UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Procedural Generation of 2D Games

Pedro Alexandre Morais de Andrade

Mestrado em Engenharia Informática
Especialização em Interação e Conhecimento

Dissertação orientada por:
Prof.º Doutor Luís Manuel Ferreira Fernandes Moniz

2020

Acknowledgement

The development of this master's dissertation relied on important incentives and supports that made everything easier, to which I will forever be grateful.

To Professor Doctor Luís Manuel Ferreira Fernandes Moniz, for his guidance and support that contributed a lot for the completion of this thesis, for his opinions, criticism, availability to answer any questions and collaboration in the search for solutions to all the problems that were arising throughout the work.

To the MAS-BioISI group, for having received and integrated me very well, which resulted in an incredible work and companionship environment, where everyone voluntarily helped each other.

To my friends, André Veríssimo, Tiago Correia, Ricardo Santos, João Gouveia, Pedro Serralheiro, Nuno Rodrigues, Sandro Correia, André Penas, Beatriz Damasceno, Mariana Oliveira, Filipa Rodrigues, João Pereira, Jorge Loureiro and my cousins Ramiro Morais and Henrique Morais, who contributed not only to the performance of important tests for this project, but also were always by my side during this phase of hard work, giving me strength and support in the most difficult moments.

Finally, I would like to express my special thanks to my family, especially my parents and uncle, for their unconditional support, encouragement, friendship, patience and total help in overcoming various problems that have arisen. I dedicate all this work to them!

Abstract

The main objective of this project is to develop a procedural generator of levels for 2D games, with the capacity of adapting the difficulty of the levels to a player's skill in a specific game. Thus, in order to implement a procedural generator with the previously mentioned features, we intend to combine two techniques: procedural content generation and dynamic difficulty adjustment. Procedural content generation is a technique which has the purpose of creating content for a game. The game content generated can be anything related to the video-game in question (e.g. characters, items, terrain, levels). Dynamic difficulty adjustment is the name of the technique used to make adjustments to the game's difficulty, depending on the overall progress of a player in a particular level.

The procedural content generator developed uses the idea of rhythms of a level as its basis (Smith et al., 2009). This approach consists on describing a level as a sequence of actions that must be done to successfully conclude it. Our methodology differs from the classical rhythm-based approach, because instead of a sequence of single actions we represent a level as a sequence of classes of actions. A class of actions is a group of actions that have the same assumed difficulty, which is defined by a mechanic description (what keys to press to perform an action). For the generation of these sequences of classes of actions, it is used a genetic algorithm whose fitness function is able to evaluate the difficulty of a sequence, which allows it to generate rhythms for diverse levels with different difficulties. After the rhythm generation process, the resulting sequences of classes of actions are going to be passed as a parameters to a geometry generator, that is going to associate each of the class of actions to a level chunk, having, in the end, a new playable level (a group of level chunks).

This approach was then tested with different games to demonstrate the generator's capacity to generalize and, to prove our definitions of difficulty, we made some tests using search algorithms and human players to make this evaluation.

Keywords: Procedural content generation, Dynamic difficulty adjustment, 2D games, Genetic algorithm, Artificial intelligence

Resumo alargado

Este projeto tem como principal objetivo o desenvolvimento de um gerador procedimental de níveis para jogos 2D, com a capacidade de adaptar a dificuldade dos níveis gerados às habilidades demonstradas por um jogador num determinado jogo.

Geração procedimental é uma técnica que tem o propósito de criar conteúdo (por exemplo: itens, personagens, paisagens, níveis, etc.) para um vídeo-jogo, tendo pouca intervenção humana e alguma aleatoriedade, em combinação com restrições ou parâmetros, para definir como deve ser realizada a geração de um determinado conteúdo de jogo. Ajustamento dinâmico de dificuldade é o nome da técnica usada para ajustar a dificuldade de um jogo, dependendo do progresso geral de um jogador num nível em particular.

Nesta dissertação, primeiramente, foi feita uma investigação e análise sobre métodos de geração procedimental de conteúdo, suas taxonomias e métodos de ajustamento dinâmico de dificuldade, de forma a ser feita uma introdução aos principais temas de debate e, consequentemente, conseguirmos fazer uma avaliação ponderada das melhores opções que queremos utilizar no desenvolvimento deste projecto.

O foco do projeto é combinar estes dois tipos de métodos, de forma a implementar um gerador que providencie uma geração procedimental de níveis que consiga adaptar a dificuldade de cada geração dependendo das capacidades do jogador, ou seja, que permita fazer um ajustamento dinâmico da dificuldade do jogo conforme o progresso geral de um jogador num nível em particular. Como a geração procedimental de conteúdo não é, geralmente, considerada uma opção viável para jogos comerciais, um objetivo secundário deste projeto é também poder contribuir, de alguma maneira, para que esta área de investigação possa proporcionar melhores opções para esta técnica de geração de conteúdos.

O gerador procedimental de conteúdo desenvolvido tem como base os ritmos de um nível, que vai dividir o desafio de geração em partes mais pequenas. Esta abordagem consiste na ideia de descrever, ou caracterizar, um nível como uma sequência de ações que têm de ser realizadas para que a sua conclusão (do nível) seja bem sucedida. A abordagem que seguimos deriva dessa ideia, no sentido que, em vez de ser uma sequência de ações, vai ser uma sequência de classes de ações. Isto devido ao facto de querermos generalizar esta metodologia para jogos de diversos géneros que, consequentemente, terão ações diferentes uns dos outros. Entende-se como uma classe de acções, um grupo de ações que vão ter a mesma dificuldade assumida, que vai ser caracterizada por uma descrição mecânica (que teclas pressionar para executar uma ação). Esta forma de organizar as ações permite-nos generalizar, pois, como referi, nem todos os jogos têm as mesmas ações, mas em todos os jogos conseguimos fazer a distinção, ou comparação, entre a dificuldade de cada ação, podendo assim, colocar as ações que têm a mesma dificuldade assumida na mesma classe.

Para a geração destas sequências de classes de ações foi utilizado um algoritmo genético cuja função de *fitness* avalia a dificuldade de uma sequência de classes de ações, o que permite gerar ritmos para diversos níveis de várias dificuldades. Uma das principais vantagens deste algoritmo é que possibilita gerar ritmos distintos para a mesma dificuldade, o que nos ajuda a ter uma grande diversidade na geração dos níveis. Após o processo de geração de ritmos, a sequência de classes de ações obtida vai ser passada como parâmetro a um gerador de geometrias que, por sua vez, vai associar cada uma das classes de ações a um determinado contéudo ou pedaço de nível, tendo no final da execução um novo nível jogável (grupo de pedaços de nível resultante da sequência de classes de ações). A associação funciona da seguinte forma: primeiro o gerador identifica qual a classe da ação, para se perceber quais as ações que fazem parte dessa classe, depois aleatoriamente escolhe uma dessas ações, com o propósito de gerar terreno ou conteúdo de jogo apropriado. Por exemplo, se a ação for saltar então vai gerar terreno ou conteúdo que obriga o jogador a ter de saltar, caso queira passar aquele pedaço de nível com sucesso.

No final, esta abordagem foi testada para quatro jogos diferentes (*Super Mario*, *Cave Copter*, *Rambit* e *Swervin Mervin*), de forma a demonstrar a capacidade de generalização do gerador para vários géneros de jogos, que foi também um dos principais objectivos deste projeto. Os géneros testados são designados: plataforma, corrida e atirador horizontal.

Para comprovar a nossa definição de dificuldade, foram feitos testes com algoritmos de pesquisa e com pessoas, de forma a avaliarmos a dificuldade e a qualidade dos níveis gerados. No caso dos testes com algoritmos de pesquisa, foi utilizado o A* como *benchmark*, com o propósito de o executar 100 vezes em cada uma das dificuldades do jogo do *Super*

Mario, de forma a podermos observar o número de níveis completos pelo algoritmo para cada uma das dificuldades e concluir se estes (níveis) estavam a ser corretamente adaptados. Os testes com as pessoas passaram por as colocar a jogar 3 níveis de cada jogo para depois disso colocarem os níveis por ordem do mais fácil para o mais difícil. No final, foi feita uma comparação entre essa ordem sugerida pelos jogadores e a nossa ordem, que assumimos que seja a correta, tendo como base a nossa definição de dificuldade.

Finalmente foi feita uma avaliação crítica dos resultados obtidos nos testes, que nos leva a concluir que implementamos com sucesso um gerador de nível capaz de adaptar os níveis de acordo com o valor de uma dificuldade alvo e também confirmou que os nossos conceitos de dificuldade foram bem generalizados para a maioria dos jogadores, pois todos conseguiram classificar os níveis dos testes de acordo com a sua dificuldade, a juntar a isso, as performances dos jogadores foram piorando à medida que a dificuldade dos níveis foi aumentando. Assim, podemos ficar com uma ideia mais concisa sobre trabalhos futuros nesta área de investigação (geração procedimental) e, mais especificamente, inovações e reparos a fazer na nossa abordagem.

Palavras-chave: Geração procedimental de conteúdo, Ajustamento dinâmico de dificuldade, Jogos 2D, Algoritmo genético, Inteligência artificial

Contents

List of Figures

List of Tables

Glossary

AI Artificial intelligence. 1, 3, 8, 15, 17–19, 33, 49, 53, 57

ANN Artificial neural networks. 20

classes of actions Classes of actions are used to organize all the possible actions of a 2D game by difficulty of execution, in order to help the generalization of our approach to different genres.. ii, xi, 1–3, 5–7, 31–43, 46–48, 50, 53, 55, 57, 58, 66, 75, 76, 86–88

DDA Dynamic difficulty adjustment. xi, 4, 10, 18–21, 31, 49, 52–55, 76, 77, 87, 88

EDPCG Experience-driven procedural content generation. 14

LBPCG Learning-based procedural content generation. 15

NPC Non-player characters. 20

PCG Procedural content generation. xi, 1, 8–12, 14, 15

rhythm Rhythms are a sequence of actions that are used to structure a level in rhythm-based approaches for procedural level generation. ii, 1, 2, 4, 6, 15, 16, 30–32, 34–40, 46–48, 50, 51, 54, 58, 74, 86, 88

rhythm-based approach It is an approach for level generation that uses what is called the rhythm of a level, which is going to divide the generation problem into smaller pieces that are usually found by using pre-made parts of game tiles. (Gillian, 2011). ii, 1–6, 30–32, 34, 55, 58, 86–88

SBPCG Search-based procedural content generation. ix, 12, 13

SOS Self-organizing system. 20

sprites A sprite is a two-dimensional bitmap that is integrated into a larger scene, most often in a 2D video game.. 15, 44–48, 75

UCT Upper confidence bound trees. 20

Chapter 1

Introduction

Procedural generation is the process of dynamically creating content, thus being perceived as an artificial intelligence (AI) task. It allows the development of less predictable gameplay, due to the induced randomness, improving the game's value and also creating a game's world within a short period of time. AI in gaming has increased over the years and procedural generation has been gaining momentum, having the potential of being a promising tool in the future of games (On et al., 2017).

Although there are prominent advantages of the applications of procedural content generation in 2D games, there are still some limitations when trying to procedurally generate levels for different genres, since most procedural level generation approaches are too specific for a particular game or genre. This happens because not all games will have the same playstyles, making it more complex to have a procedural content generation (PCG) method that is suitable for most of them as a consequence. Therefore, the effort of investigating and developing alternative PCG approaches, in order to solve these limitations, remains a current issue in this research area.

This dissertation presents a procedural level generator for 2D games, which can adapt the difficulty of the levels generated according to the player's skill, with the ability/capacity to widespread its generations to different styles of games and not just being limited to a specific game or genre. For this level generator, we developed a rhythm-based approach that utilizes the rhythms as a way to structure a new level, where a rhythm is a sequence of actions that must be made by the player to complete the level successfully.

In this project, the methodology differs from the classic rhythm-based approach (Gillian, 2011), as instead of using a sequence of single actions to define a rhythm, it is used a sequence of classes of actions, where each class has a difficulty associated with their mechanic description (which keys to press to perform an individual action or a combination of actions, for instance, pressing the space bar and the right key will execute the jump to the right action in most *platform* games). This variation of the classic rhythm-based approach seeks to solve its generalization problem, because it is inconceivable to use the same sequence of actions for different games, since most of them won't have the same possible actions. However, all games will have actions that are easier to perform than others, being possible to group them into classes, making the method more generic. For the rhythm generation we use a genetic algorithm whose fitness function has the ability to evaluate the difficulty of a sequence of classes of actions, which allows it to generate easier or harder rhythms depending on the classes of actions that are part of it. After generating the rhythms, they are used as parameters to a geometry generator, in order to associate each of the classes of actions to a level chunk (terrain, content, enemies, etc). If it is an easy class, then it will generate a level chunk that is supposedly easy to complete. For example, the run action is one of the easiest actions in most games, so it is a valid action for the easy class, in other words, it means that the run action belongs to the easy class. We can admit that an individual action like running or jumping can be perceived as easy to perform and, on the other hand, a combination of different actions (running $\rightarrow$ jumping $\rightarrow$ attacking an enemy) will be more complex to execute. We are not only going to organize the individual actions from a game into the classes, but also combinations of them, that will consequently vary their execution difficulty. In the geometry generator, there will be associations of level chunks to each of those actions (individual and combined actions), in other words, the geometry generator will have the information of what in-game content can be completed by a specific action or combination of actions. This process must be applied to all the possible actions and their combinations in a game so it can generate appropriate level chunks for each of the classes of actions. This has the purpose of correctly translating a rhythm to a valid playable level, by providing and joining level chunks that match with each of the classes from that rhythm.

In order to assess the quality of the generated content and make some corrections based on that, the level generator will first be applied to one particular game. Then the generated levels are going to be compared with the original levels from that game, so it is possible to accurately evaluate if they are valid or not. Considering that the objective of making such modifications to the classical rhythm-based approach is to make it more generic, after successfully implementing the generator to a specific game, more applications to different games will be made, so it is possible to test the generator's ability to widespread.

Finally, we did some tests to analyze the results of the levels generated and to validate the definitions of difficulty. These tests used the A* search algorithm (Millington, 2019) and human players. With the search algorithm, it was intended to evaluate the consistency of the levels, regarding their difficulty, and the dynamic adjustment technique. With this, there will be a way of verifying if the generator is correctly adapting the levels based on a difficulty value and if these adaptations are consistent. Regarding the human player test, they were made with the purpose of proving that the notions of difficulty, which were used in the development of the project, are generic enough for most players and, with this, validating the definitions of difficulty. Another objective of this test is to also confirm if the generated levels are valid, based on the opinions of the players.

1.1 Objectives

Considering that most procedural level generation approaches, for 2D games, are too specific for a particular game, it would be advantageous to develop a more generic level generator that could be applied to different 2D games. With this in mind, it was decided to introduce some modifications to the classical rhythm-based approach for level generation.

The classic rhythm-based approach consists on describing a level (from a 2D game) as a sequence of actions, so it can simplify the generation process by dividing it in smaller pieces. Then, with a grammar-based method, the geometry of the level will be structured, by matching appropriate in-game content with a specific action from the sequence that describes the level (Smith et al., 2009). Even though this is a proper level generation technique, it is very specific to a game and difficult to generalize, since not all games will have the same actions. Consequently, there will always be limitations when trying to produce generic sequences of actions, because it is not possible, in most cases, to use the same sequence for different 2D games. To solve this issue, instead of using a sequence of actions we propose the use of a sequence of classes of actions. A class of actions will organize the possible actions from a game by difficulty, allowing it to be generalized, since it is possible in most cases to rank the different actions from a game by difficulty. Each class will have a difficulty value associated to it, which it is correlated to a mechanic description. The purpose of a mechanic description is to specify how an action that is part of a particular class is executed (how many keys are pressed, the pressure that needs to be applied, the state a player must be to execute it, etc.).

Thus, the primary objective of this project is to develop a procedural level generator for 2D games that produces valid levels (that are possible to complete, with no impossible actions and challenges), by using a rhythm-based approach that has the ability to widespread its generations to different styles of games and not just being limited to a specific game or genre. Since the level generator is also required to adapt the difficulty of its generations, another essential factor is to implement an AI element that is going to dynamically adjust the difficulty of a game's level based on the player's skill or experience. A game is correctly adapted by difficulty when the generator is providing levels with suitable challenges based on a player's abilities, therefore, it is mandatory to evaluate if the implemented dynamic difficulty adjustment (DDA) technique is working as intended (adapting the level correctly to a target difficulty).

To experiment this variation of the rhythm-based approach for level generation, the developed generator must be applied to a game, in order to evaluate the feasibility of the levels generated, the adaptations made to the difficulty and to identify eventual problems that can occur. Then, the approach needs to be applied to different game genres, to verify its ability to be generalized.

Finally, it is crucial to make some tests with human players, to analyze and validate the definitions of difficulty, that are used in the generator's dynamic adjustment technique, based on their opinions about the different stages of the levels generated and performances. For this purpose, we devised a set of tests, to infer whether we were successful in the development of a rhythm-based approach for procedural level generation that can be generalized to more than just one genre and that can adapt the difficulty of its generations based on a player's skill.

Accordingly, the established objectives for this project are:

- **Investigate approaches for level generation and dynamic difficulty adjustment**. To be able to assess the most relevant and important approaches to accomplish the objectives, it is fundamental to investigate. Then, by taking into consideration the knowledge attained, it is necessary to determine the best level generation approach and dynamic difficulty adjustment technique to be used in the resolution of the project's problems. Since most level generation approaches are not generic enough, it is required to make some modifications to one of these approaches in order for it to be applicable to different 2D games.
- **Develop a rhythm generator**. We opted to modify the rhythm-based approach for level generation, thus the rhythm generator is one important part of our project, since the rhythms are going to be the base structure of the generated levels. The rhythm generator should be able to generate different sequences of actions for a specific difficulty value, being required because of this, to evaluate the difficulty associated to each action from a rhythm.
- **Develop a geometry generator**. After implementing a rhythm generator, it is necessary to translate the rhythms into levels. Therefore, the geometry generator must be able to take the rhythms as parameters and return a playable level. Considering that we also want to adapt the difficulty of the contents generated in the levels, a dynamic difficulty adjustment technique will be included in the geometry generator, in order to adapt the difficulty of the generated levels depending on a player's skill.

- **Evaluation**. A set of tests must be made to assess the ability of the procedural level generator to widespread its generated content and to check if the levels are getting correctly adapted to a specific difficulty value. Since difficulty is a very subjective concept, it is required to have a test that uses a control group, so it is possible to validate our definitions of difficulty.

- **Analysis of results**. One of the most important points to complete a project is to analyze the results produced by the approach used to solve the problem, in order to evaluate how far the objectives have been achieved and whether the project was successful. Furthermore, this analysis should clarify which improvements we should make to the approach, if any, and give insights into the future work in this area of research.

1.2 Contributions

In this dissertation, it is shown that a rhythm-based procedural level generator can be modified to be applicable to different 2D games, mainly because the possible actions in these games have a lot of similarities (in terms of what keys to press) and can usually be sorted by difficulty, that allows us to make this generalization. It is also demonstrated that a genetic algorithm can be an excellent tool to generate diverse sequences of classes of actions, having the ability to generate different rhythms for the same difficulty value.

As we discuss in the Conclusion, the results of the tests proved that we established a valid definition of difficulty for each class of actions (a group of possible actions that a player can make) in 2D games. If an action is in the same class of the other, it means that they have the same assumed difficulty. The idea of using classes of actions instead of a single action in the rhythms is what concedes the generator the capacity to generalize. This is because not all games have the same actions, but they do have actions easier than others and, therefore, it is possible to group them in different categories, from easier to harder for example.

Thus, the principal contributions of this project are:

- **Generalization of rhythm-based approach**. The classic rhythm-based approach describes a level as a sequence of actions. Then, by using a grammar-based method, the geometry of the level will be structured, by matching in-game content with an action from the sequence. Since a sequence of actions is always very specific to a particular game and complicated to generalize, we proposed some modifications. Those changes consist on using a sequence of classes of actions instead of using a sequence of actions, in order to make the sequences more generic. We applied this methodology to four different games, where two of them are from the *platform* genre, one a racing game and the other a horizontal scroller shooter. By having the results as proof, we concluded that the generalization of this level generator (that makes use of our modified version of the rhythm-based approach) was successfully made to various games of very distinctive styles.

- **Classes of actions**. We introduced the classes of actions. A class of action will help to group all the possible actions from a game by difficulty. This will grant us the ability to generalize the rhythms (sequences of classes of actions) to different games, since in most cases it is possible to sort the actions by difficulty. Each class will have a difficulty value associated to it, which it is correlated to a mechanic description.

- **Mechanic description**. We defined a mechanic description for each class of actions, that helps us to generalize our characterization of difficulty to different games (definitions of difficulty). A mechanic in-game is done by the players, that provides us what we call gameplay. It is the clarification of what keys to press or what buttons to click to execute a certain action (for instance, pressing the left key to move left). Depending on what keys to press, an action will be part of a certain class of actions. These definitions of difficulty for each class were validated with tests, by using the help of human players.

- **Rhythm generation for each difficulty value with a genetic algorithm**. We implemented a genetic algorithm whose fitness function can evaluate the difficulty of a rhythm, which allowed us to generate different sequences of classes of actions with the same desired difficulty. The genetic algorithm also provides a diversity of rhythms to the same target difficulty, making every level different, even though it has the same difficulty value.

- **Geometry generator that translates rhythms into levels**. We implemented a geometry generator for each game that uses the rhythms as parameters and, based on our definitions of difficulty, the geometry generator builds, in real-time, a valid brand new playable level. In other words, knowing that the action that we want is from a certain class of actions, the geometry generator will build content, or challenges, that can only be fulfilled using one of those actions from that class (for instance, if there is a gap in the terrain the only way we have to get past it is by jumping). Since the level is structured by the player's possible actions, the generator guarantees every time that it is possible to complete the level (does not have impossible actions/challenges).

With all this together, we can conclude that it was possible to develop a procedural level generator that can adapt the difficulty of its levels to a player, by using a sequence of classes of actions, where each class has a difficulty associated with its mechanic description, and such an approach can be generalized to different 2D games, always generating different and playable levels in real-time for the same desired difficulty (has a lot of diversity) in each run, as we report in the Conclusion.

1.3 Structure of the document

This document is organized in the following form. In Chapter 2, we introduce all the related work and concepts that we used throughout this project, including procedural content generation, dynamic difficulty adjustment and genetic algorithms. Chapter 3 will describe the methodology used for the level generation, from how the components of the generator were developed to how they were evaluated. In Chapter 4, we present the results from the tests made and analyze them on detail. Finally, in Chapter 5 we make conclusions regarding the project's success and suggest ideas for future work.

Chapter 2

Related work

In this chapter, we discuss the fundamental concepts related to procedural content generation, as it is the most important topic of our project. Afterwards, because it plays an essential role in the successful development of our work, several procedural level generation methods, dynamic difficulty adjustment techniques and genetic algorithms are analyzed. Considering that procedural level generation is the most important subject matter of this project, we give special attention to it.

2.1 Procedural content generation

Procedural content generation (PCG) is the algorithmic creation of game content – *e.g.* characters, terrains, dungeons and levels - with little human intervention, that usually implements some stochasticity which combined with constraints and parameters, will define what content and how it should be generated (Ulisses et al., 2015). Level generation is the most significant old challenge of PCG domain (Sharif et al., 2017), offering lots of advantages, such as consuming less time and money, when generating in-game contents, compared with the manual generation and providing more variety in the content that can be generated (Sharif et al., 2017), that consequently increases the replayability and longevity of a specific game. PCG is considered an AI task, since it can replace the work done by a designer or artist, allowing independent developers and small companies to overcome their lack of resources to design game environments from scratch (Sharif et al., 2017).

The main limitation of PCG is when we want to have complete control of the content generated with lots of specific details, in this scenario the manual generation achieves the best results, since it needs total human intervention, something that PCG does not have (Sharif et al., 2017).

This field of study has a long history. One of the first uses of PCG algorithms goes back to the 1980s era when *Rogue* (Michael Toy, 1980) was released with a procedurally generated dungeon that would be different each time it was played, so a level could not be repeated and the only way to beat the game was to really have the skill and not just memorize a pattern of moves to pass certain challenges (enemies, terrain, etc.). Although it is not a new area, only recently methods from artificial and computational intelligence started to be used in video-game content generation. Especially "search-based" approaches to PCG (creating an evolutionary algorithm or another stochastic search and optimization algorithm) have only recently been explored (Shaker et al., 2011).

The most common use of PCG in commercial games nowadays is the offline creation of trees and vegetation (Shaker et al., 2011). Although, there are a few commercial games that use level generation – e.g. *Rogue* (Michael Toy, 1980) and games inspired by it like *Diablo* (North, 1996) – PCG algorithms are not usually used for the generation of major game elements, because these techniques are still not seen by game developers as a reliable source of generation for such game contents (Shaker et al., 2011).

According to *Far Cry 2* (Montreal, 2008) technical director Dominic Guay, one difficulty associated with procedural generation is retaining artistic control alongside the dynamic processes, since a lot of focus must be put on the tools that surround the algorithms, to allow the systems to be properly utilized (Remo, 2008). Another drawback is the complexity of the pipeline and tools, that require much focus on the systems and rules that it can be shocking to artists, who are accustomed to more straightforward linear art tools (Remo, 2008). Consequently, given the need for a reliable fast content generation tool, the development of a better and more versatile PCG method is a very important research topic.

Besides being fast, reliable and produce high-quality content, another characteristic that is needed for PCG algorithms, in some situations, is to be controllable (J.Doran and I.Parberry, 2010). A PCG algorithm is controllable if it can take parameters that will specify the features of the generated content. These features can be described on different levels of abstraction, from geometric aspects to gameplay aspects. This is particularly useful for content that is produced by a combination of human designers and algorithms, so the human designer can specify features that the content should have that are suitable for human editing or to fit into already human tailored content (J.Doran and I.Parberry, 2010) (R. M. Smelik and Bidarra, 2010).

It is also relevant to have PCG automatically adapt to the player, for instance, generating harder/easier levels based on the skill of the player. Such adaptation has become more important as the game-playing population increases and, therefore, gets more diverse. This can also be seen as a way of captivating players by adapting the different challenges that a game has depending directly on the player's skills/experience (C.Pedersen and Yannakakis, 2010) (N. Shaker and Yannakakis, 2010).

This game adaptation can be performed using a technique called dynamic difficulty adjustment (DDA), which consists of making some changes in the parameters, scenarios and game behaviors to avoid frustrations while playing the game challenges (*e.g.* being too easy or too hard) (Silva et al., 2016).

2.1.1 Taxonomies

In this subsection, the most important types of generation in PCG are discussed and compared with each other whenever possible, by having the articles (Shaker et al., 2011) (Togelius et al., 2011) as a reference.

Online versus offline

The main difference between these two types of generation is that in the online the content is created during the run-time of the game and, on the other hand, in the offline this is done during game development. There are three different types of online generation: static, dynamic and procedural generation based on the experience of the player.

Necessary versus optional content

The generated content can be necessary or optional. Content is considered necessary if it is required by the players to progress and complete the level – *e.g.* enemies that need to be defeated, dungeons that need to be traversed, etc. – whereas optional content could be ignored or avoided by the player, in other words, it is not required to achieve level completion (*e.g.* available items or locations that could be explored or ignored). Necessary content is crucial to make progress in the game, therefore, it must always be valid. For instance, it would not be valid to have unbeatable enemies or intractable dungeons, because it would be impossible for the player to progress. On the other hand, sometimes it can be allowed to generate optional content that is "unusable" or invalid (useless weapons, unnatural layout of a scenario like a house, dungeon, etc.), since it is not necessary for the completion of the level.

Random seeds versus parameter vectors

In terms of parameters for these PCG algorithms there are two different types. The first approach is called random seeds because, it will only take a seed (to its random number generator) as input. In the other case, the algorithm could receive a multidimensional vector of real-valued parameters that are going to specify how the content and its properties should be generated. The things you could define with parameters would be for example the number of rooms in a house/dungeon, items location, etc. The challenge of using a random seed or parameter vectors is called the number of degrees of control, since the control you have of the content that is generated by the algorithm relies essentially on the parameter approach you are using, with parameter vectors you have more detailed control of how the content should be generated than with the random seed approach.

Stochastic versus deterministic generation

The generation could be either stochastic or deterministic. If it is deterministic, given the same parameters, the result content will always be the same. On the other hand, if it is stochastic, it means that given the same parameters to the algorithm it would not generate the same content (here we assume that a random number of generator seed is not considered a parameter since its only purpose is to create randomness). These different approaches help us to define the amount of variation and randomness that we want in the generated content, given identical parameters as input.

Constructive versus generate-and-test

The generation process can be either constructive or generate-and-test. A constructive algorithm will only generate content once and because of this, it needs to assure that the generated content is correct/valid while it is being built. This is normally achieved by only using operations or sequences of operations that guarantee correctness in the produced content.

A generate-and-test algorithm, like the name suggests, have a mechanism of generation and testing. Firstly, the algorithm will generate content that will be evaluated by some criteria that checks its validity and if it fails this test, the content is discarded and regenerated (rebuilt/remake). This process goes on until the content has the minimum requisites for it to pass the test, meaning that it is “good enough” based on the defined criteria.

The following table 2.1 illustrates some examples of games that use the different types of PCG taxonomies previously presented.

PCG Taxonomy	
Approach	**Game examples**
Online	*No Man's Sky* (Games, 2016), *Rogue* (Michael Toy, 1980), *Diablo* (North, 1996)
Offline	*Horizon Zero Dawn* (Games, 2017), *Ghost Recon Wildlands* (Paris, 2017), *Elite* (David Braben, 1984)
Necessary	*Call of Duty 4* (Ward, 2007), *Doom* (id Software, 1993), *Legend of Zelda* (Shigeru Miyamoto, 1986)
Optional	*Borderlands* (Software, 2009), *Galatic Arms Race* (Games, 2010)
Random seeds	*Minecraft* (Persson, 2011), *Dwarf Fortress* (Tarn Adams, 2006)
Parameter vectors	*Galactic Arms Race* (Games, 2010), *Left 4 Dead* (Studios, 2008)
Stochastic	*Rogue* (Michael Toy, 1980), *Diablo* (North, 1996), *Spelunky* (Derek Yu, 2009)
Deterministic	*Elite* (David Braben, 1984), *.kkrieger* (.theprodukkt, 2004), *Terraria* (Re-Logic, 2009)
Constructive	*Rogue* (Michael Toy, 1980), *Diablo* (North, 1996), *Spelunky* (Derek Yu, 2009)
Generate-and-test	*Sokoban* (Imabayashi, 1981)

Table 2.1: Examples of games for each PCG taxonomy

2.1.2 Special approaches

In this subsection, we analyze some special PCG approaches, including search-based procedural content generation, experience-driven procedural content generation and learning-based procedural content generation.

Search-based PCG

A search-based procedural content generation, or SBPCG, assigns the use of evolutionary algorithms of another search/optimization algorithms to generate content for games (Kerssemakers et al., 2012). A SBPCG is a special case of generate-and-test generation, with the following features (Togelius et al., 2011):

- Besides accepting or rejecting a candidate content, the test function also grades it by using one or a vector of real numbers. This test function is normally called a fitness function, evaluation function or utility function. In this case, the value of the content generated will be its fitness, which allows us to make an assessment of how good the game content is.

- Generating new candidate contents is dependent of the fitness value assigned to the previously evaluated content instances, with the objective of generating new content with higher fitness value.

To sum up, a fitness function is a tool used to accurately make an evaluation of the content's suitability in a game. The biggest challenge of this approach is to formalize a good enough fitness function that grants the desired optimizations for the content (Togelius et al., 2010). Even though this approach relies on evolutionary algorithms, it does not mean other heuristic/stochastic search mechanisms are not viable as well (Togelius et al., 2011).

In the next figure 2.1, it is illustrated the overall flow of a SBPCG and situates it in comparison with the generate-and-test and constructive approaches (Togelius et al., 2011).

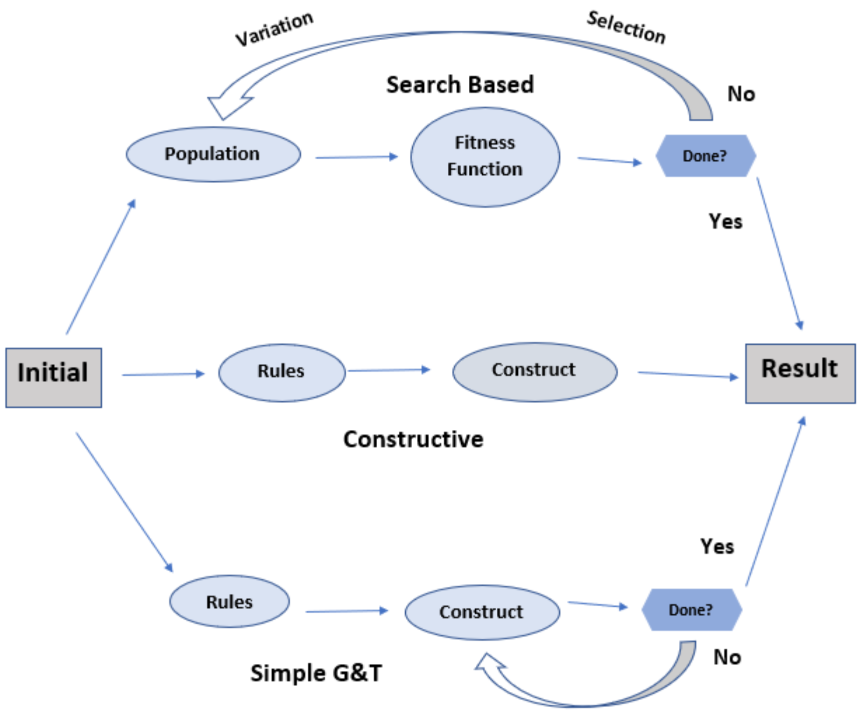


Figure 2.1: Comparison between the SBPCG, generate-and-test and constructive approaches (Togelius et al., 2011)

Experience-driven PCG

Experience-driven procedural content generation, or EDPCG, is a PCG method that proposes the usage of the player's experiences, as a tool, to make optimizations in the game content generated.

In this approach, game content is seen as building blocks of games and games as ”potentiators” of player experience. Thus, content can be seen as indirect building blocks of player experience. Since a game is defined by game content building blocks, that when played results in player experience, it is very important to be able to evaluate the quality of the content generated, search through the available content and generated content that optimizes the player experience. Therefore, the structure of this method is the following (Yannakakis and Togelius, 2015):

- Player experience modeling: player experience is modeled as a function of game content. The player will then be described by the playstyle and the cognitive and affective responses to the gameplay.
- Content quality: the quality of the content generated will be evaluated and linked to the model of the player experience.
- Content representation: content is represented accordingly to maximize efficiency, performance and robustness of the generator.
- Content generator: the generator will search in the content space for content that optimizes the player experience accordingly to the model acquired.

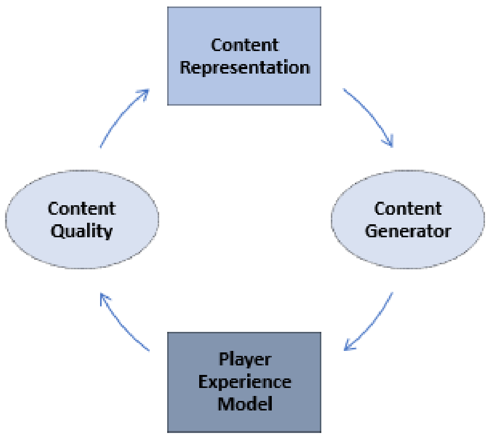


Figure 2.2: Structure of the experience-driven approach (Yannakakis and Togelius, 2015)

Neuroevolutionary preference learning can be used to build the player experience models that approximate the function between the selected subset of gameplay features, controllable features, expressivity features and affective preferences (Shaker, 2012).

Learning-based PCG

Learning-based procedural content generation, or LBPCG, is a PCG framework based on machine learning. Machine learning is a data-driven methodology for knowledge acquisition/modeling that has been successfully applied to different AI fields (from machine perception to natural language understanding) (Roberts and Chen, 2015), but it is still quite rare in games (Lucas et al., 2012).

LBPCG explores and utilizes knowledge and information gathered from game developers and public testers using a data-driven methodology. It relies on the generalization of component models trained with inductive learning, that allows us to generate robust and adaptable content to a random player. Therefore, LBPCG has several noticeable features, including avoiding evaluation functions, effectively limiting the search space to relevant content and minimizing interference to the end-user gameplay experience (Roberts and Chen, 2015).

The process of this approach uses commercial video game development as a reference (a division of the life cycle of a game into three stages: development, public test and adaptive). Thus, taking this into account, the LBPCG framework encodes the developer's knowledge on game content in the development phase, models public player's experiences in the public test stage and, finally, generates the content to the players in the adaptive stage (Roberts and Chen, 2015).

2.2 Procedural level generation

Level generation is a crucial objective for this project and, like it was mentioned before, it is a difficult area in PCG, being less common to find information on how developers use it. The popular approaches for generating levels to 2D games (mostly for *platformers* like *Super Mario Bros* (Shigeru Miyamoto and 4, 1985) and *Sonic the Hedgehog* (Yuji Naka and Yasuhara, 1991), etc.) are presented in this next subsections.

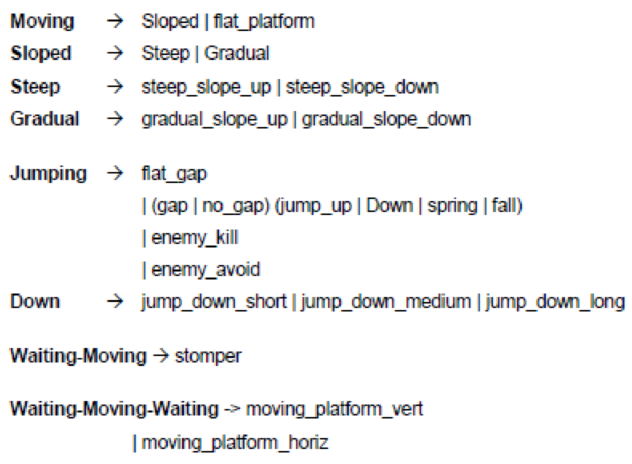
2.2.1 Rhythm-based

The main idea of this technique is the use of what is called the rhythm of a level, which is going to divide the generation problem into smaller pieces. These smaller pieces are usually found by using pre-made parts of game sprites. The most important part of this approach is the division of the problem that can be compared with the division of a music piece that is marked with some rhythm, but instead of defining the rhythm by the beats (like in a song) it is defined by using the player actions - *e.g.* jumping, running, crouching – following the rules and metrics implemented (Gillian, 2011).

The generation of the levels is made by a grammar-based method, where firstly the rhythms are generated and then geometry based on them are built. The generation will be defined by parameters that can be manipulated by a human designer and will also minimize the manually made content, relying only on the geometry components that are defined in the game's tileset. By explicitly separating the rhythms generation and the geometry generation we can represent more variety than in their pattern building approaches, where the rhythms are defined by geometry (Smith et al., 2009).

The process starts with a two-layered grammar-based approach to generate small parts of a level (rhythm groups). The first phase creates a sequence of actions that a player will make, restricted by the rhythm. Then, the second phase will convert that sequence of actions into geometry, by using a grammar-based method. To get the complete level, the geometries resulted from the rhythms are put together (Smith et al., 2009).

In the next figure 2.3, it is shown a possible grammar to be used on a geometry generator for a *platformer* game (Smith et al., 2009).



```
Moving     →  Sloped | flat_platform
Sloped     →  Steep | Gradual
Steep      →  steep_slope_up | steep_slope_down
Gradual    →  gradual_slope_up | gradual_slope_down

Jumping    →  flat_gap
              | (gap | no_gap) (jump_up | Down | spring | fall)
              | enemy_kill
              | enemy_avoid
Down       →  jump_down_short | jump_down_medium | jump_down_long

Waiting-Moving → stomper

Waiting-Moving-Waiting -> moving_platform_vert
             | moving_platform_horiz
```

Figure 2.3: Example of grammar to be used in a geometry generator (Smith et al., 2009)

2.2.2 Combining pre-made parts

Combining pre-made parts is the most popular and easier method for level generation. There will be some pre-made parts that we know work with the game and repeat them in a complex way. It is possible to do the most basic repetitions where we have obvious patterns and complex random selections that are controlled by some rules (Barrett, 2012) (Mawhorter and Mateas, 2015).

Building a whole new unique level chunk by combining pre-made parts is a very powerful way of producing diverse combinations that the player never saw before in-game. This sort of pre-made content can be seen in the game *Spelunky* (Derek Yu, 2009), where a manual made level part is lightly altered in some tiles, allowing this to provide incredible diversity (Tanya X. Short, 2017).

Even though this is a great method for unique level generations, we need to be aware that even combinatorially driven procedural content can start to look repetitive, after the player has seen hundreds of different combinations and discovers the pre-made parts. In other words, the pattern-matching human brain kicks in and the illusory curtain of uniqueness fades away (Tanya X. Short, 2017).

2.2.3 AI-based

The concept behind the AI-based approach is to create a level design AI. The most important feature that this algorithm needs to have, to be considered successful, is that it must generate levels that are possible to complete.

To prove that a game is feasible an AI player is used, that is going to confirm if the levels are valid by completing them. AI agents that can play through levels of a game is a popular research field which has diverse examples of implementations, making it accessible to integrate in a project. After having an AI player, the knowledge of the game's physics (information about the possible player actions) are given to it, so the AI can take this information and use it as constraints to test if a level is really possible to complete (Fisher, 2012).

The level design AI always has the aid of the AI player. If we have a partially generated level that we know it is feasible until now (which means, it is possible for a player to go from the beginning to the end), the level design AI can consider placing a new chunk, to extend the level. To do this, it verifies if the new generated chunk interferes with the feasibility of the existing level and if it is possible for the player to reach this new chunk (all by using the AI player). In other words, if the AI player is still able to complete the level after the placement of the new chunk, then the level design AI adds it and continues the level generation process. Otherwise, the level design AI does not add the new chunk and tries to generate a different one instead. This process is repeated as long as we intend (Fisher, 2012).

In this approach, to get all the possible levels that are feasible, the brute force attack is used. The general principle that this idea follows is to make the design algorithm as generic as possible, using parameters to define most of the characteristics of a level, to be able to randomize them (the parameters) as much as possible, resulting in plenty of level diversity (Fisher, 2012).

This technique is the most complex and least efficient method, but it is possibly the most versatile. By using random generations and other variations, it guarantees no repetitions and no observable patterns that, by using the AI player, the correctness of the generated level can be proven.

2.3 Dynamic difficulty adjustment

Some elements of a game that can be changed by using a dynamic difficulty adjustment (DDA) technique are, for instance: speed of enemies; the health of enemies; frequency of enemies; frequency of power-ups; power of the player; power of enemies and duration of the gameplay. In this section, the different types of methods for DDA are analyzed, by using the article (Zohaib, 2018) as a base.

2.3.1 Probabilistic methods

In these approaches, DDA is seen as an optimization problem where the player's progression is modeled on a probabilistic graph that maximizes engagement as a well-defined objective function. Then a dynamic programming technique with high efficiency is used to solve it. This DDA technique can be successfully applied to various genres if an appropriate progression model is constructed. The different states for level-based games can be established by two important facets: trial and level. This approach can also be defined for games that have multiple or nonlinear progressions (*e.g.* role-play games). For these cases the graph is more complex, since there is a lot of diversity in the states and many links.

2.3.2 Dynamic Scripting

Dynamic scripting is an online unsupervised learning approach for difficulty adaptation that operates many rule-bases in a game for each opponent type. The rules are designed manually by using domain-specific information and will have a probability of selection that it is correlated to the weight value assigned to it. When creating a new opponent, the rules in the script that define the enemies are taken from the rule-base depending on the type of enemy and the rule selection probability. Then, the rule-base adjusts itself by updating the weight values and by taking into consideration the rates of failure or success of the script rules.

In this technique, learning is a progressive task. At the end of an encounter, the rule weights used are updated depending on their effect in the result. The rules leading to success increase their weight, while those leading to failure have their weights decreased. The remaining rules are then adjusted properly so that the sum of all the rule-base weights is constant. This approach is normally used in the generation of new opponent tactics while increasing the difficulty of the challenges of the game's AI to match the skill of the player.

2.3.3 Hamlet system

Hamlet is a DDA system that was built with the main idea of using operation research and inventory management methods. It is going to analyze and adjust the supplies and demands of the in-game inventory and, with this, shape the difficulty depending on the player's skill. Hamlet uses metrics to capture incoming game information as the players go through the levels and then is going to predict the future state of the players in the game, based on that information. When an unwanted but preventable state is predicted, Hamlet changes game settings required to avoid it. It tries to anticipate when a player is going to struggle repeatedly and is near a state where his resources can not meet its requirements.

2.3.4 Reinforcement learning

The concept of this DDA approach is using an adaptive AI in the game to adjust the game behaviors and parameters in real-time automatically, depending on the skill of a player. It has the advantage of keeping the player engaged for longer periods and improve their general experience.

Since the DDA is going to be made in real-time, the adaptive AI requires being capable to make unexpected but rational judgments, like a human player, but should not display a thoughtless behavior. It also needs to correctly evaluate its opponent in the early stages of the game and then adjust its playing style to the opponent's skill.

2.3.5 Affective modelling using electroencephalography

Normally in a game that keeps the score, the DDA applications decisions can be made depending on the difference between the player's scores and a certain threshold value, like for instance when one player becomes much stronger than the other, we could assume that this will be a boring situation for the stronger and frustrating situation for the weaker player. In this approach, the difference is that the excitement of the players is measured and it sets the game in motion when the level of excitement goes below a threshold value. Instead of depending on heuristic scores to decide when a player is bored/frustrated with a game, the gaming experience is addressed directly for this concern. An affective-state regulation technique is implemented by using headsets to decipher electroencephalography signals and the mechanism to modify the signal to an affective state. Later, by using this affective state information, the DDA is set up by the game.

2.3.6 Artificial neural networks methods

In this subsection it is presented DDA approaches that make usage of artificial neural networks (ANN).

Single and multilayered perceptron

The main idea is to use a neural network model that maps characteristics like playing behavior, design parameters of levels and player's emotions then train it, by using game session data and evolutionary preference learning.

Upper confidence bound for trees and artificial neural networks

It is a DDA technique that uses artificial neural networks from data provided by the upper confidence bound for trees (UCT). UCT is a computing intelligence method and because of this, its performance is directly correlated with the duration of the simulation. The main concept is to train the ANN with the UCT-created data of a player's gameplay progress (*e.g.* win rates).

Self-organizing system and artificial neural networks

A self-organizing system (SOS) is a group of entities that display global system characteristics through local interactions while not having centralized control. This approach introduces a technique that is going to try to adjust the difficulty of a level by creating a SOS of Non-Player Characters (NPC). The system is going to use ANN to track the player's in-game progress and, because the ANN needs to adapt to player's different levels of skills and playstyles, an evolutionary algorithm is developed to update the vector weights in the ANN (depending on the player).

Game examples that use DDA
Archon (Associates, 1983)
M.U.L.E (Software, 1983)
Global Conquest (Software, 1992)
Crash Bandicoot (Dog, 1996)
Resident Evil 4 (Capcom, 2005)
Fallout 4 (Studios, 2015)
Mario Kart (Nintendo, 1992)
God Hand (Studio, 2006)
Homeworld (Entertainment, 1999)
Madden NFL 09 (Tiburon, 2008)
Left 4 Dead (Studios, 2008)

Table 2.2: Example of games that have a DDA mechanism

2.4 Genetic Algorithms

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution, with the purpose of being applied to optimization and search problems. It was introduced in 1975 by John Holland in his book ”Adaptation in natural artificial systems” (Holland, 1975) and has since then been improved, becoming nowadays a very useful tool.

Genetic algorithms are one of the most popular out of the existing evolutionary computation paradigms. This algorithm can be applied for a search problem, by considering a set of solutions for a problem and select the best ones (Mallawaarachchi, 2017). Normally, the population of solutions are named individuals that are represented by a fixed-length bit string, which is equivalent to a chromosome in biology, where each position of the string represents one feature (known as genes) (Forsblom and Johansson, 2018). Therefore, a group of genes is a chromosome and a group of chromosomes is known as a population (figure 2.4).

The most popular operation in genetic algorithms is the *cross-over* operator, that takes two fit individuals and cross them at a *cross-over* point, producing two new offspring (as figure 2.5). There are also other operators like *inversion* (where a part of the string is reversed) and *bit flipping mutation* (where flipping one bit of the string will produce a new offspring, as figure 2.6) (Sivanandam and Deepa, 2008).

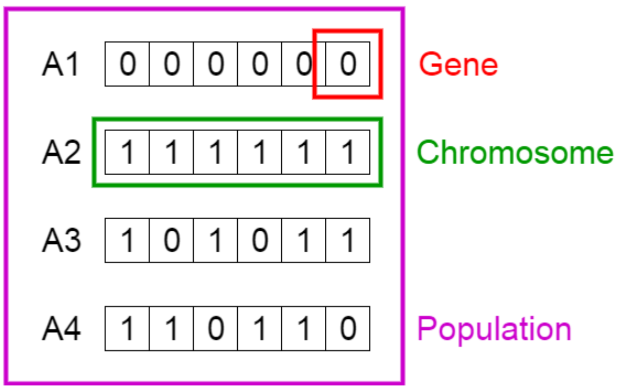


Figure 2.4: Illustration of a gene, chromosome and population (Mallawaarachchi, 2017)

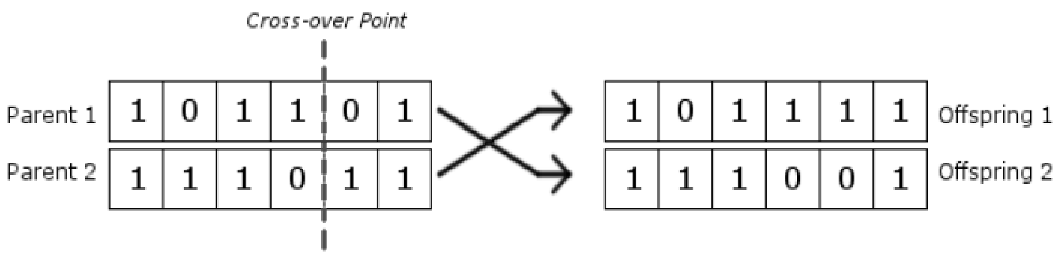


Figure 2.5: *Cross-over* operation (Forsblom and Johansson, 2018)

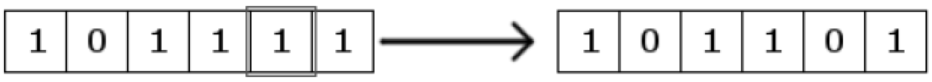


Figure 2.6: *Bitflip* operation (Forsblom and Johansson, 2018)

The first genetic algorithms only used one *cross-over* point when crossing individuals and, because of this, it was not possible to give a new offspring the features from both the head and the tail of the string. If for some reason the best features are in the head or tail, the new offspring would not have them. To solve this problem two *cross-over* points were introduced, preserving the head and the tail of the string, and the part in between is crossed (Forsblom and Johansson, 2018) (as figure 2.6).

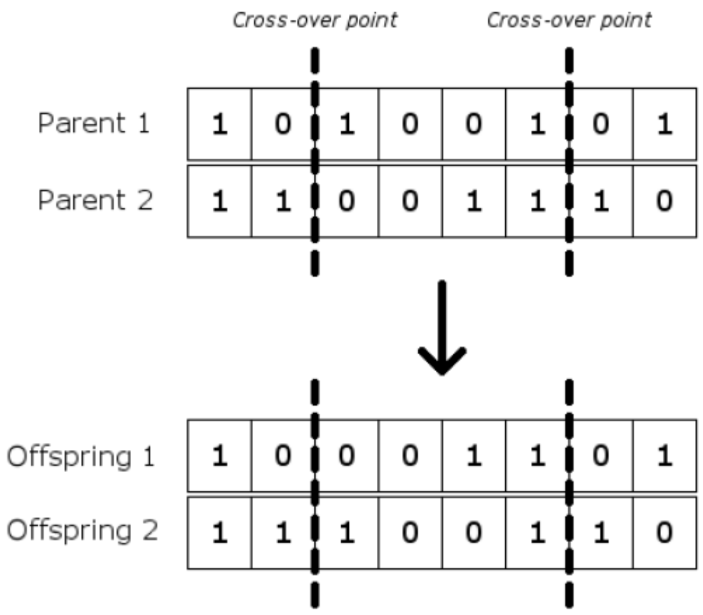


Figure 2.7: Two *cross-over* points operation (Forsblom and Johansson, 2018)

A fitness function is used to calculate how fit an individual is to solve a problem, in other words, it evaluates the quality of the solution. In a genetic algorithm, the fitness function is used to optimize value (goodness, profit, or utility) with the purpose of the strings with higher values having a bigger chance to be part of breeding offspring to a new generation (Goldberg, 1989).

There have been situations where some stochastic errors occur in genetic algorithms, that can lead to either a genetic drift, where the selection of fit individuals are favored, or premature convergence, that happens when the offspring can not outperform their parents, which results in less diversity and a non-global optimum in the search (Rakesh Kumar, 2012).

2.4.1 Selection methods

Selection is the act of choosing two individuals from the population to *cross-over*, in order to breed a better offspring. The most relevant selection methods are roulette wheel selection, tournament selection, random selection and rank selection. Selection pressure is a common term when we are selecting individuals for breeding, in other words, the higher the pressure, the higher the probability of fit individuals to be chosen, the lower the pressure, the lower the chance of a fit individual to be selected (Sivanandam and Deepa, 2008).

Roulette wheel selection

The concept of this selection method is taken from the roulette wheel. It is one of the first selection methods for genetic algorithms, introduced in 1989 by Golberg in his book ”Genetic Algorithms in Search, Optimization and Machine Learning” (Goldberg, 1989).

Each individual of the population has a portion of the roulette wheel, where the size of it is proportional to how good the fitness is (the better the fitness, the larger it is). To generate the new offspring the roulette is spun for as many reproductions as needed, where there is a minimum of two parents to breed a new offspring. When an individual is selected, it is then added to the mating pool, where it will be able to reproduce with other strings. After that, the strings in the mating pool are paired to *cross-over* in a randomly selected *cross-over* point (that cannot be the first character) (Forsblom and Johansson, 2018).

There is a problem associated with this selection that happens when the fitness values of the strings differ too much. If for instance, an individual composes 90% of the wheel, then it will be very hard to select any other individuals (Sivanandam and Deepa, 2008).

Random selection

In this selection, individuals from the population are picked randomly to be part of the mating pool for breeding. The result of this selection is, therefore, unpredictable and can cause a lot of disruption (Sivanandam and Deepa, 2008).

Ranking selection

Ranking selection is used to solve the problem that can occur in the roulette wheel selection (fitness values differ too much). This selection works by sorting/ranking all individuals from the population by fitness values. Then, it is going to be assigned a probability of selection to each individual based on their rank, instead of their fitness value. This way, the chance of selection of individuals will be more spread out. Ranking selection assures that convergence does not increase too fast, to avoid stagnation and premature convergence (Rakesh Kumar, 2012).

Tournament selection

In this selection, two individuals from the population are randomly chosen to compete. The individual with the strongest fitness wins and is then moved to the mating pool and the individual that loses is placed back into the population. The tournament is repeated until the mating pool is full, having the guarantee that the strongest/fittest individuals are there to reproduce and create a new generation (Koza, 1992).

2.4.2 Mutation

A mutation is used to help the genetic algorithm to not be stuck at a local minimum. It is going to change the population by randomly modifying its current genetic structure to provide diversity in the population. It can be made by *bit-flipping* or by swapping two random chromosomes of an individual (Sivanandam and Deepa, 2008).

2.4.3 Elitism

It is the process of allowing the strongest, or some of the strongest individuals to live in the next generation, to give meaningful improvements to the genetic algorithm performance. This happens because the best individuals could be lost due to *cross-over* or mutation, thus, reducing the genetic algorithm efficiency (Sivanandam and Deepa, 2008).

2.4.4 Genetic Algorithm Process

This algorithm simulates natural selection, where the fittest individuals are selected for reproduction (Mallawaarachchi, 2017).

Natural selection starts with the selection of the fittest individuals from a population. Then, they will produce new offspring which inherits their characteristics and will be added to the next generation. If the parents have a good fitness their offspring will have a better chance at surviving. This process keeps on iterating until a generation with the fittest individuals is found (Mallawaarachchi, 2017).

The process can be organized in five phases (Mallawaarachchi, 2017):

1. **Initial population** - The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve.

2. **Fitness function** - The fitness function will determine how fit an individual is, in other words, it will determine its ability to compete with other individuals. Therefore, the better the fitness score, the greater the probability of being selected for reproduction.

3. **Selection** - The selection phase consists on the selection of the fittest individuals that will pass their genes to the next generation (the different methods for selection were discussed previously in this section). Two pairs of individuals (parents) are required to produce an offspring.

4. ***Cross-over*** - For each pair of parents to be mated, a *cross-over* point is chosen at random from within the genes. Offspring is then created by exchanging the genes of

parents among themselves. Afterwards, the new offspring is added to the population (*cross-over* operation is represented in figure 2.5).

5. **Mutation** - In a new offspring, there is a low random probability of their genes to be subjected to a mutation. This implies that some of the bits in the bit string can be flipped.

The algorithm terminates if the population has converged, which means that it does not produce offspring that are significantly different from the previous generation or if the algorithm reached its maximum number of generations (in case it was previously defined). The last generation will be a set of possible solutions to our problem (the better the fitness score, the more suitable the solution is) (Mallawaarachchi, 2017). The following figures illustrate the previously discussed process (as figure 2.8) and an example of pseudo code (as figure 2.9).

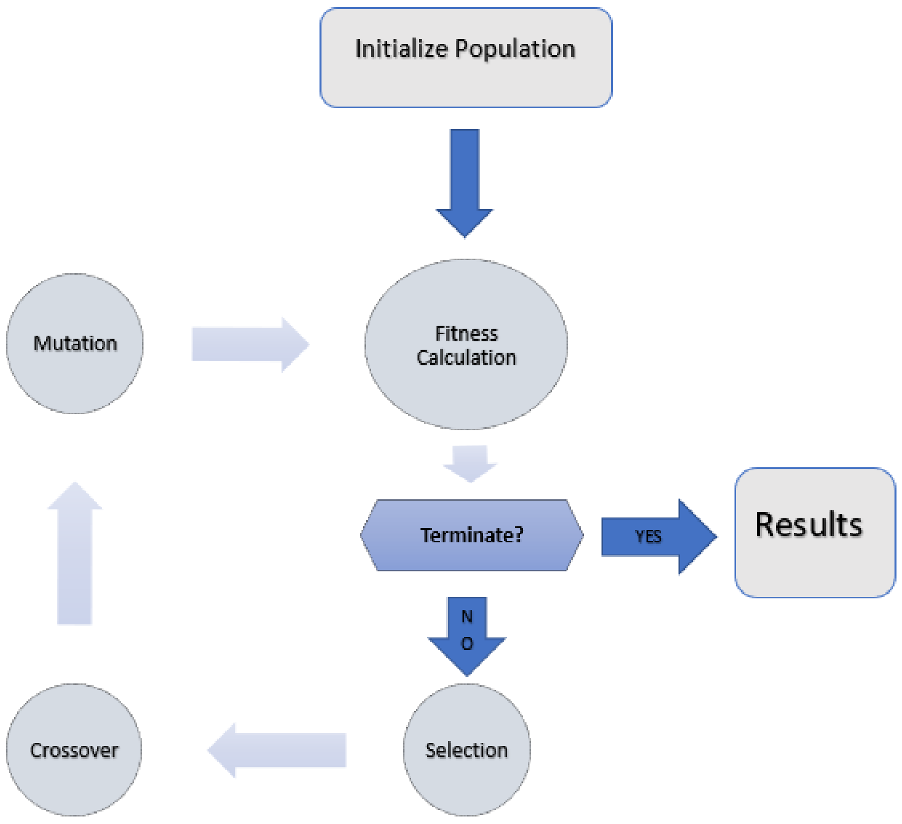
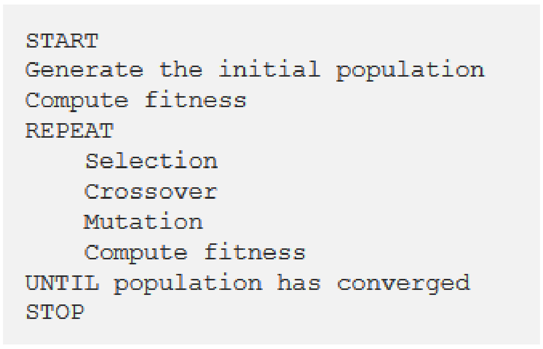


Figure 2.8: Genetic algorithm process represented by a diagram (Mallawaarachchi, 2017)



```
START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP
```

Figure 2.9: Pseudo code for a genetic algorithm (Mallawaarachchi, 2017)

Chapter 3

Methodology

This chapter describes the level generation methodology that was used for the developing of the generator. The tools used are the first topic of debate, from programming languages used to the framework, games, etc. Then, our idea of the methodology is explained and described into detail, along with the problems that we had to solve during the advancement of the project. The generator's components and their features are valuable points of focus, in order to clarify the design solution. Afterwards, methods and approaches for evaluation are also debated, since it is fundamental to have a way to critically analyze our concepts and designs.

3.1 Tools

This section aims to give insights into the process of decision making about the tools that were available for the development of the work and the tools that were used in the implementation of the level generator for 2D games.

3.1.1 Frameworks

In the initial stages of the project, it was analyzed which of the frameworks could be used for the procedural generation of levels. The main ones were Unity and Game Builder, because both of them were rather good platforms for the development of games in general. Even though these platforms were good options, the conclusion was that it would take a good amount of time to learn how to use them properly and so we opted to use tools that were already acquainted with, in order to mainly focus our attention in the development of the level generator. It was opted to work with frameworks that used Java and Python, not only for having useful and lots of libraries but also for being languages that are well known to us.

Pygame is a cross-platform set of Python modules, designed for writing games. It includes computer graphics and sound libraries created to be used with the Python programming language. It is based on the idea that the most computationally intensive tasks in a game can be abstracted separately from the main logic, which means, the use of memory and CPU are handled by Pygame code itself and not by your game code. Thus, it becomes possible to use a high-level language, such as Perl, Lua or Python to organize the structure of the games (Pygame, 2018). Being Pygame also a tool that was known to us it became our main ”workspace”.

For Java we chose the Mario AI framework, which was used for Super Mario AI tournaments, had open-sourced code in Java. In this tournament, there was a level generation track championship (procedural generation of levels) that focused on the software that could design levels for human players. This was relevant since it had a good Super Mario level generator development environment ready to be used. Since there was also a championship for best controllers (an agent that plays the game), some agents that could play the levels generated were available in this framework.

3.1.2 Games

The generator should be able to provide levels for different genres of 2D games, so, with this in mind, the games that were chosen are very different from each other in order to show the capability that the generator has to widespread it's generations.

The generator was tested in four games: *Super Mario* (Khalifa, 2009), *Rambit* (Oliveira, 2011), *CaveCopter* (Nolte, 2010) and *Swervin Mervin* (Buntine, 2017).

Super Mario and *Rambit* are *platform* games, or just *platformers*, and they are a game genre and subgenre of action games. In a *platformer* genre, the playable character must jump and climb between suspended platforms while avoiding obstacles. Environments normally have varying heights in the terrain and the player often has some control over the height and distance of jumps to avoid letting their character fall or miss necessary jumps. For the game *Super Mario*, its objective is to reach the end of the level without losing lives, by falling into the terrain gaps or getting hit by the enemy's attacks. Mario is the controllable character, and he can run left, run right, crouch and jump. The jump action is used to get past most of the challenges of the game, since it can kill and avoid enemies. For the *Rambit* game, the objective is to reach the end of the level without losing lives as well, by falling into the terrain gaps or getting hit by the enemy's attacks. Rambit is the controllable character, and he can run left, run right, jump and shoot his gun. Rambit will have 3 lives in total, that are decreased every time the player fails an action. The enemies can either be avoided, by the jump action, or defeated, by Rambit's gunshots.

CaveCopter is a Shoot 'em up, or a horizontal scroller shooter. This is a subgenre of games within the shooter subgenre in the action genre. The player's character engages in a solitary attack, often in a spaceship or aircraft, shooting at a large number of enemies and avoiding their attacks. In *CaveCopter*, the objective is to reach the end of the level without crashing, this happens if the player hits the level's terrain or if the helicopter has no fuel left. The helicopter is the controllable character, and it can be moved up, down, left, right and shoot missiles. The terrain can be avoided by using the correct movements at the time and the enemies can be defeated by the helicopter's missiles. The enemies have missiles and mines as their weapons, and if the helicopter collides with them it will lose fuel.

Finally, *Swervin Mervin* is inspired in the game *Outrun* (Suzuki, 1986), a racing game. This genre of games are either in the first-person or third-person perspective, in which the player takes part of a racing competition with any type of land, water, air or space vehicles. In general, they can be distributed along between hardcore simulations, and simpler arcade racing games. Racing games may also fall under the category of sports games. For the game *Swervin Mervin*, the player controls a car and can vary its speed (increase or decrease it) and control its movements (move right or left). The biggest challenges in this game is to avoid every type of obstacle (to not crash) while keeping the car on the road to complete the track 3 times. The player will have a time limit and every time a lap is completed the timer will reset.

The *Super Mario* (Shigeru Miyamoto and 4, 1985) that was used exists in the Mario AI tournament framework (Khalifa, 2009) and the other three games were published in the Pygame official website (Pete Shinners, 2000).

3.2 Problem analysis

The main purpose of this project is the development of a procedural level generator for 2D games, where the generator's features include: being able to generate valid levels to various games of different genres and to adapt the difficulty of those levels generated accordingly to a player's skill, or ability. Therefore, the most important problems revolve around the generalization of a level generation method, more specifically the rhythm-based approach, the rhythm generation process and the level design. Level design addresses problems such as the definition of rules that allow a good generation of levels, the representation of the level's data for each game, the whole process of generation, amongst other challenges. The geometry generator is the component that will structure and build the level, by having a rhythm as a parameter, making it directly correlated to the resolution of the level design problem.

In our level generator, we need to ensure that the level is feasible, which means that, it is possible to complete (having no impossible challenges) and that it can adapt its difficulty depending on a player's performance. To do that, it's required to implement a DDA technique into our geometry generator, in order to evaluate a player's performance on a specific game and to generate levels of appropriate difficulty. A level's difficulty is considered appropriate when it is not too easy - that can lead to boredom - and not too hard - that can cause frustration - for players in general.

To sum up, considering the objectives of this project, the most relevant problems that we have to deal with are:

- The application of the rhythm-based approach in various games of different genres.
- The rhythm's generation process.
- The implementation of a geometry generator that can translate rhythms into valid levels.
- A definition of difficulty, that will allow the generator to adapt the difficulty of the levels generated accordingly to a player's skill.
- The evaluation of the level generation approach.

Therefore, by taking into account the importance of all the mentioned aspects and challenges, we looked to harmonize them in the solutions we developed, in order to achieve the main objective that was defined for this project, as we describe in the next section.

3.3 Level generation methodology

The whole design of the level generator is based on the idea of generalizing the rhythm-based approach. Therefore, since the beginning of the development phase of the project, this was an essential point that deserved special attention. The main challenge when trying to generalize a level generation approach is to find common grounds between the different games, because there is a huge variety of genres with very distinctive actions that can be performed.

To solve the rhythm-based approach generalization problem it is proposed the definition of classes of actions, in order to use them in the rhythms of a level, rather than actions. Thus, a rhythm will be represented as a sequence of classes of actions, instead of a sequence of actions. A class of action is a group of actions with the same assumed difficulty and it is characterized by a mechanic description, whose purpose is to clarify which keys a player needs to press to execute actions from that class in general. This mechanic description is strongly related to the execution difficulty of the actions since, for instance, it would be easier to perform actions that only require the pressing of one key instead of pressing two keys alternatively. Our idea was to create five different classes, where the first four are groups of actions and the last one represents challenges (special types of in-game content that can either be enemies or obstacles). In the level generation process, since the classes of actions represent groups and not single actions, there is a selection method to chose one possible action from the classes that belong to a rhythm, before creating in-game content.

Using the classes of actions in the rhythms to structure the level generation is the most important tool for the generalization of the rhythm-based approach, because we can always assign actions from a specific game to a class of actions. Thus, making it possible to use the same rhythm to generate a level for various games without any concerns, something that it is complicated to achieve if a rhythm is a sequence of actions, since most games wont have the same possible actions. Organizing the actions of a game into groups, based on difficulty, also allows us to generate levels with rhythms of a specific difficulty value, which will be very helpful when trying to adapt the difficulty of the levels to a player's performance.

Like it was previously mentioned, in our level generation approach the level won't be represented by a sequence of actions but by a sequence of classes of actions. This way, the same sequence of classes can originate different sequences of actions, because most of the time there will be more than one action assigned to each class. For the rhythm generation process, we developed a genetic algorithm whose fitness function is able to evaluate the difficulty of a sequence, based on the classes of actions that belong to it, therefore, the fitness value will correspond to the difficulty value of a rhythm. This evaluation can be done because each class has an assumed difficulty associated to it's mechanic description, thus being possible to sort them by difficulty. Using a genetic algorithm for this process also allows us to produce and search for diverse rhythms for a target difficulty, because there are many sequences that will have the same fitness value.

After obtaining a sequence of classes of actions, we need to translate that into a level. To do this, we use a geometry generator that for each class of the sequence will generate appropriate terrain (level chunks), depending on the generation rules of the game and the current class of actions. The geometry generator will first select an action from the class of actions and then it will generate valid terrain that matches with that selected action. In other words, it will provide terrain that respects all the defined rules and that can only be successfully surpassed by the player if he uses the previously selected action. For example, if the selected action is jump, then it will generate game content that require the player to use the jump action, like some changes in the height of the terrain or some rocks or boxes that are in the way. The goal here is to use the rhythms and the classes of actions as the basis of the level that is going to be created.

In order to adapt the difficulty of the levels, we implemented an AI agent that is going to evaluate the progress of a player in a certain difficulty. This agent checks if the player successfully completed the level, or not, and how good was the playthrough, like for instance, how many lives the player lost, how many points, how fast it was or, in case the player lost, how close he was to successfully completing it. Depending on these results, the AI will adapt the current difficulty accordingly, with the intention of generating a suitable level to the player, based on the displayed skill.

Figure 3.1 illustrates by phases how the level generation process for a specific game works.

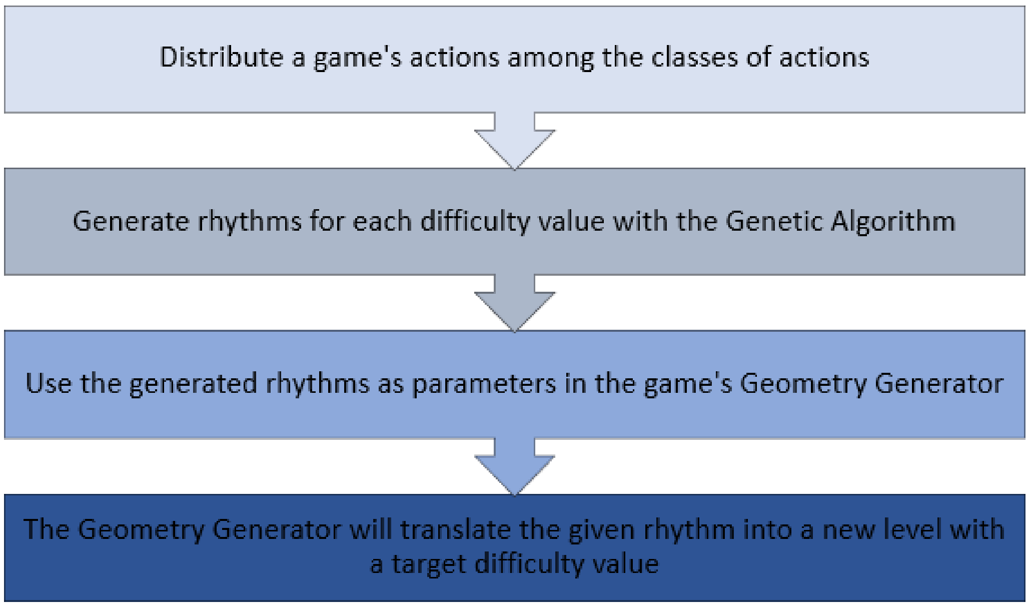

Distribute a game's actions among the classes of actions
Generate rhythms for each difficulty value with the Genetic Algorithm
Use the generated rhythms as parameters in the game's Geometry Generator
The Geometry Generator will translate the given rhythm into a new level with a target difficulty value

Figure 3.1: Generation mechanism in our methodology

In the following sections, the ideas behind the main components of our level generator are explained, including the classes of actions, the genetic algorithm, the geometry generator and the dynamic difficulty adjustment method.

3.4 Classes of actions

The most crucial challenge for this project is to be able to apply the rhythm-based level generation approach to numerous 2D games, mainly as a result of existing various genres that have lots of different aspects that influence the gameplay, from diverse tiles, styles, possible actions, enemies, challenges and so on. The generalization aspect is the most essential problem to solve, since most level generation approaches share this limitation of not being generic enough.

With this said, we made some alterations in the classic rhythm-based approach which allow us to make this generalization to different game genres. To achieve that, we had to find a way to connect the concept of describing a level as a sequence of actions to the common aspects between most 2D games. Since difficulty adaptation is an important part of the generator, it is also a challenge to include it in the problem resolution.

3.4.1 Modified rhythm-based approach

The modifications we suggest for the rhythm-based approach, in order to solve the generalization problem, is defining the classes of actions, which represent a group of actions that have the same assumed difficulty. A class of actions has a mechanic description associated to it that describes how the actions from a class are executed in general (how many keys are pressed, what state the player has to be to execute them, how long should the keys be pressed, etc.). The mechanic description is also directly linked with the difficulty of an action (pressing one key is easier than pressing two keys alternatively, for instance) and, therefore, will help us to organize all the possible actions in a game by difficulty.

The whole point of the classes of actions is to use them in the rhythms, instead of the actions. This way, we will have sequences of classes of actions instead of sequences of actions. The mechanic descriptions will help to identify the assumed difficulty of all the possible actions in a game and group them in the different classes accordingly. Since we have an assumed difficulty associated to each class, it will allow us to evaluate the overall difficulty of a rhythm, by checking which classes are part of the sequence.

With this idea as our basis, we can apply this approach to most game genres, because all games will have some actions easier to perform than others, making it possible to group them this way. The essential factor here is to accurately define the mechanic description associated with each class and to actually correlate a mechanic (what keys to press) to a difficulty value.

3.4.2 Defining the classes of actions

For difficulty in general, it was determined that a class would be ranked in four different categories: very easy, easy, average and hard. Leaving one class that represents enemies or special in-game objects (for example, in racing games the enemies are replaced with obstacles). In a rhythm, these classes of actions are represented with numbers, where 0, 1, 2, 3 and 4 are respectively very easy, easy, average, hard and challenges. The concepts related to each class of actions and it's mechanic descriptions (Table 3.1) can be described as follows.

- **Class 0 (very easy)** - This class will represent the easiest actions in games. Those actions that literally anyone can do, without having skill whatsoever. In some games, this would be the run action.

- **Class 1 (easy)** - In this class, the actions are easy to execute but will have a degree of difficulty greater than the easiest ones. Actions that are easy to execute but can be failed, with normally no repercussions. For example, run and jump vertically to a platform.

- **Class 2 (average)** - The average difficulty represents actions that will need some understanding of the game mechanics. These actions, if failed, will have a major impact on the progress of the game (losing or making it harder to win). An example of this would be to run and jump horizontally over a gap in the terrain.

- **Class 3 (hard)** - The hardest actions are a much more difficult version of the average actions. It will have the same consequences but are harder to execute than the average ones. If for instance, a horizontal jump over a gap is considered an average action then a longer horizontal jump would be a hard action, since it requires the player to perform the jump in a specific point to successfully execute the action.

- **Class 4 (challenges)** - This class will represent a challenge. These challenges can either be enemies, that the player needs to evade or attack, or obstacles, in games that do not revolve around defeating foes. For instance, in a racing game normally there are no enemies (it has adversaries, that can not be killed), therefore, the class will represent obstacles that block part of the road, making the level much harder or challenging.

Classes of Actions	
Class	**Mechanic description**
0 (very easy)	Pressing one key with no specific pressure needed to complete the level chunk OR pressing no keys at all
1 (easy)	Pressing two keys at the same time OR pressing one key but having to consider the amount of time to apply pressure on it, in order to complete the level chunk
2 (average)	Pressing two keys or more with variation between them, having to consider the amount of pressure to apply at every moment, in order to conclude the level chunk
3 (hard)	Equal description of class 2 (average), but also needing a specific past state (position, player state, speed, etc.) to complete the level chunk
4 (challenges)	Represents an enemy that we have to evade or defeat and, in some cases, represents obstacles (special in-game objects)

Table 3.1: Generic mechanic description to each class of actions

3.5 Genetic Algorithm

The rhythm generation process is one important part of our generator, since the rhythms are going to be the base structure of the levels. Therefore, it is required to have a rhythm generator with the ability to provide diverse sequences of classes of actions for each difficulty value and a way of generating rhythms of a specific difficulty value. Another important feature that the rhythm generator must have is being able to accurately evaluate the difficulty value of a rhythm, by checking which classes are part of the sequence and understand that each class of actions will have an assumed difficulty associated to it's mechanic description.

Therefore, the tasks that the rhythm generator must be able to execute are the following:

- Evaluate the difficulty value of a rhythm, depending on the classes of actions that are part of it.

- Generate different rhythms for each difficulty value, in order to promote diversity for the levels with the same difficulty value.

- Generate sequences of classes of actions for a target difficulty, with the purpose of searching for rhythms with a specific difficulty value.

3.5.1 Rhythm generation approach

We propose to use a genetic algorithm to play the role of rhythm generator where the chromosomes will represent the rhythms (inspired by the articles (Moghadam and Rafsanjani, 2017) and (Smith et al., 2009)) and the genes will represent the classes of actions. Using a genetic algorithm has lots of advantages, such as:

- Providing various combinations of sequences of classes of actions, giving the diversity that we intend.
- A fitness function that can be used to evaluate the difficulty value of the rhythms, by defining the right parameters and giving the information related to each class of actions.
- Searching for a specific difficulty value by using the fitness of each rhythm to determine how close that sequence is to a target difficulty.

After the genetic algorithm generates sequences of classes of actions that are appropriate for a specific difficulty value, they will be saved in a text file, so the geometry generator can use them in the level design. The idea is to have a set of rhythms for each difficulty value, to have some variety in the levels generated. The difficulty values can go from 0.0 (easiest) to 1.0 (hardest). Depending on the player's skill, one of these difficulties will be used (at the start it has the value of 0.5), so the rhythms from these difficulties can be selected and be used as parameters in the geometry generator. Depending on the overall performance of the player, the difficulty value will be updated, with the purpose of generating levels with the appropriate challenges for a specific player. Diversity on the levels generated by the same rhythm is also assured, since an action is randomly selected from a class in each level generation and because there are several types of level chunks that can match with a selected action.

One important part of this algorithm is its fitness function. The fitness function needs to accurately evaluate a sequence, in terms of overall difficulty. The function has, therefore, the following criteria to assess the rhythm, based on our definitions:

- **Variety of classes of actions** (V) - The function needs to evaluate the variety of classes of actions that exist in a rhythm. Needs to acknowledge that the more variety a rhythm has the harder it will be, because there will be less repetitive actions. Consequently, the more repetitive the easier it gets. Of course, repetitive hard actions won't be easy, but it would be harder if there was some more variety.
- **Number of classes of actions** (N) - This criterion is going to count how many classes of actions there are in the sequence that are not easy, which means that, it

will only count the most difficult classes (average, hard and enemies) and in the end, we have a final value that will be the sum of the number of ”hardest” classes in the sequence. This final value is meant to be maximized in the harder levels and to be minimized in the easier ones, with the objective of having a greater number of easy actions at easier levels.

- **Difficulty variance** (C) - This is a very important parameter of the fitness function that evaluates the difficulty of a rhythm. The way the function works is by going through a rhythm and, for each sequence of two classes of actions, a value of difficulty is assigned. After reaching the end of the sequence, all the assigned difficulty values are summed, resulting in the difficulty value associated to that rhythm. The difficulty values for each pair of classes of actions are numbers (the greater the number the greater the difficulty) and are based on the mechanic descriptions associated to each class. For instance, a sequence from the very easy class to the easy class will have a smaller value (which means, it is easier) than a sequence from the hard class to the average class.

3.5.2 Genetic algorithm implementation

In a genetic algorithm, to generate a new population, some operations that help to simulate natural selection will be needed. The main ones are the selection method, the *cross-over* and mutation operations. Next, we discuss how each of those operations, from this algorithm, work and how they are combined to simulate natural selection in the generation of a new population.

The process of the developed genetic algorithm can be organized in the following five phases:

1. **Initial population** - We initialize the population’s chromosomes with random sequences of classes of actions (numbers from 0 to 4). The only limitation is not placing more than one class 4 in a row and also not placing it in the last gene (class 4 indicates that next level chunk will have a challenge, so it needs other classes to be valid).

2. **Fitness function** - The fitness function will determine the difficulty value of each chromosome and depending of the target difficulty it will be assigned a fitness value. In other words, the closer the difficulty value of a chromosome is to the target difficulty the greater its fitness value (that goes from 0 to 1).

3. **Selection** - The better the fitness of an individual, the greater the probability of that individual to be placed in the mating pool. It will work like the roulette wheel selection method, mentioned in the Related work chapter. Therefore the selection

is random and the individuals with better fitness have a greater chance of being selected, since the number of times that the individual appears in the mating pool is directly proportional to its fitness value.

4. **Cross-over** - The *cross-over* operation will take two chromosomes from the mating pool and then randomly select a cutting, or midpoint. After selecting that cutting point, it will mix the first half of one of the chromosome and the second half of the other, resulting in a new individual.

5. **Mutation** - The mutation rate will be equal to 1%. The mutation operation works by randomly choosing one gene from the chromosome and randomly change it, between the possible numbers (the classes of actions).

The algorithm terminates if the chromosome with the fitness value equal to 1 is found (meaning that has the target difficulty) or if the maximum number of generations is reached, which are 100 in our case. In the end of the execution, the genetic algorithm will provide information about the last generated population, such as:

- The best chromosome.
- The fitness value of the best chromosome.
- The average fitness of the population.
- Total of generations.

Defining the fitness function and its parameters

The parameters were discussed previously, where we determined that they would be: the variety of classes of actions (V), the number of classes of actions (N) and the difficulty variance (C). Below we have the formulas used for the calculation of each parameter (Moghadam and Rafsanjani, 2017) and, for the case of the difficulty variance parameter, we also have a table to illustrate the difficulty value to each pair of classes of actions.

$$N = \sum_{RLen}^{i=0} f(x_i), \text{ where } f(x) = \begin{cases} 1, & \text{if } x \neq 0 \text{ and } x \neq 1 \\ 0, & \text{otherwise} \end{cases}$$

Where $RLen$ is the length of the rhythm and x_i is the class of action number i in the rhythm, that is not easy (can be class 2, 3 or 4). For each class of a rhythm that is not easy, the total value is incremented by 1.

$$V = \sum_{RLen}^{i=0} |\overline{x} - x_i|^2$$

Where $RLen$ is the length of the rhythm, x_i is the class of action number i in the rhythm and $\overline{x}$ is the mean of the classes of actions in the rhythm.

$C = \sum_{RLen}^{i=0} |difficulty(x_i, x_{i-1})|$

Where $RLen$ is the length of the rhythm, x_i is the class of action number i in the rhythm, x_{i-1} is the class of action number $i-1$ in the rhythm and $difficulty(x_i, x_{i-1})$ is the difficulty variance value when executing an action from the class of action x_i and then an action from the class of action x_{i-1}. The following table 3.2 illustrates this values, where the numbers represent each class.

Difficulty variance	
$difficulty(x_i, x_{i-1})$	**Difficulty value**
x_i = 0, x_{i-1} = 0	0
x_i = 1, x_{i-1} = 1	1
x_i = 2, x_{i-1} = 2	3
x_i = 3, x_{i-1} = 3	4
x_i = 0, x_{i-1} = 1 or x_i = 1, x_{i-1} = 0	1
x_i = 0, x_{i-1} = 2 or x_i = 2, x_{i-1} = 0	2
x_i = 0, x_{i-1} = 3 or x_i = 3, x_{i-1} = 0	3
x_i = 0, x_{i-1} = 4 or x_i = 4, x_{i-1} = 0	2
x_i = 1, x_{i-1} = 2 or x_i = 2, x_{i-1} = 1	2
x_i = 1, x_{i-1} = 3 or x_i = 3, x_{i-1} = 1	3
x_i = 1, x_{i-1} = 4 or x_i = 4, x_{i-1} = 1	2
x_i = 2, x_{i-1} = 3 or x_i = 3, x_{i-1} = 2	4
x_i = 2, x_{i-1} = 4 or x_i = 4, x_{i-1} = 2	5
x_i = 3, x_{i-1} = 4 or x_i = 4, x_{i-1} = 3	5

Table 3.2: difficulty variance for a sequence of two classes of actions

Regarding the fitness function, various tests were made to the generated rhythms, to conclude which of the functions produced better results and, also, to validate the parameters that were used. The fitness function F that was initially used is:

$$F = C + V + N$$

To prove that this function provided the best results, it was tested some variations of it. The goal was also to give insights about the most crucial parameters, in order to give more importance to them. The fitness functions that were tested are the following:

- $F = C$ - In this function, the only parameter is the difficulty variance. The rhythms generated had the following characteristics: some hard actions in the easiest difficulties; Perfect fitness was sometimes found in the first generations, that resulted in having no mutations (since there is no offspring) and had a very little variation of actions in the harder difficulties, resulting in sequences of the same classes of actions. Since the only parameter used is the difficulty variance, there is no control over the number of hard actions in the easiest difficulties and no control over the variation of the actions overall.

- $F = V$ - This function has the variation of actions as its only parameter. The main problem of this function was having random difficulty in all rhythms, because the only criterion used was having few variations of actions in the easiest difficulties and lots of variations in the harder difficulties. This parameter alone is inconsequential, because there is no definition of difficulty.

- $F = N$ - For this function, the only parameter is the number of classes of actions. The features of the rhythms generated by this function are: having lots of sequences of the same classes in most rhythms and some easy actions in the harder difficulties. Overall, it had better results than the function $F = V$ but still had lots of problems. These problems happen because there is no definition for the difficulty of a sequence of actions and no variation criteria.

- $F = C + N$ - In this function, we have two parameters: the number of classes of actions and the difficulty variance. Some features that were noticed in the rhythms generated are: repetitive rhythms - very common to have sequences of the same actions - and very little variation in the harder rhythms. This happens because there is no variation criterion, so, the rhythms generated only take into consideration the difficulty of the classes and the difficulty variance of the sequences. This results in very similar rhythms with almost no variation.

- $F = V + N$ - This function has two parameters, the variation of actions and the number of classes of actions. This function provides rhythms with good variance in harder difficulties and generates valid easy rhythms. It still has a problem, since there is no definition of difficulty for the sequences of actions, the rhythms generated have some easy actions in the harder difficulties and some hard actions in the easier difficulties.

- $F = C + V$ - For this function, we have two parameters, the variation of actions and the difficulty variance. From the previous functions, this is the one that generates the best rhythms overall. The only problem that occurred was having some hard actions in the easier rhythms and sometimes there were easy actions in the harder rhythms, since we do not have the N parameter. The number of classes of actions solves those issues.

- $F = C + V + N$ - This is the function that was initially used, having all the parameters that were discussed. This is the best function in terms of rhythms generated, with the best chromosome's fitness around 0.9 (close to perfect fitness), most of the time. It respects the difficulty variance definition, has a lot of variation in the harder difficulties and low variation in the easier difficulties. It is also important to note that there were no hard actions in easier difficulties and no easy actions in the harder difficulties. The only problem that was identified is that sometimes repetitions of three or two classes of actions in a row occurs in the harder difficulties. This function is the complete version of the previous one, $F = C + V$.

Since these tests were made to prove that the best results happen when using this initial function and to find which of the parameters were the most crucial, the initial function was modified by giving more importance to those parameters. After analyzing the results of the tested fitness functions, the most important parameters of the function are V (variation of classes of actions) and C (difficulty variance). With this information a new fitness functions was tested, by giving more importance (multiply them by 2) to these parameters or to both. It is possible to see that the function that had the closest results to the initial function was the $F = C + V$, because of this, even though the parameter N is important, it is less important than the others, according to the results of the tests.

Thus, it was given more importance to those parameters, to check if the results were better. The fitness functions tested were based on the variation of the importance of the variables w_1, w_2 and w_3 from the following function:

$$F = w_1C + w_2V + w_3N$$

- $F = w_1C + w_2V + w_3N$ where w_1 = 1, w_2 = 2, w_3 = 1 - This function gives more importance to the variation parameter. The resulted rhythms are overall good, with the best chromosome's fitness always around 0.9999, which means they are very close to the perfect score for that target difficulty. Although it had better hard rhythms than the initial function used, it still can improve, because sometimes it generates rhythms that are very similar in the easiest difficulties and a few easy actions in the harder difficulties. It had good results, but since the criterion that was given more importance is not related to difficulty, the appearance of some easy actions in the harder levels is somewhat expected/justified.

- $F = w_1C + w_2V + w_3N$ where w_1 = 2, w_2 = 1, w_3 = 1 - For this function, it is given more importance to the difficulty variance parameter. It generated good rhythms, with the easier rhythms very similar to the initial function, which is good. The best chromosome's fitness is, as the previous function, always close to 0.9999. The only problem found in this function is the same problem found in the initial function:

the repetition of three or two of the same class of actions in the harder difficulties. This is a better version of the initial function, since the best chromosome's fitness is better (initial has 0.9 and this one 0.9999), but the problem that needed correction remains.

- $F = w_1C + w_2V + w_3N$ where w_1 = 2, w_2 = 2, w_3 = 1 - In this function, it is given more importance to the difficulty variance and the variation parameters. Like the previous two functions, the best chromosome's fitness values are always close to 0.9999, but it generated much better harder rhythms, with much more variation, with the cases of repetition of classes happening rarely and normally not more than two. The easiest rhythms generated are similar to the initial function, which is good. This function generated very good rhythms overall, with no noticeable problems, based on our definitions of difficulty. This function is, therefore, the best, by using these parameters. This is somewhat expected, since C and V were the most important parameters, so, it is logical to have the best results by giving more importance to both of them.

Being the fitness function the most important element of a genetic algorithm, it was crucial to have the best possible fitness function, by having the essential features of the rhythms for each difficulty as a basis. This was the main reasoning behind the number of tests made, with various fitness functions to analyze.

These tests helped to conclude that the fitness function that produced the best sequence of classes of actions was the function:

$$F = w_1C + w_2V + w_3N \text{ where } w_1 = 2,\ w_2 = 2,\ w_3 = 1$$

Since it provided the characteristics that were intended. In the easy difficulties, it had lots of easy actions with very little variation and the harder it gets, the harder the actions in the rhythms and the more variation between actions there will be. This, therefore, fulfills the goals for the generation of the rhythms.

3.6 Geometry Generator

Level design problems aim to answer the question: "What should we generate?". In this case, the goal is to generate valid levels for 2D games, where there is a group of suitable challenges for the player. By suitable challenges it means that the resulted levels have achievable objectives, which in 2D games, normally mean that, it is possible to reach the end of the level (Tanya X. Short, 2017).

Another important ability that we want to integrate into our geometry generator is that, whenever possible, generate some visual or gameplay variety, trying to make as many changes as possible to the terrain generated, without looking too random. This will avoid repetitive levels that will encourage the replayability of the game, since it will, in most cases, generate different terrain/challenges.

For instance, some of the main problems regarding level design in platform games (a 2D game genre), include generating appropriate heights or gaps in the terrain, that the player can reach by using in-game actions, like jumping, for this specific case. In other words, to have a good generation, the level design must focus on generating terrain based on rules, that help us to define terrain limitations. Without those rules, it would most likely produce impossible levels for the players, which is something we want to avoid.

Thus, the most important steps we need to take, to successfully implement a rather good geometry generator are: decide what rules should we define for the level design, how the generation process would work and how to represent the data of the generated level.

3.6.1 Geometry generation approach

As previously discussed in this subsection, level rules are an essential aspect for procedural level generation since it defines how the building, or structuring, of the level's content should be made.

Rules exist to make a level more believable rather than completely random. Therefore, the main purpose is to limit the terrain generation, so it allows the geometry generator to produce suitable and valid levels. Rules that are related to the gameplay are going to control features that impact the feasibility of the level generated (should not generate impossible challenges, enemies impossible to defeat, amongst other game-changing factors), whereas rules related to sprites used for decoration are made for aesthetic reasons. This will grant game content placement in interesting ways without blocking the player's movements, which would result in an unfeasible level (not possible to complete due to the poor placement of the sprites, for example) (Tanya X. Short, 2017).

Since the rules are such an important element, one of the problems that need solving, regarding terrain generation, is the definition of valid rules for each game, in order to, not only making it visible appealing but also making the levels generated possible to complete. Our intent is also to establish rules that make the content generated somewhat similar to real levels from the original game. Thus, the main rules we defined (by having the book (Tanya X. Short, 2017) as reference) to solve our generation problems are the following:

- **Gameplay rules** - Rules that allow the generator to structure feasible challenges for the level generated, based on the possible actions that the player can execute with the controllable character. From generating the terrain in the correct heights (not too high that blocks the window, or too low, for example) to generating game challenges that can be successfully completed by executing a certain action or combination of actions (for instance, gaps in the terrain with a length that allows the player to jump across or, in racing games, making a turn that can be done without going out of the track).

- **Aesthetic rules** - The main purpose of these rules is to make the level visibly appealing, by mainly using the original levels from the game as a basis. This includes matching terrain tiles that if put together make the illusion of ”real” terrain (for example, if we have grass tiles we should put them together and not mix them with ground tiles or sometimes if there is a cliff we have certain tiles that should be placed on the edge to indicate that). It all revolves around making the level believable for the players, without using random sequences of tiles to build the level’s terrain. Another use for this type of rule is to decorate the level with sprites in different ways for each generation to make the levels more interesting, with the concern of not blocking player’s movements with certain objects incorrectly placed (platforms that block a jump, for instance).

- **Enemy disposition rules** - These rules define how and where the placement of enemies should be, depending on their characteristics. Since there are, in most games, different types of enemies with a diverse set of attacks/abilities, they should be placed accordingly to generate suitable challenges in the level. In some games, there are also enemies that remain static and others that run around. Due to this, we need to place them in specific terrains that allow the enemy to have an impact in the difficulty of the level (for example, if we put an enemy that runs around near a cliff he will most likely fall down).

- **Probability rules** - Rules that define the probability of certain game content to appear in the generated level. These contents can be items, like bonuses or power-ups, collectibles or even tiles that have an impact on the difficulty of the level, because in certain games some terrain tiles end up making the level more difficult by having smaller collision boxes (for instance, in *Super Mario* (Shigeru Miyamoto and 4, 1985) we have normal ground terrains and green platform terrain tiles that only have a collision box on the top side, making the jumping much harder).

- **Level objective rules** - Objective rules focus on guaranteeing that the objectives can be attainable so the player can progress. These rules sound similar to the gameplay rules but are very different, primarily because the gameplay rules are related to the terrain generation. Level objective rules relate to other aspects of the game that allows it to be feasible, or possible to progress. These aspects include, for instance, the time limit, that should be enough for the players to complete the objectives, but not too long, to avoid making the game too easy.

- **Handmade content rules** - We need these rules to create desirable outcomes by relying on some handmade content, that represents standard level chunks that we want to include within the procedural generations. For example, if we have a level chunk that can be completed using the horizontal jump action we want to always make sure that there is a landing spot on the other side, so we need to describe that in the rule-set. This last example is directly related to gameplay rules but we also need handmade content rules related to the aesthetics. For instance, we can define a sequence of platforms or tiles that can appear in the generation.

Generation Process

The main concerns regarding the generation process are the definition of what should be integrated, when generating a level, and how to accurately create those in-game contents. In our approach, the rhythms and the previously mentioned generation rules act together in the geometry generator, in order to generate valid levels. The rhythms will work as parameters for the geometry generator, with the goal of translating the classes of actions from the sequences into level chunks, a group of sprites that put together will form a piece of the level. Because of this, the level is generated as we go through the rhythm, by joining the level chunks that are being created from the classes of actions in the sequence.

The method used for the level chunks generation can be described as follows:

1. Identify the class of actions
2. Select one action that belongs to that class
3. Identify which types of terrains and contents that match with the selected action
4. Generate a level chunk that matches with the selected action

The implemented method will first identify the class of actions, in order to randomly select an action that belongs to it. Then, based on the action that was selected and the generation rules, a level chunk is created. The level chunk must match with the selected action, meaning that, it can be successfully completed by executing the selected action.

After going through all the classes of actions from a rhythm and joining all the generated level chunks together, the new level will be decorated with different sprites that are part of the game. These sprites include items (like power-ups or bonuses), collectibles and decorative sprites (trees, rocks, etc.) that will have a probability to appear based on the difficulty value of the level.

We do an online generation where every time we load a game it will procedurally generate a level's content based on the approach described previously. This is the main reason for making the generation process simple, so it can be made in real-time without any complementary issues.

Data Representation

How to save a level's data after the generation process is an important problem to solve if we want to replay a level. Our goal is to keep data as simple as possible, by using the most appropriate format, according to the game in question. Mainly we are using open-sourced games, therefore, to represent game content we will always analyze how the code of the game is structured and, with that information, understand which data representation makes more sense. In most cases, the levels are saved in text or *CSV* files, where their contents (terrain, items, enemies, etc.) are represented by lists of integers, where each number, or combination of numbers, matches with a specific in-game content. Even though representing a level's data with lists of integers is applicable to most cases, there are some games where there is no ”saving mechanism”, making it impossible to replay a level. We can conclude that the data representation problem for levels can be solved by using lists of integers and by analyzing the game's code, in order to - as mentioned before - keep it as simple as possible, since this is a fundamental concern when generating various levels for different games.

3.6.2 Geometry generator implementation

The geometry generator's main objective is the conversion of classes of actions to level chunks, a group of in-game contents. For each of the open-sourced games that we applied our level generation methodology, a geometry generator was implemented, to create valid level chunks, depending on the actions and sprites of that game. Thus, the geometry generator is tasked with the action selection and terrain generation, based on the class of actions and the generation rules. Since each class, normally, contains more than one action, the generator will first randomly select one of the possible actions from that class. Afterwards, depending on the selected action, the geometry generator will identify which in-game contents are appropriate, in order to create a level chunk that can successfully be completed if the player executes the selected action.

For this process, it is assumed that the division of actions into the different classes of actions is already done for a specific game. Subsequently of this procedure, the geometry generator is going to match possible level chunks to each action from all the classes of actions. It is required to know, therefore, the game's sprites (tiles, objects, enemies and other contents), in order to have the most correct and diverse level chunks for each action, avoiding generating always the same terrains to the same action. For instance, the action vertical jump can be matched with a level chunk that contains a change of the ground's height, or various objects that could be blocking the way, amongst other options. This grants us a huge diversity in the level generation because, first, the actions are randomly selected from the class of actions and the level chunk, that matches with that action, is also going to be randomly selected, making the generations as diverse as possible. Even to the same level chunks, there are lots of factors that can be changed, like for example, the height, the size, the tiles used, etc. The process that transforms a rhythm into a new level can be described in figure 3.3 as follows:

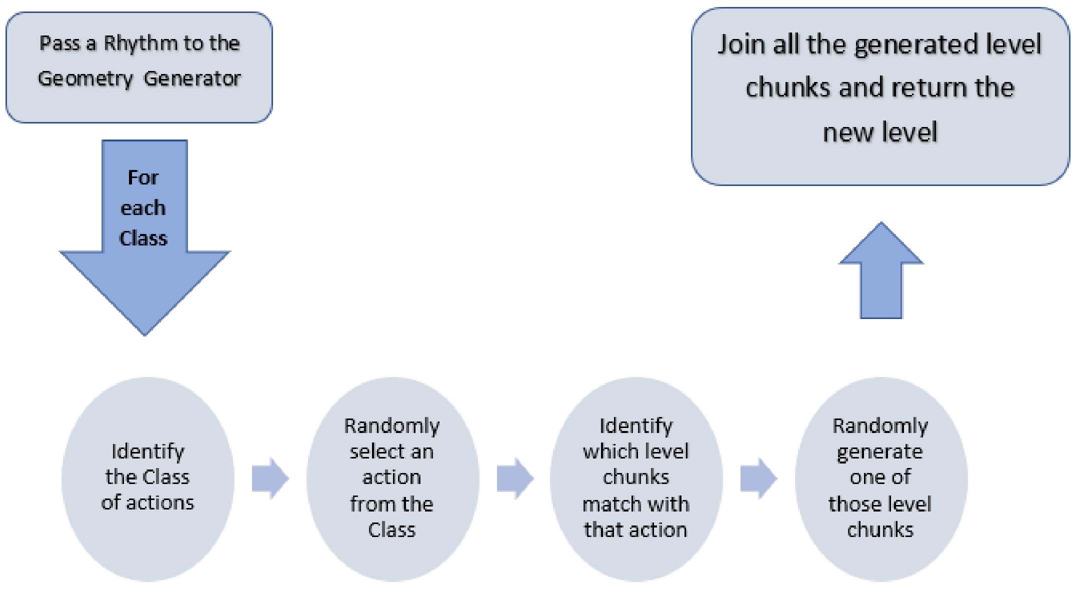
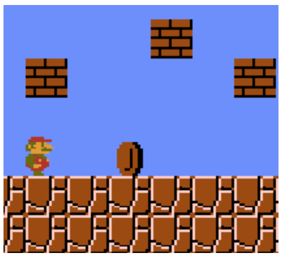

Pass a Rhythm to the Geometry Generator
For each Class
Identify the Class of actions
Randomly select an action from the Class
Identify which level chunks match with that action
Randomly generate one of those level chunks
Join all the generated level chunks and return the new level

Figure 3.2: Generation process in the geometry generator

The geometry generator also has an important role in the level's difficulty adaptation. The smaller the length of the terrain, the harder it will be, since it takes more precise movements from the player to successfully complete a level chunk. Thus, the length of the terrain generated will be directly related to it's difficulty (the harder the difficulty, the smaller the length of the terrain). These changes on the terrain can be applied to any level chunk, allowing the generator to vary the difficulty of all the actions in a game. For instance, if the action is the horizontal jump the possible changes could be to vary the size of the landing spot (platform, ground, etc.) and if the action is the turn right action (in racing games) the tightness of the curve can vary. The class of the challenges (class 4), is a special case, since it is generating enemies or obstacles, and not actual terrain. For this specific class, based on the value of the difficulty and by using the help of an AI agent (DDA technique), the enemies are adapted to the current difficulty, by giving them proper attributes (damage, health, weapons, speed, etc.) and varying them accordingly. When the game does not revolve around enemies and has obstacles instead, the number of obstacles that are generated in a level chunk is changed (the harder the difficulty, the more obstacles are generated).

The following figures illustrate some examples of level chunks generated for each class of actions for the *Super Mario* (Khalifa, 2009) game.

Class 0 (very easy)


(a) Run action

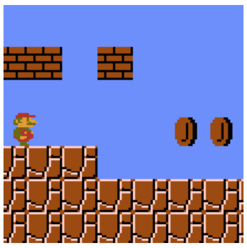
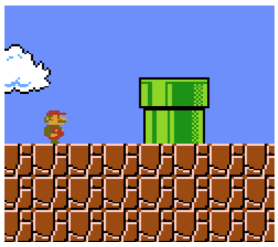

(b) Fall action

Figure 3.3: Examples of generations for class 0, by generating plain terrain (for run action) and by generating a decrease in height (for fall action)

Class 1 (easy)


(a) Vertical jump action

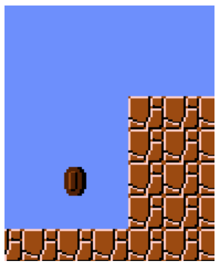

(b) Vertical jump action

Figure 3.4: Examples of generations for class 1, by generating a pipe or height increase

Class 2 (average)

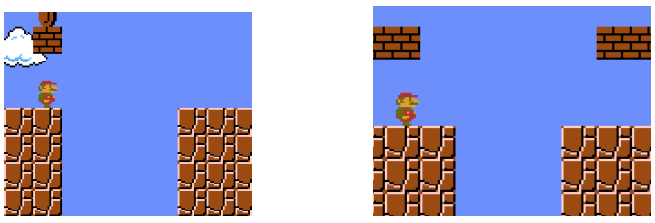

(a) Horizontal jump action (b) Horizontal jump action

Figure 3.5: Examples of generations for class 2, by generating a gap on the terrain

Class 3 (hard)

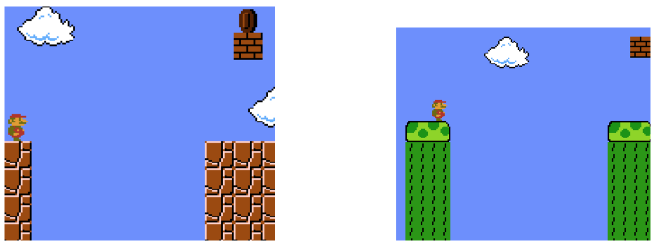

(a) Hard horizontal jump action (b) Hard horizontal jump action

Figure 3.6: Examples of generations for class 3, by generating a big gap on the terrain

Class 4 (challenges)

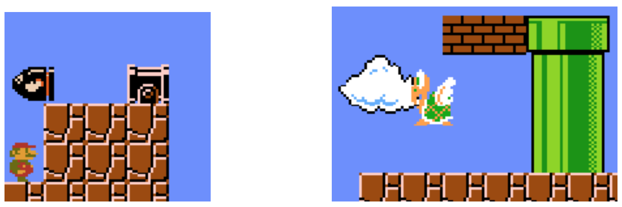

(a) Bullet Bill (b) Flying green Koopa

Figure 3.7: Examples of generations for class 4, by generating enemies

By joining these generated terrains (level chunks) a piece of a level is composed. For instance, the rhythm [4,1,1,0,3] could result in something like the figure 3.9 illustrates (in another execution it generates something different that also matches this sequence of classes of actions).

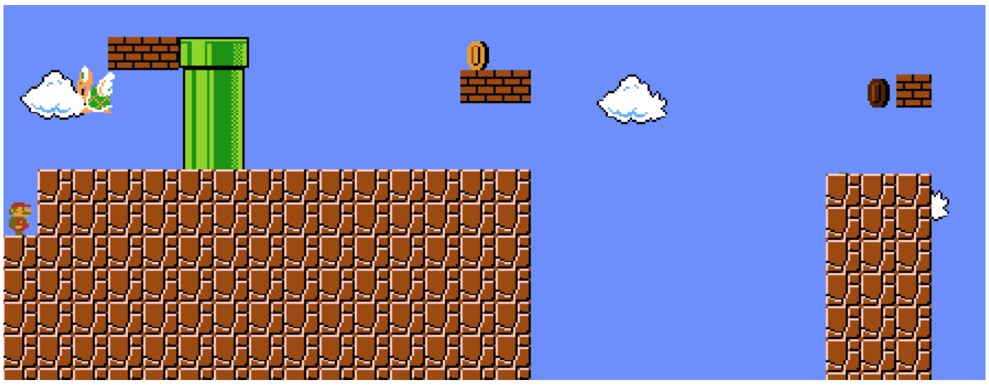


Figure 3.8: Example of a piece of a level for the game *Super Mario*

Since the enemies are not actual terrain, having the class 4 in the sequence means that the next level chunk generated will have an enemy. This is why it is not possible to have two class 4 in a row or at the end of the rhythm.

Thus, in the previously illustrated rhythm [4,1,1,0,3], what the geometry generator perceives is that for the first terrain that matches the vertical jump (class 1) an enemy should be generated, because it has a class 4 before the class 1. The full interpretation of the rhythm [4,1,1,0,3] for the game *Super Mario* is described as follows:

- **Current class of action = 4** - Identifies that the current class of action is the class 4 (challenges). The geometry generator knows that the next level chunk will have an enemy (in this case it will be terrain that matches the class 1)
- **Current class of action = 1** - Identifies that the current class of action is the class 1 and that it should generate an enemy in that terrain (since the class 4 was before this class 1), so it generates terrain that match the vertical jump action and also generates an enemy.
- **Current class of action = 1** - Identifies that the current class of action is the class 1, so it generates terrain that match the vertical jump action
- **Current class of action = 0** - Identifies that the current class of action is the class 0, so it generates terrain that match the run or fall action (selects one randomly).
- **Current class of action = 3** - Identifies that the current class of action is the class 3, so it generates terrain that match the hard horizontal jump action.
- **Ends loop through the whole rhythm** - Joins all the level chunks (or terrains) in the order they were generated, decorates the new level (platforms, coins, etc) and returns it.

3.7 Dynamic Difficulty Adjustment

A feature that we require from our procedural level generator to have is the capacity to adapt the level's difficulty depending on the skill, or ability, of a player. Thus, the problems that we need to solve are: what DDA technique should we use, how to evaluate the skill of a player and how can the generator adapt correctly.

The DDA technique should be able to evaluate the performance of a player in a certain difficulty of the game and be able to adapt the level's features by taking some factors of the player's gameplay into consideration. Factors like, for instance, the time the player took to complete the level, how many lives did he lose, how many points he acquired, how many enemies were killed and so on.

3.7.1 Difficulty adaptation approach

To solve the challenges related to the difficulty adaptation we analyzed the most common gameplay features of 2D games that influence the performances of players, in order to conclude which of them we would give more importance, when making an adaptation. The following table 3.3 illustrates the importance assigned to each aspect of a player's gameplay, to evaluate the overall performance in a certain game.

Player's Performance	
Gameplay features	**Weight assigned**
Completion Time	+3
Life Count	+2
Points Acquired	+1
Enemies Killed	+1

Table 3.3: Weight assigned to each gameplay features

We want to give more importance to the time the player took to complete the level, since it is applicable to every game and we can normally assume how challenging the level was for a player, based on that. Life count is also important in the determination of the player's performance, because it is directly related to how many times the player failed an action, but not all games will have a life count, for that reason we give a bit less importance to this factor. The remaining two aspects can be taken into consideration in some games, but not in most cases, since the generation of enemies or items/bonuses (that result in points) is mainly made randomly and it is very dependable of the current level's difficulty (the more enemies the greater the value of difficulty, the more bonuses the easier the difficulty and so on), so it is not possible, in most game genres, to infer anything relevant related to the player's skill, based on those game factors. Regarding the cases where the player loses, we will only take into consideration how close the player was to complete the level and decrease the difficulty accordingly.

These gameplay features are directly related to the game's genre, since, as it was mentioned, not all games will have a life count or enemies, for instance. Thus, it is required to consider the main objective and style of a game, in order to make a proper interpretation of what aspects can be used to correctly evaluate a player's performance. How the DDA technique should adapt the difficulty value of the levels generated is defined not only by a game's genre but also by our definitions of difficulty (mechanic descriptions of the classes of actions, the matching between level chunks and actions, difficulty adaptations of the level chunks, etc.). Our definitions of difficulty are a fundamental component for the adaptation of the levels to each player, because it is going to associate in-game contents and actions to a certain value of difficulty. Therefore, the DDA's capacity to make valid difficulty adaptations to the generated levels is directly correlated to the successful implementation of the geometry generator and the classes of actions.

3.7.2 DDA implementation

The implementation of the DDA is based on an adaptive AI that is going to automatically adjust a game's behaviors and its parameters in real-time depending on the displayed skill of a player.

In all the games, the first level's difficulty has a value of 0.5, a difficulty that is average, being appropriate (not too easy and not too hard) for a player in general. Depending on the performance, the difficulty value will increase or decrease. How much will increase and decrease will also vary.

If a player wins/overcomes a level, the value will increase. Relying on how good the gameplay was, it will increase more or less. The factors that are analyzed will directly depend on the game, but, in general, the aspects that are mainly taken into consideration are: how fast the player was and how many times the player lost a life. In racing games and other genres that do not have life counts, the time that the player took to complete the level is the main evaluation resource. In games that have life counts, it is possible to assume that the more lives a player has the fewer times he failed, meaning that, the better the performance was. In games like *Super Mario* (Shigeru Miyamoto and 4, 1985), that have collectible items (like coins), the DDA could take into consideration the number of items collected (coins), but it was opted to only analyze how fast the player completed the level, because there was not a fixed number of coins generated (it is random), so it would not be a very accurate way of measuring skill, which is a very subjective concept. For instance, if there are two levels with the same difficulty generated but one had more coins generated than the other it would be a mistake to assume skill based on something completely randomized. Since there were more coins in one of the levels than the other, there was a greater chance of evaluating the player as skilled, which would be a wrong inference for this case. The only way that it would make sense to take the collected items into consideration was if the number of items was the same for levels of the same difficulty, which is not the case.

If a player loses a level, the decrease in the difficulty will also vary. The factors that were taken into consideration here is how close the player was of successfully completing the level. This can be done by checking in which part of the level the player lost, since in most games the levels are composed by a list of rhythms that are organized from the first rhythm to last one, thus, having a way of dividing the level into parts, or blocks (levels are composed normally by three or four rhythms, where each rhythm represents a game chunk). In racing games, one rhythm is enough, because of the concept of laps. In these games, it is checked in which lap the player lost and the more laps the player completed, the smaller the decrease in difficulty it will be.

The values increase or decrease between 0.3 and 0.1 (difficulty value goes from 0.0 to 1.0). By using the DDA, the generator will pick the most appropriated value to increase or decrease. This will be directly related to the progress of the player, by taking into consideration the factors that we mentioned previously. This allows players that ”over-performed” or ”under-performed” to get a significant change in the current difficulty that they are playing and giving a small change to the players that barely win or lose a level.

In the following table 3.4, it is illustrated which gameplay aspects are going to be taken into consideration for each tested game.

Player’s Performance	
Game	**Gameplay aspects**
Super Mario	Completion Time
Swervin Mervin	Completion Time
Rambit	Life Count
CaveCopter	Level Completion

Table 3.4: Primary gameplay aspect that assesses a player’s performance for each game

Since both *Super Mario* (Khalifa, 2009) and *Swervin Mervin* (Buntine, 2017) revolve around timers, the time it took the player to complete the levels was the best way to evaluate the performance. For *Rambit* (Oliveira, 2011), since it does not have a timer, the only aspect to evaluate the quality of the playthrough is the life count (the playable character has 3 lives in total). *CaveCopter* (Nolte, 2010) is a special case, because it does not have a timer nor a life count, so the difficulty value will always vary by 0.1, that will depend exclusively on the player's ability to complete the level (if the player can complete the level, the difficulty value is increased by 0.1 and if not, it is decreased by 0.1).

3.8 Evaluation of the approach

To validate our definitions of difficulty, the DDA's ability to correctly adapt the difficulty of the generations and to verify the capacity of the modified rhythm-based approach to generalize, we need to pick the right tests to do such tasks, so our evaluation can be clear, regarding the success of the work developed. It is also essential to make the correct tests to make an assessment of future work and the improvements that the generator needs.

The generalization of the rhythm-based approach is our main concern, so it is important to apply the approach to different games, in order to evaluate the quality of the generations. We also had to define our notion of difficulty and therefore, this characterization is a decisive part of our work that needs the "support" of these tests, because difficulty is a very subjective concept, that will vary from person to person, being necessary to find a difficulty definition that favorably suits the majority of people.

With this in mind, it was determined that three tests were needed to evaluate these features of the generator.

- **Generalization test** - This test aims to evaluate the ability of the modified rhythm-based approach to generalize. In other words, the objective of this test is to check if the sequences of classes of actions that the approach provides are generic enough for most 2D games. To confirm this, it is required to successfully apply it to at least four different games with different genres. The games which we tested our level generation approach were: *Super Mario* (Khalifa, 2009), *Rambit* (Oliveira, 2011), *CaveCopter* (Nolte, 2010) and *Swervin Mervin* (Buntine, 2017).

- **AI test** - For this test, we are going to have the A* search algorithm (Millington, 2019) - one of the controllers available from the Mario AI tournament framework (Khalifa, 2009) - as our *benchmark*, which is going to play through the levels generated to the *Super Mario* game, allowing us to assess the success rate for each difficulty. It will help us to see the consistency of the levels generated for each difficulty and will help us conclude if the DDA is able to adapt the in-game content appropriately for each difficulty. If the DDA was correctly implemented, the algorithm should have more success rate in the levels that are easier to complete than it does in the harder ones.

- **Human player test** - We want to have a test that uses the help of people to analyze the feasibility of the levels generated and to verify how generic the difficulty definitions were for most players. The test consists on letting people play several levels from the games where the generation approach was applied and ask them to sort the levels from easiest to hardest, to compare that to the actual difficulty and prove that the definitions of difficulty are correct for most players. This topic is very important, since a subjective concept (difficulty) was being evaluated. Furthermore, this test will hopefully provide us the feedback we need to make assumptions about which improvements the generator needs overall and future work in this area, depending mainly on the player's opinions regarding the accuracy of our definitions of difficulty and the quality of the levels generated.

Chapter 4

Evaluation tests and results

In this chapter, we are going to present the tests and the results obtained by our level generation methodology, in order to prove that it is working as intended. The purpose of this chapter is to confirm that our level generator approach is able to generate valid levels for different 2D games and is capable to adapt the difficulty of the levels generated accordingly.

Validating our definitions of difficulty was a crucial point of focus for some of these tests, since it is a very subjective concept. These definitions include: the assumed difficulty associated to the mechanic descriptions of the classes of actions, matching level chunks with actions and difficulty adaptations to the level chunks. We considered them the most complex and subjective parts of the project, because what is considered hard for a player might not be for another, even for players of similar experience or skill, in a specific game. It was important to have an AI element capable of working as a *benchmark* for these tests and also having a control group to evaluate the levels, by playing them and giving their feedback. First we tested our level generation approach by applying it to different games, in order to evaluate the feasibility of the levels. Then, by using some of the generated levels as examples, we performed the remaining tests.

4.1 Generalization test

In this section, we present the results and explain how the implementation of the approach was made for each of the tested games, in order to evaluate if the sequences of classes of actions provided by our approach are generic enough for most 2D games.

By taking into consideration the level generation approach discussed in the Methodology chapter, we tried to apply it to four different games: *Super Mario* (Khalifa, 2009), *Rambit* (Oliveira, 2011), *CaveCopter* (Nolte, 2010) and *Swervin Mervin* (Buntine, 2017).

The main modification that we did to the classical rhythm-based approach was the definition of the classes of actions, which allowed us to use the same rhythm to generate levels for different games. With the classes of actions we will always have the means to organize the possible actions of a certain game, by considering the mechanic descriptions discussed in the Methodology chapter, thus, we can always define a level by a sequence of those classes. Even though the rhythms provided by the genetic algorithm are generic, the other components of our generator must be adapted to the different features of the games. These features include: the possible actions, the geometry generator of the levels, how should the difficulty adaptations work and the generation of challenges.

In the next subsections, we illustrate the division of each game's actions into the classes of actions, how the geometry generator translates the rhythms into levels, the difficulty adaptations for each game and other details related to the contents generated (enemies, decoration, etc.). We also display some examples of generations that match the different classes of actions with level chunks from a game. For *Super Mario*, since some examples of level chunks for each class were already given in the Methodology, we opted to show examples of fully generated levels.

4.1.1 *Super Mario*

In the *Super Mario* game, there are four relevant actions: run, fall, vertical jump and horizontal jump. The difficulty of the horizontal jump action can be varied depending on the size of the gap Mario has to cross. With this, these actions can be divided into the classes as illustrated below in table 4.1:

Classes of Actions		
Class	**Action**	**Mechanic description**
0	Run or Fall	Pressing right key, with no specific pressure
1	Vertical jump	Pressing up key and right key, at the same time
2	Horizontal jump	Pressing up key and right key, at the same time and keeping pressure until the gap is crossed
3	Hard horizontal jump	Pressing up key, right key and A key, at the same time and keeping pressure until the gap is crossed, also requiring to be on the edge of the gap at full speed
4	Enemies	In this game, the class will represent enemies that can either be evaded or defeated.

Table 4.1: Classes of actions for the game *Super Mario*

The hard horizontal jump action will describe the longest jumps that the player needs to do and the regular horizontal jump goes from small to average size gap. In *Super Mario*, the enemies can be defeated by jumping on top of them or evaded. To evade them the player needs to wait for the right timing to get past them successfully. In the following images (figure 4.1 and figure 4.2), it is shown some complete levels that were generated with our approach.

Rhythms used: [0,1,0,3,0,0,1,0,0,1] + [1,0,1,0,1,1,1,2,1,0] + [0,0,0,0,1,3,1,1,1,0]

Difficulty value: 0.2

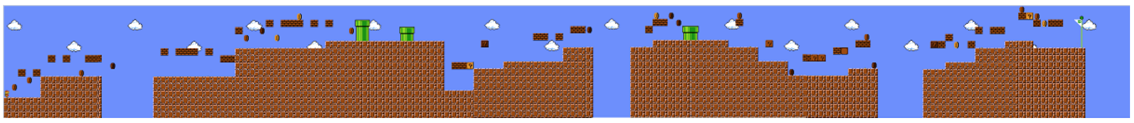


Figure 4.1: Example of a level with the difficulty value of 0.2

Rhythms used: [4,1,3,2,2,3,0,3,0,2] + [2,1,4,2,3,1,3,2,0,2] + [3,3,2,2,0,3,2,2,1,3]

Difficulty value: 0.6

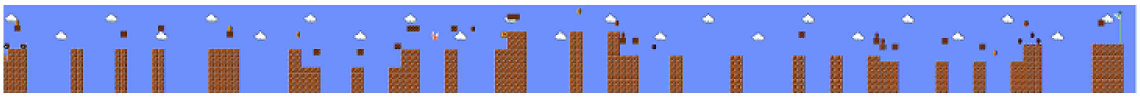


Figure 4.2: Example of a level with the difficulty value of 0.6

Generation exceptions

The terrain generation is mainly dependable on the action it should match and can have several options. In some cases, there is only one type of terrain that can match an action, in order to keep the level feasible (possible to complete). There are two cases where this happens in this game. The first case is when the next action is a vertical jump and the height of the terrain is at its maximum value. The other exception is when the next action is fall and the height of the terrain is at its minimum value. In the case where the terrain is at its maximum height and the action is a vertical jump, instead of having several ways of generating matching terrain for that action, it will only have the option of generating a pipe object, because the terrain cannot be increased anymore. For the case where the terrain is at its minimum height and the action is fall, since it is not possible to decrease the terrain more, it is going to generate terrain that matches the run action instead, because it has the same assumed difficulty (same class of action).

Generation exceptions	
Content generated	**Conditions**
Pipe	Next action is vertical jump and current height is equal to maximum height
Terrain that matches run action	Next action is fall and current height is equal to minimum height

Table 4.2: Generation exceptions for the game *Super Mario*

Difficulty adaptation

The next table 4.3 displays how the geometry generator adapts the terrains depending on the difficulty value.

Difficulty adaptation	
Game aspects	**Adaptation**
Landing Spot	The harder the difficulty, the smaller the landing space (goes from 1 to 4 tiles)
Terrain type	The harder the difficulty, the greater the probability of having the green platform terrain tile instead of the ground tile (green tile only has collision box on top)
Run and fall terrain sizes	The harder the difficulty, the smaller the size of the terrain generated for the actions run and fall (goes from 2 to 4 tiles)
Regular enemies	The easier the difficulty, the more common it is to have regular enemies
Winged enemies	The harder the difficulty, the greater the probability of the enemies being able to fly (harder to beat)

Table 4.3: Adaptations made in the game *Super Mario* based on difficulty

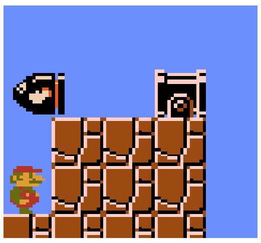
The greater the difficulty value, the smaller the landing spot and the smaller the size of the terrain generated for the run and fall actions, since these are the easier actions in the game. In this version of *Super Mario*, there are two different floor tiles: ground tile and green platform tile. The green platform tile makes the actions (mainly jumping) harder, since it only has a collision box on top (ground tile has a collision box all around it). Regarding enemies, there are various types, so the table below (table 4.4) shows the rules that restrict the generation for each one. If they are ”winged” or not it only depends on the probability associated with the difficulty value (higher difficulty translates to a higher chance of the enemy being ”winged”, since they are more difficult to avoid or defeat).

Enemy selection

Enemy selection	
Enemy type	**Conditions**
Green *Koopa*, *Spiky* and *Goomba*	Next actions are run or fall
Red *Koopa*	Next actions are horizontal or hard horizontal jumps (since red *Koopa* is the only enemy that does not fall down in the gaps)
Pipe flower	Next action is vertical jump and the terrain is at its maximum height value
Bullet Bill	Next action is vertical jump

Table 4.4: Enemy selection for the game *Super Mario*

The idea is to generate enemies that have an impact in the game and that makes levels feasible. Since red *Koopa* is the only enemy that does not fall down through the terrain gaps, it makes sense when these gaps are generated to introduce that type of enemy and use the other options (green *Koopa*, *Goomba* and *Spiky*) in terrains that do not have any gaps. When the terrain is already at the maximum height and the action is vertical jump, the generator will add a pipe, instead of increasing the height of the terrain, so it is only logical if there is an enemy in that level chunk it should be the pipe flower (represented in figure 4.3 b), an enemy flower that comes out of the pipe. Finally, if the next action is vertical jump and the terrain is not at its maximum height, the terrain is increased normally and if there is an enemy there, because of the terrain disposition, bullet Bill (represented in figure 4.3 a) makes sense to be generated, since it will have the most impact on the gameplay from that higher position.


(a) Bullet Bill

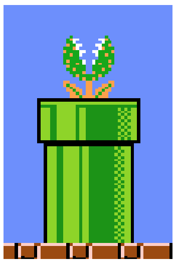

(b) Pipe flower

Figure 4.3: Examples of enemies

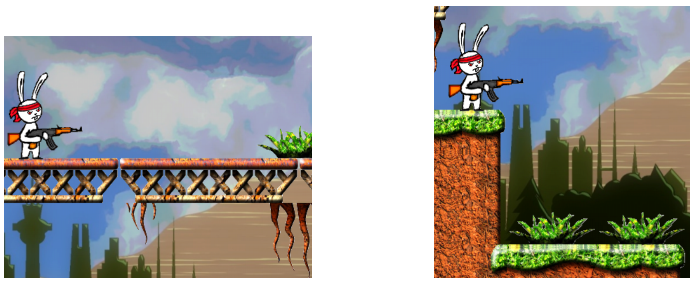
4.1.2 *Rambit*

Since *Rambit* is a platform game, like *Super Mario*, the actions are divided the same way, because the only variation is that the enemies can be defeated by shooting them with Rambit's gun, instead of jumping on top of them (remaining actions are the exact same).

Classes of Actions		
Class	**Action**	**Mechanic description**
0	Run or Fall	Pressing right key, with no specific pressure
1	Vertical jump	Pressing up key and right key, at the same time
2	Horizontal jump	Pressing up key and right key, at the same time and keeping pressure until the gap is crossed
3	Hard horizontal jump	Pressing up key and right key, at the same time and keeping pressure until the gap is crossed, also requiring to be on the edge of the gap and at full speed
4	Enemies	In this game, the class will represent enemies that can either be evaded or defeated.

Table 4.5: Classes of actions for the game *Rambit*

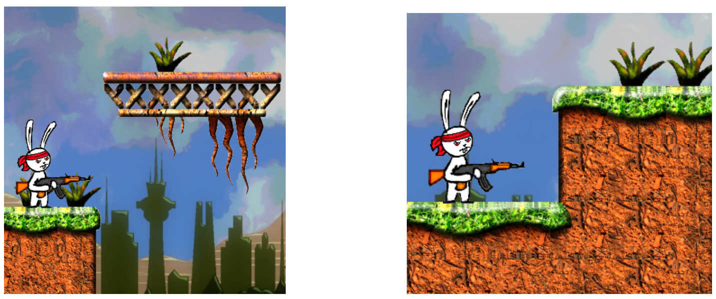
Class 0 (very easy)



(a) Run action (b) Fall action

Figure 4.4: Examples of generations for class 0, by generating plain terrain (for run action) and by generating a height decrease (for fall action)

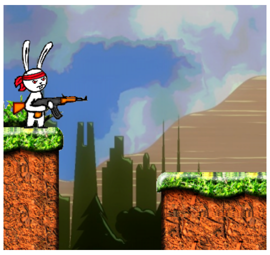
Class 1 (easy)



(a) Vertical jump action (b) Vertical jump action

Figure 4.5: Examples of generations for class 1, by generating a flying platform or a increase in height

Class 2 (average)


(a) Horizontal jump action

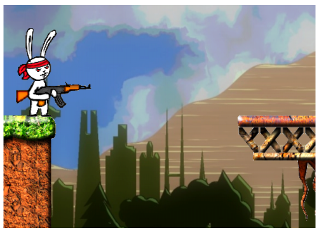
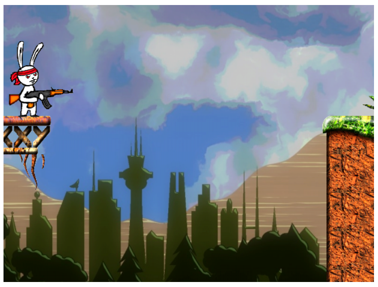

(b) Horizontal jump action

Figure 4.6: Examples of generations for class 2, by generating a gap in the terrain

Class 3 (hard)


(a) Hard horizontal jump action

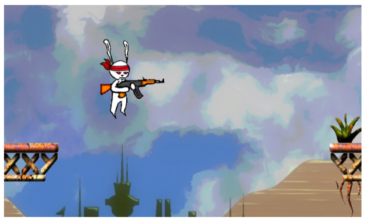
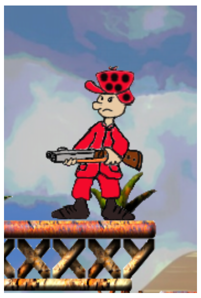

(b) Hard horizontal jump action

Figure 4.7: Examples of generations for class 3, by generating a big gap in the terrain

Class 4 (challenges)


(a) Shooter enemy

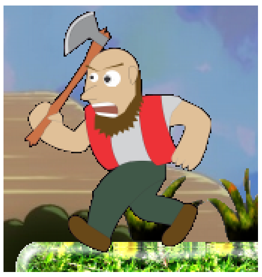

(b) Runner enemy

Figure 4.8: Examples of generations for class 4, by generating enemies

The previous images illustrate some examples of terrain generated for each class of actions for the *Rambit* game (there are other variations that can match each of these classes).

Generation exceptions

In the game *Rambit*, all the terrain is dependable on the action that it has to match. There is many options for terrain that matches a certain action, except in some cases. When the next action is a vertical jump and the height of the terrain is already at its maximum value, the only content that can be generated is a floating platform, to keep the level valid. The other exception is when the next action is fall and the terrain is already at its minimum height. In this case, since the actions run and fall have the same difficulty value (are in the same class), terrain that matches the run action is generated instead, to solve this problem.

Generation exceptions	
Content generated	**Conditions**
Floating platform	Next action is vertical jump and current height is equal to maximum height
Terrain that matches run action	Next action is fall and current height is equal to minimum height

Table 4.6: Generation exceptions for the game *Rambit*

Difficulty adaptation

The next table 4.7 will illustrate the aspects of the game that are adapted and how does that happen.

Difficulty adaptation	
Game aspects	**Adaptation**
Landing Spot	The harder the difficulty, the smaller the landing space (goes from 2 to 4 tiles)
Run and fall terrain sizes	The harder the difficulty, the smaller the size of the terrain generated for the actions run and fall (goes from 2 to 4 tiles)

Table 4.7: Adaptations made in the game *Rambit* based on difficulty

The sizes of the terrains generated for the landing spots, after a horizontal jump, and the terrain that matches the run and fall actions are getting smaller the harder the difficulty is. This happens in order to make the actions itself harder, by decreasing their execution time. For the landing spots, the smaller it is the more precise the movements of the player must be, to successfully complete the horizontal jump, and in the case of the run and fall actions, since they are the easiest actions in the game, the time that the player is actually doing them is reduced.

Enemy selection

Enemy selection	
Enemy type	**Conditions**
Runner enemy	Next actions are run, fall or vertical jump
Shooter enemy	Next actions are horizontal or hard horizontal jumps (since the shooter is static and does not fall through the gaps)

Table 4.8: Enemy selection for the game *Rambit*

The idea for the enemy selection in *Rambit* is to make the best use of the two types of enemies that are available. Since the shooter enemies remain static, it makes more sense to place them when there is a gap (terrain that matches the horizontal jump or hard horizontal jump). This is because the runner enemy can fall off the edge of the gap and since it is intended to have the optimal situations where each enemy will have the most impact, it was opted to only generate running enemies when the terrain matches actions that do not have gaps (run, fall and vertical jumps).

4.1.3 *CaveCopter*

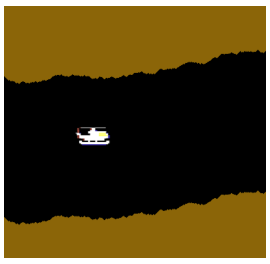
In the game *CaveCopter*, the player can either keep the helicopter's height, move it up, down, left and right (increases speed). In this genre of game, it was determined that the average (class 2) and hard (class 3) actions are a sequence of action variations. The average action will make the player variate the helicopter's height, with a successive change of the terrain's height, and in the hard actions, this variation is a more difficult version of the average action (sudden variations of height, consecutively).

Classes of Actions		
Class	**Action**	**Mechanic description**
0	Keep height	Pressing no key
1	Up or Down movement	Pressing up key or down key, with a specific pressure until the terrain increase or decrease is over
2	Low variation	Pressing up key, down key and right key, with variation on the pressure of the up and down key until the terrain variation is over
3	High variation	Pressing up key, down key and right key, with variation on the pressure of the up and down key until the terrain variation is over, also needing to be in a certain location of the terrain with a specific speed
4	Enemies	In this game, the class will represent enemies that can either be evaded or defeated.

Table 4.9: Classes of actions for the game *CaveCopter*

In this game, the enemies are UFOs that will have different attacks, depending on the difficulty, and can be defeated by the helicopter's missiles. The UFOs can also be evaded, by waiting for the right moment and dodging its attacks in an accurate fashion. The following images show some examples of terrain that match with each of the classes of actions.

Class 0 (very easy)


(a) Keep height action

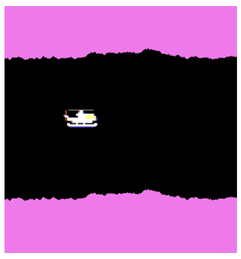
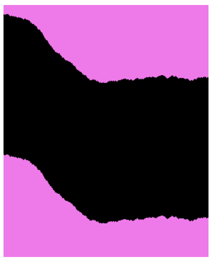

(b) Keep height action

Figure 4.9: Examples of generations for class 0, by generating terrain that keeps the height

Class 1 (easy)


(a) Down movement action

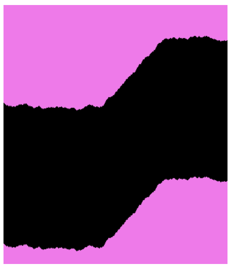
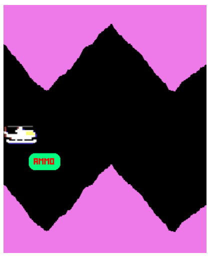

(b) Up movement action

Figure 4.10: Examples of generations for class 1, by generating increases (for up movement action) or decreases (for down movement action) in the terrain

Class 2 (average)


(a) Low variation action

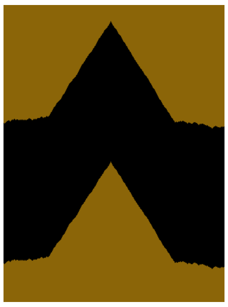
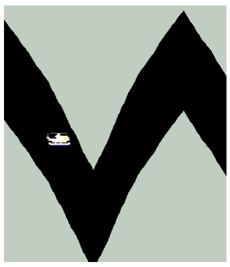

(b) Low variation action

Figure 4.11: Examples of generations for class 2, by generating small increases and decreases

Class 3 (hard)


(a) High variation action

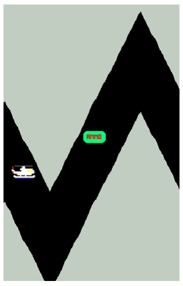
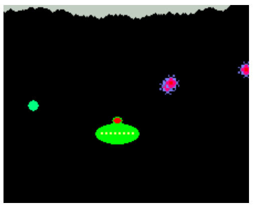

(b) High variation action

Figure 4.12: Examples of generations for class 3, by generating big increases and decreases in the height

Class 4 (challenges)


(a) UFO shooting mines and missiles

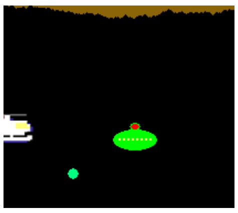

(b) UFO shooting missiles

Figure 4.13: Examples of generations for class 4, the UFO enemies

Generation exceptions

The generation process is fully dependable on the current class of actions and will generate terrain that matches the selected action (from that class). When the action is up and down movement, it will pick one of the actions randomly, in order to generate matching terrain. There is a special case for the actions low variation and high variation, where the generator can either make a decrease of terrain first and then an increase of terrain, or an increase of terrain first and then a decrease, since what it matters is that there is a variation of the terrain's height successively.

For the actions previously discussed, there are some exceptions. When the action is either up or down (same class), if the current height is equal to the maximum height value it will always select the down movement action and if the current height is equal to the minimum value, it will always pick the up movement action. These restrictions are also applied for the low and high variations when deciding whether the generation starts with an increase or decrease of the terrain's height. In order to limit the terrain, to have feasible and valid levels, specifically for this case, to avoid the terrain generated go out of the game's window. The following table 4.10 illustrates these exceptions.

Generation exceptions	
Content generated	**Conditions**
Terrain that matches the up movement	Next action is down movement and current height is equal to minimum height
Terrain that matches the down movement	Next action is up movement and current height is equal to maximum height
Variation starting with up movement	Next action is down movement and current height is equal to minimum height
Variation starting with down movement	Next action is up movement and current height is equal to maximum height

Table 4.10: Generation exceptions for the game *CaveCopter*

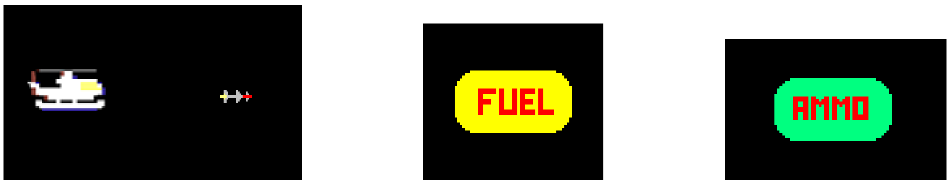
Difficulty adaptation

The next table 4.11 shows what game aspects are taken into consideration when adapting the difficulty of the generations and how these adaptations really happen.

Difficulty adaptation	
Game aspects	**Adaptation**
UFOs	The harder the difficulty, the more types of attacks the UFOs will have
Up and down movements heights	The harder the difficulty, the bigger the increase or decrease of the terrain generated for the actions up and down movement
Low variation heights	The harder the difficulty, the bigger the increase and decrease of the terrain generated for this low variation of actions
High variation heights	The harder the difficulty, the bigger the increase and decrease of the terrain generated for this high variation of actions
Distance between floor and ceiling	The harder the difficulty the smaller the distance between the floor and the ceiling terrain
Fuel and ammo	The harder the difficulty the less fuel and ammo items are provided

Table 4.11: Adaptations made in the game *CaveCopter* based on difficulty

The adaptations for the terrain in *CaveCopter* revolve around making the increases and decreases lasting longer and also decreasing the distance between the top and bottom terrain, since it will take better management of the movements of the helicopter by the player, in order to successfully complete the levels, without crashing. Regarding the enemies, they will become harder to beat or to get past, by having more variety of attacks the greater the value of difficulty. The fuel and ammo are the only collectibles in the game, and since they have a big impact on the overall difficulty, the harder the level the less of these resources the player will have available (fuel is what allows the helicopter to move and ammo provides missiles that can defeat UFOs).



(a) Copter shooting a missile

(b) Fuel bonus

(c) Ammo bonus

Figure 4.14: Examples of in-game contents

Enemy selection

There is only one type of enemy, the UFOs. The difference in difficulty is reflected on what kind of weapons it has in its arsenal. The UFOs can have missiles, mines, both of them or neither. On the easier levels, the UFOs generated won't have any weapons available and the harder the levels get, the more abilities it will have. First, the missiles are added and then, because of its features, the mines are included in the UFO's weapon set. The mines will stay in the same place after being shot, making the level much more complex to complete, since the player needs to worry about evading them and the terrain at the same time.

The power of each weapon is increased accordingly to the value of difficulty: the UFO's speed and missiles will get faster, the UFOs shoot mines more frequently and the number of mines that are placed per shot is also increased.

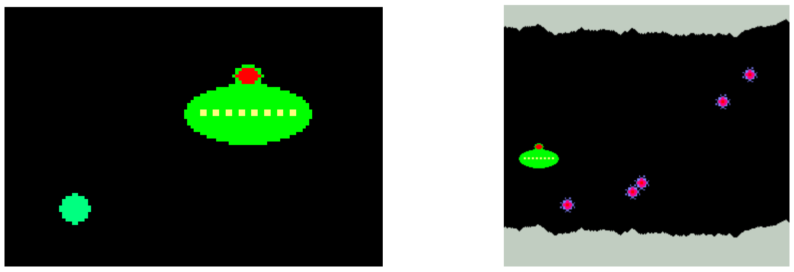

(a) UFO shooting a missile (b) UFO shooting mines

Figure 4.15: UFOs using their attacks

4.1.4 *Swervin Mervin*

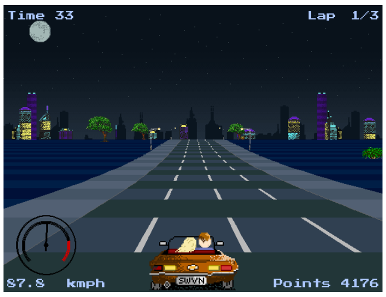
Swervin Mervin is a racing game, and because of this, it is somewhat different from the previous tested games. The main difference is that, in this game genre, there are no enemies but obstacles instead, since the other cars, that are competing with the player, are not seen as enemies, but as movable obstacles that cannot be defeated. In terms of actions, we can see in the next table 4.12 that we can either go straight, move left, move right and brake. Thus, we have the following actions.

Classes of Actions		
Class	**Action**	**Mechanic description**
0	Go forward	Pressing up key
1	Easy turn	Pressing up key and right or left key, with a variation on the pressure of the keys until the turn is over
2	Low variation OR Hard turn	Pressing up key, down key and right or left key, with variation on the pressure of the up and down key to stay on the road and a specific pressure on the right or left key until the turn is over
3	High variation	Pressing up key, down key and right or left key, with variation on the pressure of the up and down key to stay on the road and a specific pressure on the right or left key until the turn is over, also needing to be in a certain location of the road with a specific speed
4	Obstacles	In this game, the class will represent obstacles that can only be evaded.

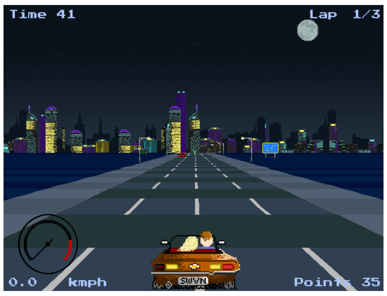
Table 4.12: Classes of actions for the game *Swervin Mervin*

The important aspect that needs to be noted, is that the difference between an easy turn and a hard turn is the size and how much degrees the curvature of the turn will have. An easy turn is short with a small degree in the curvature and a hard turn is long with an average to a big degree in the curvature. These easy and hard turn actions always start and end on straight road, meaning that it first generates straight road, then the turn and finally adds more straight road, in order to differentiate them with the difficulty of the low and high variation actions. The variation actions represent successive turns in different directions were there is a left turn then a right turn, or a right turn then a left turn, instead of a turn and straight road, like the easy and hard turn actions (will only have one turn generated in the middle of the straight road). The difference between the low and high variation is that in the low variation the successive turns are easy turns and the high variation are successive hard turns (the difference between the easy turn and hard turn was discussed previously). Even though it is hard to show the whole road generated to each class, we have some examples of generations that match each class of actions.

Class 0 (very easy)



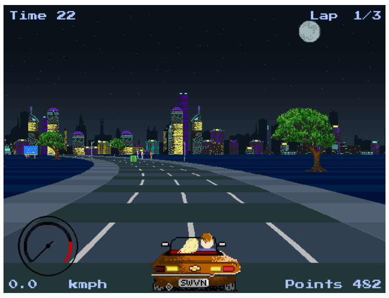
(a) Go straight action



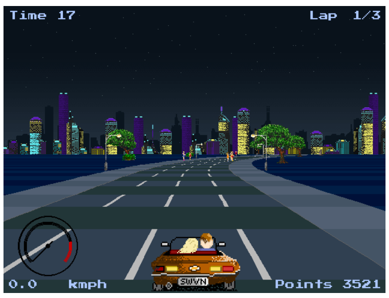
(b) Go straight action

Figure 4.16: Examples of generations for class 0, by generating straight road

Class 1 (easy)



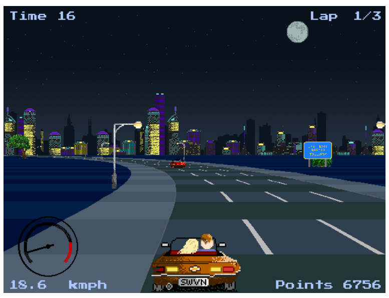
(a) Easy turn action for the left



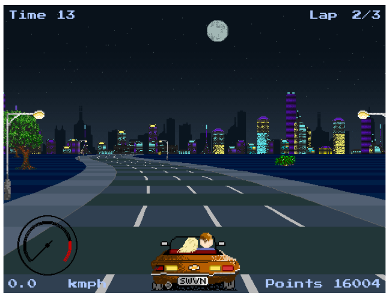
(b) Easy turn action for the right

Figure 4.17: Examples of generations for class 1, by generating easy turns to the left or right

Class 2 (average)



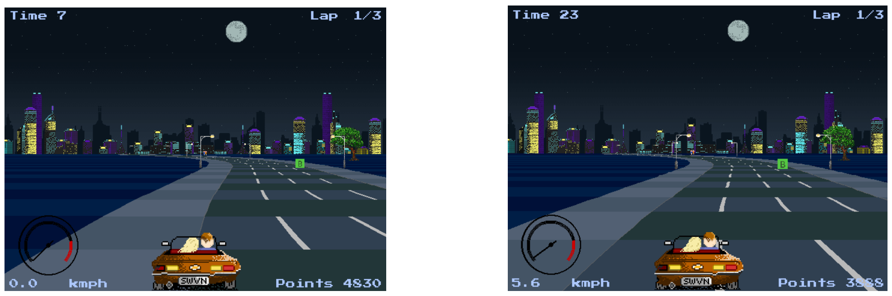
(a) Hard turn action



(b) Low variation action

Figure 4.18: Examples of generations for class 2, by generating a hard turn (to the left or right) or by generating a variation of easy turns (for low variation action)

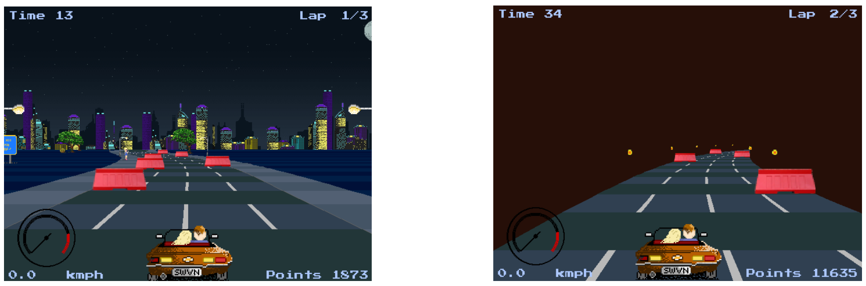
Class 3 (hard)



(a) High variation action (b) High variation action

Figure 4.19: Examples of generations for class 3, by generating a variation of hard curves

Class 4 (challenges)



(a) Obstacles (b) Another example of obstacles

Figure 4.20: Examples of generations for class 4, by generating obstacles

On the right image of class 2 (figure 4.18 b) and both images of class 3 (figure 4.19), it is shown the low and high variation, respectively. The instant when the screen was captured was in the middle of the action's matching road, which is right before the end of the first turn and in the start of the second one, which goes in the opposite direction.

Generation exceptions

For all the games, when testing the level generation approach, there were always some exceptions in the generation. For this game, it was the first time that there were no exceptions, so the road generated is totally restricted by which action it should match. In the classes where there are two actions, it will randomly select one of them and the direction of the turns for the actions easy turn, hard turn, low variation and high variation are also picked randomly. For the low and high variation, only the first turn's direction is randomly selected, since the second turn will go in the opposite direction (if the first is a left turn, then the second would be a right turn, for instance).

Difficulty adaptation

In the following table 4.13, it is described what aspects of the game were taken into consideration when adapting the difficulty and how they were made.

Difficulty adaptation	
Game aspects	**Adaptation**
Easy curves	The harder the difficulty, the bigger the curvature degree and the size of the curves generated
Hard curves	The harder the difficulty, the bigger the curvature degree and the size of the curves generated
Time bonuses	The harder the difficulty, the less time bonuses will be generated in the road
Obstacles	The harder the difficulty, the more obstacles are going to be generated (2 to 6 obstacles)
Movable obstacles	The harder the difficulty, the more cars will be generated and the faster they will be

Table 4.13: Adaptations made in the game *Swervin Mervin* based on difficulty

The adaptations that are done in this racing game are very similar to the other games, since we are changing some factors mainly related to the characteristics of the terrain generated (like the size), the features of the enemies or obstacles (to become tougher to get past the harder the level's difficulty) and varying the number of bonuses available. Thus, to all the turns, the greater the value of difficulty the harder it will be to successfully drive through them. It was determined that the degrees of the curvature in those actions and the size of them will greatly impact the overall difficulty of each track, because if a turn has more degrees, or is longer, it will require a better positioning of the car to actually get past them, without crashing or getting out of the road.

The other cars are a huge part of this game, since they are the player's competition and will work as movable obstacles. Therefore, in order to increase a level's difficulty, we generate more and faster cars (the greater their speed, the harder it is to avoid colliding with them). The other cars are seen as movable obstacles, as previously mentioned, so we opted to not used them in class 4 (represents challenges), because it is not possible to predict which part of the level they will impact. Thus, we decided to use the static obstacles, illustrated in the images of class 4 (figure 4.20), instead.

Regarding the obstacles, they are adapted to the difficulty value in a very simple way. The greater the difficulty of the track, the more obstacles are going to be generated (the number of obstacles goes from 2 to 6) on the road, for each class 4 that exists in the utilized rhythm.

(a) Static obstacle

(b) Movable obstacle, other cars

(c) Time bonus

Figure 4.21: Examples of in-game contents

4.1.5 Analysis of results

This last section allowed us to describe all the details related to the implementation of our level generation approach to each of the games. It was possible to notice some common grounds in the games besides the rhythms being the same for all of them, which were:

- **Class of action distribution** - For *Super Mario* and *Rambit*, since both of them are *platformer* games, the way we organize their actions is identical (they have the same actions) and for *CaveCopter* and *Swervin Mervin*, even though both games have very different styles, their actions are quite similar, regarding their execution, so we organized them with the same criterion. This happened because in these games the challenge of their levels is to avoid colliding or getting out of the path (not getting out of the road or not colliding with the ground or floor), the easier actions revolve around maintaining a certain movement and the hardest actions revolve around big variations in the movements.

- **Geometry generator** - Every geometry generator will be different for each game, because every game will have a different geometry and visuals (sprites), but the structure will be the same, because the number of classes of actions is constant to all of them. Although it is not possible to have a generic geometry generator, there are still features that are very similar for the *Super Mario* and *Rambit* games, since they are the same genre, we can generate similar level chunks (as we illustrate in the generalization test). For *CaveCopter* and *Swervin Mervin* there was some similarities found too, which were the variations generated (or the absence of variations) in the terrain, even being possible to notice some resemblance between the *CaveCopter*'s terrain and the *Swervin Mervin*'s track or road, from a different angle. Regarding challenges (enemies or obstacles) generation, the criteria was the same to all games, which was to place them in the locations they had more impact, as it is explained in the previous section.

- **Difficulty adaptations** - The adaptations made to vary the difficulty of the levels is very similar in all games, since we are always manipulating the sizes of the level chunks in order to make it more difficult to get past (the space is smaller and takes more precise movements the greater the difficulty value). The enemies are adapted with the same logic, the greater the difficulty value, the more powerful, or more abilities, they have. For *Swervin Mervin* the obstacles are adapted with the same concept as the enemies, but since we can not increase the obstacles abilities we just generate more obstacles the greater the difficulty value.

To sum up, by verifying that we were able to utilize the same sequences of classes of actions in four different games and that the other components of the generator were possible to adapt without any issues, we can conclude that the modified rhythm-based level generation approach was successfully developed and widespread to different 2D games.

4.2 AI test

The DDA's main purpose is to vary the difficulty value of the generated levels, depending on a player's performance, and to adapt the in-game contents that are generated in a level. The DDA uses the difficulty value to adapt the generated content accordingly and to select a rhythm to be used as the base structure of a level. For this test, we want to evaluate the capacity of the DDA to correctly generate content for levels with a specific difficulty value, and to evaluate the consistency of the difficulty of the rhythms used for each value. Thus, we have applied the A* search algorithm (Millington, 2019) as a *benchmark* to the *Super Mario* levels, in order to confirm that the DDA and the rhythms are working as intended, by checking how the difficulty value variations affect the level completion rates.

We chose this way to evaluate the generator's DDA by having into account that, on one hand, we had available some implementations of search algorithms (Millington, 2019) for the *Super Mario* game in the Mario AI tournament framework (Khalifa, 2009) and, on the other hand, because the A* algorithm is one of the most complete, optimal and efficient search algorithms in the fields of computer science, we opted to use it as a *benchmark*. Thus, for each difficulty value 100 levels of the *Super Mario* game were generated in real-time and it was counted how many times the algorithm successfully completed a level from a specific difficulty value, to help us conclude if the DDA is correctly generating content based on a difficulty value and if the set of rhythms associated to each difficulty value have a consistent degree of difficulty.

If the DDA is generating according to the level's difficulty and the rhythms associated to each difficulty value are consistent, the expected behavior is the A* algorithm failing the completion of more levels as the difficulty value increases. This test is important to validate the DDA's generations based on a difficulty value, since this value varies depending on the player's performance. For instance, if a player successfully completes a level, then the difficulty value is going to increase and the DDA should be able to adapt the generated content to match with that value.

4.2.1 Results

The results of the application of this test are illustrated in the following graph (figure 4.22), where the X-axis represents the level difficulty value and the Y-axis represents the number of levels completed (out of 100):

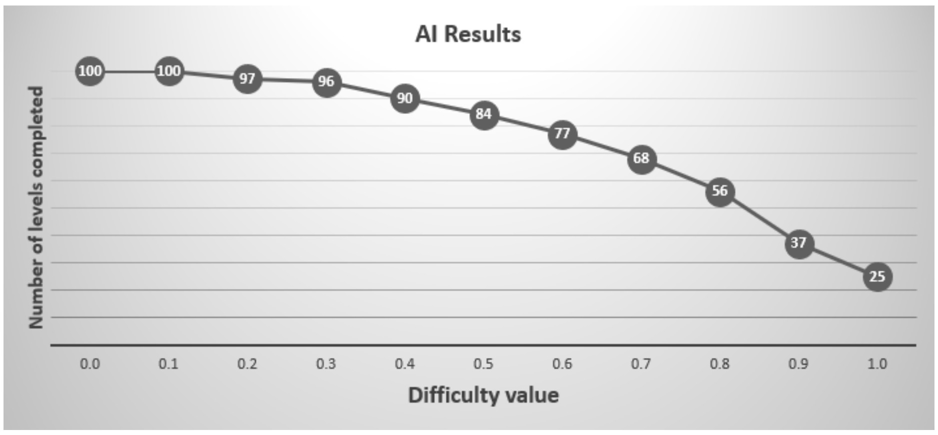


Figure 4.22: Results from the AI tests

It is possible to conclude, based on the results, that the rhythms used are consistent with their difficulty values and that the DDA is correctly generating in-game contents according to a difficulty value, since the greater the difficulty, the fewer times the A* algorithm was able to complete a level. This means that, as the difficulty value increases, the levels are getting harder to complete, which is what we intend.

The percentage of completed levels per difficulty value is slowing decreasing as the difficulty increases. This allow us to infer that, not only the rhythms, but also the challenges and contents of the generated levels are being accurately and consistently adapted to each difficulty value, since we do not observe a considerable drop in the completion rates. In order for the DDA to appropriately adapt the generations to a difficulty value, we implemented parameters in the geometry generator, with the purpose of controlling some features of the level's contents (the size of the landing spots, enemies attributes, number of power-ups, among others). These parameters get to their maximum value of difficulty at the value 0.9, this being the main reason we believe the biggest variation of completed levels occur from the 0.8 to 0.9 (less 19 levels completed).

4.3 Human player test

In this section, the results from the tests that use the help of human players to evaluate the difficulty definitions are presented. The main purpose of this test is to validate our definitions of difficulty, since it is a very subjective concept that can differ from player to player, being crucial to be generalized the best as possible. It is expected that in most cases the players are able to recognize the difficulty values for each of the levels that are provided.

For these tests, it was given remote access of our computer to the players that are going to test some of our generated levels. Several applications that provided the tools needed were assessed, like *TeamViewer* (GmbH, 2005) and *Microsoft Teams* (Microsoft, 2016), but they had some delay in the actions, that would greatly impact the performances of the players in the levels. Because of this, we looked for other options, so the players would have the best conditions possible to play the levels. We ended up using the *Quick Assistance* application for remote control from *Windows 10*, since in most cases it had no noticeable delays.

The test consisted on providing to the players tutorial levels and then, three levels with different difficulties of each game we applied our level generation approach. We opted to first give the players tutorial levels so they could learn the objectives and possible actions of each game, before playing and sorting the test levels by difficulty, based on their opinion. The goal is not only to see if they are able to sort them correctly (based on our definitions of difficulty) but also to check if their performances match with the difficulty value of each level (for the harder levels it would be expected worse performances). It is important to note that the order of the levels varies from game to game. It is impossible to guess the correct order before playing, thus the results are only based on the opinions and thoughts of the players after seeing the levels. It was also asked the players to categorize themselves as casual, regular or persistent players, so it would be possible to take this into consideration when analyzing their performances (would be normal if a casual player had bad performances even in easy levels). In the following subsection, it is illustrated the results regarding these tests.

4.3.1 Results

In this subsection, the results from the human player test are displayed and analyzed. In total 15 tests (9 casual, 3 regulars and 3 persistent players) were made, which means each of the levels was played 15 times. For each game, it is illustrated in graphs the performance of the players and in a table the success rate of each player category in the three provided levels. To evaluate the performances, the levels were divided in three parts (first part, second part and final part), in order to evaluate into more detail the gameplay ”quality” of each player. These parts will represent how far a player got in a level, assuming, of course, that the player lost. In order to have the most information possible, when a player is able to complete a level, there are some features of the game that allows us to evaluate the ”quality” of the playthrough, like for instance: how many lives the player had in the end of the level, the time it took to complete the level and so on.

Like it was previously mentioned, the correct order of the levels by difficulty is completely random being, because of this, impossible to guess which level is the hardest before playing all of them. The following table (Table 4.14) illustrates the correct order, from easiest to hardest, of the levels, that we randomly assigned to each game (the difficulty value of each level is inside brackets).

Correct order of the levels	
Game	**Order**
Super Mario	Level 2 (0.2) → Level 1 (0.4) → Level 3 (0.6)
Rambit	Level 1 (0.2) → Level 3 (0.4) → Level 2 (0.6)
CaveCopter	Level 2 (0.4) → Level 1 (0.6) → Level 3 (0.8)
Swervin Mervin	Level 3 (0.4) → Level 1 (0.6) → Level 2 (0.8)

Table 4.14: Correct order of the levels by difficulty

In the following figures and tables, the performances of the players for the *Super Mario*, *Rambit*, *CaveCopter* and *Swervin Mervin* levels are illustrated and analyzed accordingly.

Super Mario

Super Mario				
Level	**Casual** (9)	**Regular** (3)	**Persistent** (3)	**Total** (15)
Level 1 (0.4)	0%	0%	0%	0%
Level 2 (0.2)	0%	66%	100%	27%
Level 3 (0.6)	0%	0%	0%	0%

Table 4.15: Success percentage for the *Super Mario* game

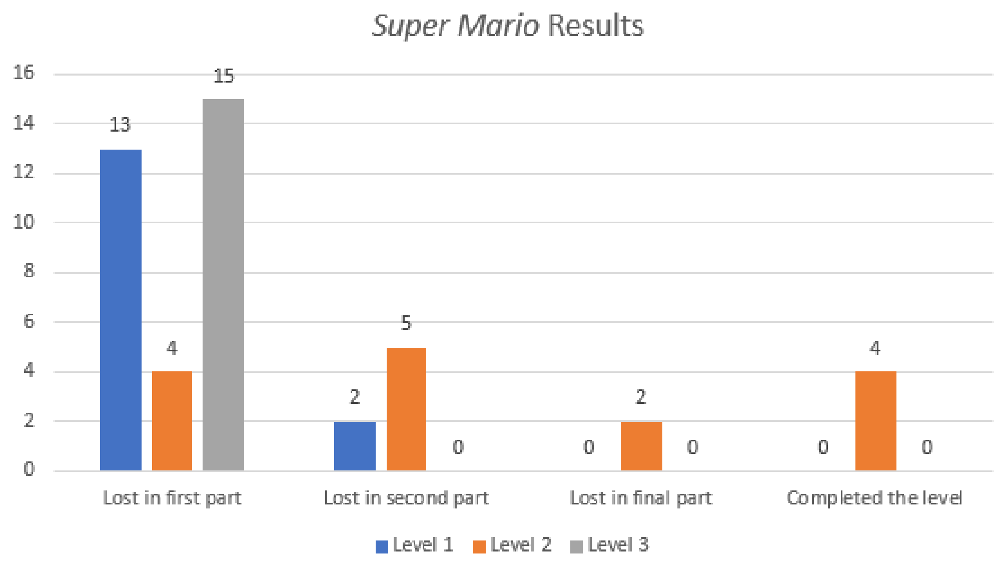


Figure 4.23: Results from the *Super Mario* levels

All the 15 players guessed the right order of the *Super Mario* levels. Performance-wise, it is possible to see in figure 4.23 that the level that had the worst performances was level 3 followed by level 1 and finally level 2, which had better results (most completed levels). This information matches correctly with the difficulty of each level. By comparing the results between level 1 (difficulty value = 0.4) and level 3 (difficulty value = 0.6), the results are similar, even though level 3 had the worst performances. This happens exclusively for the *Super Mario* levels, mostly because it is the hardest game to play, by consequence of its fast-paced playstyle that made it difficult to play, not only for the casual players but also for the regular and persistent players.

Rambit

Rambit				
Level	**Casual** (9)	**Regular** (3)	**Persistent** (3)	**Total** (15)
Level 1 (0.2)	100%	100%	100%	100%
Level 2 (0.6)	22%	33%	66%	33%
Level 3 (0.4)	44%	100%	66%	60%

Table 4.16: Success percentage for the *Rambit* game

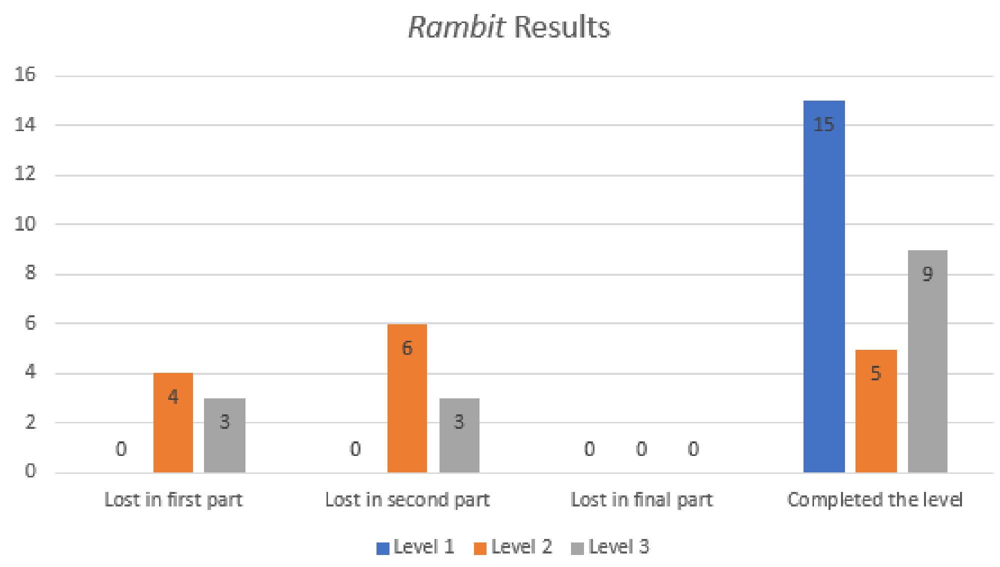


Figure 4.24: Results from the *Rambit* levels

All the 15 players guessed the right order of the *Rambit* levels. Regarding performances, it is possible to see in figure 4.24 that the easiest levels, since all players were able to complete, was level 1 and the level that had the worst performances was level 2 followed by level 3, that had 4 more players completing it in comparison with level 2 (the hardest level). This information matches our expectations based on the difficulty values of each level. The performance details about the completed levels are illustrated in the next figure 4.25.

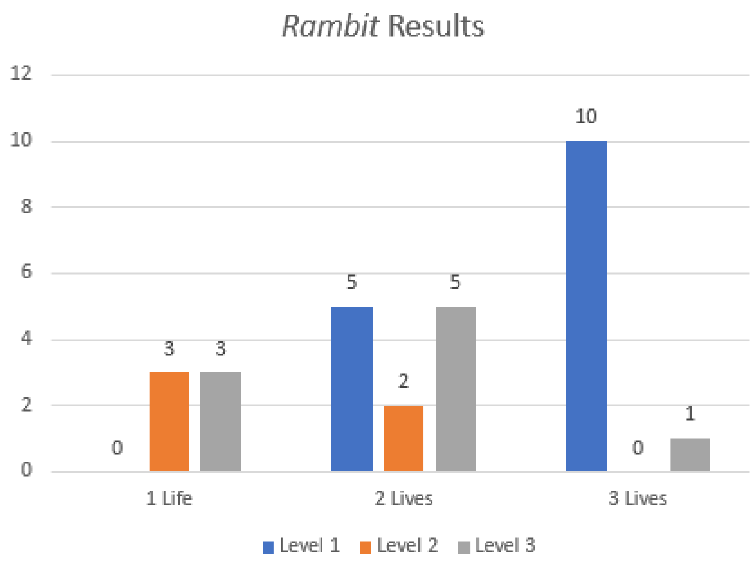


Figure 4.25: Level completion details for the *Rambit* levels

By analyzing the completed levels, it is possible to also confirm that the best results happen in the easier level and get ”worse” as the difficulty increases. The way this is verified on *Rambit* is by checking how many lives the player lost (has 3 lives in total). For level 1, 10 players did not lose a single life while the others lost only one, for level 2, out of the 5 players that completed it, 2 of them lost one and the rest lost two. Finally, in level 3, out of the 9 players that completed the level, 1 was able to not lose any lives, 5 only lost one and the rest lost two. With this information, it is clear to see that the greater the difficulty value of the *Rambit* level, the fewer completions there are and, for the ones that got completed, it is possible to notice that there was more failure in the harder ones, meaning that it was more complicated to successfully get past its challenges.

CaveCopter

CaveCopter				
Level	**Casual** (9)	**Regular** (3)	**Persistent** (3)	**Total** (15)
Level 1 (0.6)	44%	66%	100%	60%
Level 2 (0.4)	88%	100%	100%	93%
Level 3 (0.8)	11%	33%	0%	13%

Table 4.17: Success percentage for the *CaveCopter* game

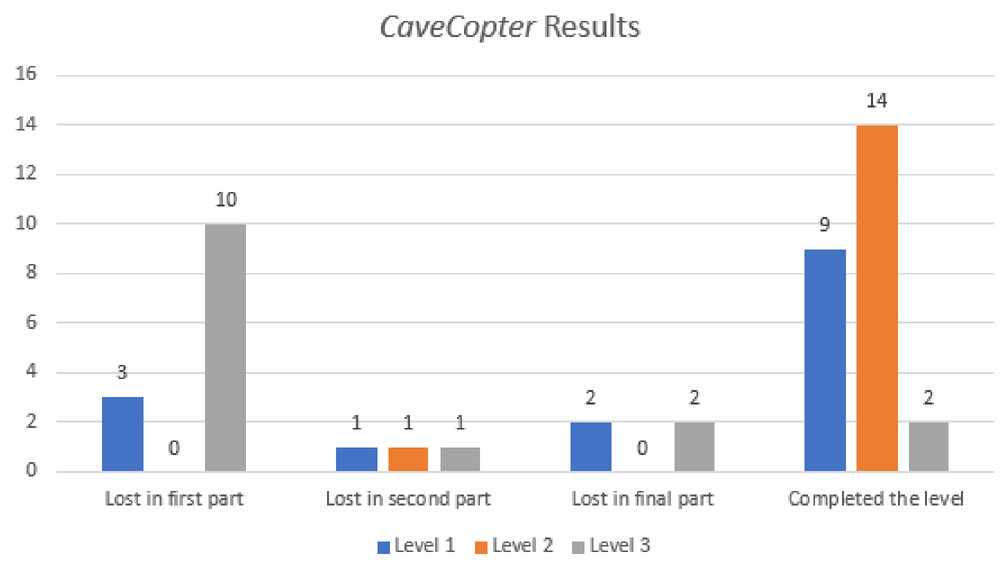


Figure 4.26: Results from the *CaveCopter* levels

First of all, it is important to mention that this was the only game that did not had example levels, since *CaveCopter* did not have a level saving mechanism, generating everything online in the tests.

Even though there was the possibility of, for example, the level with the greater difficulty generate the easiest version of itself making it similar to the second hardest level and, because of this, making it more complicated to distinguish from each other, the players were still able to order them all correctly by difficulty.

By considering the players' performances in figure 4.26, it is clear which level is the easiest and which is the hardest, since in level 2 only one player was not able to complete it and the hardest level only 2 were able to finish it successfully (level 3). Even with the complications mentioned in the beginning (possibility of confusing levels), if the results for level 1 and 3 are compared, we can distinguish them very easily, since most players in level 3 couldn't get past the first part. This also happens with level 2 and 1, since in level 2 almost all the players completed the level, something that does not happen in level 1 (6 players were not able to finish it). This allows to conclude that the level generator for *CaveCopter* was consistently generating terrain and challenges appropriate to the difficulty value.

Swervin Mervin

Swervin Mervin				
Level	**Casual** (9)	**Regular** (3)	**Persistent** (3)	**Total** (15)
Level 1 (0.6)	11%	66%	33%	27%
Level 2 (0.8)	0%	0%	0%	0%
Level 3 (0.4)	88%	100%	100%	93%

Table 4.18: Success percentage for the *Swervin Mervin* game

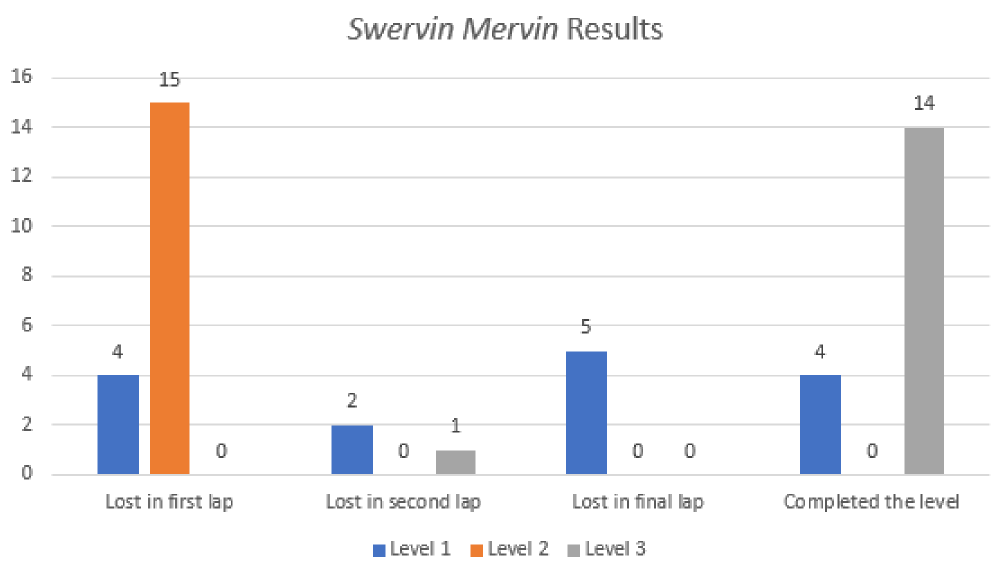


Figure 4.27: Results from the *Swervin Mervin* levels

All the players were able to order the levels correctly. The performance results in figure 4.27 also match with our expectations, since the harder the tracks got, the fewer players were able to complete them. To be able to evaluate the quality of the performance into more detail, we wrote down the time it took each player to successfully complete a level. The performance details about the completed levels is illustrated in the following figure 4.28. The 3 options correspond to the remaining time when the level was completed (between 1 to 5 seconds, between 6 to 10 seconds or between 11 and 15 seconds).

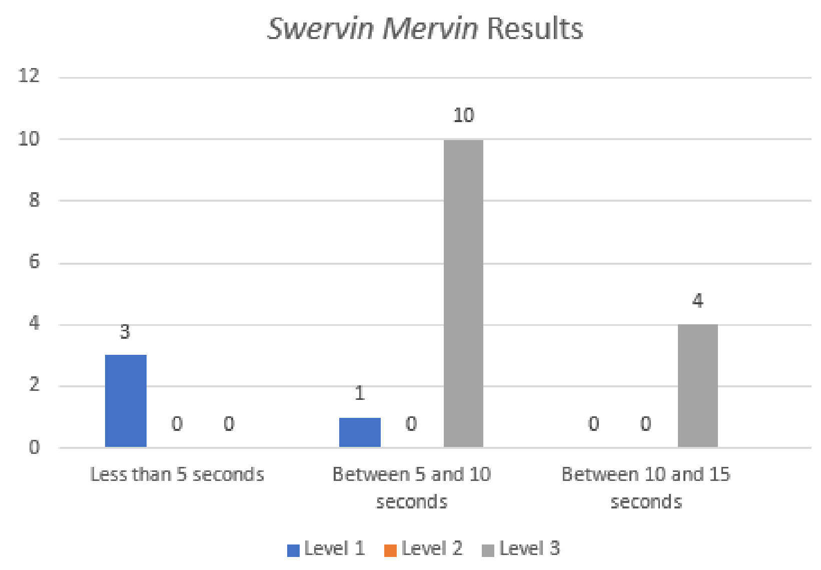


Figure 4.28: Level completion details for the *Swervin Mervin* levels

For the easiest track (level 3), 10 out of the 14 players, which were able to complete it, finished the track with more than 5 seconds to spare and the rest of them had more than 10 seconds remaining on the timer when they crossed the finish line. For level 1, out of the 4 players that completed the track 3 had less than 5 seconds to spare and only 1 had more than 5 seconds remaining. Meaning that most players barely were able to complete this level. Level 2 was the hardest track and the performances clearly show that, since all the players lost in the first lap, so there is no information about its completion performances.

4.3.2 Analysis of results

It is possible to conclude that the definitions of difficulty were generic enough for most players and the levels generated are possible to complete (feasible). By taking into consideration the performances on each level, it is possible to verify that they get worse, the greater the difficulty value of the level. This shows that the level's terrain and challenges, that were generated, are being adapted correctly and the definitions for each class of actions are accurately characterized, since they are the base of the level generator. Another very important information attained from these tests is that every single player was able to order the levels by difficulty correctly. This means that, even though some were not capable of finishing the levels, they could still distinguish the levels from each other. Thus, not only the gameplay but also visually the levels are being correctly adapted, based on their difficulty value. The feasibility of the levels was also confirmed since, according to the players, even thought some levels were hard to get past, they still could see themselves completing those levels if they had more time to practice, meaning impossible challenges were not generated (valid generations).

Some remarks regarding the visual aspects of the game were also made by the players and the most positive comments were about the games *CaveCopter* and *Swervin Mervin*, that accordingly to the players looked human-made, whereas *Super Mario* and *Rambit* seemed randomly generated in some parts. This is crucial, because it gave insights about upgrades that could be made, regarding the geometry generator for the *platformer* games, or maybe it looked less ”natural” than the other two games simply because of the features of the genre itself (has gaps in the terrain, something that does not happen in the other tested game genres). The different player categories (casual, regular and persistent), had no significant impact in the performances, since all the players had more complications in the harder levels. Initially it was anticipated that it would make sense if the casual players could not complete even the easier levels and that the persistent players could complete the harder levels with no problems. Thus, we included these aspects into the test, in order to justify some of those situations in case they occurred. We believe the player categories did not had that much of an impact as a result of all the players having no experience in the tested games.

Chapter 5

Conclusion

The main objective of this project was the development of a procedural level generator for 2D games, which could adapt the difficulty of the levels generated depending on the player's skill, with the capacity to widespread its generations to different games.

In the first stages of the project, an investigation was made on the procedural content generation area, in order to acquire the most useful knowledge for the development of our level generator approach. Taxonomies, procedural level generation approaches and dynamic difficulty adjustment techniques were analyzed, since those were some of the most important topics of this project. After deciding what our approach would be, we also investigated and analyzed genetic algorithms, because it plays an important role in our level generation approach.

The methodology of our generator revolves around the rhythm-based approach for level generation, that describes a level as a sequence of actions that need to be completed by the player, if he wants to successfully get past the level's challenges. In our level generation approach, we proposed some modifications to the classical rhythm-based approach, by introducing the classes of actions. The classes of actions are a way for us to organize actions, depending on their difficulty, and allowed us to make the rhythm-based approach more generic, being possible to be applied to different 2D games, since all games will have actions with different difficulties and similar ways of performing them, which provides us the means to divide them (the actions) into the different classes. For the generation of our rhythms, we used a genetic algorithm whose fitness function has the ability to evaluate the difficulty of a certain rhythm, based on our definitions of difficulty (mechanic descriptions), for each class of actions in the sequence. After being generated, the rhythms are saved in files that are going to be used in the geometry generator of each game. The geometry generator will take random rhythms from the target difficulty's files as parameters, with the purpose of generating appropriate and valid in-game content for each of the classes in the selected rhythms.

In order to confirm the ability of the approach to widespread its generations and to validate our definitions of difficulty we conducted several tests to make assessments about the level generator. First we applied our level generation approach to four different 2D games: *Super Mario*, *Rambit*, *CaveCopter* and *Swervin Mervin*. Then, to confirm our notions of difficulty and to very if the DDA was correctly adapting the generations to a difficulty value, we used the $A*$ search algorithm to play through the levels of the game *Super Mario*, in order to verify how many levels the algorithm was able to complete, for each difficulty value and, with this, confirming that the level generator was generating and adapting accordingly to the value of difficulty assigned to a level. Since we also wanted to determine if our definitions of difficulty (for the classes of actions, the level chunks and the adaptations to the terrain) are generic enough for most players, the level generator was tested by a control group, to play some of the generated levels, with the purpose of demonstrating this.

By having the original levels of the tested games as a basis and the opinion of the same control group that did the playthrough tests, we were able to conclude that we successfully applied this modified rhythm-based approach to games with very different genres and playstyles. Regarding the tests that looked to validate our definitions of difficulty we could conclude that we successfully implemented a level generator that is able to adapt the levels accordingly to the value of a target difficulty and also confirmed that our concepts of difficulty, for each class of actions, were generic for most players, since all of them were able to identify the difficulty of the levels from the tests and their performances matched that as well (the greater the difficulty of a level, the worse the performances were). Thus, we consider that all the results we obtained in this work allow us to consider that it was successful in view of the defined objectives. However, there are some aspects that can and should be improved in the future.

Regarding future work and aspects that can be improved in our level generation approach, we considered the following two: The first one is the online generation of the rhythms instead of being offline. Because the genetic algorithm was implemented in Java and we have games from Pygame we had to generate them offline and save them in files, in order to let the geometry generator of those games use the generated rhythms. The other improvement would be the ability to generalize to game genres where the levels that can not be described by a sequence of actions that allows a player to complete the level. This is the biggest flaw in our approach, since our methodology revolves around generating levels based on sequences of actions and, in these specific games there is no sequence to follow (levels that does not change scenario or levels where the direction that a player needs to go to progress does not remain the same). It is very complicated to represent these types of levels with our rhythms, mostly because we can not associate a difficulty value to the action of changing directions (it is not harder to go left than right, for instance) and we can not generate terrain based on actions if the terrain remains the same for the whole level, meaning that we can not structure it by a specific sequence of actions (fighting games, for example). These problems are the main concerns for future work but we could also include the implementation of a better fitness function for the genetic algorithm (with different parameters, or other values used in the difficulty variance table) and the improvement of the geometry generator for *platformers* (to look more natural and less random).

In order to conclude the dissertation and sum up this chapter, we can enumerate the level generator's qualities and future improvements as follows:

Qualities of our level generation approach

- The classes of actions allowed us to successfully generalize the rhythm-based approach for level generation
- The implementation of a DDA that correctly adapted the generations according to each difficulty value and based on our definitions of difficulty
- A rhythm generator that provided the tools to: evaluate the difficulty of each rhythm, generate different rhythms for each difficulty value and search for rhythms of specific difficulties
- A geometry generator which not only was able to translate rhythms into a valid levels but that could also help the DDA in the difficulty adaptation task
- Definitions of difficulty that are generic enough for most players

Future improvements to our level generation approach

- Introduce an online rhythm generation, to avoid saving the rhythms into files
- Generalize the approach for games where the levels can not be described as sequences of actions
- Test other fitness functions, with other parameters or different values for the current ones
- Improve the geometry generator for *platformers*, so the levels look more human made

Bibliography

Associates, F. F. (1983). Archon: The light and the dark. Electronic Arts.

Barrett, S. (2012). Herringbone wang tiles.

Buntine, A. (2017). Swervin mervin. https://github.com/buntine/SwervinMervin/. [Online; accessed January 20, 2020].

Capcom (2005). Resident evil 4.

C.Pedersen, J. T. and Yannakakis, G. N. (2010). Modeling Player Experience for Content Creation. *IEEE Transactions on Computational Intelligence and AI in Games*.

David Braben, I. B. (1984). Elite. Acornsoft.

Derek Yu, A. H. (2009). Spelunky.

Dog, N. (1996). Crash bandicoot. Universal Interactive, Sony, Konami, Vivendi Games, Sierra, Activision.

Entertainment, R. (1999). Homeworld. Sierra Studios.

Fisher, J. (2012). How to make insane, procedural platformer levels.

Forsblom, J. and Johansson, J. (2018). Genetic Improvements to Procedural Generation in Games.

Games, E. (2010). Galactic arms race. Microsoft Windows.

Games, G. (2017). Horizon zero dawn. Sony Interactive Entertainment (PS4), Microsoft Windows.

Games, H. (2016). No man's sky. Sony Interactive Entertainment (PS4), Microsoft Windows.

Gillian, S. (2011). Launchpad: A rhythm-based level generator for 2-d platformers.

GmbH, T. (2005). Teamviewer. https://www.teamviewer.com. [Online; accessed May 8, 2020].

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning.

Holland, J. H. (1975). Adaptation in natural artificial systems. *The U. of Michigan Press*.

id Software (1993). Doom. GT Interactive.

Imabayashi, H. (1981). Sokoban. Thinking Rabbit.

J.Doran and I.Parberry (2010). Controllable procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*.

Kerssemakers, M., Tuxen, J., Togelius, J., and Yannakakis, G. N. (2012). A procedural procedural level generator generator. *2012 IEEE Conference on Computational Intelligence and Games, CIG 2012*, (September):335–341.

Khalifa, A. (2009). Mario ai framework. https://github.com/amidos2006/Mario-AI-Framework. [Online; accessed October 16, 2019].

Koza, J. (1992). Genetic programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems) . *A Bradford Book*.

Lucas, S. M., Mateas, M., Preuss, M., Spronck, P., and Togelius, J. (2012). Artificial and Computational Intelligence in Games (Dagstuhl Seminar 12191). *Dagstuhl Reports*, 2(5):70.

Mallawaarachchi, V. (2017). Introduction to genetic algorithms - including example code. https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3. [Online; accessed October 3, 2019].

Mawhorter, P. and Mateas, M. (2015). Procedural level generation using occupancy-regulated extension.

Michael Toy, Glenn Wichman, K. A. (1980). Rogue: Exploring the dungeons of doom. Amiga, Amstrad CPC, Atari 8-bit, Atari ST, Commodore 64, DOS, Macintosh, TOPS-20, TRS-80 CoCo, Unix, ZX Spectrum.

Microsoft (2016). Microsoft teams. https://www.microsoft.com/pt-pt/microsoft-365/microsoft-teams/free. [Online; accessed May 10, 2020].

Millington, I. (2019). *AI for Games*, pages 195–291. Third edition.

Miyamoto, S. (1981). Donkey kong. Nintendo.

Moghadam, A. B. and Rafsanjani, M. K. (2017). A genetic approach in procedural content generation for platformer games level creation. *2nd Conference on Swarm Intelligence and Evolutionary Computation, CSIEC 2017 - Proceedings*, (February):141–146.

Montreal, U. (2008). Far cry 2. Ubisoft.

N. Shaker, J. T. and Yannakakis, G. N. (2010). Towards Automatic Personalized Content Generation for Platform Games. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Nintendo (1992). Mario kart.

Nolte, H. (2010). Cavecopter. http://www.pygame.org/project-Cave+Copter-1405-.html. [Online; accessed December 16, 2019].

North, B. (1996). Diablo. Blizzard Entertainment Electronic Arts (PlayStation).

Oliveira, T. P. (2011). Rambit. https://www.pygame.org/project-Rambit-2141-.html. [Online; accessed November 5, 2019].

On, C. K., Foong, N. W., Teo, J., Ibrahim, A. A. A., and Guan, T. T. (2017). Rule-based procedural generation of item in Role-Playing Game. *International Journal on Advanced Science, Engineering and Information Technology*, 7(5):1735–1741.

Paris, U. (2017). Tom clancy's ghost recon wildlands. Sony Interactive Entertainment (PS4), Microsoft Windows, Xbox One.

Persson, M. (2011). Minecraft. Mojang.

Pete Shinners, M. B. (2000). Pygame. https://www.pygame.org/news. [Online; accessed December 16, 2019].

Pygame (2018). Pygame — wikipedia, the free encyclopedia". https://en.wikipedia.org/wiki/Pygame. [Online; accessed 16 December, 2019].

R. M. Smelik, T.Tutenel, K. J. d. K. and Bidarra, R. (2010). Integrating procedural generation and manual editing of virtual worlds. *Proceedings of the ACM Foundations of Digital Games*.

Rakesh Kumar, J. (2012). Proceedings of the International Conference on Information Systems Design and Intelligent Applications 2012 (INDIA 2012) held in Visakhapatnam, India, January 2012.

Re-Logic (2009). Terraria.

Remo, C. (2008). MIGS: Far Cry 2'a Guay on the importance of procedural content.

Roberts, J. and Chen, K. (2015). Learning-based procedural content generation. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(1):88–101.

Shaker, N. (2012). Towards Player-Driven Procedural Content Generation.

Shaker, N., Togelius, J., Yannakakis, G. N., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., Pasquier, P., Mawhorter, P., Takahashi, G., Smith, G., and Baumgarten, R. (2011). The 2010 mario ai championship: Level generation track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4):332–347.

Sharif, M., Zafar, A., and Muhammad, U. (2017). Design Patterns and General Video Game Level Generation. *International Journal of Advanced Computer Science and Applications*, 8:393–398.

Shigeru Miyamoto, N. R. and 4, D. (1985). Super Mario Bros.

Shigeru Miyamoto, T. T. (1986). The legend of zelda. Nintendo.

Silva, M. P., Silva, V. D. N., and Chaimowicz, L. (2016). Dynamic difficulty adjustment through an adaptive AI. *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, pages 173–182.

Sivanandam, S. and Deepa (2008). Introduction to Genetic Algorithms. *Springer*.

Smith, G., Treanor, M., Whitehead, J., and Mateas, M. (2009). Rhythm-based level generation for 2D platformers. *FDG 2009 - 4th International Conference on the Foundations of Digital Games, Proceedings*, (January 2009):175–182.

Smith, A.M., M. (2011). Answer Set Programming for Procedural Content Generation: A Design Space Appraoch. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200.

Software, G. (2009). Borderlands. 2K Games.

Software, O. (1983). M.u.l.e. Electronic Arts.

Software, O. (1992). Global conquest. Microplay Software.

Studio, C. (2006). God hand. Capcom.

Studios, B. G. (2015). Fallout 4. Sony Interactive Entertainment (PS4), Microsoft Windows, Xbox One.

Studios, T. R. (2008). Left 4 dead. Valve Corporation.

Suzuki, Y. (1986). Outrun. Sega.

Tanya X. Short, T. A. (2017). *Procedural generation in game design*, pages 57–70.

Tarn Adams, Z. A. (2006). Slaves to armok: God of blood chapter ii: Dwarf fortress. Bay 12 Games, Kitfox Games.

.theprodukkt (2004). .kkrieger.

Tiburon, E. (2008). Madden nfl 09. EA Sports.

Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2010). Search-based procedural content generation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6024 LNCS(PART 1):141–150.

Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186.

Ulisses, J., Gonçalves, R., and Coelho, A. (2015). Procedural Generation of Maps and Narrative Inclusion for Video Games.

Ward, I. (2007). Call of duty 4: Modern warfare. Activision.

Yannakakis, G. N. and Togelius, J. (2015). Experience-driven procedural content generation (Extended abstract). *2015 International Conference on Affective Computing and Intelligent Interaction, ACII 2015*, pages 519–525.

Yuji Naka, N. O. and Yasuhara, H. (1991). Sonic the hedgehog. Sega.

Zohaib, M. (2018). Dynamic difficulty adjustment (DDA) in computer games: A review. *Advances in Human-Computer Interaction*.