TÉCNICO LISBOA

Building a music rhythm video game

Ruben Rodrigues Rebelo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Rui Filipe Fernandes Prada

Examination Committee

Chairperson: Nuno João Neves Mamede

Supervisor: Prof. Rui Filipe Fernandes Prada

Member of the Committee: Carlos António Roque Martinho

November 2016

Acknowledgments

I would like to thank my supervisor, Prof. Rui Prada for the support and for making believe that my work in this thesis was not only possible, but also making me view that this work was important for myself.

Also I want to thank Carla Boura Costa for helping me through this difficult stage and clarify my doubts that I was encountered this year.

For the friends that I made this last year. Thank you to Miguel Faria, Tiago Santos, Nuno Xu, Bruno Henriques, Diogo Rato, Joana Condeço, Ana Salta, André Pires and Miguel Pires for being my friends and have the most interesting conversations (and sometimes funny too) that I haven't heard in years. And a thank you to Vânia Mendonça for reading my dissertation and suggest improvements.

To my first friends that I made when I entered IST-Taguspark, thank you to Élvio Abreu, Fábio Alves and David Silva for your support.

A small thank you to Prof. Luísa Coheur for letting me and my origamis fill some of the space in the room of her students.

A special thanks for Inês Fernandes for inspire me to have the idea for the game of the thesis, and for giving special ideas that I wish to implement in a final version of the game.

And specially to my mother Maria de Fatima Rodrigues for the support and sacrifices she made for me for all these years. I wouldn't be here without her.

Resumo

A indústria de videogames tem vindo a crescer ao longo dos anos devido às suas contínuas inovações e criatividade. Um dos gêneros de jogos de vídeo existentes é o jogo de vídeo baseado em música, em que o jogo é influenciado pela música e ritmo. Jogos de vídeo baseados em música tiveram um caminho irregular ao longo dos anos, mas hoje em dia eles estão se expandindo não só com uma jogabilidade pouco tradicional, mas com dispositivos inovadores que melhoram a experiência.

Neste trabalho, desenvolvemos um protótipo de um jogo de vídeo de ritmo e música, com características de outros jogos baseados em música de ritmo, em que a jogabilidade é influenciada pela música que é tocada. Para este fim, foi construído um motor de jogo para apoiar um sistema de partículas e criamos um editor de níveis para criar níveis para o jogo.

Finalmente, foi avaliada a experiência de jogo com os jogadores e se o jogo alcança a experiência de ritmo de outros jogos deste género. A partir dessa avaliação, podemos concluir que o nosso jogo oferece uma experiência essencial de um jogo de ritmo da música.

Keywords: Música, Jogo de Vídeo, Ritmo, Desenvolvimento de Jogo de Vídeo, Editor de Níveis, Motor de Jogo

Abstract

The video game industry has been growing over the years due to its continuous innovations and creativity. One of the existing video game genres is the music-based video game, in which the gameplay is influenced by the music. Music-based videogames had a irregular path over the years, but nowadays they are expanding not only with traditional gameplay, but with innovative devices that improve the experience.

In this work, we developed a rhythm music video game prototype, with characteristics from other rhythm music-based games, in which we try to influence the gameplay with the music that is played. For this purpose, we built a game engine to support a particle system and we created a level editor to create music levels. Finally, we evaluated the game experience with users and whether the game achieves the rhythm experience of games of this genre. From such evaluation, we can conclude that our game delivers the essential experience of a music rhythm game.

Keywords: Music, Video Game, Rhythm, Game Development, Level Editor, Game Engine

Contents

List of Figures

1

Introduction

This document reports the creation of a music video game, in the context of a master's thesis. Below is presented the motivation for the development of such a video game, the goals intended to achieve with this project and the outline for the remaining sections of the document.

1.1 Motivation

The gaming industry has grown extensively in the last 5 years. According to "2015 Essential Facts About the Computer and Video Game Industry" [1], in 2014, this industry had sold over 135 million video games and generated more than 22 billion dollars in revenue.

Among the different video games genres, there is one called music video games. Music video games are a video game genre where the gameplay is entirely oriented to the interaction of the player actions in synchronization with the music that is played. These actions are normally on the beat of the music. The beat is often defined as the rhythm listeners would tap their toes to when listening to a piece of music.

Over the years, the music-based genre of videogames had an irregular path of evolution. It started on the second half of the 90s by enforcing its presence in Japan, with games like Dance Dance Revolution. During the 2000s it came to the West, with the game franchise Guitar Hero and Rock Band, which included a controller to try to emulate the experience of using a authentic musical instrument and reaching as a popular video game genre in the 2008, behind the action video games.

But it was during the 2009 that this genre had a decline, because the market was saturated by spin-

offs from core titles, which led to a nearly 50% drop in revenue [2] , leading to a step back for further expansion in 2010. Despite these setbacks, the music video game market continues to expand, introducing dance-based games that incorporated motion control from Kinect, and Rocksmith, that allows the players to play the songs using a real guitar.

Music-based video games have strong emphasis on music in the game, and the player's enjoyment of the video game is directly related to the specific music that is being played during gameplay. Since player tastes for music vary widely, if a song is enjoyed by one player, it might be unappealing to a majority of other players, making the music preferences fragmented according to the different players.

Most music-based video games are created upon the music of a specific popular artist or from a collection of music from a variety of artists for a "general audience" of video game players. But there are video games that are customizable to the point that the player can choose which music should be played by downloading content, in order to reach more players.

Also, some music video games have multiplayer mechanics, either cooperative gameplay or competitive gameplay, improving the video game experience for the players.

Playing this type of game is normally synonymous of performing a show. And one of the type of shows that is highly appreciated is the show of fireworks. But among this type of firework shows, the ones where there is music and the music is synchronized with the explosions give a better experience.

Considering the situation in which music-based video games are, in our work, we intend to tackle the issues that arise when building a music video game by making use of knowledge from areas of computer engineering and the area of game design, in order to create a fun video game where the gameplay is influenced by the music that is being played using the theme of fireworks.

1.2 Goals

The main objective of this work is the development of a music-based video game, where the music or song influences the gameplay with the theme of syncronized fireworks. We also develop a level editor for a easy level creation. We intend our game to:

- Have a song sincronized with the gameplay.
- Have a particle system for simulating firework explosions.
- Have a fluid, lag-free performance.
- Verify that the music can influence the game experience of the video game.
- Be influenced by existing video games.

The challenge will be to design such a videogame that can be appealing to a large audience.

1.3 Document Structure

The rest of the document is organized as follows:

- The *Background* section describes the history of music-based video games. Additionally, there is a brief description of the principles presented in most music video games.
- The *Related Work* section describes the study of the state of the art on music-based video games and how the music in these games influences the gameplay.
- The *Game Design* section has a description of the general architecture, and the main features that are implemented, in terms of game design.
- The *Implementation* section describes the steps taken to implement the various game design decisions followed in the game design section and the technical challenges that we had during development.
- The *Evaluation* section explains the tests that were performed and which metrics that were used in order to validate the our work.
- Finally, the *Conclusion and Future Work* section sums up the important aspects and decisions taken for future work.

2

Background

In this section, we will start by presenting the history of music-based videogames and then we will present some principles of this game genre.

2.1 History of Music-based Video Games

Before the current music video games for consoles, personal computers and arcade machines, this genre of games started with a electronic toy Simon, built in 1978 by Howard Morrison and by the inventor Ralf Baer. Launched by Studio 54 in New York, Simon had become a cultural icon in the beginning of the 80s, making big success not only in the USA but also in other countries, selling millions of units [3].

In Simon, the player has to press a sequence of randomly generated buttons, each one producing a musical note. For each successful sequence the player answers, a new note is added to the sequence, making it more extensive and requiring a bigger memory effort. It will be that characteristic of pressing the buttons at a given order would became the essence of music video games until today.

At the beginning of the second half of the 80s in Japan, the music-based videogames received a lot of attention by the developers, becoming more and more popular. One game was Otocky (ASCII Corporation, 1987) a 2D shooter for the Famicom System, where a note was played on each of directions the player could fire, making it possible to compose music as the player played the video game [4].

In 1996, a musician named Masaya Matsuura used the concept of Simon and with the collaboration of Sony launches for the first Playstation the game PaRappa the Rapper, where the player plays a

puppy rapper in search for the conquest of his love interest. PaRappa needs to overcome situations, like learning kung-fu or even takes a driver's education course to get his license through his adventures. The player presses the buttons in the controller with timing and in sequence for the protagonist to win the rap duels, approaching his objectives. The game was one of the ten most sold games in Japan in the year of 1997 [5].

One of the biggest differences in the industry of video games between Japan and the West (America and Europe) was on the popularity of the arcades. While in the USA, the arcades were on the top in the decade of 1970, and today are in decline, in Japan the arcades continue to have a significant presence on the urban landscape. And it was in the arcades that the sequence games of music have became a popular genre and cultivated a financial success in the West in the beginning of the XXI century. In Beatmania (Konami, 1997), we have control of a similar DJ equipment, turntable included, where the player sees graphic symbols falling in lines, and when those symbols reach the bottom of the screen, they must press the respective buttons to execute a movement. This design of notes "falling" in cascade have became one of the models to the following music-based video games.

Konami still continued to evolve their arcade machines. Dance Dance Revolution (Konami, 1998) became a pioneer game in using the body as the controller of the video game. The player uses his feet to press the buttons on a "carpet" following the commands of the screen, and with that, starts to execute dancing movements. Guitar Freaks (Konami, 1998), Drum Mania (Konami, 1999), Keyboard Mania (Konami, 2000) follow the same characteristics from Beatmania with different controls, like with a guitar, a drum and a keyboard.

The plastic musical instruments started a commercial success in the West with iconic videogames like Guitar Hero (RedOctane, 2005) and Rock Band (MTV Games, 2007) both developed by Harmonix. While Guitar Hero is a sophisticated version of Guitar Freaks, with a plastic guitar as the controller, Rock Band came along with the option of a drum, a microphone for vocals, the use of two guitars.

But in 2009, the market became saturated with spin-offs from the core titles, which led to a nearly 50% drop in revenue for music game publishers [2] and within a few years, both Rock Band and Guitar Hero series announced they would be taking a hiatus from future titles.

Even with these setbacks, the music videogames continued to expand, introducing a number of dance-based games like Just Dance (Ubisoft, 2009) and Dance Central (2010, MTV Games), that incorporate the use of motion controllers and camera-based controls like Kinect. Also the addition of Rocksmith (2011, Ubisoft), a game with the unique feature of allowing players to plug in virtually any electric guitar and play.

Existing games started to change their business model, namely by relying on downloadable content to provide new songs to players. And the introduction of the new generation of console hardware has also helped the return of Guitar Hero and Rock Band titles in late 2015.

2.2 Principles in Music Video Games

Because adequate metaphors are necessary to build understanding in music-based video games, a grammar was build to constitute the principles for rhythm music-based games [6].

In this analysis of structures of games, the authors concluded the existence of certain features on this genre: active scores, rhythm action, quantisation, synaesthesia, play performance, free-form play and sound agents.

Active Score Music It provides a collection of sequences that are used to compose a new score. This concept adapts the score to each performance, turning everyone into a composer. This composition is made in real time, and the score is composed by pieces of composition.

Rhythm Action In this type of gameplay, there is little freedom of expression. The player is stricted to the rules and must react to specific stimuli displayed on the screen and communicated by sound. In the end, players achieve a score and performance points. Performance points are given by the successfull hits, misses and accuracy. There are multiple difficulty levels, and progress depends on the player choices.

Quantisation The beats are set as the base grid that the notes are going to be performed. The player performs an action, but this action is only performed at the right beat providing a immersive environment of visual and sound sensations.

Synaesthesia Synaesthesia is the involuntary neurological state in which different sensations are coupled. This is provided, in video games, by having a background music and synchronise the onscreen action with the fitting samples, for example the visual background vibe with the beat of the song being played. The game rhythm provides the player with hints on where the obstacles are going to be placed and what kind.

Play as performance Performance is when the player plays a guitar, dances or chant songs, offering the player with a wide range of expression, categorizing sometimes as party games. This type of games provides a dedicaded hardware with an specific controller for music games.

Free-form play Offers the player the opportunity to perform based on his expression. It is normally turned into a toy providing instruments to experiment with.

Sound Agents The visual elements and the accompanying sound provides the environment for the game. The action performed to the objects are the trigger for the sound.

It is important to add that this genre falls between either of three groups: they are rhythm games, electronic instrument games or musical puzzles. In rhythm games the player presses buttons in the provided rhythm and this is progressed by increasing speed and complexity of the rhythms. Later, results

are displayed based on performance expressed as a score, allowing competitive play. In electronic instrument games the rhythm - and the melody as well - is generated by the player, the game intends to provide freedom to the player to create his expression. In the puzzle games the player surpasses the challenges by playing a certain musical sequence provided by tunes.

3

Related Work

In this section, we will refer some of the state of the art in music videogames and discuss the influence of music to the gameplay. To finish, we will make a brief discussion, explaining our line of thought to this work.

3.1 Osu!

Osu![1] is a rhythm game originally for Microsoft Windows, developed by Dean "peppy" Herbert in 2007. The gameplay in Osu! is largely similar to a game for the DS "Osu! Tatake! Ouendan!", where a song is played and the player must perform certain actions in accordance with the beat. However , it contains many features that were not in the original game, including multiplayer and also Beatmap (level) creation, among others.

3.1.1 Gameplay

The main objective of the game is to successfully hit enough game objects (hit circles, sliders and spinners) before the health bar depletes to 0 and the song is over. In each level or Beatmap a song is played and the player must hit the game objects at accordance with the rhythm of the song to earn points.

[1] https://osu.ppy.sh/

The hit circles are objects that must be hit within the timing associated with the object. When the hit circle appears, a ring of approach starts shrinking around the circle, and the player must click in the circle when the ring is within the border of the circle. The slider appears as two hit circles with a path to follow. When the ring approaches one of the circles, the player hold and follow the path to the other circle, and then release the button. The spinner is a big circle that appears in the middle of the screen, and the player must hold the button and circle the pointer around the screen to earn points.

Every time a successfull hit object is hit, a multiplier is added and is used to calculate the points earned by that hit object. The bigger the multiplier, the more points are earned. If the player misses the game object in time, the multiplier is reset to 0.

At the end of the level or Beatmap, the player views how many points he earned, how many successfull hits, misses and multiplier he achieved, as well as a statistical graphic of performance. The player earns a grade of performance out of SS, S, A, B, C, D.

More experienced players can play more difficult Beatmaps, and try to get a better score by completing the level without many misses to increase the multipliers. Difficulty levels include easy, normal, hard, insane and expert. One Beatmap can have multiple difficulty levels.

For playing the game, the player can use only the mouse to point and click, or the mouse with the help of the keyboard, or even use a digital tablet to point and click at the markers at the timing of the level.

The game still provides with a level editor, where players can create their own levels or Beatmaps, and share in the community of the Osu! game. After that any player can download new Beatmaps at the site and play.

3.1.2 Discussion

The game falls in the rhythm action category. The player must hit the objects in time with the rhythm of the song, and the difficulty level increases by pushing the players limit, by increasing the speed and complexity of the levels.

Also according to the principles of music games it falls in the rhythm game genre, where the player plays the game in reaction to the screen stimulus. At the end of the level, a score of the performance of the player is displayed.

The influence of the music in the gameplay depends on how the level or Beatmap is created. In the creation of the beatmap, the player creates the points of the beats, the points where the notes will be placed in accordance with the time, by setting by hand the BPMs (Beats Per Minute) of the song. This setup of BPMs creates the base in which the game objects are placed in timing with the song. Afterwards, the creator of the level places the hit game objects (hit circles, sliders and spinners), by which the creator interprets best in his own creativity.

Therefore the levels are created by the players, and not by some algorithm. After created, the levels or Beatmaps are placed in a judging stage, where some players of Osu! evaluate the levels before they can be shared in the site, and be played worldwide. It is important to add, that the creation of beatmaps can follow a set of guidelines that are displayed on the site of the game [7] , in which can help in the

Figure 3.1: Osu! gameplay.

creation the level or Beatmap. These guidelines go from simple patterns with the game hit objects to technics used to combine effects of the hit objects.

The music that appears in the Beatmaps is from a variety of artists to give the large audience of the game the choice to download and play the levels.

3.2 Audiosurf

Audiosurf[2] is a puzzle/rhythm hybrid video game created by Invisible Handlebar and released in 2008 for PC. The gameplay of this game mimics a race in a track in which the player must collect blocks to accumulate points and this track is created by the music chosen by the player. The audio of the game won the Independent Games Festival 2008 Excellence in Audio Award, with the hit soundtrack composed by Pedro Macedo Camacho.

The main feature of this game is the ability to use the player personal songs to generate a level track. Afterwards it is possible to compare scores in an online leaderboard, for the competitive players.

[2] http://www.audio-surf.com/

3.2.1 Gameplay

The objective of this video game is to use the ship, controlled by the player, to reach the end of the track by collecting blocks.

In the beginning, after the player selects the song for the track, the level track is generated. The level track has segments uphill, downhill and curvatures for the sides following the attributes from the song. Along with the track, there are blocks that the player must collect to score points. The goal is to collect these blocks and form clusters of 3 or more of the same color. The more blocks accumulated in the cluster, the more points are scored. The blocks in the default settings of the game, appear in hot colors such as red and yellow for more points, or cool colors such as blue and magenta that are worth less.

The players are also awarded extra points for there achievements at the end of each track. These feats can be from finishing a song with no blocks left on the grid or collecting all blocks from a certain color.

In each track the player can be awarded medals such as bronze, silver and gold for each difficulty level, awarded at the end upon reaching the total points required.

The goals of the game can also change depending on the game modes available to the game. For example, in the “Mono” game mode, players must collect the color blocks, regardless of the color, and avoid the grey blocks. If the player can avoid the grey blocks until the end, a 30% bonus is awarded to the additional sum of points earned. Upon completing the track, the score is uploaded to the score server, together with the length and form, any feats achieved and the character used. Scores are stored per song title, allowing people to compete. From the moment the length and the form of the track are uploaded, it is possible to detect mislabelled or variant versions of songs in the score browser, and also the option to report possible fraudulent scores. There are three different high score lists for each song, for casual, pro, and elite characters respectively, characters with which the player can play in the game.

Sometimes, during gameplay, Overfill may occur. Overfill is caused when the player attempts to add a block to a column which is full and has no blocks forming a match. Doing so will cause the column to empty all blocks, causing a substantial loss of points. It is not possible to cause overfill for about a second after you fill a column up to the top - blocks moved to this column are simply not added.

The player has the ability to choose from 14 different characters distributed between 6 types and 3 difficulty levels, each of which have different abilities that can be performed to aid in the completion of the level. There are also powerups in some levels (depending on the difficulty), that add some gameplay variation to how the level is completed.

The game also allows the player to choose how to control the ship, either by mouse, arrow keys, number keys or a gamepad.

3.2.2 Discussion

The category of this game is a musical puzzle, where the player collects blocks and matches them in clusters of three or more blocks of the same color.

The genre falls onto the rhythm action, where the player avoids and collects blocks in synchronization

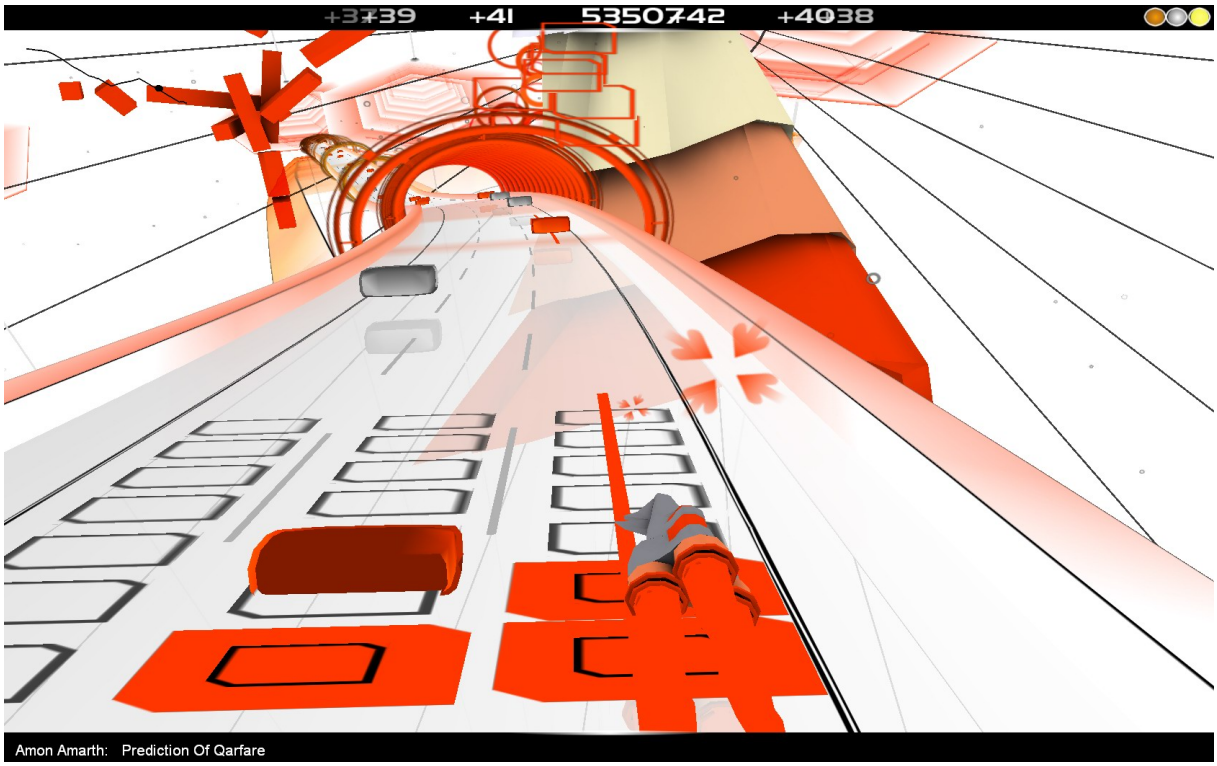

Figure 3.2: Audiosurf gameplay.

with the rhythm of the music being played.

The influence of the music on the gameplay depends on how the level is generated. In a interview the creator of Audiosurf Dylan Fitterer [8] stated that: The track of the level is generated based on frequency analysis. When the music is more intense, there is a steep downward slope and when the music is calmer, the ship is going the way uphill. Many tracks have uphills and downhills with speed bumps described by the song analysis. The blocks color are determined by the spike and placed in the timeline of the song by the analysis. So the most intense music spikes are blocks, and the less intense become the blocks to dodge.

This game is a example in which a level can be created according to audio analysis of a selected song.

3.3 Guitar Hero

The Guitar Hero[3] series is a series of rhythm music games developed by Harmonix in 2005. In this series, this video game is played by using a special controller emulating a guitar. The main gameplay is to press in the respective buttons of the Guitar-shaped controller, as they appear on the screen to score points and keep the virtual audience excited.

It was created to be a party game, where with the guitar-shaped controller the players pretend to be rockstars.

[3] https://www.guitarhero.com/

3.3.1 Gameplay

The core gameplay of Guitar Hero is similar to the gameplay of Konami's rhythm video game Guitar Freaks. It is recomended to play the game with the guitar controller, but it is possible to play with the standard controller instead.

The video game is played with an extended guitar neck shown vertically on the screen, and as the song progresses, colored markers indicate notes travelling down the screen in syncronization with the music. The note colors and positions match those of the five fret keys on the guitar controller. Once the note(s) reach the bottom, the player must press or play the indicated note(s) by holding down the correct fret button(s) and hitting the strumming bar in order to score points. A Rock Meter changes depending on the sucess or failure caused by the player (denoted by red, yellow, and green sections). If the Rock Meter drops below the red section, the song will automatically end, and the virtual audience boos off the player by his performance.

Player score is incremented for every successfully note hit, and by hitting a long series of consecutive sucessful note hits, the player increases their score multiplier. The game does not focus on accuracy, therefore as long as the player hits each note in the respective window of action, the player receives the same number of points.

In selected special segments of the song there are glowing notes outlined by stars, and by sucessfully hitting all notes in this series it fills the "Star Power Meter". The Star Power Meter can also be filled by using the whammy bar (the lever on the guitar) during sustained notes witin these segments. When the Star Power Meter is half full, the player can activate the "Star Power" by pressing the select button or momentarly lifting the guitar in the vertical position. When the Star Power is activated, the score multiplier is multiplied by two until the Star Power is depleted. The Rock Meter also increases more dramatically when the Star Power is active, making more easier for the players to make the Rock Meter stay at high level. In more recent games of the Guitar Hero Series, the Star Power that could not be incremented until it was fully drained was changed, removing this restriction so the player can collect Stars as you play the song even on Star Power mode.

Notes can be a single note, or composed of two to five notes that make a chord. Both single notes and chords can also be sustained, indicated by a colored line following the marker. The player can hold sustained note(s) keys down for the entire lenght for additional points. During a sustained note, a player may use the whammy bar on the guitar to alter the tone of the note. Also, regardless of whether sustains are hit early or late, if the fret is held for the full duration of the hold, the game will always award the same amount of score increase for the note. In addition the games support virtual implementations of "hammer-ons" and "pull-offs", guitar-playing techniques that are used to successfully play a fast series of notes by only changing the fingering on the fret buttons without having to strum each note. Sequences where strumming is not required are indicated on-screen by notes with a white outline instead of the usual black one. Notes can now be played while a sustained note is being played.

In Guitar Hero World Tour were introduced drums and vocal tracks in addition to lead and bass guitar. Drum tracks are played similarly to guitar tracks, where the player must strike the appropriate drum head

or step down on the bass drum pedal on the controller when the notes pass the indicated line. Certain note gems, when using a drum controller that is velocity-sensitive, are “armored”, requiring the player to hit the indicated drum pad harder to score more points. Vocal tracks are played similar to games of this type of genre, where the player must match the pitch and the pacing of the lyrics to score points. In Guitar Hero 5 allows players to create a band of up to four players using any combination of instruments.

Figure 3.3: Guitar Hero III gameplay.

Figure 3.4: Guitar Hero guitar.

3.3.2 Discussion

Guitar Hero series' categories, in the music-based videogames principles are rhythm action and play as performance. In rhythm action, the player must perform the actions that are shown in the screen and at the end there is a score and performance points depending on the execution of the actions. In play as a performance, the player plays the game in a special guitar that tries to imitate a real guitar, therefore the player can perform like a real guitar rock star.

Guitar Hero falls in the rhythm game genre, where the game button pressing speed and complexity progresses with the speed and complexity of combinations of chords in different difficulty settings.

Since the fact that this video game series uses notes and chords (combinations of notes), to be played in the "guitar", it seems like the player is playing a real guitar. The music really influences the gameplay, since the notes and chords that must be played are in syncronization with the song.

In Frets on Fire, an open source Guitar Hero clone videogame for the PC, with similar base gameplay, it is possible to create new levels with the correspondent song. In this game there are four difficulty levels: Supaeasy, Easy, Medium, Amazing. The difficulty increases as the number of frets (notes) available to play increases, and as the BPM (beats per minute), timing between notes decreases. In Frets on Fire, the level is created in a level editor, much by attempt and error, and not automated. Diverse configurations can be created depending on the creativity of the creator.

The music in Guitar Hero is a collection of songs of many popular artists, with the intent of targeting the game to a broad audience.

3.4 Dance Dance Revolution

Dance Dance Revolution[4], abreviated as DDR, is a music video game series produced by Konami. It was introduced in Japan in 1998 as part of a series of music games by Bemani series and is the pioneering series of rhythm and dance genre in video games.

In this game the player stands on a "dance platform" or stage and has to hit colored arrows laid out in a cross with their feet to musical and visual cues. Players are judged by how well they time dance to the patterns presented to them and are allowed to choose more music to play to if they receive a passing score.

Dance Dance Revolution is an arcade video game only for arcade machines.

3.4.1 Gameplay

The core gameplay involves the player, stepping his or her feet to correspond with the arrows that appears on the screen and the beat. During normal gameplay, arrows scroll upwards from the bottom of the screen and pass over a set of stationary arrows near the top (known as the Step Zone). When the scrolling arrows overlap the stationary ones, the player must step on the corresponding arrows on

[4] http://www.ddrgame.com/

the dance platform, and the player is given a judgement for their accurancy of every note played (From highest to lowest: Marvelous, Perfect, Great, Good, Almost, Miss).

There are different types of arrows. For instance, Freeze Arrows, which is a long green arrow that must be held down until the tail reaches the Step Zone, producing an "O.K.!" judgement if the player succeed or "N.G." (Not Good) if the player fails to do so, or Shock Arrows, walls of arrows with lightning effects that must avoided, which are scored the same as Freezes (O.K./N.G.); if they are stepped on, a N.G. is awarded, the life bar decreases, and the steps became hidden for a short period of time.

Successfully hitting the arrows in time with the music fills the "Dance Gauge", or life bar, while failure to do so drains it. If the Dance Gauge is fully depleted during gameplay, the player fails the song, usually resulting in a game over. Otherwise, the player is taken to the Result Screen, which rates the player's performance with a letter grade and a numerical score, among other statistics.

Aside from play style Single, Dance Dance Revolution provides two other play styles: Versus (Player 1 side of play style Single and player 2 side of play style Single playing together) and Double (One Player utilizes both pads to play). In earlier versions also have Couple/Unison Mode, where two players must cooperate to play the song; this mode later became the basis for "TAG Play" in newer games.

Figure 3.5: Dance Dance Revolution gameplay.

Figure 3.6: Dance Dance Revolution arcade machine.

3.4.2 Discussion

This video game falls in the rhythm action and play as performance categories. In rhythm action the player must react to screen stimulus, and “dance” in the “carpet” by stepping on the correct positions. In the end there is a score and performance grades acording to the player actions. The player “dances” as a performance of the given inputs percepted.

It is a rhythm game genre, where the complexity increases as the speed of the arrows and arrows

sequences complexity increases.

Stepmania, a Dance Dance Revolution clone videogame for the PC, with similar gameplay, gives the oportunity to create levels for the game. First the creator has to setup the BPM (beats per minute) and afterwards the creator places the arrows on the level timeline. Arrows must be placed in a certain order, following a set of rules: the flow of arrows must mimic the existance of a “carpet”, because it has to flow as a dance with the feet (there are some combinations impossible to recreate as the player dance).

The songs in Dance Dance Revolution are from a set of a variety of artists to reach a broad audience.

3.5 Rez

Rez is a rail shooter music videogame released in Japan by Sega in 2001 for Dreamcast and Playstation 2. The game main novelty at the time was the use of sounds and melodies as the player target and destroys foes with electronic music instead of sound effects found in most rail shooters.

3.5.1 Gameplay

Rez is a rail shooter in which the player takes control of an onscreen avatar travelling along a predetermined path. The player does not control the overall path, only the avatar’s position on the screen. The player targets enemies by holding a “lock-on” button while moving an aiming recule over up to 8 enemies. Once the “lock-on” button is released, the avatar fires shots that target to each target. Failure to hit an enemy or a projectile in time may cause a collision, which reduces the player’s current evolution level by one and changes the avatar form. The game is over if the avatar is hit while at its lowest possible level. At higher evolution levels, the avatar appears as a humanoid figure, while it appears as a pulsating sphere at the lowest level.

Some enemies drop power-up items when destroyed. Two different items enhance the player’s avatar by increasing his/her “evolution bar” by one or three points, respectively. Another item enables the player to trigger an “Overdrive”, which releases a continuous shower of shots at all the enemies of the screen for a short period of time. In some game modes, score bonus items also appear periodically.

The game consists of five main areas. The first four are divided into ten sub-sections and conclude with a boss battle. The final area contains a large number of sections and a boss rush, in which the player must fight variations of the bosses from the first four areas. The player goes afterwards to the final boss.

The boss for each area features a variable difficulty scale, depending on the player’s performance leading up to that point. In addition, completing all five levels unlocks alternate gameplay modes, color schemes and secret areas.

Unlike most games, Rez contains almost no sound effects or spoken language. Instead, the game is set to electronic music, which plays in the background and gradually evolves as the player moves along sections. The music is enhanced by musical effects generated by the player’s actions, enemies and surroundings. Player actions are usually locked to the rhythm of the music, such that shots and hits against enemies occur exactly on each beat (as opposed to occuring in real time). Graphical elements

such as polygons that make up the player's avatar, as well as the background elements, also "beat" in time with the music.

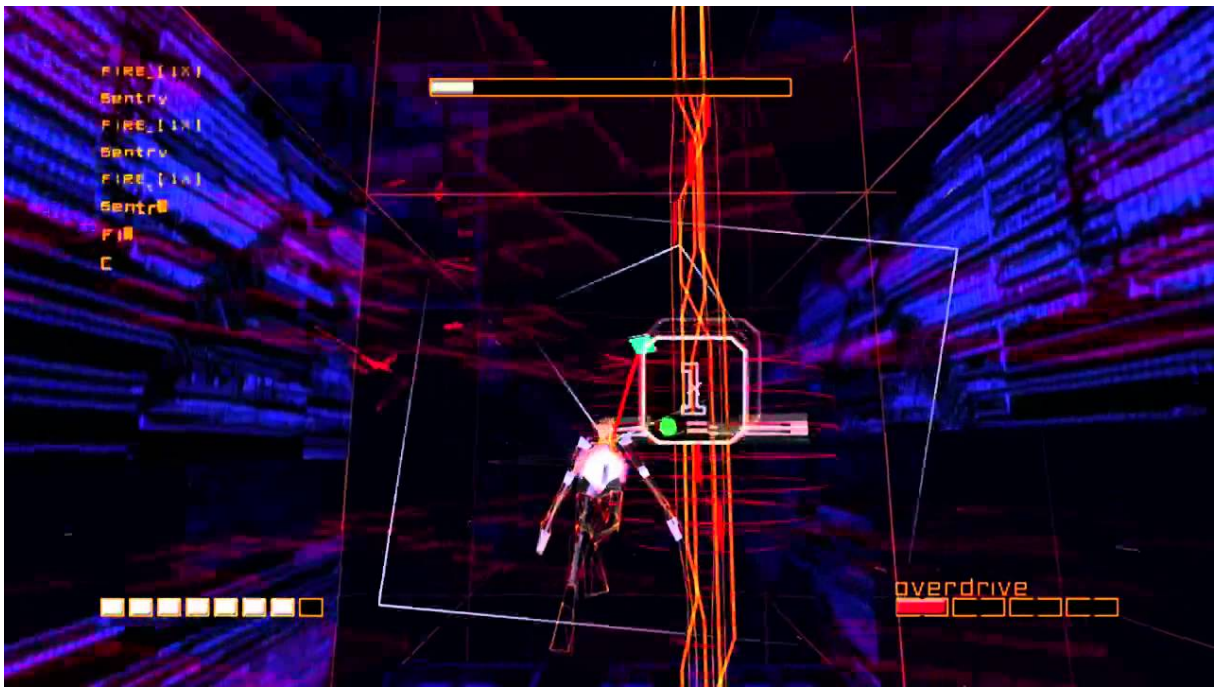

Figure 3.7: Rez gameplay.

3.5.2 Discussion

The video game falls in the quantisation and synaesthesia categories. In quantisation, the beats of the electronic music, on the background, mark when the exact time when the enemies are hit by the shots fired by the player. In synaesthesia, the background objects of the game vibrate with the beat of the song on the level.

3.6 Videogames Discussion

Nowadays there are a lot of music videogames in the market, and therefore is not possible to study all the games with different gameplay mechanics. The games that were analysed are the more significant in the diversity of gameplay.

Videogames that require special devices similar to a "guitar" of Guitar Hero or a "carpet" to dance as Dance Dance Revolution, with the intention to the player pretend to be a real performer, require a special API and adapters be developed. This kind of API and special adapters may be difficult to find, therefore our solution would not have this type of qualities.

Videogames that create levels with the song from audio analysis (similar to Audiosurf), have strong emphasis in the algorithm, while any type of song could be used (not requiring a dedicated soundtrack). Since the scope of this project is to build a video game, the algorithm could exceed the time required to complete the project (maybe audio analysis could be used as future work).

Games that require a dedicated soundtrack, like Rez, need some work to create such content. Such content can be difficult to come up available or even create.

Osu! type of gameplay is appealing to develop something similar. Simple gameplay, different difficulty levels, content created by players are some of the main points in this video game.

However this game did not only had a version for PC. There exists a video game Osu! version for the Android platform as well for the iPhone/iPad platform, but with some differences in the gameplay. Osu!droid is the Android version of the game and it has basically the same gameplay as the Osu! version in Windows. Osu!stream for the iPhone/iPad has the same basic gameplay with the addition of two new gameplay types of interactions: the hold, and the two syncronised touches. The hold is when the player must hold the button until the circle slice fills up; the two syncronised touches is represented by a straight line connecting the two circles and they must be pressed at the same time.

Next we will present the project that was be made for this thesis.

4

Game Design

In this chapter we describe the game design of the project. We will start with the base concept, then describe the gameplay, system requirements and the design of a possible level editor for the video game. Finally we present a overview discussion of the game design. It is important to have in consideration that this is the game design for this specific prototype.

4.1 Concept

Many or maybe almost all people are fascinated by a show in which the dark sky lights up with little lights that follow a certain dance in the sky: firework show. Firework can be used alone, but some firework shows are displayed with a music in the background to make the show more entertained. However when the music is synchronized with the explosions of the fireworks, give an interest effect on people, so this thesis project game is recreating that type of show, but giving the player the control over the show.

The main concept is for the player to execute actions while the song is playing in the background and perform a show of fireworks, actions which are sinchronized with the song being played. Each show is considered a level. In each level, the player explodes fireworks at the rhythm of the song, and depending on his performance, the player receives a score. The performance depends on the timing of a given action of the player.

At the end of the level, there is an overview of the performance of the player in that level, where the

Figure 4.1: Fireworks show.

plaver can view how many right hits and the misses he had made.

The levels are categorized by their difficulty, so the player has the opportunity of choosing the levels that he wants to play.

The video game platform is a touchscreen device in a 2D perspective, where the actions that can be performed from simple presses to static holds and moving holds. Therefore the level design follows a series of combinations of actions for the player to perform that are sequencial, giving the sense of flow during gameplay. Besides the actions, firework effects take place upon the actions performed by the player, displaying the show. The fireworks have different colors and different visual effects, from simple comet-like fireworks to its explosions, which are performed at the beat of the song.

The variation of visuals of the firework and the pacing of actions will try to give the impression of different types of shows, providing a different experience.

4.1.1 Preliminary Interviews

Before we started designing the gameplay aspects from the concept, we decided to do some ”field work” by investigating what kind of players would like this type of video game and what characteristics this game project should have. In these interviews these potential players had dificulties in imaging or visualizing the concept just from simple words and images in order to give us more ideas for the

gameplay aspects of the game. Even so, there was some information that we found useful. There was also some deviation on the interviews: players of the music rhythm games would give less original ideas, with reference to other games as a comparison, while non-players of this type of games would give more original ideas that would be interesting to use in this project.

In this semi-structured interview we started with simple questions on the player aspect of the interviewer, type of player (casual or hardcore), what kind of games they played and if they played any rhythm game. Afterwards the concept was presented and the participant was asked how he could describe the game as they imagined, with the possibility of drawing. These descriptions allowed to further evolve the gameplay base idea.

Since the concept had no description of gameplay mechanisms, the participants gave some ideas that we already had, and gave some thought and also some new insights. Some interviewers told that exploding fireworks randomly would not be fun, and it would be best to have some targets to press at the rhythm of the music. Another suggestion was that only exploding by pressing at the targets would be boring, so as a solution, some kind of different type of gameplay interaction could be used to bring variety, also with multiple colors and explosions sizes and effects. For different timings the game should award the player different points to add to a global score. There were also some ideas for the visuals and visual feedback for the levels that were taken in consideration.

In a macrovision of the game, some interviewers gave ideas for using card systems similar to the ones from games like *SuperStar SMTOWN*[1], and some sort of global highscore to compare with other players. Other participants gave the idea of objectives that would replayability to the levels.

So as a final note from this interviews, we decided to create for the first prototype a simple version of the game, as described in the next section.

4.2 Gameplay

The main objective of the game is to explode the fireworks sucessfully, by touching the screen at the right timing, before the firework show enthusiasm meter deplets to 0 or before the song is over. The player explodes fireworks to earn points, depending on the timing for performing the touch. At the end of the level, there is an overview of the score, with the successful explosions and misses, giving a grade to the performance.

The enthusiasm meter is similar to the rock meter from Guitar Hero, and shows how a supposed virtual audience is enjoying the firework show. At the end of the level this virtual audience will show on the screen along with the performance score. If the enthusiasm meter deplets to 0 before the level ends, this virtual audience will show and boos of the player performance.

The enthusiasm meter is composed by a specific number of chances in which the player can miss or fail the opportunity to attain accuracy on the action performed in a specific context. For every miss or fail, the player loses chances and for every perfect accuracy action he regains chances. This maximum of chances depend on level difficulty.

[1] `https://play.google.com/store/apps/details?id=kr.co.dalcomsoft.superstar.a&hl=pt_PT`

When a firework is ready to explode, the player can see the position where the firework will explode. Along with the position, a scaled version of the position to press is displayed, shrinking at that specific position (Figure 4.2). When the two indicators overlap, this indicates the timing to press. Therefore there is a window of time to accept the press as successful. Missing this window is counted as a miss. It is possible to press the position within this window of action, and the action is awarded a score, depending of how close is to the right timing of the action. When this object appears, the player has to press its center to perform the action at the right timing.

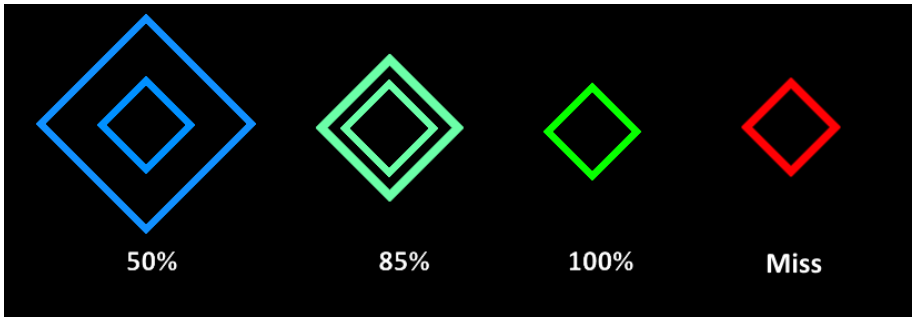

Figure 4.2: Timing indicator.

To better acknowledge the right timing, a color is associated with the right timing. The colors work as gradients of the specific phases of the indicator. It starts with blue fading in at a certain limit before the timing, then it shrinks to the right timing to a green color and after this timing it goes to red fading out. This type of identification goes for every action indicator.

Each indicator of action is referenced as a percentage value of the accuracy: 100% means that is in timing and below that value is designed as the distance percentage to achieve 100% accuracy. For example, 80% of accuracy it means that it takes little while to press the indicator at 100% accuracy. If the value is above 100% it means that it passed the window of action (red color) and it is considered a miss.

There are two types of fireworks: the tail comet (Figure 4.3) and the burst explosion (Figure 4.4). For every explosion there is a comet that previously starts at the bottom of the screen and performs a path to the specific position in which the firework will explode. When performed at the right timing, in the specific window of time for action, the firework explosion will be performed, or else there is not going to be an explosion, leaving the comet without explosion.

Using the results of the interviews, we decided to implement some game objects that perform gameplay interactions throughout the game. The game interactions discussed are: simple touch, hold touch with path to follow and sequence of multiple presses. Next we will present each interaction in more detail, and afterwards we will discuss the score for each object in more detail.

We also designed the game objects to not overlap, and only draw the first to touch in it's entirely, while the next one (that is somehow overlaped) only draw part of it, visualizing what game object to make action first. Example in Figure 4.5.

In the simple touch, a indicator of action will start, only including the timing indicator shown in Figure 4.2). While the outer edge is approaching the edge of the game object, that we named as note game

Figure 4.3: Firework comets.

Figure 4.4: Firework burst explosion.

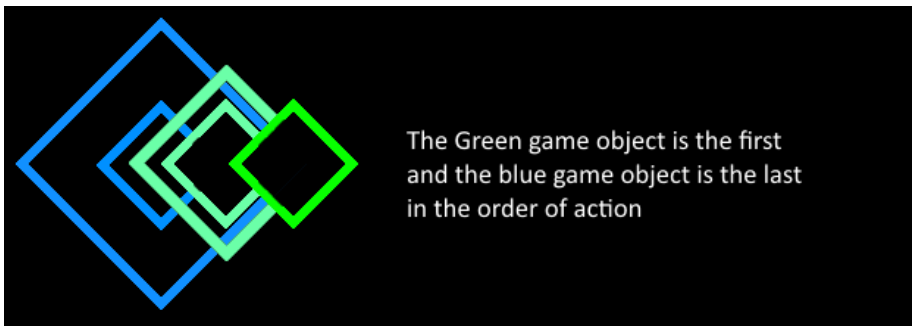

Figure 4.5: Overlap indicators.

object, a comet like firework will come from the bottom of the screen and go to the specific point in time to the explosion. If the note object is pressed at the right timing a explosion burst firework is performed. The window of action is: if the hit accuracy is below 70%, is considered a miss. Between 70% and 90% is considered an ”OK” accuracy and above 90% is considered perfect. For this game object is a comet-like trajectory is followed to the point of explosion, and the explosion is a firework burst. For every value of accuracy above 90%, there is going to be an extra explosion, giving the feedback of a timing touch successfully done.

For the hold touch, the user has to follow a certain path, denoted as slider (Figure 4.6), which is represented by a proximity indicator and a straight line which the touch must follow. The proximity indicator starts to shrink and after reaching the right position it starts moving automatically on the path. Since the button is small and it is difficult to know where the indicator is moving and at which speed, and the touch must follow, it includes an orange edge large enough to be visible for the player to know where to follow. If the player starts pressing while the object is already following the path, it is counted, but with a decrease in score. This game object is shown comets coming from a path to a specific direction. Firework comets will be generated along the path.

The multiple presses (Figure 4.7), are represented by the proximity indicator with a number in the center. The number represents the number of presses that have to be performed for the game to accept the action as successful. After the proximity indicator disappears, a new indicator, a blue big edge, appears and starts to shrink. This blue edge represents the time that is needed to perform the multiple

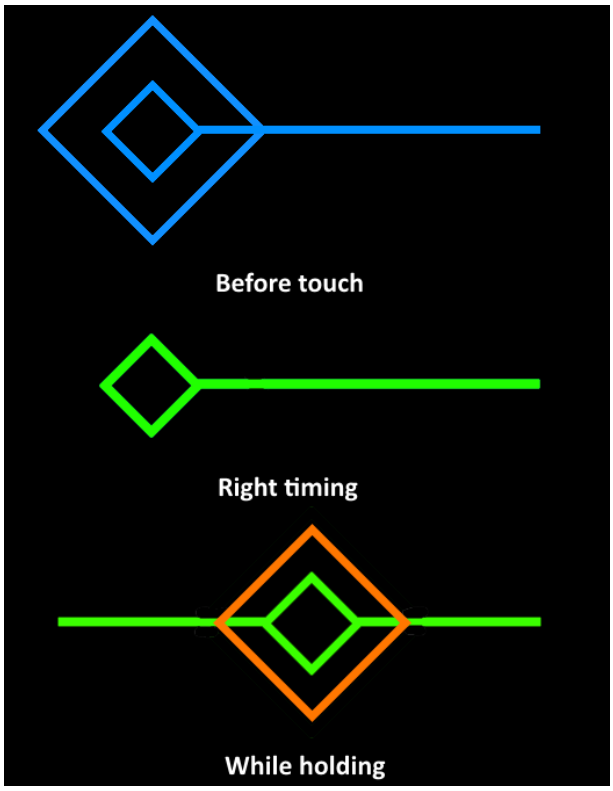

Figure 4.6: Slider game object.

presses. When the blue edge overlaps the multiple presses it will disappear. Missing the window, or not performing the correct number of presses will be accounted as a miss. For every touch, the number of presses indicated in the center will decrease until it is completed. For every hit firework explosions will burst from that spot.

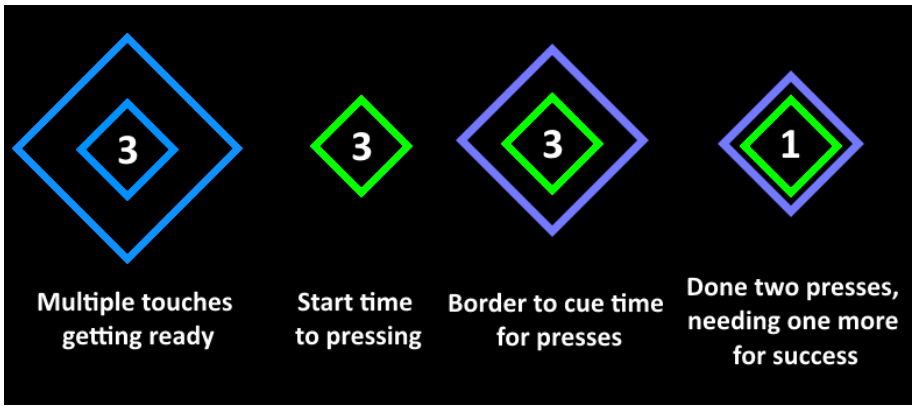

Figure 4.7: Multiple presses game object.

Another form of interaction of game objects that was discussed with the interviewers was having the same time actions and the hold actions. The same time actions are associated with the note object, where is two note objects connected by a line and the timing is the same, needing to press these objects at the same time. However this type of action is not only for notes, it can be performed by different configurations of action objects: note with note, note with slider, note with multiple presses and slider

with multiple presses. The same time actions are represented by a black line that connects the two game objects that need to be pressed at the same time. The remaining action would be a hold. This action was not much discussed in the interviews, but it would be a kind of press where the player would have to remain in the same place for some time to the action to be successful. But we will not discuss this action design because of time constraints.

For the feedback of the actions, we decided to have an indicator that the action was performed in accordance for the different game objects. This feedback appears after the action is performed or missed.

For the note game object a edge equal to the game object will have a color determined by the accuracy: below 70% of accuracy (a miss) is represented by red color, between 70% and 90% (”OK”) accuracy is represented by a blue color, and finally above 90% (Perfect) accuracy is not represented, since in this accuraccy level, the main explosion and an extra burst occur. The burst's color is different from the one in the main explosion. Missing the opportunity is represented by red color. This feedback gradually disappears over a very small time.

The slider game object is represented by the same feedback presented previously at the end of the slider. It is a mean value of accuracy between the first touch and the release touch. The first touch is in accordance with the exact time of touch, and the release takes into consideration the exact time of release. This means the better accuracy mean value between this two exact times, the better it is the performance. At the end of this action a edge equal to the to the center of the slider game object will have a color determined by the accuracy with the same color feedback presented previously. Since the slider appears comets coming from the line of the slider, if the player does not follow the slider correctly, new comets will not be generated.

For the multiple touches, we account for the accuracy time of the first touch and the presses needed to perform.

Along with the performance points there is a combo counter that represents the successful consecutive touches that the player performed. This value is going to be used to the score of the successful hit of the objects. This means that each time a miss is performed, the counter is reset to 0 and each sucessfull hit, increments the countes's value by one.

The score points are counted by each suceessful action performed. Missing an action will be not give points. The accuracy value will be multiplied by 50 if the accuracy was inside the OK window, or multiplied by 100 if the accuracy is bigger than 90%. Also each instant this score points is multiplied by the combo counter at that giving time. This means that bigger accuraccy and bigger the combo counter more points earned.

At the end of the level, there is a screen that shows the performance that the player achieved in that level. The score screen contains, the number of misses, the number of ”OKs” timings, the number of ”Perfect” timings, the final score, the accuracy on all level and a Grade letter of that level performance. Grades can assume values SS, S, A, B, C, D, E. The conditions for each grade are:

- **Grade SS** If the player performed more than 100% of Perfects in all level game objects.

- **Grade S** If the player performed more than 90% of Perfects, 0% misses and less than 5% of Oks in all level game objects.
- **Grade A** If the player performed more than 80% of Perfects and 0% misses OR more than 90% of Perfects in all level game objects.
- **Grade B** If the player performed more than 70% of Perfects and 0% misses OR more than 80% of Perfects in all level game objects.
- **Grade C** If the player performed more than 60% of Perfects in all level game objects.
- **Grade D** In other cases different of the above.

To increase difficulty and present some challenge, the speed of the timing indicator is variable. For expert players the speed can be increased, and for more novice, the speed can decrease.

4.2.1 Discussion

In this section we explain what we have done for the prototype of the game. The complete development of the videogame was considered as an objective, but time constraints directed the development of the game for this thesis towards a first playable prototype for playtesting.

Since there was no artist assigned for this first prototype, we focused on making it playable and more fluid. Therefore the final prototype does not feature the virtual audience telling how good was the performance of the player. However, we still take it into consideration, because it would add another layer of immersion to the game and it would make the experience better.

There is a enthusiasm meter to give the player the oportunities miss while playing the game level. It is setting in an easy difficulty, because we are not aiming for difficulty in this prototype, but there is a setting in which the player can change the game's difficulty level.

Also due to time constraints, we decided to implement the note game objects and the slider, but we are leaving for future work the simultaneous interactions, multiple touches and holds.

During the development of this thesis, we also decided to change the way the fireworks are associated with the slider game object. At first, we thought of having multiple firework explosions bursts along the path, at the rhythm of the song, but later in the development we decided to change, because an explosion is more likely associated with presses, and not holds. We change, this into comets along the path. We commited to maximum of 6 comets along the path, but in future interactions it should be many comet fireworks along the path.

The accuracy for this prototype is set to a value meant to be easy, but since is configurable, it can be later changed for closer timing experiences.

Is important to state that previously, we had decided to make the timing indicator as a clock going to the right timing, making a count down, but it would be easily hidden by the user's hand, so, we decided for an Osu! game like mechanic of showing the timing indicators.

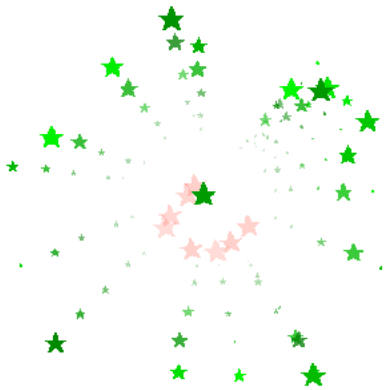

Figure 4.8: Firework Type 1. Random burst.

Figure 4.9: Firework Type 2. Circular burst.

We also based our grade system on Osu!'s, since it could be seen in there site[2], and it would be fair way to grade the performance of the player.

One question that arised, after the interviews and that was a constant influence in the design process, was what kind of songs would be best suited for this game. This means a variety of combinations that the levels can demonstrate, but maybe not all songs work with fireworks. This is one of the questions that we discussed with potencial players to find the answer as reported in chapter 6, and to be followed in future work.

4.3 Firework Effects

Figure 4.10: Firework Type 3. Elyptical burst.

For the fireworks theme, we need a way to display the fireworks. For this purpose we considered two

[2] http://osu.ppy.sh/wiki/Main_Page

ways of doing this: by creating a particle system to recreate such effect, or by making a sequence of images (batch of images) that have the various sequences of a firework explosion. And we decided to follow the development by implementing a particle system.

In Chapter 5 we describe how we created the particle system. This system recreates two effects: the comet-like fireworks (a firework comet), where fireworks move along a path to a specific point, leaving a trail of particles and the burst explosion that recreates the explosion of the fireworks. We decided to make three types of firework explosions:

- The **Random burst** is a effect where the particles just go in random directions of a center point.
- The **Circular burst** is a effect in which the particles draw a circle as they burst in equidistant from the center.
- The **Elyptical burst** is a effect where the particles draw a elypse as they burst from the center. The elyptical burst can be rotated to recreate a different visual effect when exploding.

4.4 Game Requirements

In this section we will present the game system requirements to create the game:

- The game should run in mobile devices with touchscreen capabilities and the operating system Android for both smartphone and tablet.
- The game requires to play songs from files, an essencial feature since is a rhythm video game.
- The game has to be able to time the actions at a accurate timing, as it is a rhythm game.
- The game shoud use a particle system to create the fireworks effects.
- The game should allow to play levels, created by a level editor created especialy for the game. A level file is created and saved by the level editor, and loaded and played in the level player.
- Finally, the game should allow to create levels that give a sense of a immersion in the song with synchronized fireworks explosions, for different difficulties.

These are the main requirements for the game, but more specific requirements are needed to cope with technical requirements of the challenges needed to accomplish the goals.

4.5 Level Editor Design

To allow the creation of the levels to be played, this game needs a level creator. In this section we will describe the various important functionalities that the level editor features.

The level editor has two main views: the editor view and the visualizer. In the editor view, the user can place and edit the actions, and edit the fireworks of the level. The visualizer view can have a preview of how the level would like on the level player.

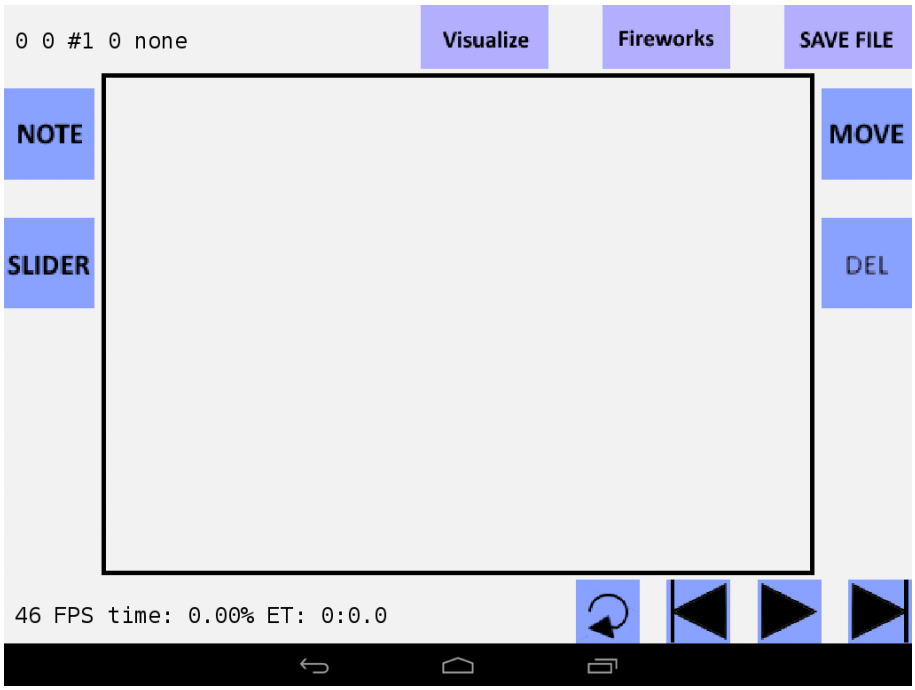

Figure 4.11: Level Editor screen.

Figure 4.11 shows the level editor with the editor view.

In any view the level can be saved by pressing the save file button at the top right corner of the editor. It level is saved to a file that contains the song name and the different game objects and fireworks assocciated with them.

The control of the levels works in the timeline of the song. Therefore the user needs to go forward and backwards in the timeline to progress in the level. The game objects appear at timings they are assigned to (fading in before time and fading out after the time) or remain invisible instead.

The buttons in the level editor display actions or functionalities, while the interior of the rectangle is where the game objects corresponding to actions in the level player are placed. The buttons are separated in the following categories: the bottom buttons are sound timeline controls, the upper buttons are views to be displayed, the buttons on the left are the actions of the game objects, and finally the buttons on the right are meant to edit the game objects.

The first and the most important controls of the editor are the sound controls. The sound controls make it easy to navigate between the timeline of the song. There is the Play button, that plays the song, and pauses it if pressed again. Pressing Play gets the timeline to start moving in real time, made possible by the system clock. Then there are controls to navigate backwards or forward in the timeline. These buttons (placed next to the play button) make the song advance or go backwards in 100 ms. This number was chosen because it is not too small nor too large to move in the timeline. There is also a Replay button to go back to the beginning of the timeline, to make easy for the creator to preview the flow of actions from the beginning.

Next to the controls there is the timeline information of the song: the time scale in minutes, seconds and a decisecond to make more accurate the song information. Next there is the information of the

progress of the song with in a percentage. There is also a FPS counter, to facilitate the monitorization of how heavy the particle computations are.

On the top left corner is the number of game objects, and the beginning and end indices of the game objects vector, refering to the visible objects at that instant in the timeline.

In the editor view there are two modes of edition: level mode and fireworks mode. The first mode is for the placement of game objects (notes and sliders) on the timeline. The other mode is to edit the fireworks associated with the game objects. In both views the timeline controls of the song and the information of the time of the sound are kept in the same position.

4.5.1 Level Mode

In the Level Mode can place the game objects: the notes and slider game objects. Instances of these objects can be placed in the timeline. The rectangle where we place the game objects has a grid, so we can know the distance between the objects. All the placed objects are snapped to the grid lines. The placement timing of the game objects is made in accordance with the song time that is played. The placement of the note in the editor is made by selecting the note button, that turns red as game object currently selected. From that moment on, the level creator can place note objects in the locations of his choice. For each note there is a location and a time associated with it.

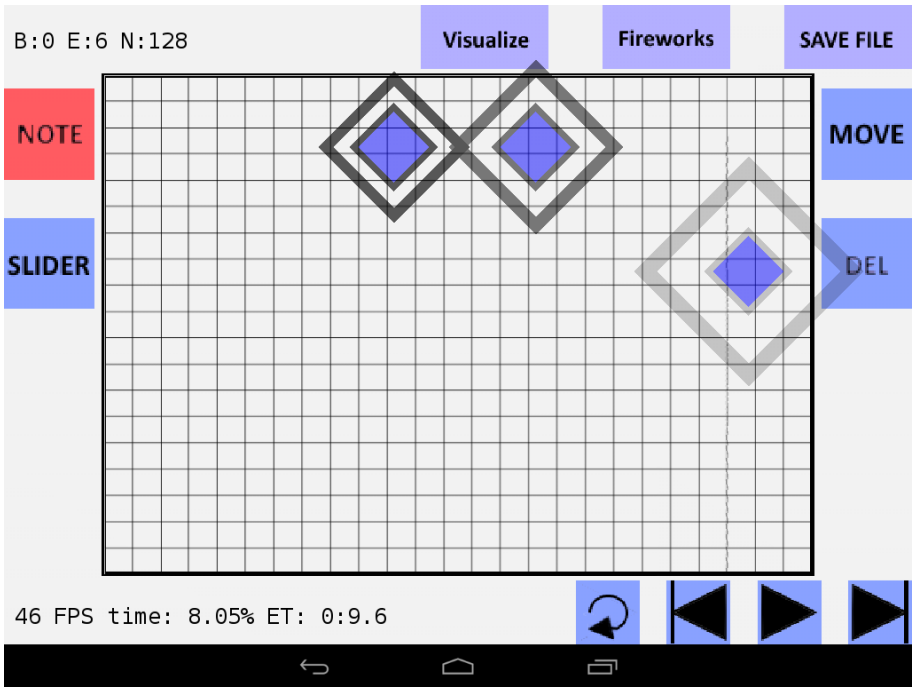

Figure 4.12: Level Editor screen with three note game objects.

The placement of the slider in the editor is made by selecting the slider button and placing sliders in the desired locations. For each slider there is associated a location of beginning, a location of end, the time of beginning and the time of end. This makes possible to create variations regarding the action timing and the length of the paths to be followed.

At the right side of the area to place the game objects, there are two buttons to edit the game objects

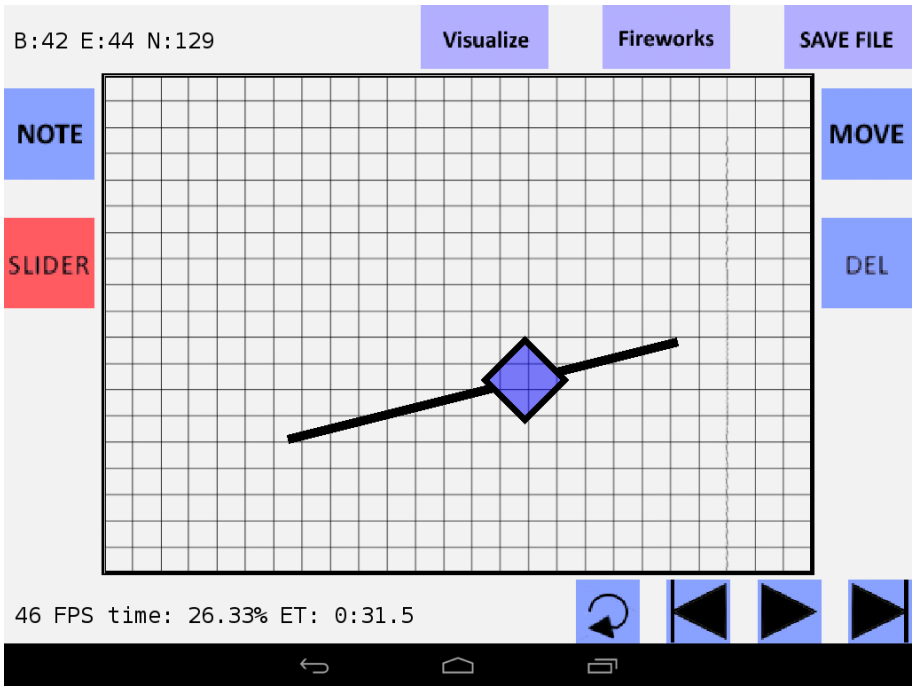

Figure 4.13: Level Editor screen with slider game object.

placed in the timeline: the move button and the delete button. The move button moves the game objects of their location, with the initial time that was placed. In the slider object, the slider keeps its form, direction and distance between the initial point and the end point, only moving its initial location. The delete button deletes the game object.

The note game object is represented by a blue square with a black border. When the song is playing, there is an extra scaled version of the black border that scales to the black border of the note. This extra border starts transparent and becomes opaque during the passing of the game object. The timing to perform the action is when the two black borders overlap, and afterwards the object is made transparent.

The slider game object is represented by a blue square with a black border and a black line. When it starts to show, a scaled version of the black border appears and shrinks to the other border at the right timing. Afterwards, the blue square and the black border moves along the black line representing the movement need to perform during play time.

When the game object is moved the blue square becames a red square, representing a move action.

4.5.2 Fireworks Mode

After the placement of the game objects, the fireworks associated with the game objects can be edited. The fireworks entity is a group compose of a tail and an explosion. The tail appears 3 seconds before the timing of the explosion, and the explosion effect is played after the timing.

To start editing the fireworks of the game objects, first a game object must be already placed in the editor level mode, and then change to firework edit mode by pressing the Firework button.

In the firework editor scene, the game object is represented by the firework explosion point and by a trail of dots (the path followed by the firework until reaching the explosion point). When the explosion

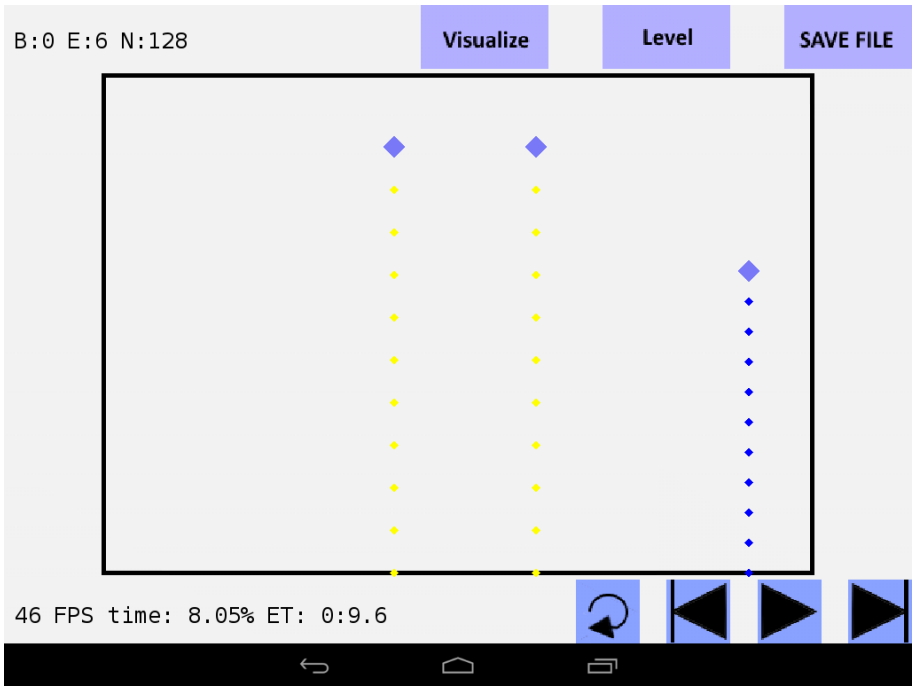

Figure 4.14: Level Editor firework screen with notes game objects.

point is pressed, one can edit the firework effect associated with that particular game object.

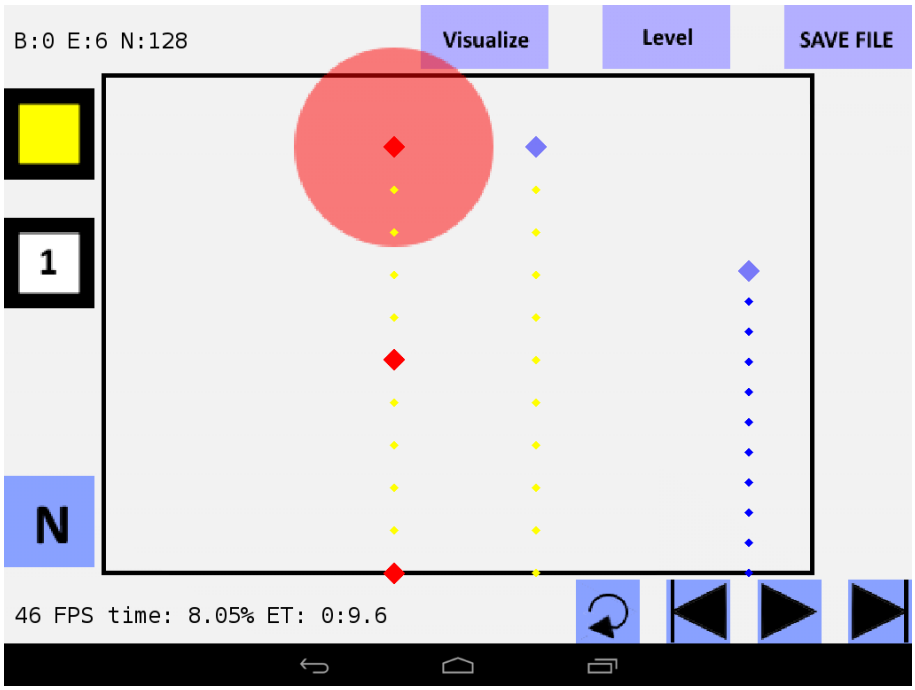

Figure 4.15: Level Editor firework screen with an editable firework note game object.

Associated with the firework there is one color for the tail and the explosion, for simplicity. When the game object is selected, the color of the firework can be changed by pressing a color button at the left side of the editor screen. This button alternates between the following colors: Green, Red, Gray, Yellow and Blue. Next to the color button there is a button to select the type of fireworks: Type 1 is a random

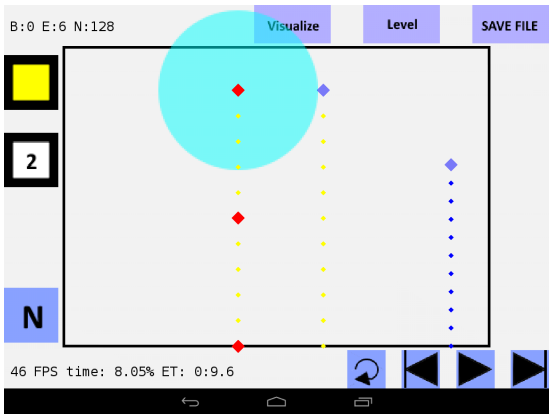

Figure 4.16: Level Editor firework screen with note firework edit game object type 2 firework explosion.

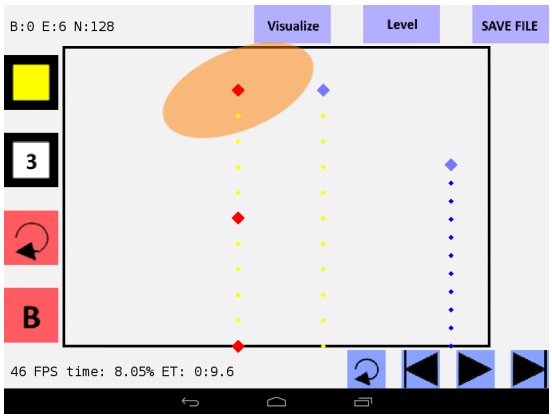

Figure 4.17: Level Editor firework screen with note firework edit game object type 3 firework explosion.

direction explosion, type 2 is a circle explosion and type 3 is a elyptical explosion. The type 3 explosion adds a new button, to set the rotation of the elyptical explosion, adding 22.5 degrees for each time is pressed. When the explosion is selected, there is a visual cue to identify of the 3 types of fireworks it is at the center of the explosion: a red circle if type 1, a big cyan circle if is type 2 and a elyptical orange circle if it is type 3.

There is also a button to change the size of the explosion to normal (N) or big (B), below the color, type and rotate buttons. Figure 4.16 and Figure 4.17.

Along with the selection of the type and color of the firework, it is possible to change the trajectory of the firework tail (Figure 4.18), this is done by moving its control points of the firework tail. When a game object is selected and the firework edit options are shown, the control points of the curve of the firework are displayed. The creator can then drag these control points and place them where he wants. There is a visual feedback of the tail with the group of dots lined up that changed into a curve when changing location of the control points. Only the bottom control point and the middle control point can be moved. The bottom control point moves along the bottom line of the rectangle. If the middle control point is moved above the center of the middle of the other two points, it gives an effect of the comet tail going fast at the beginning and slowing down to the point of the explosion, due to the manipulation of the control points.

The procedure described above applies to the note game objects, but there are differences in the case of the slider game objects.

When one starts to edit the fireworks of a slider game object, initially there are two tails entities connected by a line, representing the firework tail edit for the initial point and the end point of the slider. If the first comet in the slider is selected a plus and minus buttons appears at the right corner of the editor. These buttons let the creator of the level to add more points of explosions along the line of the slider. Each point that is added is equal spaced to the line of the slider. There is a maximum of 6 fireworks instances along the line of the slider, and the minimum of 2, the initial one and the final one. Each of these comet firework instances along the slider are independent of each other, meaning, they can have different colors.

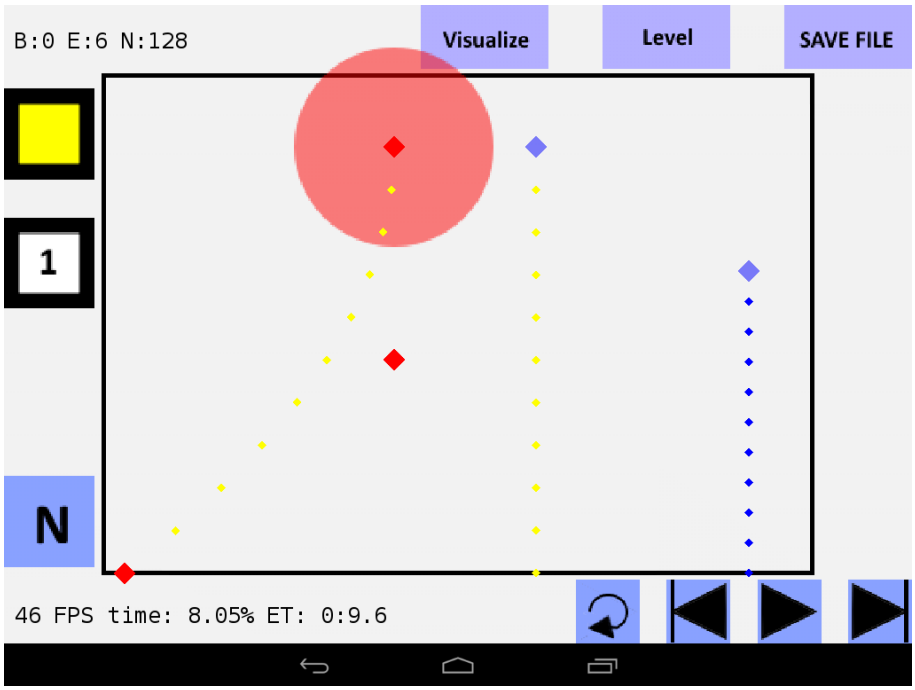

Figure 4.18: Edit of tail trajectory of the firework.

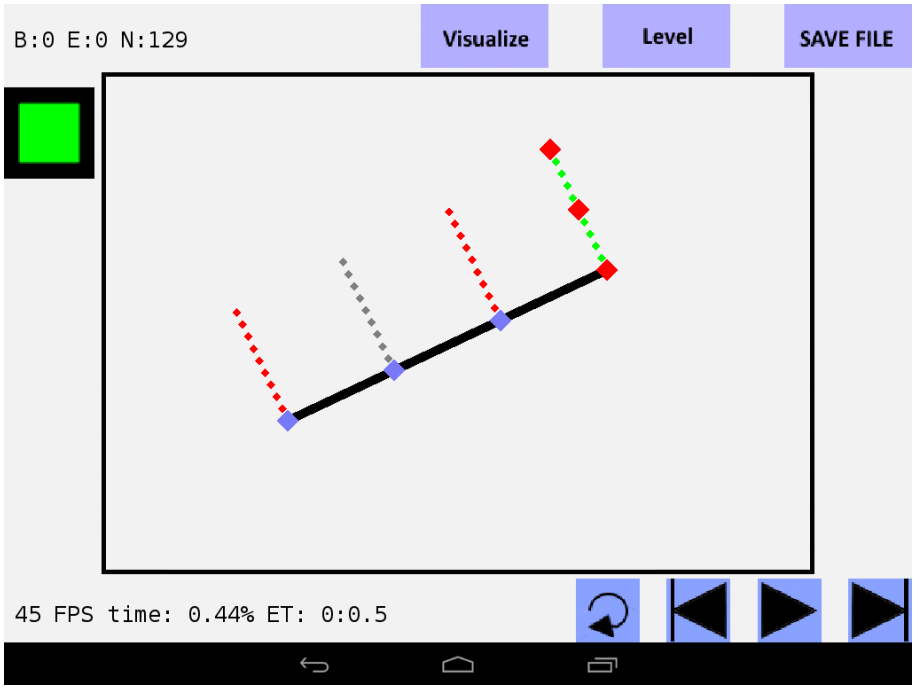

Figure 4.19: Edit of comets of firework slider game object.

It is important to add that this slider is an unfinished feature, and therefore is not properly finish because of time constraints. The purpose of the slider was to generate comets without a specific number, but keeping generating as long as the button was pressed

The slider is only for straight lines, generating the specific number of comets on the path.

4.5.3 Visualizer Mode

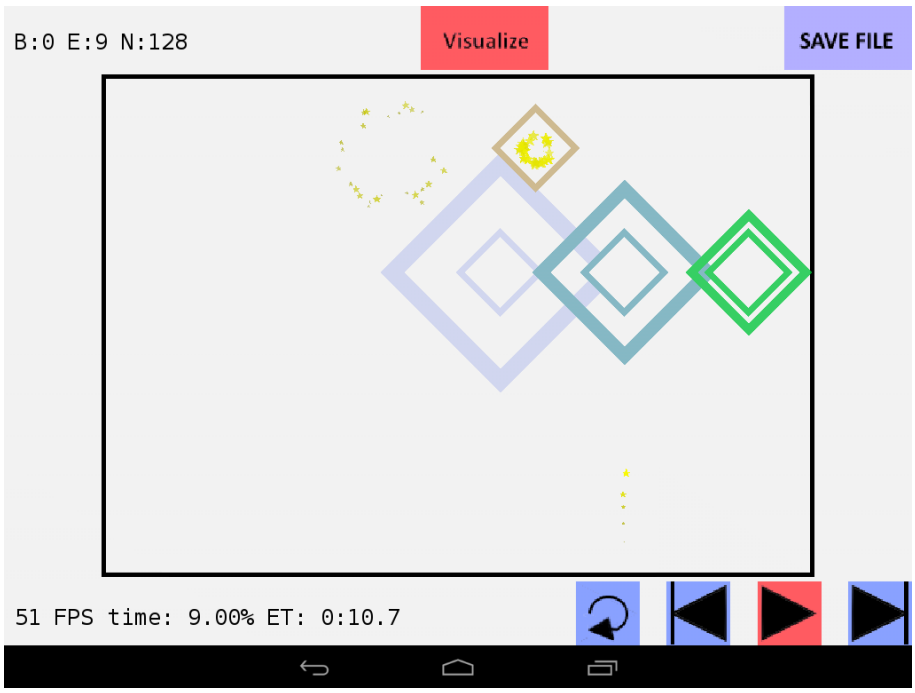

Figure 4.20: Level Editor Visualizer screen.

In all the screens, both in the Level mode and the in Firework mode, there is a button for visualizing the way the level flows in a simple preview of the level. This button can be pressed in any time in the timeline, making it useful to preview how the level should look.

In this preview, the game objects are shown as in the level player, with the colors correspondent to the timings (from blue to green, and red), and the explosions of the fireworks with the correspondent colors, paths and burst effects edited in the Firework mode. To exit this view, one just needs to press the button again.

In the next section we present the way of saving the levels created by the level editor.

4.5.4 Saved Levels

To play levels created by the level editor, level files must be created, saved and then played by the level player. The saved files include the following information about levels:

- Song filename location.
- Delay factor.
- Sequence of game object information along with the firework information.

The delay factor is the speed at which the gameobjects appear with the proximity edge. The higher the value, the slower the proximity edge moves to the correct timing edge. Analogously the lower the value, the faster the proximity edge moves.

The information for the note object and the information for the slider object is different. For the note object, there are two lines of information. The first line starts with the type of game object, followed by the x and y coordinates of the position. The next line refers the fireworks information: the time for the tail to start its movement, the 3 points of the curve of the tail, the color and finaly the type of firework (type 1, type 2, type 3) and angle of firework (used only for fireworks of type 3) and if it is big or normal size.

For the slider object, there is one line for the slider information, plus n lines for the fireworks comet in the path of the slider. The first line has the type of game object, followed by the start time and end time of the slider, next the point information of the starting point and the end point of the slider (the x and y coordinates of position). This line finishes with the number of fireworks comets on the slider. The next lines are the fireworks, with the same information described above for the note object. The number of lines for comet fireworks are equal to the number of fireworks comets of the first line of the slider information. For each comet has the time of the tail to start its movement, the 3 points information of the curve of the tail and the color.

Next we describe how levels for this game can be made.

4.5.5 Creating Levels

This way of creating levels is not optimized, but it is the way that was design.

For creating a new level, the creator opens the level editor, and the player saves the level. In this case, a new file is saved in the hard memory drive as a temporary level file, where the creator can see the information saved. There the creator edits this file directely with a file editor and in the section of the song location, he writes the filename path of the song for the level, saves the file and exit.

When the creator opens again the level editor, the editor loads the temporary level file, with the song of the temporary level file. This allows to load different songs without a file browsing application. Then the creator can edit the level, and exit the editor whenever he wants. Only when the level is finished, the creator can rename the temporary level file to a appropriate name and when the editor is opens again, it will not load the previous temporary file and the creator can create a new temporary level file, always ready for editing.

4.6 Level Player

Before the game starts, the game only starts, when the player touches the screen, when the message that the game level start after touching the screen.

The level player scene is composed by the enthusiasm meter, containing opportunities at the left top corner, below it is the accuracy of the level. At the right top corner is the total score, and below it the current score of the action performed that is added to the total score. At the left bottom corner is the duration of the song in percentage, and at the bottom right corner is the combo (chain) counter. It is possible to pause the game at any time. In this screen the player can resume the game, restart the game or exit to the selection screen. The selection screen is a simple screen where the player can select what level to play.

Figure 4.21: Level Player screen.

There is also a screen of game over where the player can restart the game or exit the level.

At the end of the level, when the song is over, and the player did not lose the game, a performance screen like is shown in 4.22. In it is the total score, the misses count, the ok timings count, the perfect timings count, the average accuracy, and the largest combo made by the player. There is also the grade letter of the performance. In this screen the player can restart the level or go back to select levels.

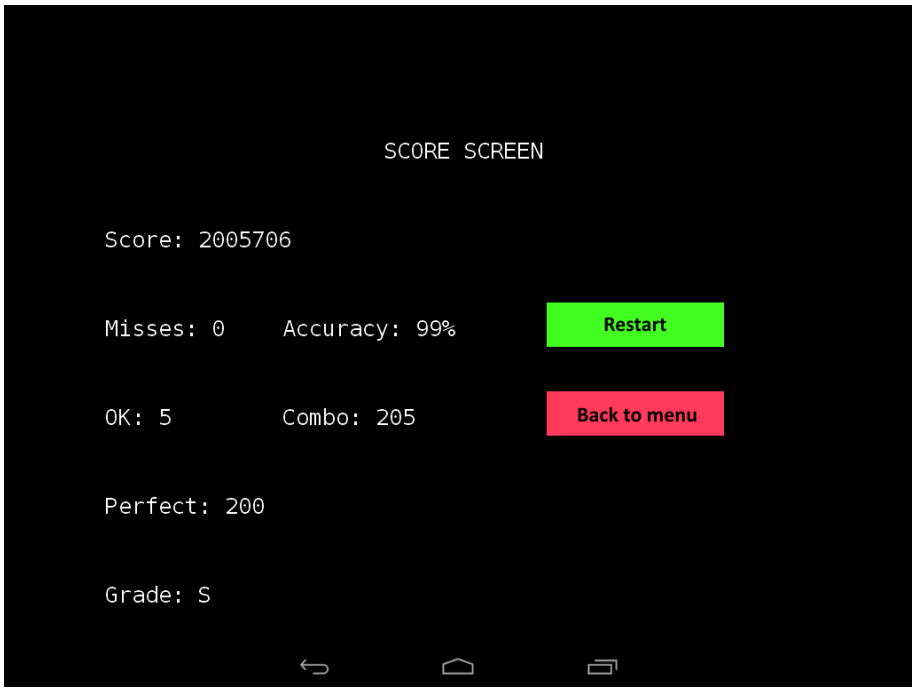

Figure 4.22: Performance screen.

4.7 Level Design

The levels created for the evaluation were inspired by the ones in Osu!, but without all type of objects that Osu! has, since our game only has two type of action objects available to play, the note game object and the slider game object.

Making levels for rhythm games is a complex task. There are not many resources on how to make interesting levels, therefore, it greatly depends on the creator's creativity. The placing of game objects and (for our game) the variation of fireworks was deliberate to create more interesting levels, with special explosions on "high beats" in the music.

For creating this type of levels, based on Osu! standard levels, we need two things: patterns and variation. Patterns are used as simple repeatable patterns at similar excerpts. These patterns can even be mirrored from any direction, to create variety. Variation means to use different sets of combinations to create as different flows into the game. Is important to add that, while analysing levels from Osu!, we found some kind of continuous path that certain patterns and variations performed, and we tried to recreate such path in our levels. We believe that level design for rhythm music games can be an interesting subject for creating levels, and some literature of this kind of subject would be great to have for the future.

When we started building the levels, we found an issue. Even though it was possible to create levels simply by placing the game objects in the timeline, placing the objects at the correct milisecond time of the beat was almost impossible. In other games, like Osu! that also features an editor, the correct placement of objects was exactly at the beat, because there was a setup of the BPM (Beats per minute) while, in our game, the player makes the setup of the ruler of the timeline with the possible points to place the objects.

Taking this issue into consideration, we created the levels. We did not make a BMP ruler for the timeline, because this issue was only noticed when we were making the levels, and there was not enough time to implement such feature, however, we tried to evaluate such problem in our playtest with the players.

4.8 Discussion

In summary, our game falls in the rhythm action category. There is the need to hit the objects in time with the rhythm of the song, and as difficulty level increases, the speed and complexity of the levels challenges the player.

Also according to the principles of music games it falls into the rhythm game genre since the player plays the game in reation to screen stimulus, and at the end there is a score on performance of the players actions.

Up to this day, there are many music rhythm games in the market for the Android platform, many have similar features, only a small handfull of games have certain gameplay features that distinguish of other games. In the current state of our game, there is not such a feature that makes it stand above

others. We will, in future work, try to find such features that make our game unique.

Other thing to be said is that we focus more in the gameplay of a level and making the fireworks particles system work in the game. We believe that the main feature that stands out is the use of a particle system that simulates fireworks. At first we thought of using batch of images that simulated the firework explosions, but we decided to make real particle system as a technical feature, and as follows in the next chapter, we indeed succeeded.

We will start stating the technical challenges that we faced and the solutions for building this prototype in the Implementation section.

5

Implementation

In this chapter we are going to describe the various steps that we took to construct our project. We will also describe the various challenges that we faced and the solutions that we chose to make a playable prototype of our project.

5.1 Building The Game Engine

Since there was no viable option for an engine for our target device, we decided to build our own engine from scratch.

At first we decided to use Android SDK to develop our game prototype. Android SDK, that stands for Android Development Kit, is a set of comprehensive set of development tools that provide a debugger, a set of libraries and a handset emulator to create Android applications in Java. However, Android SDK does not provide any game library, it only provides a set of libraries to develop applications with Android Interface standards, not giving much liberty to create games.

Instead of using Android SDK, we decided to use Android NDK, (standing for Android Native Development Kit), which contains a set of libraries in C and C++ language. However, using this native library to create the game prototype increased the complexity of our engine, because many standard and necessary libraries to help in the development were not available in the native library. Concretely, there was the need for the engine to be able, at least, to:

- Render a image into a 2D texture.

- Render text from a given string.
- Playback sound from file.

Despite the disadvantages of using Android NDK, we still decided to use it (combined with C++) due to the following reasons::As described in [9] making particle systems would drop considerably in Android SDK; also, there is a demonstration of particle systems implemented in Android NDK [1] that reported little FPS drop and maintained the fluidity of perception of the application. Additionally, there are several statements that say that Android NDK is faster than Android SDK, and Google recommends using the NDK for 'CPU-intensive applications such as game engines, signal processing, and physics simulations' [10], specially for our case of particle systems for our firework explosions.

Since the device that we were using only had a dual core, we decided to implement a simple main loop design, despite the fact that [11] argues that a multithreaded main loop would give best performance increasing speedup by two. We decided so, mainly because we did not find a working example of this type of multitheaded loop, and the only simple example found used a single game loop. Perhaps for future work we could try to find or build ourselves a better multithreaded game loop. This choice of game loop engine was also taken into consideration because of the particle system. More on game loops in [12].

Following the example in [13], from a source code provided freely for examples purposes, we implemented the simple game loop and used the integrated OpenGL 2.0 ES version for graphics. This version of OpenGL was used because of the specifications of the device we initially used for, a ZTE Blade G Lux[2] smartphone.

The main specifications are:

- Resolution 480x854 pixels.
- Android OS, v4.4.2(KitKat)
- Dual-core 1.3 GHz
- Memory 512 MB RAM

Even with a device with low specifications considering the current standards, we decided to structure the engine accordingly.

5.1.1 Architecture of the Engine

In this section we present the architecture of the engine we built for this project.

5.1.2 AndroidMain and NativeActivity

The **AndroidMain** function, is the main function that is called when the application starts for the Android environment.

[1] `https://www.youtube.com/watch?v=4ZrcIBDcPTI`

[2] `http://www.gsmarena.com/zte_blade_g_lux-7057.php`

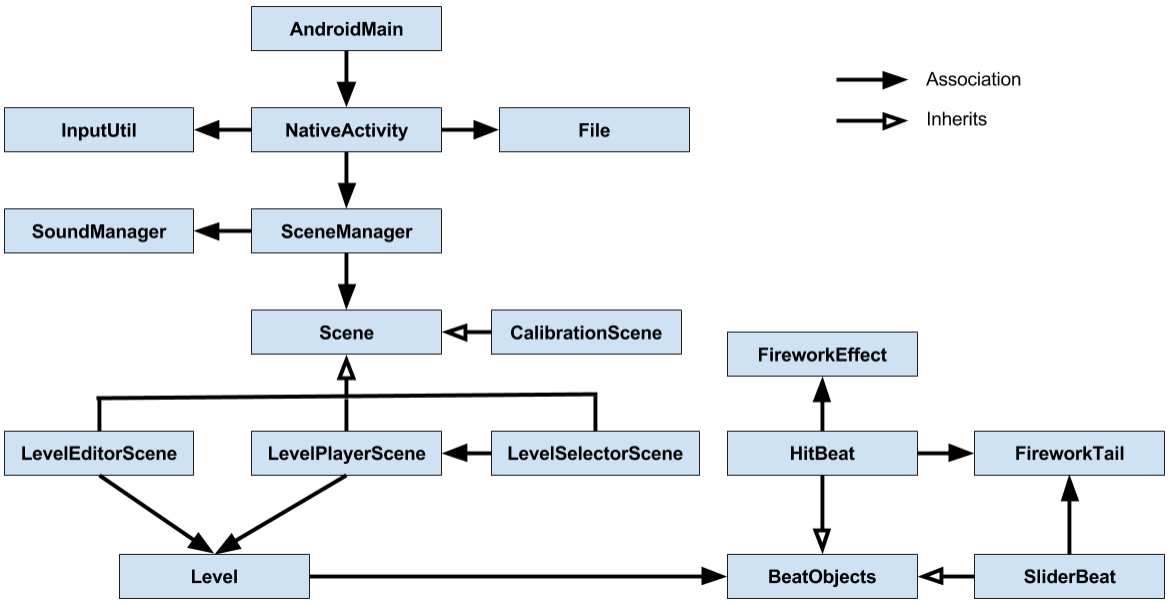

Figure 5.1: Engine Architecture.

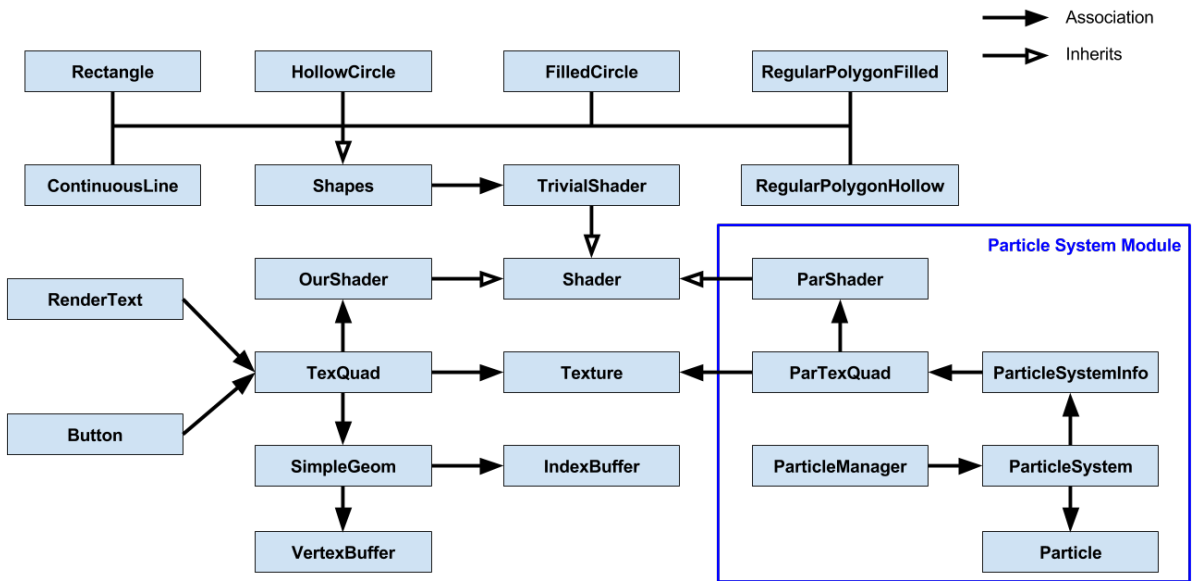

Figure 5.2: Helper classes.

The **NativeActivity** class that implements an Android activity purely in native code, like is shown in [3], It reads files from the asset folder with the **File** class and read the inputs from the device with the **InputUtil** class.

5.1.3 Scene Management

For better management of scenes we decided to build a struture of scenes, based in one of the Android samples for NDK [13], that worked by making the transitions between scenes. This is done in

[3] `https://developer.android.com/reference/android/app/NativeActivity.html`

the **SceneManager** class and **Scene** class. It represents the main game loop as a simple main loop described in [12].

5.1.4 Displaying Images

The **Texture** class is responsible for displaying images from files we decided to first use a library to load PNG images. We chose PNG images because they included an alpha channel and use lossless compression. Other choice would be TGA image format, but it is uncompressed, and such images need to be in the asset folder of the application and could use more space.

Before we started to read PNG files, we needed to load assets from the asset folder of the Android project. Afterwards we needed to use a library to load and decode the PNGs images. We found two main libraries that we could use: LodePNG[4] and libpng[5]. Libpng was the only working example that was found, and therefore we used in our engine.

To display images, we needed to create geometry, pass the images into textures and render the geometry with the textures. To do this we use a simple texture quadrilateral polygon mesh, with **TexQuad** and we use a generated **SimpleGeom** mesh in specific classes **VertexBuffer** and **IndexBuffer**, using the **Texture** into it. It was also necessary to create a OurShader class to render bind with the texture. **OurShader** is an inherited from base **Shader** class, which provides a shader for rendering a image into a given geometry, also allowing to tint the color base of the texture. All these classes are available in the opensource NDK project from Google [13].

5.1.5 Rendering Text

To render text into the screen, in **RenderText** class, we decided to use a external library, FreeType[6], which makes it possible to read font files and use this information to render text. For this purpose we load the font and render all the characters into one single bitmap texture, that is loaded always for every character. Then we create one simple **TexQuad** for each single character, with the texture coordinates of the character in the characters texture. This way was chosen instead of loading a texture for each character in runtime since that would consume render time and lead to an unecessary overheat. This way, this does not happen because only one big texture is loaded into memory and it is used to make each character.

5.1.6 Playing Sound

To play sounds we need to load and play files from the asset folder and/or from the storage drive of the device. At first we tried to use mpg123 library[7], but we faced some difficulties in having the player and the decoder to work and there was not enough working examples that would allow to overcome those difficulties. However, we later discovered that there was a internal standard embedded hardware-

[4] http://lodev.org/lodepng/

[5] http://www.libpng.org/pub/png/libpng.html

[6] https://www.freetype.org/

[7] https://www.mpg123.de/

acelerated audio API in Android that was possible to connect via Android NDK: OpenSL ES [14]. We built a class for the sound, called **SoundManager**. We put an singleton instance of this class in the SceneManagement, because this is the most centralized class in all engine, and also having in mind that sounds can be played in transitions between scenes. We save all loaded audio files into a internal memory map that provides fast search for various files. The interface of this class allows to load into memory a MP3 sound file, from asset folder or the storage device, the playback controls of PlaySound, StopSound, PauseSound, ResumeSound and EraseSong. There are also functions for GetSoundPosition, SetSoundPosition and GetDuration.

5.1.7 Button Class

To provide fast recreation of a button input in the touchscreen, we created a **Button** class. A button has two textures embeded into two internal TexQuads. These two textures are for the buttonwhen is focused (i.e. the button is pressed and remained in this state) and when it is unfocused. We give a specific coordinates, scale, dimensions of area of touch and a tint color.

5.1.8 Shapes

To render the game objects we generate the geometry in the **Shapes** class. The available shapes are: **HollowCircle**, **FilledCircle**, **Rectangle**, **RegularPolygonFilled**, **RegularPolygonHollow** and **ContinuousLine**.

In the first stages of development, the game objects described in the previous chapter were circles, filled circles and hollow circles. Later we changed to diamond shape geometry provided by **RegularPolygonFilled** and **RegularPolygonHollow**.

The **HollowCircle** and **FilledCircle** creates a circle with a given radius with using a TrivialShader. Trivial Shader stands for a simple shader that has one given color. Having this geometry is possible to render this object multiple times in different locations, different scale and different colors. The difference between these two is that **HollowCircle** contains a extra argument, a line width.

The **Rectangle** creates a rectangle with a given width and height and line width. Having this geometry is possible to render this object multiple times in different locations, different scale and different colors.

The **RegularPolygonFilled** is a polygon with a given radius and N number of corners with regular angles between them. It makes it possible to create a Equilateral Triangle with $N = 3$, a Square with $N = 4$, Regular Pentagon with $N = 5$, a Regular Hexagon with $N = 6$ and so on (Figure 5.3). Having this geometry is possible to render this object multiple times in different locations, different scale and different colors. It has associated an OnPress function to determine whether the point pressed in the screen is inside such polygon or not. We made that after $N = 5$ the area is not considered a polygon but a circle button.

The **RegularPolygonHollow** is a polygon with a given radius, a line width and N number of corners with regular angles between them. The N value is the same as RegularPolygonFilled. The Regular

Polygon Formula was provided by [8]

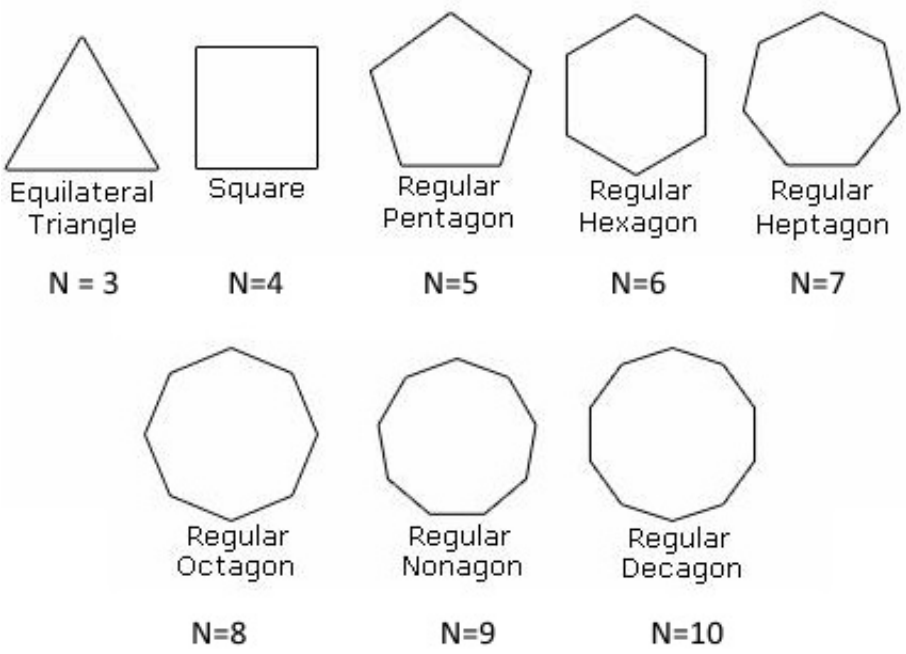

Figure 5.3: Regular Polygons.

5.2 Particle System

As previously mentioned in this chapter, we based on the assumption that it is possible to recreate a particle system in Android NDK. So with that in consideration, we searched for an engine to produce such effect, and we found HGE [15] a C++ game engine for Windows. In the demonstration of this engine[9], (more than 10 particles systems) were generated and running at 60FPS.

After recreating the engine particles class in our Engine, we needed to update the particles with a count of seconds passed in each frame. To do this there were two versions of time systems we could use: the monotonic clock or the realtime clock. However we found, by printing both clocks in the screen of a test scene in the device, that there were some differences of miliseconds between the clocks. At the time we decided to use the monotonic clock of the device, since we wanted to compute elapsed time between updates. The monotonic clock always go forward, but the realtime clock can go backwards and forward.

In terms of how it updates its time, this system uses a function that returns a delta time of its previous call. But at first, we only used an update time that was given in the function of each frame of the **Scene**, but that caused a sightly jump in animation, because the time was saved and used on other computational heavy operations in the scene and not updated until next call of frame update.

The particle system uses a basic structure, with a **Particle** class that represents the structure of a single particle. This class contains:

[8] `http://stackoverflow.com/questions/3436453/calculate-coordinates-of-a-regular-polygons-vertices`
[9] `https://www.youtube.com/watch?v=4ZrcIBDcPTI`

- The coordination of the particle **Location**.
- The vector representation of the **Velocity** of the particle.
- The **LocationDelta**, representing the location shift at each movement.
- The **Gravity** of the particle.
- The **RadialAcceleration** of the particle.
- The **TangentialAcceleration** of the particle.
- The **Spin**, the rotation variable of the particle.
- The **SpinDelta** value that increments at the Spin of the particle.
- The **Size** of the particle.
- The **SizeDelta** value that increments at the Size value of the particle.
- The **Color** of the particle.
- The **ColorDelta** value that increments at the Color value of the particle.
- The **Age** value of the particle.
- The **TerminalAge** value of the particle.

Then there is a **ParticleSystem**, which contains an array of particles in memory. The Particle System properties are stored in a structure called **ParticleSystemInfo**. To simulate an explosion, like in Figure 4.8, Figure 4.9 and Figure 4.10, we associate a comet like effect for each particle of the burst explosion by associating to each particle a ParticleSystemInfo for the tail effect. This way of creating a particle system was based on Unity base Particle System class.

The ParticleSystemInfo has the following information:

- The **Sprite**, corresponding to the ParTexQuad of the particles.
- The **Tail**, corresponding to the ParticleSystemInfo for the tail effect.
- The **Number of Emission**, which represents how many particles this particle system generates.
- The **Lifetime**, i.e. for how long the particle system will live.
- The **Burst**, which is a boolean value to say if it is a burst or not.
- The **Destination**, which is a boolean that makes a the particle move in a fixed incremental movement.
- The **DestinationRadius**, which is the fixed incremental value of radius in the x and y direction.
- The **DestinationAngle**, which represents the fixed incremental value of angle.

- The **ParticleLifeMin** and **ParticleLifeMax**, which represents the range of life of the particles.
- The **SpeedMin** and **SpeedMax**, which represents the range of speed for the particles.
- The **GravityMin** and **GravityMax**, which represents the range of gravity of the particles.
- The **RadialAccelMin** and **RadialAccelMax**, which represents the range of radial acceleration of the particles.
- The **TangentialAccelMin** and **TangentialAccelMax**, which represents the range of tangential acceleration of the particles.
- The **SizeStart** and **SizeEnd**, which represents the change in size of the particles over time.
- The **SpinStart** and **SpinEnd**, which represents the change in spin rotation of the particles over time.
- The **ColorStart** and **ColorEnd**, which represents the change in color of the particles over time.

The **Burst** effect is accomplished by generating all particles number in a moment instance, instead of the automatic emission over the **Lifetime**. This burst represents the firework explosion.

The **LocationDelta**, **Destination**, **DestinationRadius** and **DestinationAngle** are designed to recreate the bursts of type 2 and type 3 in reference of the section 4.3. The **Destination** valued true is of type 2 or type 3. Type 2 has a **DestinationRadius** of x and y with same value, while type 3 has x and y different values. The **DestinationAngle** referes only for type 3. If **Destination** is set to false it is a burst of type 1.

During development we faced the challenge of changing the values of the particle system that we were using. Instead of building a particle system editor with all changable values, we decided to load the particle systems from file in the device. This way, it was easy to simply change values and load the file information and view directly the effects of the particle system.

A **ParticleManager** manages the particles, it controls the creation, the render, the update and the deletion of particle systems.

5.2.1 Discussion

Here is the example of particles sytems configuration files for a main explosion burst, Figure 5.4 and for tail particle system, Figure 5.5 for the effect of Figure 4.8. Recall that each particle of the main burst particle systems, generates tail particles systems, and the main burst particles are not visible, only the tail particles systems are visible.

We made a testing scene were we were able to create to a maximum of **a range of 45 to 50 particles systems** of this kind, which make a total maximum of **12500 particles** (50 systems, 25 particles generated in main particle system and 10 particles for each tail particle system) in an instance. We achieved a range of 17 to 21 FPS (Frames per Second) drop, from a constant of 60 FPS. It is important to refer that every Android device has a VSYNC functionality [16] and maintains the maximum of 60 FPS. The testing scene only ran with particles systems.

```
sprite: particlew.png
nEmission: 25
fLifetime: 2
bburst: 1
fParticleLifeMin: 0.1
fParticleLifeMax: 2
fDirection: 6.28319
fSpread: 6.28319
fSpeedMin: 100
fSpeedMax: 150
fGravityMin: -50
fGravityMax: -50
fRadialAccelMin: 0
fRadialAccelMax: 0
fTangentialAccelMin: 0
fTangentialAccelMax: 0
fSizeStart: 0.1
fSizeEnd: 1.5
fSpinStart: -20
fSpinEnd: 20
colColorStart: 1 1 1 1
colColorEnd: 0 0.1 1 0.1
```

Figure 5.4: Firework comets.

```
sprite: particlew.png
nEmission: 10
fLifetime: 5
bburst: 0
fParticleLifeMin: 0.5
fParticleLifeMax: 1
fDirection: 0
fSpread: 0
fSpeedMin: 0
fSpeedMax: 0
fGravityMin: 0
fGravityMax: 0
fRadialAccelMin: 0
fRadialAccelMax: 0
fTangentialAccelMin: 0
fTangentialAccelMax: 0
fSizeStart: 1
fSizeEnd: 0.1
fSpinStart: 0
fSpinEnd: 0
colColorStart: 1 1 0 1
colColorEnd: 1 0.5 0 0.1
```

Figure 5.5: Firework burst explosion.

Even though we could achieve this values, we realized that it is not the efficient way of creating these effects, since there is a draw call of OpenGL for every particle, and given that we later found that maybe we could achieve better performance if we made a draw call for a big collection of particles. We tried to change, by separating the **ParTexQuad** and **ParShader** for particle systems of the main **TexQuad** and **OurShader** but because of time constraints we decided to implement what is essencial for this thesis and let this feature for future work.

5.3 Game Scenes

Next we will present a series of Scenes and in each we state the technical challenges that we faced and how we found a possible solution for them.

5.3.1 Building Level Editor Scene

We will only use in consideration of this final version of the level editor scene, since during development we created a series of mini prototypes for each feature of the level editor to be tested before we merged into the main level editor scene.

The first step for this Level Editor Scene was to make a simple playback controls of the sound with the specific time position of the song. We started by placing the buttons of the playback controls: the play button, the pause (in the same place as the play button), the moving forward and moving backwards.

During this phase we faced a problem: the time elapsed in miliseconds had different values from the time elapsed in the sound file of the sound manager. This is because the sound file from the OpenSL ES pointer of the current position of the song does not progress at the same pace as the date time of the device. The sound pointer of the current position of the song is updated without a constant value from the background thread playing the sound file. To solve this problem we devised the following solution:

every time the sound pointer position was updated, the time elapsed from the song was also updated to accomodate the sound position and the time elapse, as it follows:

```
currenttime += clock()/1000.0f - previoustime;
previoustime = clock()/1000.0f;

long position = (long) m_soundmanager->GetSoundPosition(songName);
if(position != lasttimesong)
{
        currenttime = (currenttime + position)/2.0f;
        lasttimesong = position;
}
```

This way, even for milimeters the song time is timed correctly. Then for the move backwards in time and move forward in time we decided to move 100 ms backwards and forward in the time elapsed, and update the position of the sound file. We also added a restart button for easyly restarting the level editing and viewing all changes from the beginning.

When we decided to place the game objects (**BeatObjects**) into the level, both the note object (**Hitbeat**) and the slider (**SliderBeat**), we wanted to do in a centralized way, so the screen to play the game would have the same information and not repeat the code. For this purpose we created the **Level** class, that contains all the **BeatObjects** and the current time of visibility for each object. It is important to add that the **LevelEditorScene** had the song controls, and then it sends the current time to the **Level** class, that displays the correct timelapse.

Since the current time could go forward and backwards with this previous code, we decided, after the game objects were ready to be displayed, an internal count timer would start and update its value to the point of the timing, instead of having a specific time to render that would go forward and backward. With an internal timer, the objects would render smoothly with the timing indicators moving with the correct timings.

Then after we added several game objects in the **Level**, we found that some where not necessary to render from a vector of objects. So we used a *start index* and a *end index* to range the game objects to render. This way, we do not render unnecessary objects.

The next issue that we faced was that we needed to recreate the effect of when game objects overlap, maintaining the first active above all others. To accomplish this we used the stencil buffer feature to achieve such effect.

```
glEnable(GL_DEPTH_TEST);
glEnable(GL_STENCIL_TEST);

glStencilFunc(GL_LEQUAL, 2, 0xFF);
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
glStencilMask(0xFF);

for(int i = m_idbegin; i <= m_idend; i++)
{
        m_beatobjectvector[i]->RenderGameObject(timepass);
}

glDisable(GL_DEPTH_TEST);
glDisable(GL_STENCIL_TEST);
```

When we start to render the game objects, we only draw them in the screen if they are not already drawn in the mask. If it is already drawn, the overlapping parts of the new game object will not be drawn.

This is accomplished because we draw the hollow polygon first, and then the filled polygon occupying the center of the hollow object, in transparent color.

We also used the function that returned the delta time of the previous call of this function, for a more fluid animation of the game objects, instead of using previously the same delta time for a entire frame computation.

After we could place note buttons, we enable them to be moved or deleted. Moving them only changed their value of the coordinates, while deleting is erasing them from the list, and updating the indexes if the deleted object is the last element of the list.

Then we created the firework editor scene. For every note game object is associated with a firework explosion (**FireworkEffect**) and a firework tail comet (**FireworkTail**). Both this firework explosion and the tail comet are loaded from a file of particle systems.

We then recreated the movement of the firework tail, by updating the firework tail position, making the effect of tai, as a simple straight trajectory. Then we added a button to change its color for both the trail comet and firework explosion.

Next, to recreate the effect of a curve trajectory of the firework tail, we created its movement from a bezier curve. We could set up the control points and the following algorithm would perform the movement.

```
1  float dt = (timepass - m_beforetime)/(m_starttime - m_beforetime);
2
3  float xa = getPt(m_vec1.x, m_vec2.x, dt);
4  float ya = getPt(m_vec1.y, m_vec2.y, dt);
5  float xb = getPt(m_vec2.x, m_vec3.x, dt);
6  float yb = getPt(m_vec2.y, m_vec3.y, dt);
7
8  float x = getPt(xa, xb, dt);
9  float y = getPt(ya, yb, dt);
10
11 particle->MoveTo(x,y);
```

The before time is the time that the firework tail starts, and *start time* is the time to be at the specific explosion place.

Then we made possible to change the type of the firework. We placed the buttons in the same place, changing the button and the current firework type for every time it is clicked. For every time there is a type 3 firework, a rotation button appears to rotate 22.5 degrees counterclockwise.

Then we added the slider game object. Like it was stated, at first we made it with a different set of firework tails and explosion bursts along the path, all of them edited independently. Later we changed so that they were simple comets, only changing their color, and also their control points to achieve some kind of trajectoy effect for each comet.

At this point, we decided to create the save file for the level, as explained in Section 4.5.4. We saved the files in the storage of the device.

To view if the level was done correctly, we designed a visualization mode, Were we chose the default timings for the explosions.

During the development of the Level Editor, we also found a timing delay between the sound and the visual cue. We found this was due to latency of the device, so we created a Calibration Scene.

5.3.2 Building Level Calibration Scene

Because of the latency of the device, we made a device calibration scene, in which we made a visual indicator like a metronome, and we asked to tap along with the indicator, calibrating the video lag, like is said in [17]. There was a countdown from 0 to 4, in loop, and when 0 was displayed, a square image would appear, and the user was to press at that time. In this scene there was visible a value of the difference between the exact time and the time pressed, giving the latency for the device. This value was the mean value of the actual difference and the previous difference in time. After many trials, and if the value did not change, that meant that the value was the lantency.

Due to time constraints, we developed this calibration scene in separate from the rest of the prototype and used it to change the latency value in a configuration file. The new time was for the rendering of every game object:

```
for(int i = m_idbegin; i <= m_idend; i++)
{
        m_beatobjectvector[i]->Render(currenttime + calibrationtime);
}
```

5.3.3 Building Level Player Scene

Since we had a visualizer of the level, it was easy to transpose such visualizer into the scene to play the game.

We added the enthusiasm meter, and save the counters of misses, ok timings and perfect timings in the Level class. We added the accuracy, computed using the following formula:

```
Accuracy = (OK_count * 50 + Perfect_count * 100) / ((Miss_count + OK_count + Perfect_count) * 100.0f);
```

We give less points to the Ok counts and more points to the Perfect count.

The score of a given action is a count of the percentage of accuracy multiplied by the score associated with the current timing performance (OK is 50 and Perfect is 100) as shown in the following formula:

```
if(current_percentage_accuracy > 70% && current_percentage_accuracy < 90%)
{
        current_score = 50;
}else if(current_percentage_accuracy >= 90%)
{
        current_score = 100;
}

total_score += current_percentage_accuracy * current_score;
```

The associated grade was already discussed in Section 4.2.

6

Evaluation and Analysis

In this chapter, we present the procedures and results of evaluating the prototype of this dissertation as a form of playtesting.

6.1 Playtesting

For the evaluation we used a tablet ZTE E8Q[1]. We created 2 levels with the level editor. Both in normal difficulty, meaning that there was no variation on the delay variable of speed of appearance of game objects. Both songs were faded out at the end of the second chorus.

The first level song chosen was Justin Timberlake's "Can't Stop The Feeling". This level was composed by note game objects and sliders. In the beginning there was a small amount of note objects, which increased at from introduction to the first verse. In transitions between stanzas there were no game objects to be played. The sliders were located at both pre-choruses, and there was a heavy sequence of note game objects in the two choruses. In terms of level design, we created simple patterns that we repeated constanty in the sections of the song. Both choruses have almost identical variation. The game objects were spaced and did not overlap.

The second level song chosen was Sia's "The Greatest". This level was only composed by note game objects, but in a smaller proportion than in the first song. In the beginning it started with a heavy amount of note game objects only at the first verse. There were no game objects at both pre-choruses,

[1] http://specsen.com/tablet-pc-zte/zte-e8q-3g-8gb/

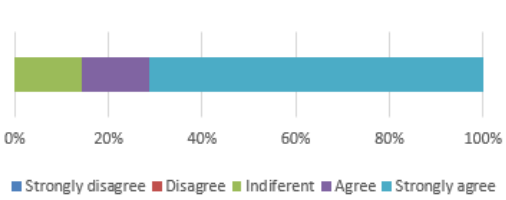

Figure 6.1: Like the song of the first level.

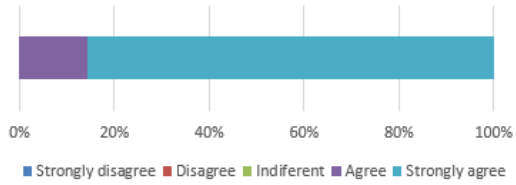

Figure 6.2: Prefer to play the level with song playing of first level.

and at the choruses the game objects display the same pattern. In other sections of the song, instead of the choruses, the patterns had variations and no repetition. The game objects were overlapping at some of the time.

Participants tested our game in a room, with no distractions. We started by explaining that they would play 3 levels of a prototype of a game with a basic description, and asked to answer a questionnaire and answer some open questions as a interview. They were asked to start by responding the first part of the questionnaire with some questions, and then they played a level twice. One time with the song playing the background and another without the song, alternating the order for each participant, to assess if playing with music would increase the score, therefore the experience. Next, they would answer another part of the questionnaire, and play a third time, and afterwards they would do the rest of the questionnaire.

The questionnaire was anonymous. The questionnaire started with an assessment of the player profile. Then after they played the first two levels, they would answer a second part of the questionnaire related to what they played. After the third level, whey would answer the rest of the questionnaire. There were questions about the experience of the players and some questions about gameplay. There were questions with open answer and likert scale answers. Along the answer of the questionnaire, we asked for some questions as a interview to take note of possible changes to the future work.

6.2 Results

We had 14 players to playing our protoype. The participants were between 21 and 28 years old. More than 60% of the participants considered themselves as hardcore players, and 57% play video games everyday.

After playing the first two levels, when asked if they like the song, 70% strongly agreed, and when asked whether they prefer the level with the song on the background, more than 80% strongly agreed. When asked why, they listed as reasons that the music predicted the game objects appearance, it was more fun, and without music, the level seemed longer.

We also, wrote down the score of the performance of the players, when played the level with and without the song, but the results were inconclusive, and we thought that was because of the factor that the player was unfamiliar with the gameplay and performed poorly the first level, and by the second time they knew the sequence from previous interaction with the level.

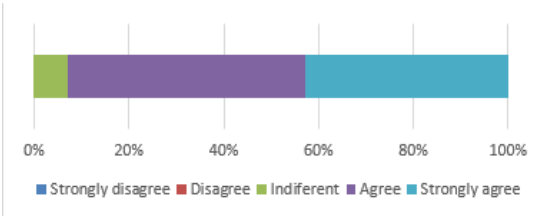

Figure 6.3: Like the song of third level.

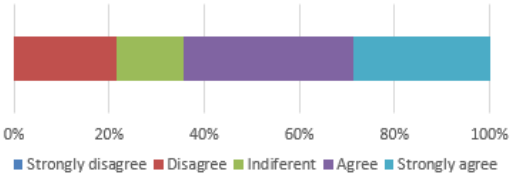

Figure 6.4: Prefer to play the third level in comparison with second level.

After playing the third level, when asked if they preferred the song in comparison to the previous level, they mostly agreed, and when asked if they preferred the third level comparing to the first level with song they also mostly agreed. When asked why, they stated that it was because the music was less ”happy” than the previous, and more challenging as well.

We decided to ask this question because of the difference in level design. The first level had repeating patterns and the the third had variation of patterns without repeating extensively. However, the results did not give much insights in that matter, perhaps there should been more specific questions, but such subject is out of the scope of this project.

Next we assessed the Game Experience Questions for In-Game experience from [18], as shown in Figure 6.5.

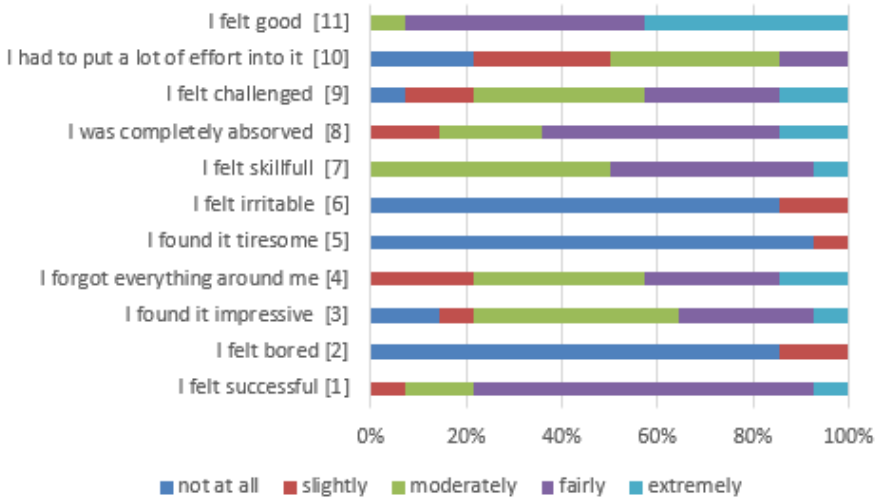

Figure 6.5: Game experience questionnaire.

In reference to [18], in terms of Competence (questions 2 and 9) it can be assumed as a moderate result in skill and success, since the opportunity system was too easy to regain lives and not much unforgiving. For Immersion (question 3), this result can be related to the fact that the music and the gameplay were sightly unsyncronized, but this was because of level design, since the objects can be placed anywhere in the timeline and the creator sense of what beats of the song should be more suited to the game objects. In Flow (questions 4 and 8) there was slight sense of flow, for those who liked the songs, but because of the unsynchronization it could have affected the experience. In Challenge

(questions 9 and 10) there was none, because it was easy to gain opportunities and the players did not worry losing. Negative effects (question 2 and 5) seemed to have good values, the game was not boring or tiresome, having received many comments on good aspects of the game (discussed later). Finally the Positive effects (question 11), received comments because of the sighly desincronization of the music, but also other reason that might have contributed to these values, because the first song was more ”uplifting” and the second more ”sad”.

In terms of game engine and gameplay directions, we asked some more questions in Figure 6.6.

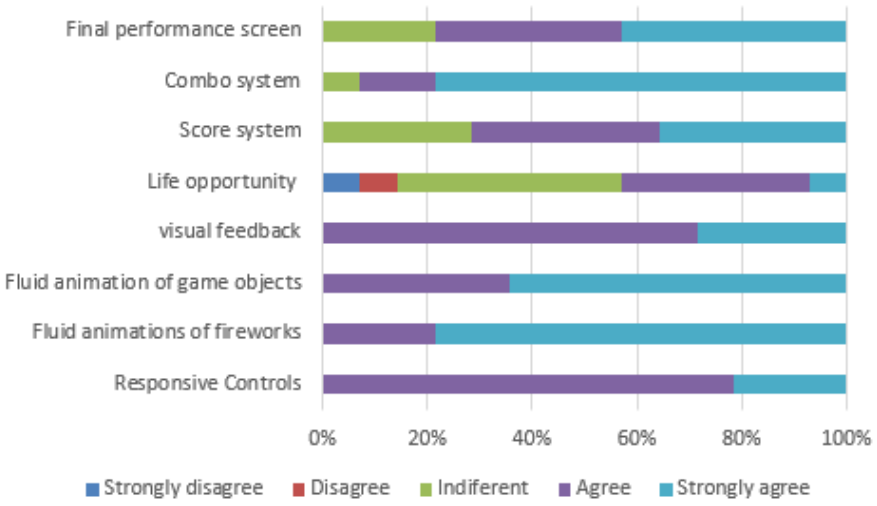

Figure 6.6: Game engine and gameplay mechanics.

In terms of responsive controls, these results may be because of desynchronization of the level and the song, but also it is also possiblly causes, according to some players, by the response of the touch of the tablet, because some use the point of the finger for touch and it need a sightly more pressure for the tablet to accept as a touch. Also the slider game object was not well understood, only few were able to successfull understand its function. Either the movement of the finger was too slow or too fast. Also during questionnaire, players gave some feedback on the slider, and how to better improve it.

The animations of the fireworks and the animations of the game objects were viewed as fluid, reaching one of our goals. The visual feedback achieved moderate results. We believe it is because of the slider problem, but also because of the sight desincronization of the song and the level.

Life opportunity had the most indiferent feedback, maybe because there was no challege in losing opportunities. The players also liked the score system explanation and understood the combo system (more accuracy = more points, more combo = more points). Regarding the final scene of score performance, there were not many players that paid attention to this screen, maybe because we said at the beginning of the playtesting that the score was not important. We later acknowledged it as a flaw, since it influenced the experience.

When asked if the music and the gameplay were related, and asked how, we had many answers, as the music helps to know when the game objects would appear and when to touch them; the rhythm of the music was synchronized with the game objects; there was a choreography of fireworks and the

song. However some players refer the lack of synchronization of the beats of the song and the level objects.

When asked what they liked less about our game, many referred the slider game object. It is something that needs to be worked on.

Many gave improvement suggestions, that were taken in consideration.

And at last, when asked if the players would play the game again 71% strongly agreed and the remaining 29% only agreed. This leads us to think that much work is needed for completion and for this players to play again.

7

Conclusions and Future Work

The main purpose of this work was to develop a music rhythm game for the Android mobile platform, by creating levels with the help of an editor. Even though we found issues (such as the difficulty in placing the game objects at the right beat time, because there was no time snap ruler for the game objects, due to time constraints), we can say we accomplished other things necessary for making a game. The results show that even with the small unsynchronization, the game can still be considered a rhythm music video game. It has the main points of a music rhythm game: the player must react to visual stimuli to achieve a score and performance points in the rhythm of the song being played. However, the tests were inconclusive on whether the song would influence the gameplay, because we planned that the players would perform better if playing the level with the song instead of without song. This may be because we did not allow the player to be more familiar with the controls before testing this hypotesis. We made this test right out in the beginning, and we could have had better results if we had players playing a test level before testing what we wanted. Also, the taste of music did influence the appreciation of the level.

In terms of other subobjectives, we were able to create a game engine that implemented a fluid particle system and fluid game visual animation, as shown in the results. This implementation, created a lag-free performance for the player. The gameplay was strongly influenced by Osu!, an existing videogame.

Overall, the game experience had good results, but our use of simplicity and avoidance of challenging, in terms of opportunities of life that were easy to regain, may have affected this part of the

experience.

We know that we need a lot of things to work, specialy the unsynchronization. For this we thought, in future work to create some kind of music analysis and give cues locations in time to when place the game objects in the level editor. If we accomplish this, maybe we can start working on improvements shared with the playtesters that we will take in consideration. Other issue is to remake the slider game object gameplay, because it seemed difficult to understand the way to interact with it.

Every tester agreed that they would like to play this game again, and we will focus our future work to deliver such experience to the these players and possible future players.

Bibliography

[1] "Industry facts - entertainment software association." http://www.theesa.com/about-esa/industry-facts/. Accessed: 2016-10-12.

[2] "Is the era of music video games really over?." http://mashable.com/2011/09/26/music-video-games/. Accessed: 2016-10-12.

[3] "Simon turns 30: The history of the toy and gaming's first grudge." http://www.1up.com/features/simon-turns-30. Accessed: 2016-10-12.

[4] "Hardcore gaming 101: Otocky." http://www.hardcoregaming101.net/otocky/otocky.htm. Accessed: 2016-10-12.

[5] "The magic box - 1997 top 30 best selling japanese console games." http://the-magicbox.com/Chart-BestSell1997.shtml. Accessed: 2016-10-12.

[6] M. Pichlmair and F. Kayali, "Levels of sound: On the principles of interactivity in music video games," in *Proceedings of the Digital Games Research Association 2007 Conference" Situated play*, Citeseer, 2007.

[7] "Mapping techniques - osu!wiki." http://osu.ppy.sh/wiki/Mapping_Techniques. Accessed: 2016-10-12.

[8] "Catching waveforms: Audiosurf creator dylan fitterer speaks." http://arstechnica.com/gaming/2008/03/catching-waveforms-audiosurf-creator-dylan-speaks/. Accessed: 2016-10-12.

[9] A. V. LOPES, "SISTEMA DE PARTÍCULAS PARA DISPOSITIVOS MÓVEIS NA PLATAFORMA ANDROID," 2012.

[10] "Java vs c app performance – gary explains." http://www.androidauthority.com/java-vs-c-app-performance-689081/. Accessed: 2016-10-12.

[11] K. Nesbitt, J. Tulip, J. Bekkema, and K. Nesbitt, "Multi-threaded game engine design Multi-threaded Game Engine Design," no. October, 2016.

[12] M. McShaffry and D. Graham, *Game Coding Complete*, ch. 7, pp. 175–193. Delmar Learning, 4th ed., 2012.

[13] “android-ndk/endless-tunnel at master · googlesamples/android-ndk · github.” https://github.com/googlesamples/android-ndk/tree/master/endless-tunnel. Accessed: 2016-10-13.

[14] “Opensl es - the standard for embedded audio acceleration.” https://www.khronos.org/opensles/. Accessed: 2016-10-14.

[15] “Github - masonm12/hge: Archive of haaf’s game engine..” https://github.com/masonm12/HGE. Accessed: 2016-10-13.

[16] “What is vsync in android?.” https://nayaneshguptetechstuff.wordpress.com/2014/07/01/what-is-vsyc-in-android/. Accessed: 2016-10-17.

[17] “How do rhythm games stay in sync with the music?.” https://www.reddit.com/r/gamedev/comments/13y26t/how_do_rhythm_games_stay_in_sync_with_the_music/. Accessed: 2016-10-14.

[18] D. Kort, “The Game Experience Questionnaire,” 2013.