ni-SMB] Nintendo, 1985, *Super Mario Bros*

Music:

- [APa-Eye] Alan Parsons Project, 1982, *Eye In The Sky*
- [Ju-†] Justice, 2007, *†*
- [CaBr-Tri] Carpenter Brut, 2017, *Trilogy*
- [DaTe-WOC] Dan Terminus, 2015, *Wrath of Code*

Programs:

- [DaTy-PMG] Daley, Tyran, 2017, *Procedural Music Generator*, https://assetstore.unity.com/packages/audio/music/procedural-music-generator-99791
- [Aiva] Aiva Technologies, *Aiva*, http://www.aiva.ai
- [JD] *Jukedeck*, https://www.jukedeck.com

Documentation:

- [Reiners-CAM] Reiners, Paul D.; *Cellular automata and music,* IBM's developerWorks, https://www.ibm.com/developerworks/java/library/j-camusic/
- [WoTo-HIW] 2005, *Wolfram Tones, How it Works*, http://tones.wolfram.com/about/how-it-works
- [WoTo-Gen] 2005, *Wolfram Tones, Generator,* http://tones.wolfram.com/generate
- [RoE-3x] Roosendaal, Eric; *On the 3x+1 Problem, http://www.ericr.nl/wondrous/*
- [GaLy-3x] Garner, Lynn E.;
- [FaAn-InPrAud] 2007, Farnell, Andy, *An introduction to procedural audio and its application in computer games*

Assets:

- [DG-DT] Demigiant, *DOTween*, http://dotween.demigiant.com/
- [Unity-PPS] Unity, *Post Processing Stack*, https://github.com/Unity-Technologies/PostProcessing
- [Unity-SA] Unity, *Standard Assets*, https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351
- [Aliy-WC] Aliyeredon, 2016, *White City*, https://assetstore.unity.com/packages/3d/environments/urban/white-city-76766
- [CaPa-RS] Caballero, Pablo, 2018, *Relabeat's Samples*
- [OrGr-Qu] Ortiz, Gregory, '*Seso*', *Quantum Font,* https://www.behance.net/gallery/63174797/QUANTUM-FREE-FONT
- [ZgDo-CGB] Zgraggen, Dorian, *CG Bookcase*, https://www.cgbookcase.com/

People:

- [CaPa] Caballero, Pablo, https://pablocaballeromusic.com/
- [DaTy] Daley, Tyran, https://stickandbindlegames.github.io/

Tools:

- [Unity] Unity, *Unity 2017.4,* https://unity3d.com
- [AD-3DS] Autodesk, *3ds Max 2019,* https://www.autodesk.es/products/3ds-max
- [MS-VS] Microsoft, *Visual Studio 2017,* https://www.visualstudio.com
- [Git] *Git*, https://git-scm.com/
- [GL] Gitlab, 2011, *Gitlab,* https://gitlab.com
- [G-GD] Google, *Google Docs*, https://www.google.com/docs/about/

Other

- [LiuTo-SiRiTr] Liu, Anna; Tokin, Mike, 2006, *Singing Ringing Tree*, Lancashire, United Kingdom

---

Thanks to:

- **Tyran Daley, https://stickandbindlegames.github.io/**
- **Nacho Miralles Tena**
- **José Martínez Sotoca**
- **Pablo Caballero, https://pablocaballeromusic.com/**
- **Kiko Alfaro**