

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Music-Based Procedural Content Generation for Games

Pedro Maria Resende Vieira de Castro



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Pedro Gonalo Ferreira Alves Nogueira

July 30, 2017

Music-Based Procedural Content Generation for Games

Pedro Maria Resende Vieira de Castro

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Jorge Manuel Gomes Barbosa

External Examiner: Luís Filipe Guimarães Teófilo

Supervisor: Pedro Gonçalo Ferreira Alves Nogueira

July 30, 2017

Abstract

Procedural content generation (PCG) has been used to create content for video games since 1980's. It allows an easier creation of levels and other game content while providing the player with a more refreshing and less boring game-play due to the constant game level variety.

Although the process of PCG is said to be automatic, it is usually used with an interface where the level designer has to choose and define the creation parameters for each kind of level desired. The purpose of this thesis is to create a new paradigm, where the whole level is created using the existing designed game content and parameters without the need of anyone's intervention. The whole choice of the parameters and settings used would be defined by music.

We can extract various kind of information from any piece of music, for which different Musical Information Retrieval (MIR) tools can be used. And then using this information we can map them in various different ways to select our parameters, creating in this way a paradigm where the focus is changed from a low-level parameters input method, to a higher one where these are altered and interpreted from a music track, which in turn would result in different kinds of levels and game-play by simply using a new piece of music. The mapping from the features extracted to the developed PCG methods had positives results that shows that it is possible to positively influence the PCG methods with music.

Resumo

A geração de conteúdo procedimental (PCG) tem sido usada para criar conteúdo para vídeo jogos desde os anos 80. Esta permite uma criação de níveis e outro conteúdo de jogos mais fácil permitindo ao mesmo tempo uma experiência mais inovadora e menos aborrecida ao jogador devido à sua constante variedade.

Embora o processo de PCG ser considerado automático, ele é normalmente utilizado em conjunto com uma interface onde os designers de níveis tem que escolher e definir os parâmetros de criação para cada tipo de nível desejado. O objectivo desta tese é criar um novo paradigma, onde o nível é todo criado utilizando o conteúdo do jogo e parâmetros disponíveis sem haver a necessidade de intervenção de alguém. Toda a escolha de definições e parâmetros é então feita pela música.

É possível extrair varias informações de qualquer peça musical, para isso, várias ferramentas de extração de informação musical (MIR) podem ser usadas. Usando estas informações é então possível mapear as diferentes formas de seleccionar os parâmetros, criando desta forma um paradigma onde o foco muda da escolha de parâmetros de input de baixo nível para uma de alto nível onde os mesmos são alterados apenas pela escolha de musica resultando em diferentes níveis e gameplay para diferentes musicas. O mapeamento entre as features extraídas e os metodos PCG desenvolvidos tiveram resultados positivos, que mostram que é possível influenciar os metodos PCG com música.

Acknowledgements

First I would like to thank my supervisor Pedro Nogueira for giving me the opportunity to work on a project I really enjoyed, in an area I wanted to have more experience.

Next I would like to thank all of my friends who supported me and kept me company during this last semester of work.

Finally, I would like to thank my family for all the patience and support given all these years.

Pedro Castro

“Not all those who wander are lost.”

J.R.R. Tolkien, *The Lord of the Rings*

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation and Goals	2
1.3	Outline	3
2	Procedural Content Generation	5
2.1	Introduction	5
2.2	PCG's taxonomy	6
2.3	Methods and generated content	7
2.4	Games Examples	9
2.5	Procedural content generation and music	11
3	Music Information Retrieval	13
3.1	Introduction	13
3.2	Music Analyses and Features	13
3.2.1	Low-level Descriptors	13
3.2.2	High-level Descriptors	14
3.3	MIR Tools	15
3.3.1	Essentia	15
3.3.2	Spotify	16
3.4	Summary	16
4	Generating the Game Content	17
4.1	Dungeon Map Structure	17
4.1.1	Map Structure Generation	18
4.2	Level Design	21
4.2.1	Backus-Naur Form Grammatical Evolution Level Generator	21
4.3	Assets Creation	22
4.3.1	Genetic Evolution for Assets Creation	23
4.4	Environment	24
5	Mapping MIR Features to PCG Parameters	27
5.1	Extracted Features	27
5.2	Influencing the Parameters	28
6	Results and Experiments	31
6.1	Game Results	31
6.1.1	Game Concept	31
6.1.2	Results	32

CONTENTS

6.2	Experiments	35
6.2.1	Experiments Results	36
6.2.2	Experiments Final Analysis	43
7	Conclusion	45
7.1	Future Work	45
	References	47
A	Fitness and Mapping Functions	49
B	Experiments Script	53
C	Features and Parameter Values	55

List of Figures

2.1	Types of game content that can be procedurally generated	6
2.2	Example of perlin noise uses for different dimensions	8
2.3	Squiggle weapon trend	10
2.4	Shield weapon trend	10
2.5	Wall weapon trend	10
2.6	Spelunky level representation	10
3.1	Beat Capture	14
4.1	Space Partition	19
4.2	Room Creation	19
4.3	Simple and straight corridors	20
4.4	Centroid and confusing corridors	20
4.5	Player Stat Distribution Example	23
4.6	Monster Desired Stats Example	24
4.7	Items Desired Stats Example	24
5.1	Map for Low BPM and Relaxed Music Track	28
5.2	Map for High BPM and Aggressive Music Track	28
5.3	Relation between Features and Parameters	30
6.1	Map Structure and Environments Results	33
6.2	Player Stats Results	34
6.3	Level Generation and Assets Results	35
6.4	Average Score Given	37
6.5	Average Score Correctness	37
6.6	Map Relation Statistics	39
6.7	Environment Statistics	40
6.8	Player Stats Statistics	41
6.9	Level Generation and Assets Statistics	42

LIST OF FIGURES

List of Tables

3.1	MIR Tools automatic extracted features available	15
6.1	Runs Description	36

LIST OF TABLES

Abbreviations

AI	Artificial Intelligence
BNF	Backus-Naur Form
BSP	Binary Space Partitioning
cgNEAT	content-generating NeuroEvolution of Augmenting Topologies
DF	Dwarf Fortress
ESA	Entertainment Software Association
EA	Evolutionary Algorithms
GAR	Galactic Arms Race
GG	Generative Grammars
IF	Image Filtering
MIR	Musical Information Retrieval
PCG	Procedural Content Generation
PCG-G	Procedural Content Generation for Games
PRNG	Pseudo-Random Number Generator
stats	statistics

Chapter 1

Introduction

The presence of video games in our lives keeps increasing, reaching millions of players every day. The U.S. Entertainment Software Association (ESA) reports video games total consumer spend, to be superior to 23 billion dollars, with 52% of frequent consumers having the opinion that money spent on games as a form of entertainment to be of higher value than other forms of entertainment like DVDs, music or going to the movies. With the Gaming Industry increasing and evolving every year and the high demand for quality in gameplay and design [Entb], the need for fast, cheap and innovative techniques for creation of content increases as well.

1.1 Context

Nowadays, the cost of production for high-quality games is increasingly high, requiring sometimes a few hundred people and 30-40% of the game budget just to create the game content. This has developed to such a point that it is becoming a bottleneck for the game budget and development time [HMVI13].

Procedural Content Generation (PCG) presents itself as a possible solution for the high costs in developing and scaling of game content by programmatically creating varying and personalized content [Bom16].

PCG can help reduce time and memory usage needs, it also greatly improves the diversity of content which can complement some lacks of imagination even when not being used to automatically create the final content. New methods of PCG are developed every year, focusing on adapting the game content to the player's actions or other kinds of input.

Audio is a very important component that highly contributes to the game's experience, and although sometimes overlooked, it is still considered as one of the core facets of computational game creativity [LYT14]. There are many ways to integrate sound and music into games, many games use it as a way to increase the player's immersion and overall experience, a way to hint some environment changes like danger and objectives, or even by trying to adapt the music and

audio as a whole to the personal experience of every player, like in Halo¹ where the soundtrack was composed of multiple layers that depended on several game factors. Other games tried to implement the music as a core mechanic of the game like, Guitar Hero² or as a byproduct of the game mechanics where the user's actions create the audio like in Bit.Trip's Runner³ where each level had its pickups and obstacles placed in a way to form the desired final music. The problem all of these games have is that the relation between the audio and the mechanics had to be developed and synchronized beforehand.

In order to be able to use features of any music to create game content or other mechanics based on, for example, rhythm or loudness, a lot of games started to try and use Music Information Retrieval (MIR) methods or adaptations of those, to extract information from any music file that can be used in game development.

1.2 Motivation and Goals

Every person is different from one another and even the same person, depending on their mood or disposition, can feel like playing a different kind of game. How can game developers adapt to this? Can a game adapt to the user's current need?

Adaptability is a concept related to replayability, where PCG is used to adjust content in relation to the player's skill or actions [Bom16], although this can improve replayability and potentially diversify the player base, encouraging different play styles or skills based on the player's previous actions, it is almost never used in commercial games due to the problems it can create.

To be able to adapt the game to someone's taste without relying on previous inputs, nor asking the player directly for input, for example, with the use of questions, is a problem that has been tackled numerous times. Using a music track as input for the generation could allow for a whole new PCG paradigm, where the information from a music track would be used as input. This way a player could select a music that described what he felt like playing, resulting in the game matching, in some way, to his mood. Of course, different players could have a different idea of what should result from a certain music but using the MIR algorithms to extract low-level and high-level music features, it should be possible to achieve a high degree of similar results for the same kind of music while still keeping them all different.

To study this approach there is a need to develop 3 different modules in which the music feature and PCG game are connected. The first module has a music track as input and should return some selected music low-level and/or high-level features. The second module should consist of a game created via PCG methods, and lastly, the third module would be the mapping of the features to the PCG methods.

The first will be developed based on the research made of known algorithms and methods/tools, while the third one will be either developed from the scratch using previously trained

¹Bungie, 2001

²Harmonix, 2005

³Gaijin Games, 2010

models. By the end of the development phase, it should be possible to extract the required information from any music, map them in the desired way to the PCG module and produce a playable game level which should be similar when the same song or similar songs are used.

1.3 Outline

This document structure is divided into Literature Review (chapter [2-3](#)), Methodology (chapter [4-5](#)), and Results/Conclusions (chapter [6-7](#)).

Introduction

Chapter 2

Procedural Content Generation

This chapter is part of the Literature Review and reviews the domain of PCG methods and their applications to the creation of certain game content, while providing analyses on the methods that have been or could be used for this purpose. It also lists and explains some previous applications of these methods in successful games.

2.1 Introduction

PCG can be referred to as the "algorithmic generation of game content with limited or no human contribution" [TCL⁺13]. The use of PCG in games has been present since the early eighties in games like *Rogue*¹ and *Elite*² where it was used to create dungeons and star systems, this level generation was somewhat regarded as random generation, but the random here does not refer to content created without any consideration for structure. Stochasticity is the aspect of randomness in the generation process, as it determines how much the content varies between runs with identical parameters [HMT13].

PCG is mostly referenced by its positives aspects because of its ability to generate content without the need to have designers work on them, thus allowing for a faster and cheaper method of creating content while trying to keep the same level of quality, but there are other advantages to these methods. PCG is able to improve a game's replayability, thanks to its stochastic aspect which makes the game content always different from the one generated before, even when the same parameters are involved, and is also able to add adaptability to the game by providing the PCG algorithms with inputs that can in some way influence the result of the generated content.

Before talking more about PCG and analyzing all of the different PCG methods and techniques, there is a need to identify exactly what kind of content is normally generated for games. Hendrix et al. [HMT13] divide game content created by PCG into 6 categories: Game Bits, Game

¹Toy, Wichman, Arnold, Lane, 1980

²Acornsoft, 1984

Procedural Content Generation

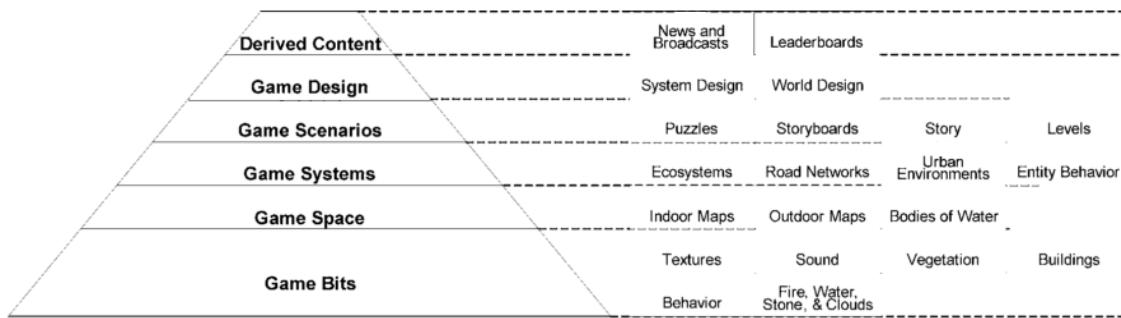


Figure 2.1: Types of game content that can be procedurally generated as divided by Hendrix et al. [HMVI13]

Space, Game systems, Game Scenarios, Game Design and Derived Content. These categories are structured as a virtual pyramid, in Figure 2.1, where the categories at the top can be built with elements from the ones beneath them. Behaviors are not part of the game content but can be created beforehand and attributed in the PCG method.

Game bits normally refer to the most elementary units of game content like textures, sound, buildings, object properties, along with other types that make up the base to create other kinds of content. Game Space, on the other hand, refers to the environment where the game action occurs. This can range from simple dungeons to other more complex indoor and outdoor maps and is one of the contents that is commonly used in roguelike games.

Game Systems are mostly used for the environments, ecosystems, and interactions between different entities and of all the content present in Game Scenarios, the levels are the most popular and representative of the group. Game Design and Derived Content are the ones currently less developed in terms of algorithms and implementations in commercial games.

2.2 PCG's taxonomy

With all the different approaches available it is useful to briefly highlight some differences and similarities between them and that will be shown using a revised taxonomy by Shaker et al. [STN16]

PCG can either be online or offline, the main difference being if the content is continuously generated during game-play or previously created, either before the game session or during game development. Furthermore it can also be divided into necessary or optional where the former represent content that is required for the completion of the game level and the latter represents optional content that can be replaced or discarded by the player at will, this makes it freer since there is no need for it to be always correct while the necessary content has to be more restricted to ensure its correctness.

Another important aspect of PCG methods is the degree and dimensions of control of the generated content. This has to do with the stochasticity of the process and the way one can control the desired result of the process. A good example of this is content generated randomly through the use of a seed where using the same seed creates the same result. Opposed to stochasticity,

Deterministic PCG enables the recreation of generated content previous generated when the same inputs are used. PCG methods can also be categorized according to their **adaptability**, and to whether they are constructive or generate-and-test.

Adaptive PCG, as opposed to static PCG, takes into account the users previous actions when creating new content. Commercial games use mostly static PCG because adaptive could result in future errors or even possible exploits, but it has gained a lot of attention in terms of investigation.

2.3 Methods and generated content

Hendrix et al. [HMVI13] divide PCG for games (PCG-G) into 5 group of methods: Pseudo-Random Number Generator (PRNG), Generative Grammars (GG), Image Filtering(IF), Spatial Algorithm, Modeling and Simulation of Complex System and finally Artificial Intelligence (AI). From these 5, PRNG, GG, IF and Evolutionary (EA) from the AI group will be analyzed in terms of objectives, algorithm and type of game content it is normally used for.

Pseudo-Random Number Generator

Pseudo-Random Number Generator (PRNG) algorithms produce a deterministic sequence of random numbers where each random number is calculated using the previous as part of its calculation. PRNG algorithms are used to create most Game Bits, while also being used, in combination with other methods, for the creation of other categories of game content like the generation of indoor and outdoor maps(Game Space), ecosystems and puzzles for Game Systems and Scenarios, respectively. When it comes to Game Design, PRNG is mostly used to support other methods.

Perlin Noise [Per04] is one of the most common PRNG methods, it can be used in different dimensions for different purposes and it has very good results in the creation of textures, clouds, fire and even other, more complex, content like ecosystems in Figure 2.2.

Generative Grammars

Generative Grammars (GG) originate in linguistics and are a model that define a grammar, or set of rules, to describe a set of linguistic phrases. Dormans [Dor10] shows us an example of how CG can be used for PCG-G to define the level of a game with this grammar as an example:

1. Dungeon -> Obstacle + treasure
2. Obstacle -> key + Obstacle + lock + Obstacle
3. Obstacle -> monster + Obstacle
4. Obstacle -> room

³<http://flafla2.github.io/2014/08/09/perlinnoise.html>




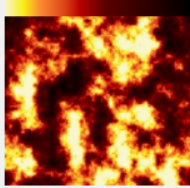
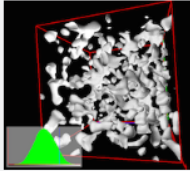

Noise Dimension	Raw Noise (Grayscale)	Use Case
1		 Using noise as an offset to create handwritten lines.
2		 By applying a simple gradient, a procedural fire texture can be created.
3		 Perhaps the quintessential use of Perlin noise today, terrain can be created with caves and caverns using a modified Perlin Noise implementation.

Figure 2.2: Example of perlin noise uses for different dimensions ³

This grammar can help generate possible dungeons that follow this set of rules and can be further developed to cover shortcomings like the minimum size and possible mandatory events. In short, GG can be used to produce sound, vegetation, buildings and also, used together with PRNG, for the generation of indoor maps.

Image Filtering

Image Filtering (IF) algorithms allow for the application of filter matrix to a pixel matrix transforming the original pixel matrix into a new one. Examples of most commonly known filters are blur, motion blur, find edges, sharpen and other examples that use different matrices and parameters like factor and bias.

IF is applied in PCG-G mostly in combination with other methods in the creation of texture, vegetation, water, fire and even complete outdoor maps (erosion, smoothing).

Evolutionary Algorithm

An Evolutionary Algorithm (EA) is a stochastic search algorithm inspired by Darwin's natural selection, an example of these methods being Genetic Algorithms. EA are commonly used with the search-based approach and are based on the fact that given a population, there is a combination of their proprieties that results in a very good solution. This kind of approach needs a population of individuals, a representation method, and a fitness evaluation function. Using this to iterate over generations of the population leads to good results that can be used as content.

2.4 Games Examples

In this section we showcase some games that use PCG-G and explain how and which content is created.

Rogue

Rogue was developed around 1980 and is considered as one of the first games to use PCG as well as the progenitor of all *roguelike* games. It's a single player game that uses random dungeon generation algorithm to create a new level every time the player goes down a set of stairs, this makes the game repeatable with a very low chance of repeating the same level.

Elite

Elite is a space exploration and trading game from 1984 where the space, systems, and scenarios are fully procedurally generated. It was one of first to use PCG to enable the compression of vast information into parameter values in order to overcome hardware restrictions creating thus complex and a vast collection of star systems, galaxies and trade items.

Dwarf Fortress

Dwarf Fortress(DF)⁴ is a game where its main play mode consist of managing individual dwarfs in a fantasy world to create an underground fortress. DF has a very high learning curve, being many times criticized by its difficulty and it mixes the roguelike genre with a *construction and management simulator*. It is a good example of a game where many different aspects of the game, such as the game world, geology, ecosystem, back story and mostly any other aspect, are the result of PCG, even going as far as to generate the color and style of the dwarf's clothes individually. However, these PCG methods are normally simple processes resulting in limited variation of generated game worlds.

Galactic Arms Race

Galactic Arms Race (GAR)⁵ is an online multiplayer space shooter that was developed to feature the automatic creation of particle-based weapons that were generated using PCG based on previous user preferences. GAR uses a custom version of the NEAT evolutionary algorithm called content-generating NeuroEvolution of Augmenting Topologies(cgNEAT) [HGS09] to create the different kind of particle-based weapons and it had surprising results, creating new and creative weapons that neither the players nor the designers had thought about but that were interesting for the gameplay (2.3)(2.4)(2.5).

⁴Bay 12 Games, 2006

⁵Evolutionary Games, 2010



Figure 2.3: Squiggle weapon trend



Figure 2.4: Shield weapon trend

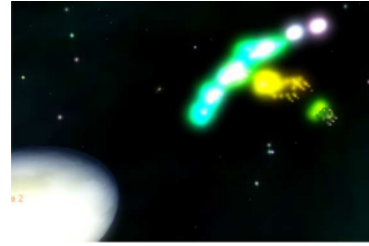


Figure 2.5: Wall weapon trend

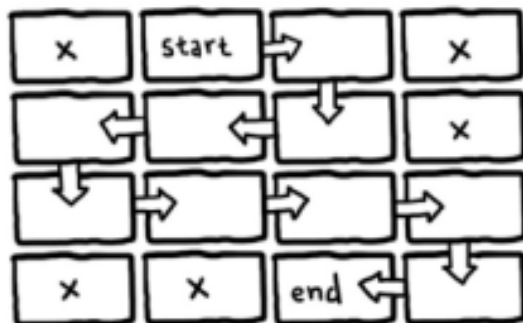
Spelunky

Spelunky⁶ is a 2d platform game where each level is a result of PCG methods, allowing for endless variations of level content. Spelunky levels consist of 16 rooms in a 4x4 grid and each room is represented by 80 tiles in a 8x10 matrix in Figure 2.6. Spelunky rooms are not always connected but can be opened using bombs and other items, the PCG used guaranteed the playability of the level in its entirety.

BeatTheBeat

BeatTheBeat is a mobile game that combines three different mini games design to utilize and test the use of music as input of PCG. The game uses automatically extracted features from the music that are then used to control some of the game's mechanics, like the rhythm in the *Tap by Tap* mini-game, or the tower that only activate to certain aspects of the music in *Music Tower Defense*. BeatTheBeat also has an interesting aspect in which the player is not always playing the game with the features from the song playing, this was used to check if the player noticed any differences [JSL⁺12].

⁶Mossmouth, 2008



a)

```
0000000011
0060000L11
0000000L11
0000000L11
0000000L11
0000000011
0000000011
0000000011
1111111111
```

b)

Figure 2.6: Spelunky level representation [STN16]

Beat Hazard

Beat Hazard⁷ is an arcade shoot-em-up where a player, in control of a spaceship has to destroy ships and asteroids to survive until the end of the music. To play one must choose a song, and the levels time, enemies, visual and sounds are related to the chosen song.

Symphony

Symphony⁸ is a shoot-em-up game where the player maneuvers the ship in order to avoid enemies and destroy them. The game focus on the replayability and the relation of the music chosen to play, with the gameplay. Symphony uses music analyses to try and understand the mood and feel related to the game, with each song/level completed you can unlock new weapons and difficulties which allows for a greater replayability even for the same song.

2.5 Procedural content generation and music

As can be seen in some of the last examples, there are already some games that make use of music analyses to either change the content of the game or to adapt the game mechanics to the music. However, there are very few examples of games whose content is a result of PCG that is affected by the song, changing the game-play of the level in a way unrelated to synchronization of mechanics with the rhythm, creating different and noticeable results.

⁷Cold Beam Games, 2010

⁸Empty Clip Studios, 2012

Procedural Content Generation

Chapter 3

Music Information Retrieval

This chapter is the second part of the Literature Review and reviews the domain of MIR methods, its evolution and current state together with some information about what can be extracted what it means.

3.1 Introduction

Music Information Retrieval(MIR) research started in the 1990's and has, since then, experienced a constant increase in trend as a research field. In its early days, MIR research, focused mostly in working with symbolic representations of music pieces like MIDI but with the fast, constant upgrade to computing power, MIR research was able to evolve, and nowadays it's a field of research whose focus is centered around extraction and inference of meaningful features from music, indexing of music and search/retrieval schemes [SGU14]. From these three, the extraction of music features will be the main focus of the analysis.

The music descriptors, or features, extracted from the music can be classified according to three main criteria: "Abstraction level: which covers from low-level signal descriptors to high-level semantic descriptors; Temporal scope: descriptors that refer to the interval of scope analyzed that can be an instant, a segment or to a complete music piece; and Musical facets: that refer melody, rhythm, harmony/tonality, timbre/instrumentation, dynamics, structure or spatial location." [SGU14]. The extracted music features can help describe a music's characteristics, help find similar music, be used to classify music gender, *danceability* and others.

3.2 Music Analyses and Features

3.2.1 Low-level Descriptors

Low-level music descriptors are extracted using time and frequency domain representations, which are in turn obtained from the audio signal. This music descriptors that are computed directly, or

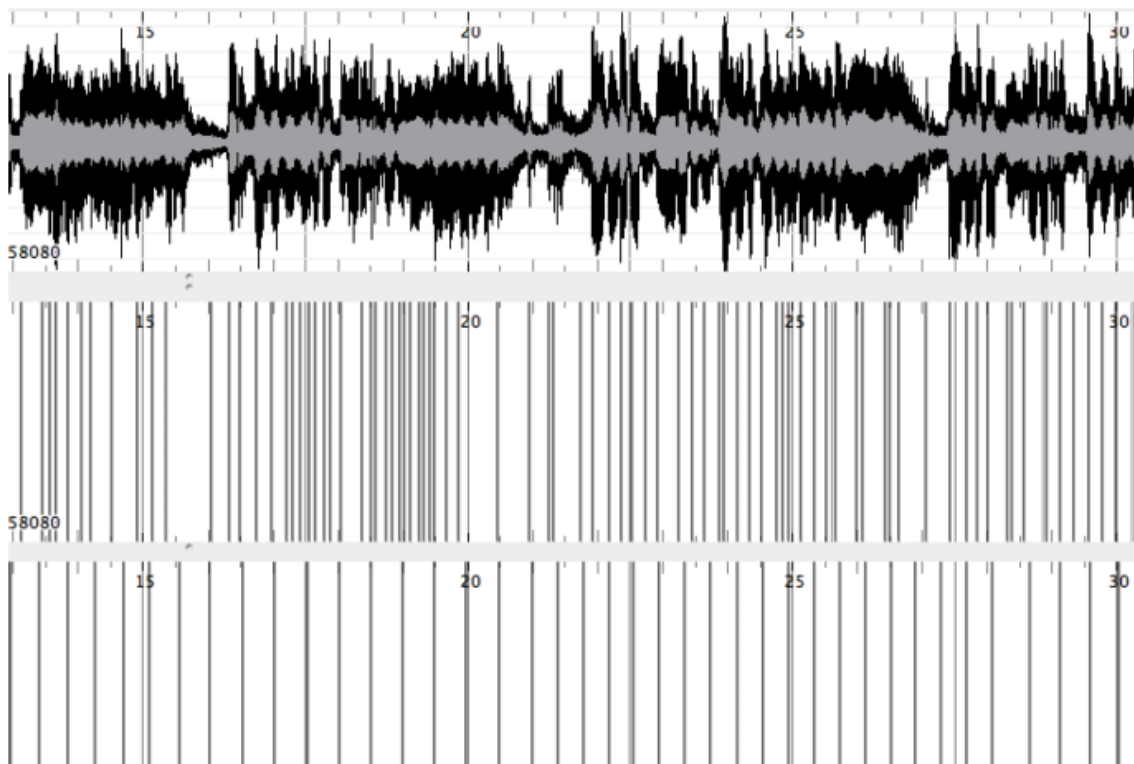


Figure 3.1: Beat Capture [SGU14]

in a derived way, from the audio signal don't have a lot of meaning in a direct use but are very important in the computation of higher level descriptors. Low-level music descriptors are mainly composed of loudness, timbre, spectral moments and pitch .

While the loudness and timbre can help compute rhythm, instrument or genre, pitch descriptors are one of the core descriptors used to process melody, harmony and tonal descriptors. The pitch descriptors main goal is to estimate periodicity in music signals which can be hard and require different approaches depending on the proprieties of the music track [SGU14].

Melody and chords can be used to describe tonality which tells us the key used in the music. Rhythm descriptors are separated into four different components: timing, tempo, meter and grouping in Figure 3.1 for example, beats-per-minute, beat positions and loudness. Another important feature is the novelty detection and segmentation, that can segment a music track into different parts that contain different proprieties.

3.2.2 High-level Descriptors

All of the past descriptors can be used to infer higher level descriptors either by classification or auto-tagging. Classification classifies a music into different into set categories while auto-tagging assigns semantic labels to the track. These can be used to predict high-level descriptors like instrument, mood, culture, genre, danceability among others.

3.3 MIR Tools

There are numerous tools, frameworks and libraries. Some examples of these are: Sonic Visualizer¹ (visual application), Essentia² (C++ library with command line compiled extractors), Spotify WEB API³ (previously "The Echo Nest"), libROSA⁴ (Python library), jMIR⁵ (Java library), meyda⁶ (Web Audi API for custom files). Although in terms of audio analysis these tools have similar algorithms implemented, and can all be used for the same purpose, the focus of our study will be Essentia and the Spotify web API because these offer a number of high-level music descriptors using their pre-trained models which can be seen in Table 3.1.

	Essentia	Spotify
Acousticness	X	X
Danceability	X	X
Dance rhythm	X	-
Gender	X	
Genre	X	
Instrumentalness	X	X
Key	X	X
Loudness	X	X
Mood	X	-
Rhythm	X	X
Speechiness	X	X
Timbre	X	X
Tonal	X	X

Table 3.1: MIR Tools automatic extracted features available ('X' for fully available, '|' for available without MIR and '-' for limited)

3.3.1 Essentia

Essentia is an open-source C++ library with python bindings that can be used for audio analyses and MIR, it's a library that focuses on the algorithms that analyse audio signal and it is complemented with Gaia which is used for the measures and classifications of the result of audio analyses, enabling in this way an extraction of high-level descriptors.

Essentia can analyse any music file, enabling the extraction of the features from any music track, it provides trained models for the classification of high-level descriptors but also provides the means to train custom models easily. Using Essentia can be done with the library, where one can decide exactly how to apply the algorithms and which ones to apply, however, Essentia also provides pre compiled extractors for both the audio analysis and high-level descriptors extractor.

¹<http://sonicvisualiser.org/>

²<http://essentia.upf.edu/>

³<https://developer.spotify.com/web-api/>

⁴<https://github.com/librosa/librosa>

⁵<http://jmir.sourceforge.net/>

⁶<https://meyda.surge.sh/>

3.3.2 Spotify

Spotify integrated a framework previously known as The Echo Nest, into its Web API. Every song present in the Spotify library has been run through a series of audio analyses and model classifications whose results are available via the Web API. Although this tool can't be used for every song, or music file, due to its limitation of having to be present in the library, since all analysis are already done, it proves itself a very fast tool. Another setback it contains is the need of an internet connection and premium Spotify account.

3.4 Summary

As can be seen, there are a lot of tools available for the extraction of music features, and although some of them are high-level and can be used directly, others require a lot of knowledge in the area or computation of the same through some methods like machine learning to extract meaningful data. Another thing to consider is the fact that less and less people have music stored in their computers, so future implementations should take that into account.

Chapter 4

Generating the Game Content

The focus of the thesis is having the game content created by the procedural content generation (PCG) methods be influenced, in a whole, by a music track as opposed to the use of music features like BPM in the game mechanics. The type of game chosen for this was a single-player 2d dungeon roguelike where the player must complete a dungeon floor created by its chosen music input. However, the game and the generated content are completely separated, that is, the whole process of content generation ends with the creation of two files that contain the representation of all created content which, in turn, is read by the game and used to generate the game level.

This chapter is the first part of the Methodology and describes the game content chosen to be procedurally generated, and how they could vary for different levels. All of the content is generated offline, before the game begins, and the implementations of the PCG algorithms was done in Java, without recurring to AI libraries or frameworks, since there was a need to make small alterations during the implementations. These alterations are needed to better control and parametrize the methods, so that it was possible to create many different kinds of highly variable game content.

4.1 Dungeon Map Structure

Desired Content

Most roguelike games use PCG methods to create different maps or room structures for every level and it is one of the most common procedurally generated content in game development.

To have a map structure that can have a great deal of variation and effect on the gameplay it was decided to have it be defined by **rooms** and the **corridors** that connect them. This way the map could vary in terms of overall size, the rooms sizes, the density of rooms and the connections between them.

4.1.1 Map Structure Generation

The implemented map generation is based on two different PCG methods, used for dungeon generation: space partitioning and agent-based dungeon growing.

4.1.1.1 Binary Space Partitioning

Binary Space Partitioning (BSP) is a stochastic method used to recursively subdivide a given space. For this implementation we use it in one of its simplest way, to divide a rectangle into smaller rectangles using a BSP tree. We start with a matrix, where each side has a length of at least 3. Afterward, we begin dividing from the root node, either horizontally or vertically, and adding the children nodes to the tree, this is done recursively until the leaf nodes can't be divided any further.

```

1 Load Root as first Node;
2 Divide the Node Horizontally or Vertically;
3 if Region is Divisible then
4   | Choose a random valid position to divide;
5   | Create 2 new Nodes for each region;
6   | Repeat from Line 2 ;
7 else
8   | End branch;
9 end
```

Algorithm 1: Binary Space Partition Algorithm

To decide if a node should be further divided we use two parameters that control the acceptable partition sizes, **minimum partition size** and the **maximum partition size**. That is to say, if dividing the node would result in a region with an axis smaller than the minimum partition, it wouldn't be divided, and if the chosen dividing axis is bigger than the maximum partition size then it must be divided. Anything in between is decided randomly according to the ratio between the length of the side length and the max partition size. This limit is restricted to the orientation of the division so while it is impossible for a region to have both dimensions bigger than the maximum partition size, it is possible for one of them to be bigger. This allows for a bigger randomization and variety in Figure 4.1.

After having the map completely divided, two points for each region are chosen at random in order to create a room in that region restricted by its **minimum size** parameter which prevents the creation of rooms too small for their purpose in Figure 4.2.

4.1.1.2 Agent-based Path Creation

Having our desired space divided into rooms, we still need to connect them to one another. In order to achieve that, while also providing a more diverse and parameterizable final result, that connection was based on Agents.

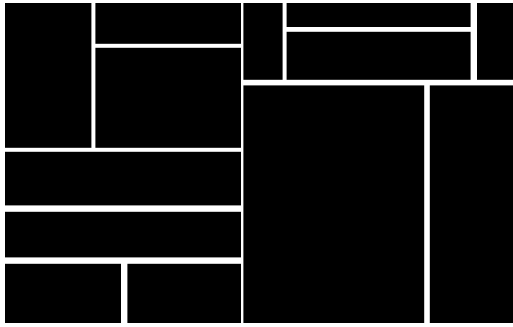


Figure 4.1: Space Partition

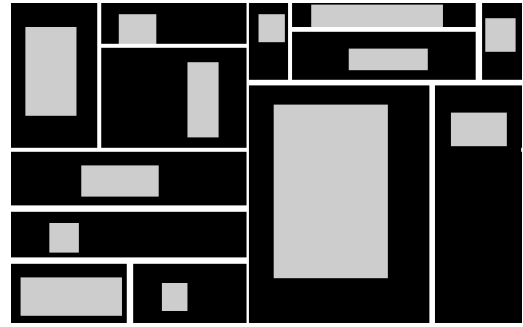


Figure 4.2: Room Creation

An Agent is responsible for the connection between two rooms and it does so by walking freely from a point in the starting room to another point in the destination room, thus creating an open path while walking to its destination. Since the objective of using agents is to prevent the same kind of connections every time, they have a certain freedom when walking.

There are two types of Agents: **Blind Agents** and **Connector Agents**. Connectors have freedom of turning but always walk the right amount in both directions, as such they only alter the shape of the path and not their length. Blind Agents, on the other hand, can move freely but the more they walk the more influenced they are to walk straight to the destination.

```

1  StartPosition ← Get Random Position from Room1;
2  EndPosition ← Get Random Position from Room2;
3  XLeft <- EndPosition.X - StartPosition.X;
4  YLeft <- EndPosition.Y - StartPosition.Y;
5  while Current Position != EndPosition do
6      Move Position in Current direction and update XLeft and Yleft;
7      Turn <- Random(100);
8      if Turn < Turn Probability then
9          if Connector Agent or Blind Agent forced To Finish then
10             Change Direction to the other Axis in order to approach destination;
11          else
12             Change To Random Direction in the other axis;
13             Reduce Stamina;
14          end
15      end
16      Update Position In Axis;
17 end

```

Algorithm 2: Agent Behaviour

With a way to connect rooms all that's left is to decide which rooms connect to which. To ensure that all rooms are connected to one another, an adaptation of a Spanning Tree with some parameters that could control the way the rooms are connected. This was done so in order to

be able to vary between simple one-way maps, confusing maps with all rooms connected to one another or centroid maps, where one room in the middle is connected to every other.

```

1 Room1  $\leftarrow$  Get Middle Room;
2 Mark Room1 as visited;
3 Choose Agent to use;
4 Room2  $\leftarrow$  Get Unvisited Room;
5 Set Agent's Turn Probability; Agent Connects Room1 to Room2;
6 if Map uses Simple Connections then
7   | room1  $\leftarrow$  room2;
8 else
9   | Get Random Percentage ;
10  | if Percentage < Probability of centroid map then
11    | room1  $\leftarrow$  room1;
12  | else
13    | End branch;
14  | end
15 end
16 Mark Room2 as Visited Repeat from Step 3 until all Rooms visited;

```

Algorithm 3: Room Connection Order method

Two different connections to the same map could be, for example, a low probability centroid map that uses Connector agents with low turn probabilities in Figure 4.3, and a high probability centroid map that uses both agents with high turn probability in Figure 4.4. The problem with creating corridors this way is that some rooms end up being extended when either the agent walks alongside the limits of the room or when he turns a lot right outside. To solve this, and because the rooms definition is needed, we apply a short algorithm that checks if a free space is surrounded by 3 cells of the same room, before assigning it to that room.

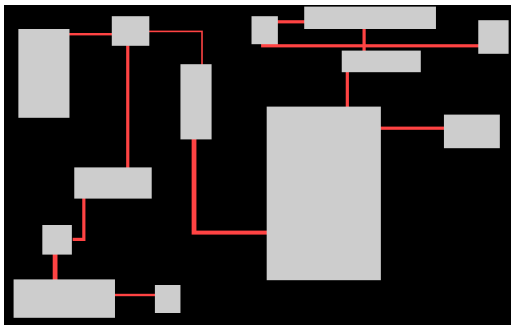


Figure 4.3: Simple and straight corridors

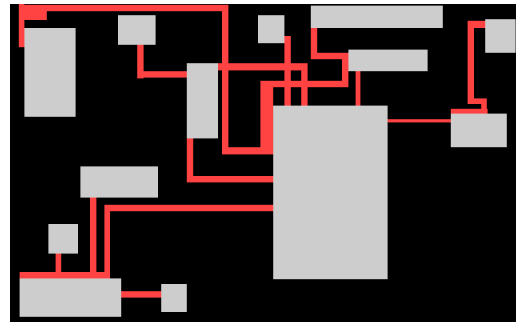


Figure 4.4: Centroid and confusing corridors

4.2 Level Design

Different maps alone aren't enough by themselves, in order to create more consistent and interesting levels, one of the generated content needs to be the level design. Since the map structure is mostly made of rooms, the level design is going to decide on what each room contains (Room Type).

Desired Content

The main objective is to have a list of rooms that always contain the essential rooms and the option of having extra rooms to occupy the remaining rooms. This ensures that small maps still have the most core rooms and that big maps have interesting content. It also allows us to control the level of danger or agitation in the level.

4.2.1 Backus-Naur Form Grammatical Evolution Level Generator

The generation of levels, missions and quests has been an object of study in PCG-G and Generative Grammars have been used for this [Dor10]. In order to generate the Level for our map we will need to know what rooms to create, and for that, we use a Backus-Naur Form (BNF) Grammar to create the rooms for our map:

- 1 $S \rightarrow \langle X \rangle? \langle Y \rangle \langle \text{Treasure Room} \rangle \langle \text{Boss Room} \rangle \langle \text{Locked Exit} \rangle;$
- 2 $S \rightarrow \langle X \rangle? \langle Y \rangle \langle \text{Treasure Room} \rangle \langle \text{Boss Room} \rangle \langle \text{Boss Room} \rangle \langle \text{Locked Exit} \rangle;$
- 3 $X \rightarrow \langle X \rangle \langle X \rangle?;$
- 4 $X \rightarrow \langle X \rangle \langle X \rangle? \langle Y \rangle;$
- 5 $X \rightarrow \langle \text{Zoo Room} \rangle \mid \langle \text{Shop Room} \rangle \mid \langle \text{Bombs Room} \rangle \mid \langle \text{Trap Room} \rangle;$
- 6 $Y \rightarrow \langle \text{Temple} \rangle \langle \text{Fake Temple} \rangle?;$

Algorithm 4: Level BNF Grammar

The grammar by itself is enough to generate a basic structure for the level but it can't be properly parameterized nor adapted to the map. To solve this we combine the BNF grammar with genetic evolution [TJH], this allows us to create a fitness function that can be parameterized and affect the outcome of the generated level. The biggest problem this creates is that the resulting level can have conflicts with the map structure, for example, the number of rooms between the map structure and the final grammar may differ. There are two ways to solve this problem, the first one is to add a variable to the grammar that guarantees the correct number of rooms. The second one is to take that into account in the fitness function which is how we implemented it.

First we consider the following 6 parameters [0,1]:

- Aggressive
- Confusing
- Fun

- Trickery
- Calmness
- Luck

These represent the objective of the generated level and each of the Room Types available increases or reduces the multiplier of a corresponding parameter. Adding these values together with a relation between the number of rooms in the map and the number of rooms in the level, we create a fitness function that ensures the creation of a random level that is related to the parameters.

4.3 Assets Creation

Player

The player stats define the abilities one starts with, whether the character is faster, stronger or more resilient, and is commonly changed in-game before the game starts by selecting different characters or creating your own. The idea is to eliminate the limited choices, creating a different character with balanced stats for each level. The player stats can all be average or some can be higher albeit sacrificing others. The chosen stats for our player were:

- Health
- Armor
- Attack
- Attack Speed
- Speed
- Luck

Monsters and Items

Like the player character, the monsters and items placed around the map are procedurally generated and may have different stats and forms. The monsters are similar to the player but their stats are different, namely:

- Health
- Attack
- Attack Speed
- Blast Speed
- Speed
- Size

Generating the Game Content

<i>Tank Player</i>	<i>Normal Player</i>	<i>Ninja Player</i>
Health: 4	Health: 2	Health: 2
Armor: 2	Armor: 1	Armor: 1
Attack: 2	Attack: 2	Attack: 2
Attack Speed: 1	Attack Speed: 3	Attack Speed: 3
Speed: 1	Speed: 2	Speed: 3
Luck: 1	Luck: 2	Luck: 1

Figure 4.5: Player Stat Distribution Example

This allows for all kind of creatures to be created, be it a large strong monster, small swarm creatures or even disguised statues. The items in turn mainly differ in their type: weapons, food, relics, and their grade. The "random" generation of these assets along with the level generation allows for a much bigger combination of different kinds of challenges and strategies for each level.

4.3.1 Genetic Evolution for Assets Creation

Considering the player, monsters, and items as assets, these can all be generated by a simple genetic evolution algorithm and controlled by their respective fitness function. With this in mind, we implemented an interface for the generation using genetic evolution that could be applied to different assets, converting them to and from bit sequences. Then using the fitness function we influenced the final result based on multiplier parameters for each stat of the object and their combinations. The implemented genetic evolution is the most basic with a simple random chance of mutation in the bits of the new generation. The population used was 100 and the number of best results kept was half of that. The new generation is a copy of the best 50 results that then go through the mutation. After a predetermined number of cycles, only the best one is chosen.

Player Stats Parameterization

The player fitness function takes most in count the balance between the stats, this prevents the creation of a very strong or a very weak character. This is implemented using *Cost*. Every increase in a stat has a cost and the higher the stat the higher the cost of a one value increase, this means an average of all stats would create something like {2,1,2,3,2,2}, and one more focused in one stat would create {4,1,1,2,1,2}.

Since we want to be able to control in some way the final result we have 6 parameters, one for each stat, which were named multipliers. The stats with higher multipliers should be more valued, but not too greatly, allowing the creation of characters by its main attributes. Every stat is represented by a 3-bit sequence in the gene sequence, totaling a 15-bit sequence to represent each individual.

Generating the Game Content

<i>Boss Monster</i> — Health: 6 Attack: 4 Attack Speed: 1 Blast Speed: 1 Speed: 1	<i>Cannon Monster</i> — Health: 3 Attack: 1 Attack Speed: 3 Blast Speed: 3 Speed: 0	<i>Minion Monsters</i> — Health: 1 Attack: 1 Attack Speed: 1 Blast Speed: 1 Speed: 3
---	---	--

Figure 4.6: Monster Desired Stats Example

Monster stats parameterization

The monsters genetic evolution bit sequence is similarly composed of 3-bit values for each stat. The main difference between the generation of monsters and the player is that it has some attributes that function as parameters and attributes at the same time, as well the lack of the multipliers for each parameter. The monster parameters are its size, difficulty, movement state and its vision range. The size of the monster influences the speed and health of the monster in the fitness function. The difficulty increases the limits of the balancing function while the movement state and vision range influence the attack, attack speed, and blast speed.

Item stats parameterization

The items are rather simple, they only contain two stats, a 3-bit value that represents the type of item and another 3 that represent its value. Since the items in itself don't require much control on their produced results the fitness function is only composed and affected by the number of items of each type already created and saved in the map. This is to avoid having the whole map full of the same kind of item. Items can also be Real or Fake. This affects their use but is more conveniently set after the item creation.

4.4 Environment

One of the best ways to highlight, in an explicit way, the differences for each level is the environment or theme of the level. A game in a closed cave dungeon, a labyrinth in the desert or path in

Health Up Item — Real : True Value : 3 Type : 1	Damage Down Item — Real : Fake Value : 3 Type : 3	Health Up Rare Item — Real : True Value : 8 Type : 1
---	---	--

Figure 4.7: Items Desired Stats Example

Generating the Game Content

a forest bring a different perception of the level and when combined properly may help the player visualize the more subtle differences between the levels. To represent this a variable containing the RGB color is created and mapped to a kind of environment that uses the same color. This way, although the kinds of environments are still limited by the ones created, the color in itself helps represent the environment of the level.

Generating the Game Content

Chapter 5

Mapping MIR Features to PCG Parameters

This chapter is the second part of the Methodology and explains the work done in module 1 and 3, with the extracted features and the mapping to the PCG parameters.

Having the desired game content generation methods implemented, all that's left is to make use of their parameters and find a way to influence them with a music track. As discussed previously, Music Information Retrieval (MIR) can be used to extract music features from a music track and although there are a lot of tools that can do this, the focus of study during the literature review was the Spotify API and Essentia. In the end, Essentia was chosen because of its freedom of use for any track and because it simplified the process of reproducing the song in the game.

5.1 Extracted Features

Essentia can be used as a library, but for simplification purposes, we compiled their command line executables that allow us to extract low-level music features for any track into a JSON file.

Most of these features are very technical and would require knowledge in signal analysis to use directly. One way to solve this would be to apply machine learning to these features and label the desired parameters. This would require a big dataset and a complete knowledge of the desired labels, although possible this could create many problems and take too much time, for example, every need of a new kind of result would require the training of a new model and the designation of new labels for the dataset. Essentia already has several trained and available SVM models that can be compiled with Essentia using Gaia, allowing us to run the low-level music features and returning a JSON with high-level results from each of those models.

Using these Models and extracted music features we can try and relate their values to our parameters in the PCG algorithms.

5.2 Influencing the Parameters

In order to better take advantage of the available extracted features, the connection and relation used is a result of trial and error in the creation of a relation between the values available, and the desired effect of the change in the algorithm's parameters. For ease of reading, we will refer to extracted features from music as features and to PCG methods parameters as parameters.

Map size and Structure

For starters, the map size needs to be related to the music, longer music needs bigger maps so having established a lower and upper bound for the map size, the final value is interpolated. Having the map size ready we want to influence the room size and their number. We want faster-paced music to have a lot of rooms and slower paced ones to have fewer rooms albeit bigger ones, this is achieved using the **Beats-per-minute (BPM)** feature to define the **Minimum/Maximum Partition Size** and **Minimum Room Size** parameters.

For the room connection, the **Agent's Turn Probability** parameter is calculated using the **Aggressive** and **Relaxed** high-level features creating simple connections on relaxed music and confusing ones otherwise. The **Centroid Map Probability** parameter is related to the **Sad** feature making the maps less confusing.

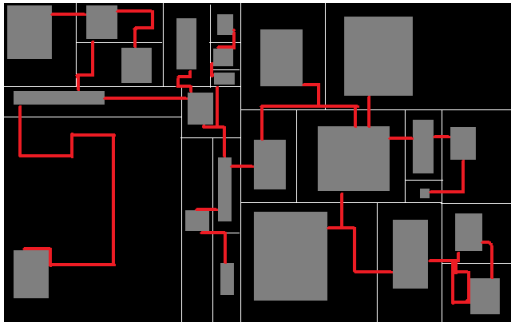


Figure 5.1: Map for Low BPM and Relaxed Music Track

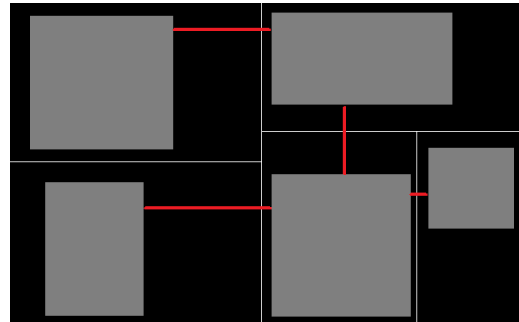


Figure 5.2: Map for High BPM and Aggressive Music Track

Room Selection

For Level Generation we relate the Grammar Evolution fitness function parameters: **Aggressive**, **Confusing**, **Fun**, **Trickery**, **Calmness**, and **Luck** to a combination of the following High-Level values: **Aggressive**, **Party**, **Happy**, **Electronic**, **Aggressive**, **Danceable** and **Relaxed**. Prioritizing in this way the creation of rooms that are categorized as being more compatible with the music. Adding new room types to the grammar only has the cost of creating the relation between that room type and the parameters in Appendix A.

An example of a good level generation for a joyful song would contain a lot of rooms with treasures and a sad one would have more trap rooms and temples for a calmer game-play.

Player Multipliers

The player multiplier attributes are related to its stats and were mapped in two different ways. While the Attack, Armor and Luck multipliers were calculated combining high-level values, the Speed and Life use low-level values, respectively, BPM and average loudness. The Attack Speed multiplier is attributed from the Dance rhythm identified and its values are directly mapped according to its string value. Relaxed and sad songs have a player with more armor or health resulting in a calmer and safer game-play while aggressive and fast songs have a faster and stronger player. The speed and attack speed multipliers result from interval or string multi-case selections but all other formulas used can be found in appendix [A](#).

Monster and Items Parameters

The monster and items parameters aren't actually directly influenced by the music features but by the type of rooms they belong to. For example, Boss Rooms create stronger and bigger monsters while Zoo Rooms create a lot of small and faster monsters with low life points, at the same time, Treasure Rooms will contain more valuable Real Items while trap rooms can contain fake items. Since the type of rooms is influenced directly by the music, the monster and items are indirectly influenced as well. An exception to this is described below.

Using Genre to Influence Parameters

After all, parameters have been calculated, the extracted genre from the Dortmund data set is used to influence either the level generation parameters or directly one of the monster stats multipliers. This is a simple increment or decrement of the already set values that in the case of monster multipliers is zero. This influence is a multi choice increment/decrement and so there are no formulas for it in the appendix.

Conclusion

All of the above-mentioned relations are a combination of additions and multiplications of the extracted values, often achieved by trial and error. A graphic of the relations is shown below([5.3](#)), examples and the final results can be seen in the Results and Experiments([6](#)), and as mentioned the exact formulas used in the relations can be seen in the Appendices([A](#)).

Mapping MIR Features to PCG Parameters

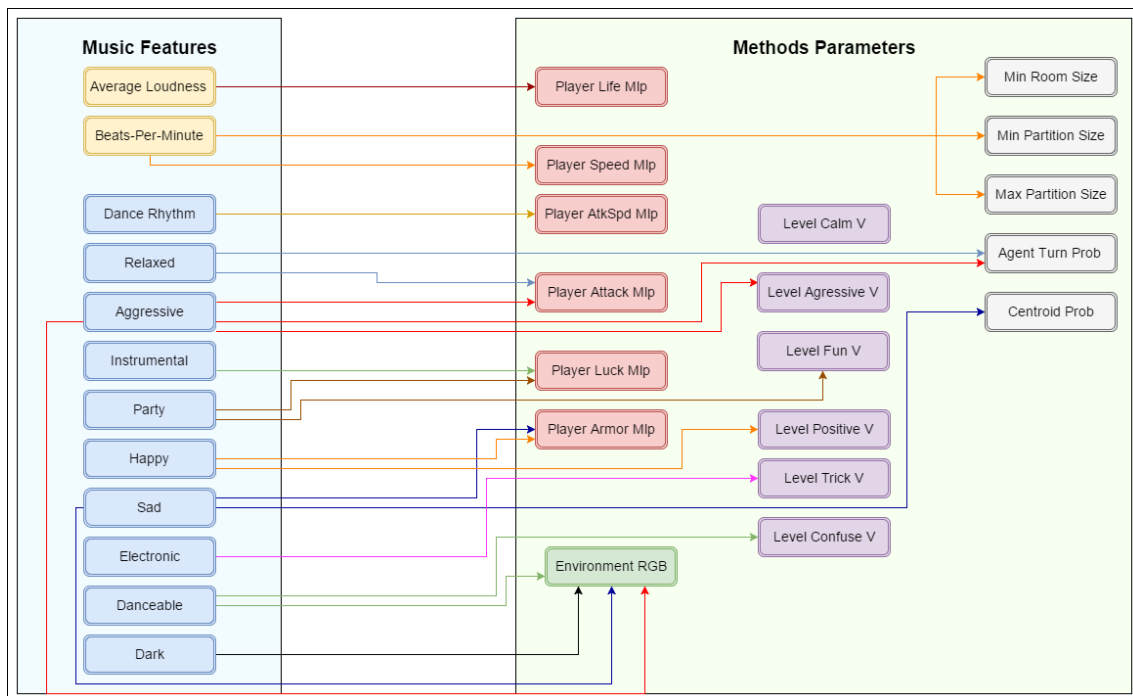


Figure 5.3: Relation between Features and Parameters

Chapter 6

Results and Experiments

This chapter describes the game created as proof of concept, showing some of its mechanics and describing its results for some music tracks. It also contains the results of the experiments with the objective of validating the mapping and the changes according to the music.

6.1 Game Results

6.1.1 Game Concept

As previously described, the developed game is a 2D rogue-like, developed with Unity, that reads the files resulting from the procedural content generation for a music track. It then creates a playable game level where it is possible to analyze the results visually and by experiencing the gameplay alongside the music.

First, the game saves the color generated for the environment and chooses an environment related to that color, for example, choosing an icy labyrinth for the color blue or a fiery one for the color red. Next, it uses the generated map structure to instantiate the floor and the walls, generating the base where the rest of the content will be loaded to while also identifying each of the rooms to its corresponding Room Type.

Afterward, the Player, that is equipped with a ranged blast attack and a shield, has its stats loaded from the generated content and the shield and blast colors are set to the environment color before being positioned in the starting room. All that's left is to load the monsters and items.

There are 5 types of Monsters: Minions, Spiders, Cannons, Statues and the Boss. When the game loads a monster's stats it decides its type and creates it in their corresponding position of the room. Items, in turn, have 8 types: Sword, Boots, Shield, Helmet, Hour Glass, Meat, Carrot and Potato, and are also decided by their loaded values and placed in their rooms.

The game's objective is to explore the dungeon while collecting items and killing monsters, trying to obtain the highest score until either the song ends, or the player's health is reduced to 0.

6.1.2 Results

We will use the game to better observe and to test the result of the content generation while comparing the desired results with the ones obtained for each content. For a more efficient comparison we will compare the results from the following songs:

- 1 Brianstorm - Arctic Monkeys
- 2 Take On Me - a-ha
- 3 Anarchy in the U.K. - Sex Pistols
- 4 Marche Funèbre - Frédéric Chopin

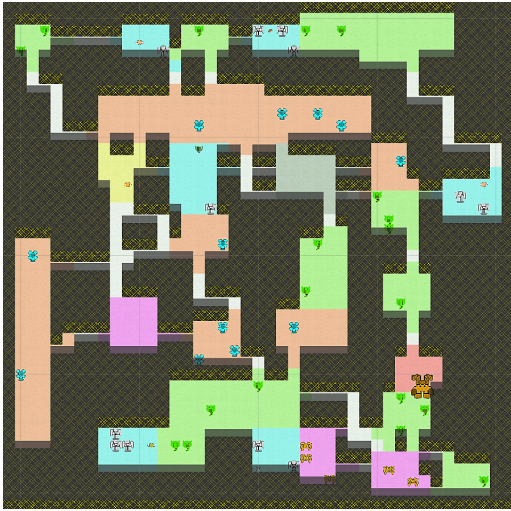
Although different runs of the algorithms create new content every time, the focus of the comparison doesn't change.

Map Structure and Environment Results

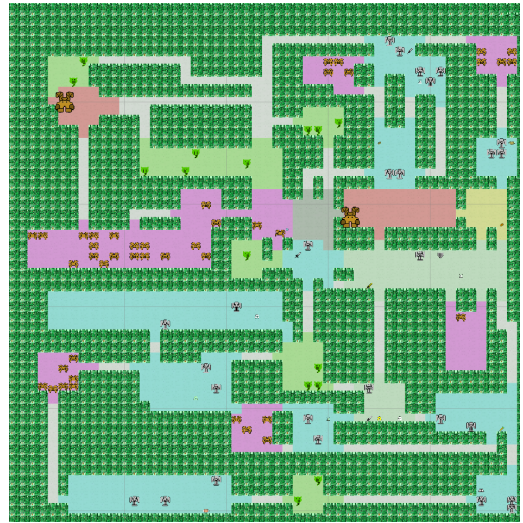
As shown in the previous chapters, the map structure is mostly influenced by the song rhythm, the Beats-per-minute, as well as the aggressiveness, or relaxation, and the Sadness of the song. The main objective behind this is to create simple maps with big rooms and straight corridors for calmer songs, and confusing maps full of small rooms, all connected to one another, for louder and faster songs.

The final result of the generation of the map structure close to what was desired and can be easily noted when comparing very different songs, song 4, which is the calmest and slowest, creates a simple map structure with big rooms while song 1, 2 and 3 create a more complicated one, full of little ones all together. In terms of the environment, we can also observe some good results, although limited to the implementations, the resulting color always seemed somewhat relevant to the audio. In song 4 we can observe an ice environment derived from the color blue; in song 2 we have a forest to represent the green color resulting from the music's relaxing and rhythmic melody. While both song 1 and 3 are aggressive and fast songs, song number 1 is more melodic this is what differentiates the resulting environment between Thunder (yellow) and Fire (red).

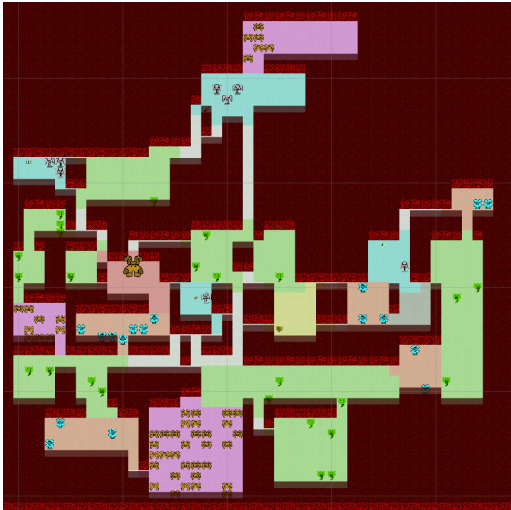
Results and Experiments



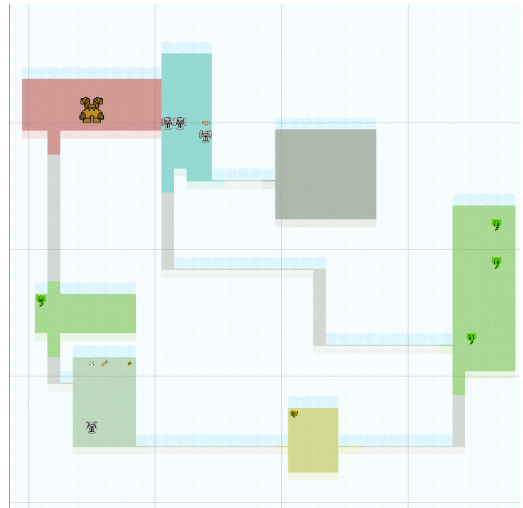
Song 1 - Brianstorm



Song 2 - Take On Me



Song 3 - Anarchy in the U.K.



Song 4 - Marche Funèbre

Figure 6.1: Map Structure and Environments Results

Player Stats Results

In terms of the Player stats, the results are sometimes harder to identify. Having the life and speed be affected directly by the low level features, loudness and BPM respectively, these 2 stats are easier to relate to the song but improved the probability of, for a music track, the player stats multipliers being high in too many parameters, which in turn can result in a well-related stats being left with a low score. This can be observed in song 3, where the life, attack and speed modifiers are all high and some instances of the levels create a player with very high life and attack but low speed, and a low speed in song 3 negatively impacts the game more than the life or attack would.

Results and Experiments

Nonetheless, when the music only sets 1 or 2 parameters high the results are good in comparison with the desired. Song number 4 creates a player with a lot of armor since it's a sad, dark and sad song. Song number 2 creates a player with an average speed and damage, sacrificing armor and life to give it and even higher attack speed. Finally, song 1 creates a fast player with good damage and attack speed.



Song 1 - Brianstorm



Song 2 - Take On Me



Song 3 - Anarchy in the U.K.

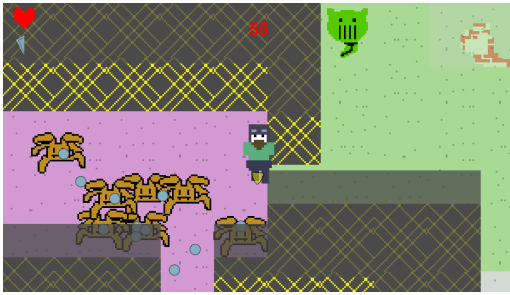


Song 4 - Marche Funèbre

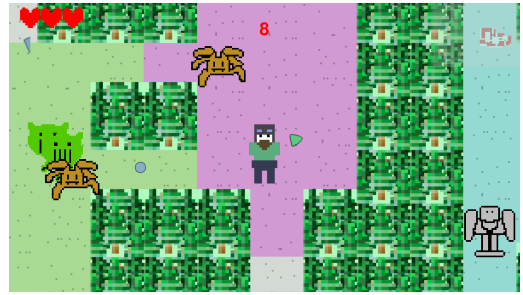
Figure 6.2: Player Stats Results

Level Generation and Assets Results

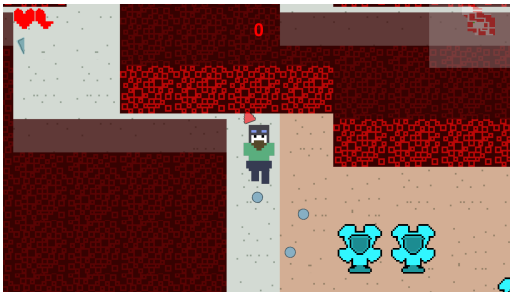
The final result of level generation is the monsters and items placed in each room, so while we also color the floor of each room according to its type, what really matters is the kind of monsters created and their quantity, along with the items each room holds. Song 1 and 3 are good examples where this works well, both of them have maps full of aggressive and high-pressure rooms like cannons and minions that don't let the player play through the level calmly. Song 4 is supposed to be an example representative of calm game-play with many calm rooms like temples where the player that play slowly through them in order to get stronger but while this also happens, the calm nature of the song sometimes creates rooms so big that there are very few extra rooms where the level generation can place them. Song 2, on the other hand, has a lot of variety since the song isn't strongly aggressive nor sad, still, it can be noticed that there are very few temple rooms since the song is not that calm.



Song 1 - Brianstorm



Song 2 - Take On Me



Song 3 - Anarchy in the U.K.



Song 4 - Marche Funèbre

Figure 6.3: Level Generation and Assets Results

6.2 Experiments

In order to validate the results discussed above, we carried out an experiment with 14 participants with the purpose of having them play through the levels for the same 4 songs and get their input on the relation of the content to the music being played. Each participant would play the game with one song of their choosing and afterward 4 times, one for each of the songs presented before, by a random order for a total of 5 runs. A complete script of the experiments can be found in the appendixes.

In order to have a better notion of both the positive and negative relations as well to prevent biased positive scores the first and fourth song had their playing song switched, the experiments were designed as such:

Results and Experiments

	Song used in the PCG	Song playing during run
Run 1	Song 1 - Brianstorm	Song 4 - Marche Funèbre
Run 2	Song 2 - Take On Me	Song 2 - Take On Me
Run 3	Song 3 - Anarchy in the U.K.	Song 3 - Anarchy in the U.K.
Run 4	Song 4 - Marche Funèbre	Song 1 - Brianstorm
Run 5	Participant's choice	Participant's choice

Table 6.1: Runs Description

6.2.1 Experiments Results

The data that resulted from the experiments was the score given by each participant in each run about the 4 relations of the generated content to the song. The scale of the score was $[-2,2]$ where 0 represented lack of relation between the content and the song, 1 represented a good relation and 2 a very good relation and the opposite for the negative values.

First, we analyzed the total average score each participant gave throughout the 5 runs in Figure 6.4. This average would ideally be bigger than 0 and close to 1, since run 1 and 4 should receive negative scores and the other 3 runs should receive positive ones. The results were good, but these alone can't differentiate if it was a result of negative and positive scores or just low scores between 0 and 1 in all songs, so we converted the scores given in runs 1 and 4 by multiplying them by -1 so that we can analyse how well, in comparison to a perfect average of 2 for every participant, the scores were (Fig 6.5). We can see that the averages change a lot but are still rather high with no one below 0, which is a good indicator for the results but we can also see that some previous high average are low which is an indicator of participants giving high scores for every run, so we will analyse in more detail.

Results and Experiments

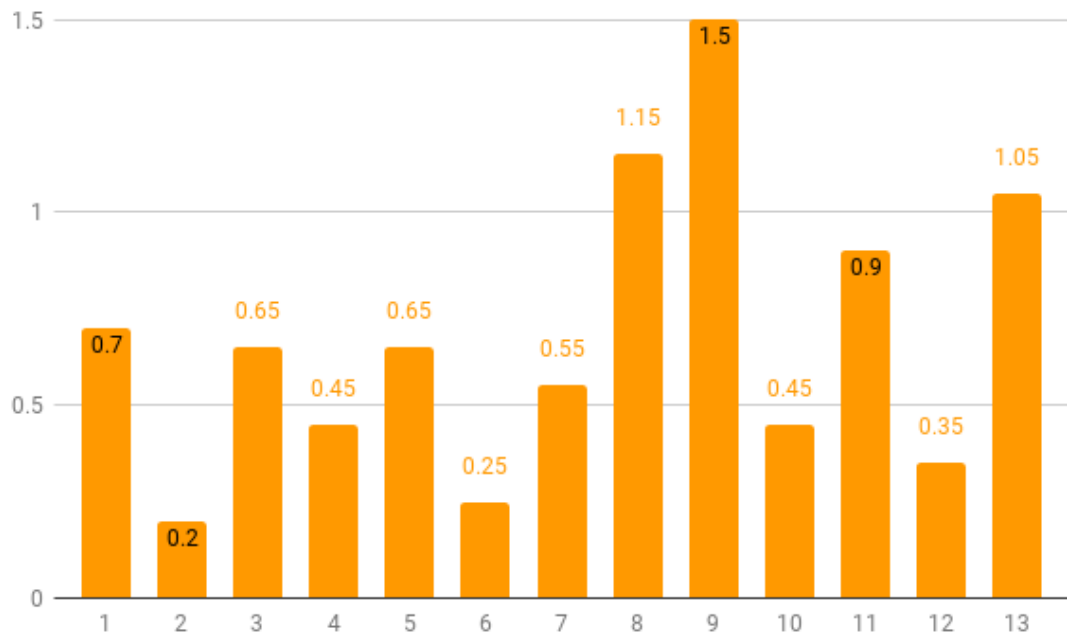


Figure 6.4: Average score given throughout all runs by each participant

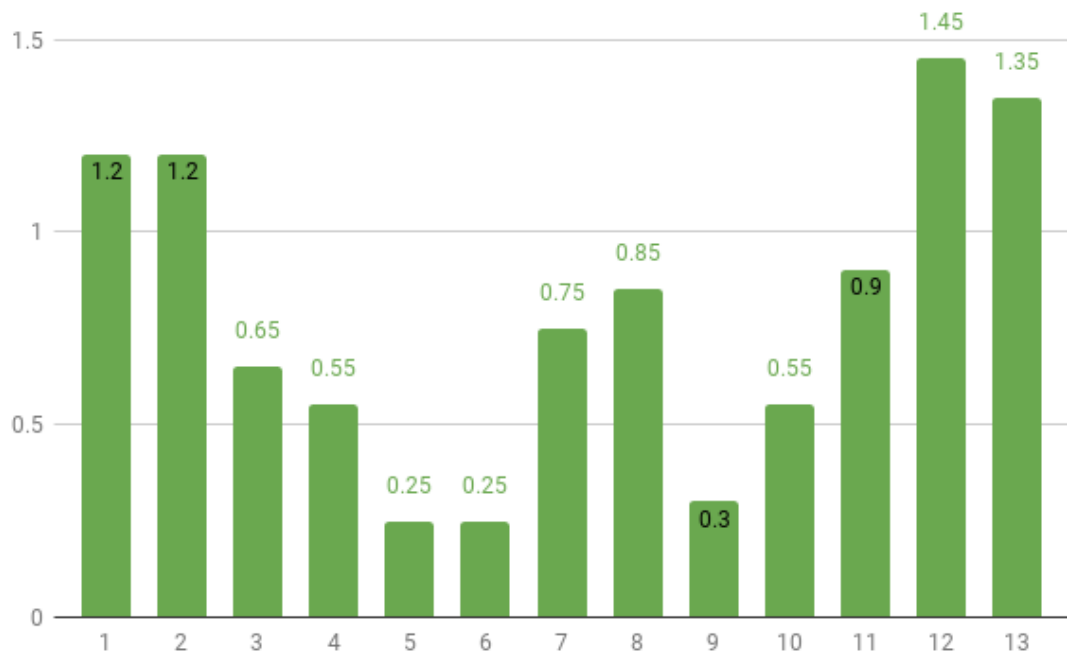


Figure 6.5: Average score given throughout all runs by each participant considering negative scores in fake runs as positive and positives as negative

6.2.1.1 Individual Relation Results

The graphs of the scores average represent the runs from 1 to 5 by order from left to right, while the complete scores graphs represent the vote of the participants for each of the runs, the runs are also represented from left to right.

In terms of the relation of the map structure with the song, in Figure 6.6, most participants believe it to have an effect correctly. Except for some cases of run 1, where most of the agents didn't think it was a bad structure for the music, the overall runs, even run 4 where the relation was also wrong had a good score distribution. This shows us that the song's impact on the map structure seems to be noticeable and correct to the user's expectations of the song.

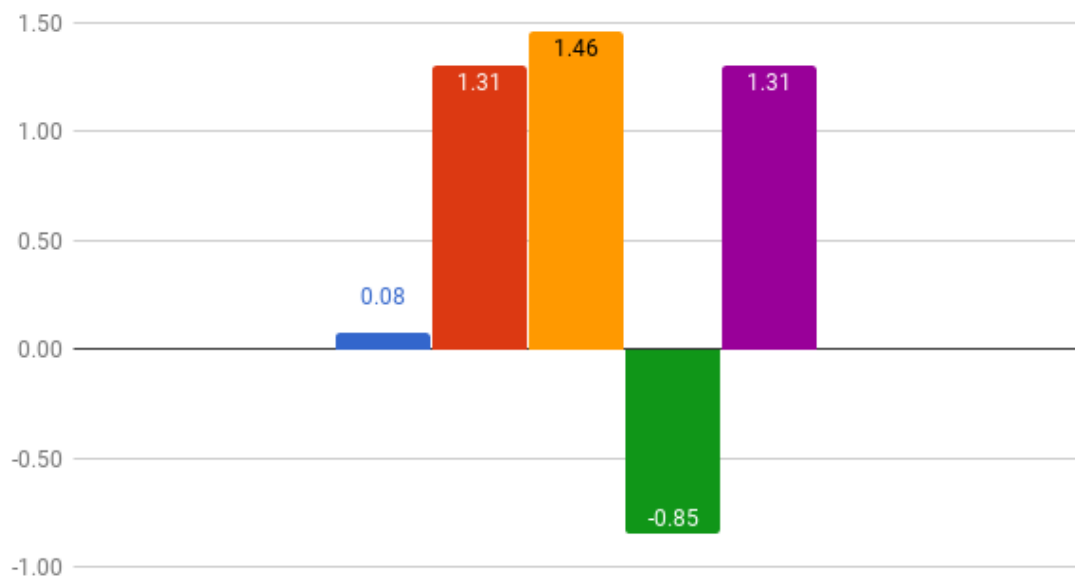
The environment results, in Figure 6.7, are similar to the map relation results but overall weaker and further from the optimal result. This could be a result of the limited environment implemented for the color but is most probably because it's a much more subjective matter.

Player Stats have the same kind of results, Figure 6.8, as the map in the first 4 runs, good values on 2, 3 and 4 with run 1 being not so good, but we see a lower score on run 5, this can be a result of some of the songs chosen to have 3 or more strong attribute multipliers and the ones chosen to focus being different from the expected by the participant.

Finally, the room's composition and distribution were where the results, in Figure 6.9 where worse. Although runs 3,4 and 5 have high scores, and runs 1 and 4 low not only are the latter both bigger than 0, we can also observe a lot of 0 scores. This is due to the fact that more aggressive rooms also have a lot of temples and shops in order to provide items to the player, while in number these leads to calmer rooms only having fewer rooms and the calmer game-play not being entirely recognized.

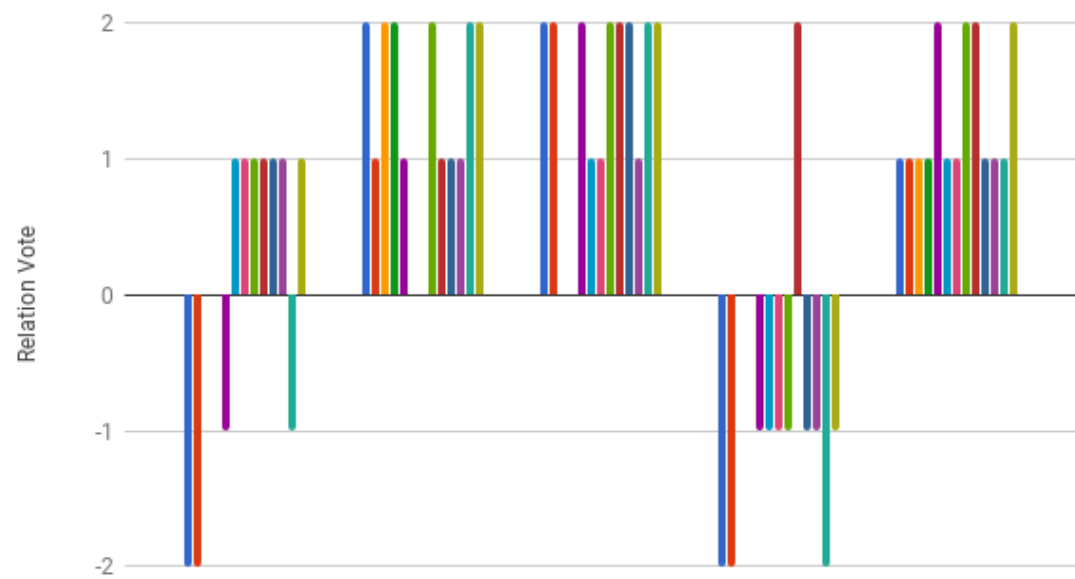
Results and Experiments

Map Structure Avg



Average Scores

Map Structure All

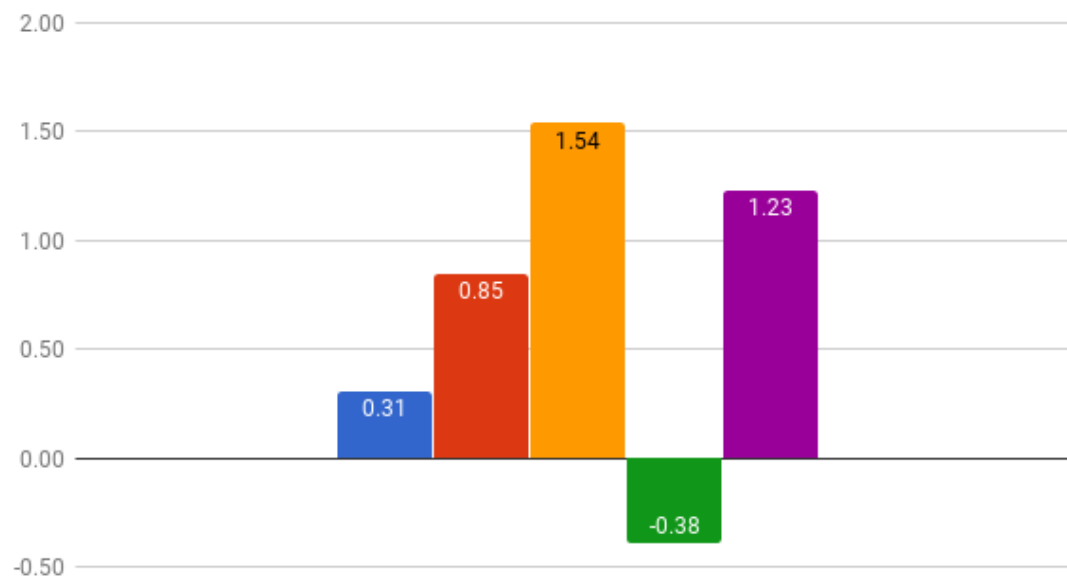


Individual Scores

Figure 6.6: Map Relation Statistics

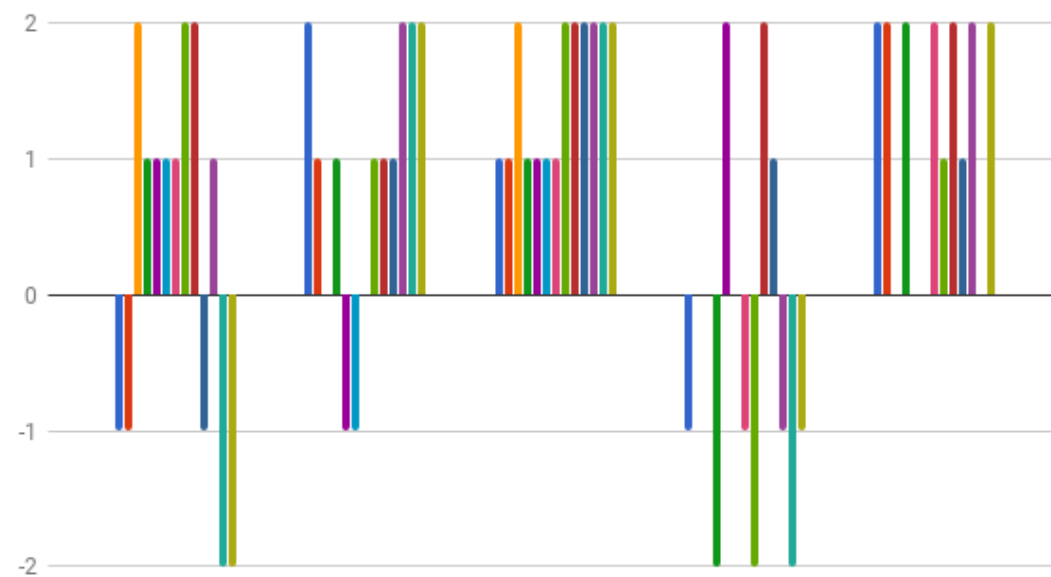
Results and Experiments

Environment Avg



Average Scores

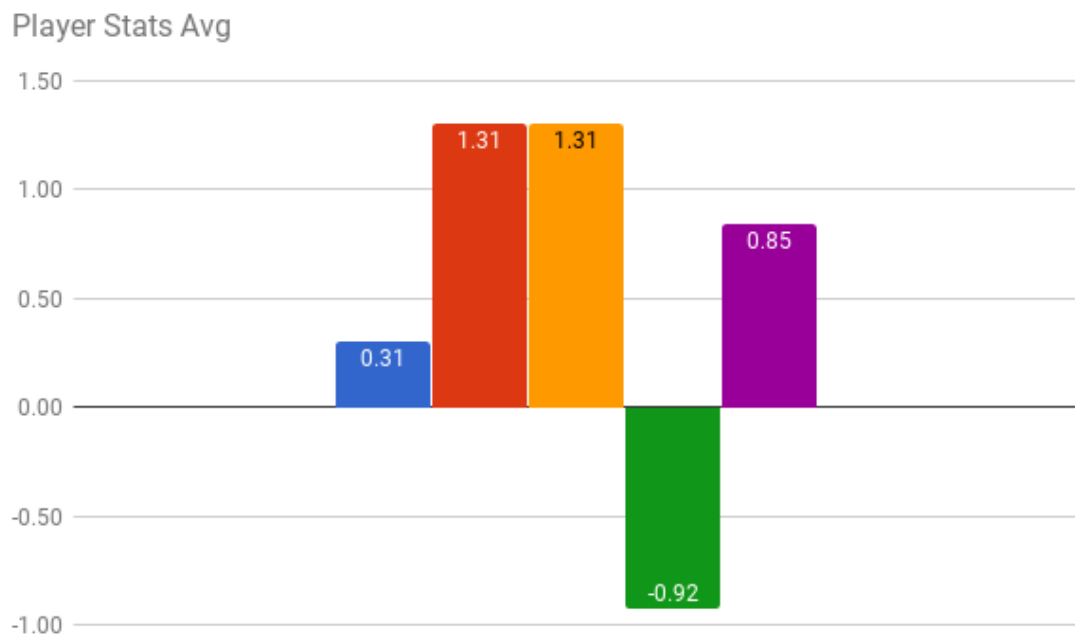
Environment All



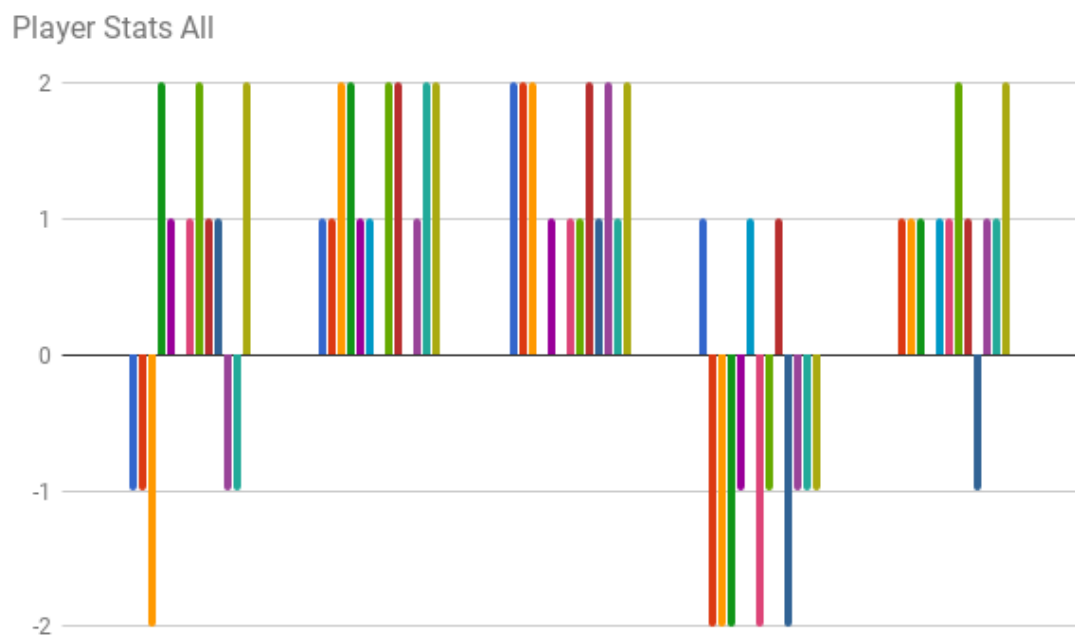
Individual Scores

Figure 6.7: Environment Statistics

Results and Experiments



Average Scores

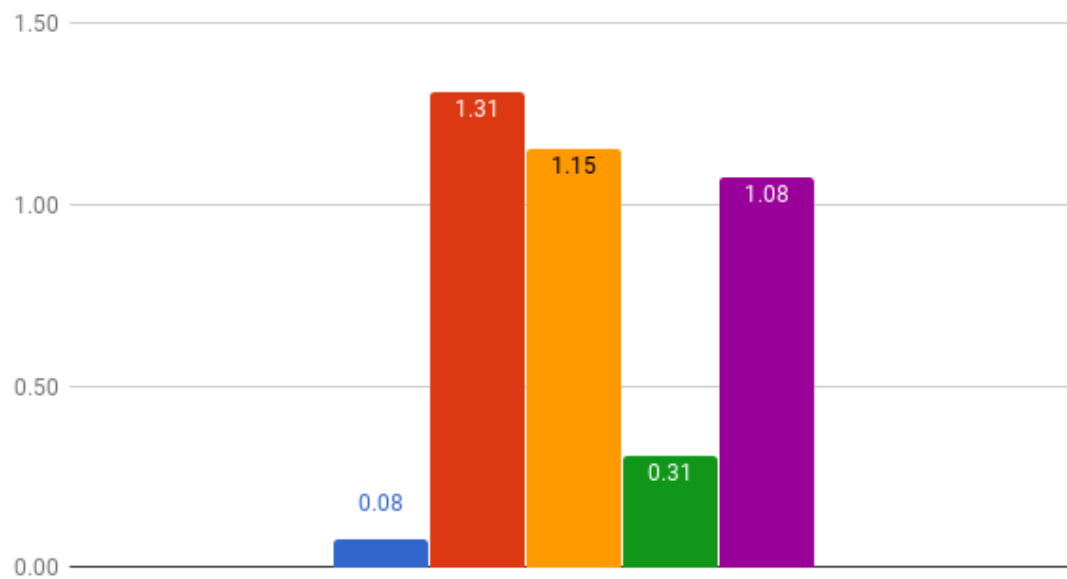


Individual Scores

Figure 6.8: Player Stats Statistics

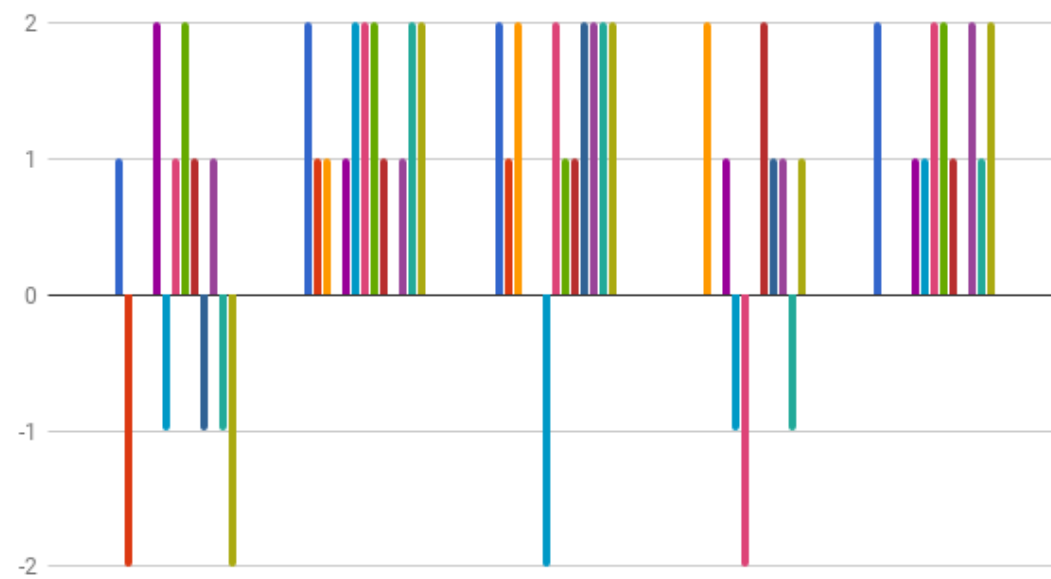
Results and Experiments

Rooms Avg



Average Scores

Rooms All



Individual Scores

Figure 6.9: Level Generation and Assets Statistics

6.2.2 Experiments Final Analysis

After analyzing all of the collected data we can say with confidence that the procedurally generated content is positively affected by the song. Even if we consider that the participants give scores that are too positive, we can see that run 1 and 4 always had an average score much lower and always with individual negative scores present while run 2,3 and 5 have barely any negative scores and mostly have a high average. The main problems were when playing a calm and slow song in a map generated for a fast one, which is the case of run 1, the opposite, run 4, had very good results. This means that when people hear a fast song they are expecting fast and aggressive game-play and notice if it doesn't happen, but the opposite seems to need more work since participants always seemed to find something in the fast game-play that could be connected to the slow and calm song.

Another aspect which seemed to not work so well was the relation of the song to the rooms and, monsters and items distribution. This is probably due to the low variety of room types available as well as the number of necessary rooms (the ones which are always generated) to be a major part of the rooms available for some generations of calm and slow songs.

In conclusion, the results of the experiments, although not optimal, are good and can validate that the final results of the PCG are in fact positively affected by the music.

Results and Experiments

Chapter 7

Conclusion

The purpose of this thesis was to implement procedural content generation (PCG) methods that could be correctly influenced by features extracted from any music track. This purpose was achieved with good, albeit improvable results. Considering the limits of implemented assets for content in the rooms, monster, and items, and the mathematical formulas created, the results obtained can be easily differentiated for different songs and create some levels with different gameplay. A bigger variety in assets and better calculated formulas could help in the generation of more accurate and diverse content but the overall result and influence would be the same.

7.1 Future Work

The future work in this area, should be in the refinement of all mathematical formulas, along with the improvement and creation of new PCG methods or implementation for new types of content. The mapping between the features and parameters could also be refined.

Different PCG methods could create more variation with new content or even with different options for the same content, this would help since we could better differentiate similar songs. It could also solve the problem of some iterations having worse results. The refinement of the mathematical formulas in the fitness function and the mapping would ensure that the defined parameters are properly initialized and valued.

Apart from this, the use of data mining models with labels specific for the desired intended results could also be used albeit only as a specific tool. A different approach on the type of game and game content created could also be explored along with a possible initial calibration for each user like it is usually done with biofeedback sensors.

Conclusion

References

- [AN] Calvin Ashmore and Michael Nitsche. The Quest in a Generated World.
- [BM16] Daniel Bengtsson and Johan Melin. *Constrained procedural floor plan generation for game environments*. PhD thesis, Blekinge Institute of Technology, Department of Creative Technologies, 2016.
- [Bom16] Henri Bomström. *The application of procedural content generation in video game design*. PhD thesis, 2016.
- [CAMY14] W Cachia, L Aquilina, H P Martínez, and G N Yannakakis. Procedural generation of music-guided weapons, 2014.
- [Col09] Karen Collins. An Introduction to Procedural Music in Video Games. *Contemporary Music Review*, 28(1):5–15, 2009.
- [Dor10] Joris Dormans. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 1:1—1:8, New York, NY, USA, 2010. ACM.
- [DT] Steve Dahlskog and Julian Togelius. Procedural Content Generation Using Patterns as Objectives.
- [EL13] Oliver Eriksson and Philip Lindau. Evaluating an adaptive music system in an adventure game environment, 2013.
- [Enta] Entertainment Software Associaton (ESA). U.S. Video Game Industry Generates \$30.4 Billion in Revenue for 2016 - The Entertainment Software Association.
- [Entb] Entertainment Software Associaton(ESA). WHAT’S INSIDE.
- [FO15] Nuno Filipe and Bernardino Oliveira. *Music-Based Procedural Content Generation for Games*. PhD thesis, 2015.
- [HGS09] Erin Jonathan Hastings, Ratan K Guha, and Kenneth O Stanley. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263, 2009.
- [HMT13] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: {A} survey. *Tomccap*, 9(1):1, 2013.
- [JSL⁺12] Annika Jordan, Dimitri Scheftelowitsch, Jan Lahni, Jannic Hartwecker, Matthias Kuchem, Mirko Walter-Huber, Nils Vortmeier, Tim Delbrügger, Ümit Güler, Igor

REFERENCES

- Vatolkin, and Mike Preuss. BeatTheBeat music-based procedural content generation in a mobile game. *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*, pages 320–327, 2012.
- [LYT14] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Computational Game Creativity Games : the Killer App. *Proceedings of the 5th International Conference on Computational Creativity*, pages 46–53, 2014.
- [Per04] Ken Perlin. Implementing improved perlin noise. *GPU Gems*, pages 73–85, 2004.
- [SGU14] Markus Schedl, Emilia Gomez, and Julian Urbano. Music Information Retrieval: Recent Developments and Applications. 2014.
- [Sil14] Diogo Albuquerque Valente Silva. *Development of artificial intelligence systems for stealth games based on the Monte Carlo Method Recurso eletrônico*. PhD thesis, Porto:, 2014.
- [SSG⁺17] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural Content Generation via Machine Learning (PCGML). *CoRR*, abs/1702.0, 2017.
- [STN16] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.
- [Stu14] Bob L Sturm. The State of the Art Ten Years After a State of the Art: Future Research in Music Information Retrieval. *Journal of New Music Research*, 43(2):147–172, 2014.
- [SYT] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards Automatic Personalized Content Generation for Platform Games.
- [TCL⁺13] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O. Stanley. Procedural Content Generation : Goals, Challenges and Actionable Steps. *Artificial and Computational Intelligence in Games*, 6:61–75, 2013.
- [TJH] Julian Togelius, Tróndur Justinussen, and Anders Hartzen. Compositional procedural content generation.
- [TSD] Julian Togelius, Noor Shaker, and Joris Dormans. Grammars and L-systems with applications to vegetation and levels.
- [TYSB10] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-Based Procedural Content Generation. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna I Esparcia-Alcazar, Chi-Keong Goh, Juan J Merelo, Ferrante Neri, Mike Preuß, Julian Togelius, and Georgios N Yannakakis, editors, *Applications of Evolutionary Computation: EvoApplications 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoS-TOC, Istanbul, Turkey, April 7-9, 2010, Proceedings, Part I*, pages 141–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [ZDG14] J R Zapata, M E P Davies, and E Gómez. Multi-Feature Beat Tracking. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(4):816–825, 2014.

Appendix A

Fitness and Mapping Functions

Fitness Functions: The functions that follow are the fitness functions used for Player, Monster and Items, as well as the ones for the level generation evolutionary algorithms. Parameter variables are represented with a P in the end, temporary variables have a T at the end and class variables have a M at the end.

Player

$$scoreT = (lifeM - 1)^2 + (attackM - 1)^2 + (armorM)^2 + (speedM - 1)^2 + (luckM + 1)^2 \quad (A.1) \\ + (attackSpeedM - 1)^2$$

$$fitnessValueM = 1 - \frac{scoreT^2}{16} + 0.1 * (playerSpeedMultiplierP - 1) * (speedM - 1) \quad (A.2) \\ + 0.1 * (playerLifeMultiplierP - 1) * (lifeM - 1) + 0.1 * (playerAttackMultiplierP - 1) * (attackM - 1) \\ + 0.1 * (playerArmorMultiplierP) * armorM + 0.1 * (playerLuckMultiplierP) * luck \\ + 0.1 * (playerAttackSpeedMultiplierP - 1) * (attackSpeedM - 1)$$

Monster

$$scoreT = (lifeM - 1)^2 + (attackM - 1)^2 + (blasSpeedM)^2 + (speedM - 1)^2 \quad (A.3) \\ + (attackSpeedM - 1)^2$$

$$fitnessValueM = 1 - ((\frac{scoreT}{difM} - 0.8) * 10)^2 + 0.1 * (monsterSpeedMultiplierP - 1) * (speedM - 1) \quad (A.4) \\ + 0.1 * (monsterLifeMultiplierP - 1) * (lifeM - 1) + 0.1 * (monsterAttackMultiplierP - 1) * (attackM - 1)$$

Fitness and Mapping Functions

$$\begin{aligned}
&+0.1 * (\text{monsterBlastSpeedMultiplierP}) * \text{blastSpeedrM} \\
&+0.1 * (\text{monsterAttackSpeedMultiplierP} - 1) * (\text{attackSpeedM} - 1) \\
&+ \text{sizeM} * (\text{lifeM} - 2)^2 * 0.1
\end{aligned}$$

Item

The item function takes is different for some kind of items, but the difference is only to penalize the repetition so we will present the regular one. The variable *valueM* is either 1 for normal items or 2 for rare items.

$$\text{fitnessValueM} = 1 - ((\frac{\text{effectM}}{\text{valueM}} - 0.8) * 10)^2 \quad (\text{A.5})$$

Level Generation

$$\begin{aligned}
\text{fitnessScoreM} = 1 - &(\frac{\text{generatedRoomCountM}}{\frac{\text{roomCountM} * 3}{4}} - 1)^2 \quad (\text{A.6}) \\
&+ (1 - \frac{\text{aggressiveM}^2}{\text{roomCountM}}) * 0.17 * \text{levelAgressiveP} \\
&+ (1 - \frac{\text{confuseM}^2}{\text{roomCountM}}) * 0.17 * \text{levelConfuseP} \\
&+ (1 - \frac{\text{funM}^2}{\text{roomCountM}}) * 0.17 * \text{levelFunP} \\
&+ (1 - \frac{\text{trickM}^2}{\text{roomCountM}}) * 0.17 * \text{levelTrickP} \\
&+ (1 - \frac{\text{calmM}^2}{\text{roomCountM}}) * 0.17 * \text{levelCalmP} \\
&+ (1 - \frac{\text{positivekM}^2}{\text{roomCountM}}) * 0.17 * \text{levelPositiveP}
\end{aligned}$$

Mapping Functions: The functions below, are the ones used in the mapping process but don't include all of the processing done in the same, as described in chapter 5. PCG parameters are the variables with a *P* in the end, and the music extracted features are the ones with *F* in the end of the name.

$$\text{mapSizeP} = (\frac{(\text{fLength} - 300)}{240} + 0.3)^2 + 30 \quad (\text{A.7})$$

$$\text{minRoomSizeP} = (1 - \frac{\text{beatsPerMinuteF} - 80}{120})^2 + 3 \quad (\text{A.8})$$

$$\text{minPartitionSizeP} = \frac{\text{minRoomSizeP}^2}{2} \quad (\text{A.9})$$

Fitness and Mapping Functions

$$\mathit{maxPartitionSizeP} = \mathit{minPartitionSizeP} * 2 + 1 \quad (\text{A.10})$$

$$\mathit{agentTurnProbP} = \mathit{aggressiveF} * 25 - \mathit{relaxedF} * 20 \quad (\text{A.11})$$

$$\mathit{CentroidProbP} = \mathit{sadF} * 100 \quad (\text{A.12})$$

$$\mathit{rgb}[0] = \mathit{aggressiveF} * ((1 - \mathit{darkF}) * 0.2 + 0.8) \quad (\text{A.13})$$

$$\mathit{rgb}[1] = \mathit{danceableF} * ((1 - \mathit{darkF}) * 0.2 + 0.8) \quad (\text{A.14})$$

$$\mathit{rgb}[2] = \mathit{sadF} * ((1 - \mathit{darkF}) * 0.2 + 0.8)) \quad (\text{A.15})$$

$$\mathit{playerLifeMultiplierP} = \mathit{loudnessF}^2 * 4 \quad (\text{A.16})$$

$$\mathit{playerAttackMultiplierP} = (\mathit{aggressiveF} - (\mathit{relaxedF} * 0.3))^2 * 4 \quad (\text{A.17})$$

$$\mathit{playerArmorMultiplierP} = (\mathit{sadF} - \mathit{happyF}^2 + 0.2)^2 * 4 \quad (\text{A.18})$$

$$\mathit{playerLuckMultiplierP} = (\mathit{instrumentalF} * 0.4 + \mathit{partyF} * 0.4 + 0.2)^2 * 4 \quad (\text{A.19})$$

$$\mathit{levelAggressiveP} = \mathit{aggressiveF} - \mathit{relaxedF} * 0.2 \quad (\text{A.20})$$

$$\mathit{levelFunP} = \mathit{partyF} \quad (\text{A.21})$$

$$\mathit{levelPositiveP} = \mathit{happyF} \quad (\text{A.22})$$

$$\mathit{levelTrickP} = \mathit{electronicF} \quad (\text{A.23})$$

$$\mathit{levelCalmP} = \mathit{relaxedF} - 0.2 * \mathit{aggressiveF} \quad (\text{A.24})$$

$$\mathit{levelConfuseP} = \mathit{danceableF} \quad (\text{A.25})$$

Fitness and Mapping Functions

Appendix B

Experiments Script

Project Introduction: The objective of this project is to create different levels for each song by creating game content procedurally, every time the game loads, that is influenced by the music.

Experiment: In this experiment you will play 5 levels for 5 songs, 4 of them are predefined and you can chose the 5th. The order of each run is random and after playing all runs you will be asked to evaluate the relation of the generated content with the music you hear during the game, the scale is [-2,2] where 1 and 2 mean a good or very good relation and -1 and -2 the respective opposites, if you can't evaluate it you can answer zero. If you don't remember or if you loose too quickly you can replay the level to get a better feel of the levels components.

The Game: The game is not the focus of the project so the graphics and mechanics can have some bugs. The objective is to kill all monsters and catch all treasures to try and have a bigger final score. There is a minimap on the top right and your life and shield is displayed on the top left. You can move the player using WASD keys, left clicks are used to fire and holding the right mouse button activates the shield.

Appendix C

Features and Parameter Values

Name	funeral.ogg	01-Brianstorm.ogg	takeonme.ogg	Anarchy.ogg
Aggressive	0.019153118	0.95595646	0.019576311	0.99995255
AvgLoudness	0.73357064	0.94885933	0.860991	0.94586086
BpmG	89.869354	164.93948	166.6205	133.82442
Danceable	0.0033090115	0.93834066	0.9397618	0.0
Dark	0.27391356	0.8792585	0.024171293	0.043872356
Electronic	0.1325261	0.19775265	0.090806365	0.36431146
Happy	0.23369092	0.3482933	0.80697596	0.9173547
Instrumental	0.99999994	0.01008898	0.024024427	0.0017903447
Length	151.59583	170.39673	227.20581	212.48436
Party	0.0023611188	0.71947753	0.4590966	0.8313982
Relaxed	0.97089165	0.09041214	0.093225956	0.045971394
RhythmDance	Tango	Jive	VienneseWaltz	ChaChaCha
Sad	0.9591051	0.020888686	0.25074565	0.03391403
RoomSimpleConnections	true	true	true	true
MapSize	39	41	49	47
MinPartitionSize	8	4	4	4
MaxPartitionSize	17	9	9	9
MinRoomSize	4	3	3	3
Centroid	95	2	25	3
AgentTurnProb	1	22	1	24
RGB	5 1 231	201 197 4	5 238 64	253 0 9
PLifeM	2	3	2	3
PAttackM	0	3	0	3
PArmorM	4	0	0	0
PSpeedM	1	2	2	2
PLuckM	1	0	0	1
PAtkSpeedM	1	1	2	3
LevelAgrV	0.019153118	0.95595646	0.019576311	0.99995255
LevelCalmV	0.9808469	0.04404354	0.9804237	4.7445297E-5
LevelConfuseV	0.0033090115	0.93834066	0.9397618	0.0
LevelFunV	0.0023611188	0.71947753	0.4590966	0.8313982
LevelPostiveV	0.23369092	0.3482933	0.80697596	0.9173547
LevelTrickV	0.1325261	0.19775265	0.090806365	0.36431146