

TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING



DIPLOMA THESIS

**“Personalised Platformer-Game Generation
via PCGML Algorithms”**

Savvas Chaitidis

Thesis Committee:

Michail G. Lagoudakis, Associate Professor

Katerina Mania, Associate Professor

Georgios N. Yannakakis, Professor (University of Malta)

Chania, July 2021

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ



ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**“Εξατομικευμένη Δημιουργία ενός Platformer Παιχνιδιού
μέσω PCGML Αλγορίθμων”**

Χαϊτίδης Σάββας

Επιτροπή:

Μιχαήλ Γ. Λαγουδάκης, Αναπληρωτής καθηγητής

Κατερίνα Μανιά, Αναπληρώτρια καθηγήτρια

Γεώργιος Ν. Γιαννακάκης, Καθηγητής (Πανεπιστήμιο Μάλτας)

Χανιά, Ιούλιος 2021

Abstract

In recent years, the game development industry has shown an enormous interest in the field of Procedural Content Generation (PCG). Especially the autonomous generation of levels for video games has become a popular subject of study. Generative Adversarial Networks (GANs) are a machine learning method that has been used over the last decade and proved to be capable of generating media content and even game content. For this thesis, a new video game that combines the characteristics of a Platformer and a Dungeon Crawler has been developed in Unity from which a handcrafted dataset has been created. A GAN was successfully trained on this dataset and was therefore able to generate new level rooms. Since the core of this work was to generate rooms whose difficulty scale is adjusted based on the player's performance, the input latent vector to the generative model was found by using an Evolutionary Strategy. Specifically, the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) was applied which searched through the latent space of the GAN by utilizing the static characteristics of each room in form of a fitness function. The output room of this process was then given a difficulty score by evaluation and placed in the correct spot of the 3x3 grid which constitutes the whole level. The experiments of this thesis are resulted from having a variety of player models and actual players play the game and show how capable the trained GAN model is in generating novel example outputs that fit the player's performance.

Περίληψη

Τα τελευταία χρόνια, η βιομηχανία ανάπτυξης παιχνιδιών έχει δείξει τεράστιο ενδιαφέρον στον τομέα της διαδικασίας αυτόματης δημιουργίας περιεχομένου (Procedural Content Generation - PCG). Ειδικά η αυτόνομη παραγωγή επιπέδων για βιντεοπαιχνίδια έχει γίνει ένα δημοφιλές αντικείμενο μελέτης. Τα Γενετικά Ανταγωνιστικά Δίκτυα (Generative Adversarial Networks - GANs) είναι μια μέθοδος μηχανικής μάθησης που έχει χρησιμοποιηθεί την τελευταία δεκαετία και αποδείχθηκε ικανή να δημιουργεί περιεχόμενο πολυμέσων και περιεχόμενα παιχνιδιών. Για αυτήν την εργασία, ένα νέο βιντεοπαιχνίδι που συνδυάζει τα χαρακτηριστικά ενός Platformer και ενός Dungeon Crawler αναπτύχθηκε στην Unity, από το οποίο δημιουργήθηκε ένα σύνολο δεδομένων. Ένα GAN εκπαιδεύτηκε με επιτυχία στο συγκεκριμένο σύνολο δεδομένων και κατά συνέπεια μπόρεσε να δημιουργήσει νέα επίπεδα του παιχνιδιού. Δεδομένου ότι ο πυρήνας αυτής της εργασίας ήταν να δημιουργηθούν επίπεδα των οποίων η κλίμακα δυσκολίας προσαρμόζεται με βάση την απόδοση του παίκτη, το διάνυσμα εισόδου στο μοντέλο δημιουργίας βρέθηκε χρησιμοποιώντας μια Εξελικτική Στρατηγική. Συγκεκριμένα, εφαρμόστηκε το Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) που αναζητούσε τον λανθάνοντα χώρο του GAN χρησιμοποιώντας τα στατικά χαρακτηριστικά κάθε επιπέδου με τη μορφή μιας συνάρτησης καταλληλότητας. Στη συνέχεια, στην έξοδο αυτής της διαδικασίας δόθηκε βαθμολογία δυσκολίας με αξιολόγηση και τοποθετήθηκε στο σωστό σημείο του πλέγματος 3x3 που αποτελεί ολόκληρο το επίπεδο παιχνιδιού. Τα πειράματα αυτής της εργασίας προέρχονται από μια ποικιλία μοντέλων παικτών και πραγματικών παικτών, οι οποίοι έπαιξαν το παιχνίδι και επέδειξαν πόσο ικανό είναι το εκπαιδευμένο μοντέλο GAN στη δημιουργία νέων επιπέδων που ταιριάζουν στην απόδοση του παίκτη.

Acknowledgements

First, I would like to take this opportunity to express my deepest appreciation to everyone who stood next to me during my studies from the very first moment till the last one. A special thank you to Prof. Georgios N. Yannakakis for guiding and encouraging me throughout my thesis showing his support and availability whenever I needed him. I would also like to thank Prof. Katerina Mania and Prof. Michail G. Lagoudakis for accepting being part of the thesis committee. Furthermore, I would like to recognize the invaluable assistance that my family provided during all my years of study at the Technical University of Crete that proved monumental towards this milestone in my life. Thank you.

Contents

Chapter 1	1
Introduction	1
1.1 Thesis Contribution	1
1.2 Structure of the thesis	2
Chapter 2	3
Backround	3
2.1 AI and Games	3
2.2 Generative Adversarial Networks	6
2.3 Latent Variable Evolution	7
2.4 Covariance Matrix Adaptation Evolution Strategy	8
2.5 Procedural Content Generation	10
2.6 Unity Game Engine	11
Chapter 3	12
GANgeon Escape	12
3.1 Game Concept	12
3.2 Game Components	14
3.3 The Level Layout	15
3.4 Character Movement	19
Chapter 4	20
Implementation	20
4.1 Creating a handcrafted dataset	20
4.2 Setting up the structure of the GAN	23
4.3 Training the GAN model	24
4.4 LVE to create levels with specific difficulty	26
4.5 Evaluating the level as a whole	30
Chapter 5	32
Results	32
5.1 Non-agent-based testing	32
5.2 Player testing	36
Chapter 6	43
Conclusion	43
6.1 Limitations	43
6.2 Future Work	44

6.3 Summary.....	44
References	46
Appendix A – Additional player analysis data.....	48

Chapter 1

Introduction

Generative Adversarial Networks (GANs) [1] is a recently emerging generative algorithm that utilizes Machine Learning (ML) algorithms to generate novel output instances of media content, such as images, videos and audio. There has also been a growing interest in utilizing the functionalities of the GAN to create content for video games, such as visuals, audio, mechanics and whole levels. Especially game domains that can provide a sufficient amount of pre-existing levels can be used as a training base for the GAN to generate novel levels with characteristics and features similar to the ones designed by the developers of the game. Such domains that have been used in prior research for procedurally generated content (PCG) are *Spelunky* (Mossmouth, LLC, 2013), *No Man's Sky* (Hello Games, 2016), *Super Mario Bros* (Nintendo 1985) and *StarCraft* (Blizzard Entertainment, 1998). Therefore, PCG is a game AI area which has existed since the early 1980s and has seen an enormous growth of interest since then with a highly promising future of implementations [2].

From the developers' side of view, having PCG systems during development could lead to numerous benefits. First, the production time of an upcoming title would be lowered significantly, allowing developers to focus on other aspects, such as marketing, to increase the games success chances. More importantly, since game content can be generated on its own, the development costs would also face a serious reduction. These two factors can lead to an overall better product that ships faster to the market with less effort and costs.

1.1 Thesis Contribution

The purpose of this thesis is to make use of PCG-ML algorithms [3] in order to generate game content in form of levels that are personalised and adjusted to each player and their performance. For this purpose, Generative Adversarial Networks (GANs) are applied and trained on a custom dataset of levels from the game *GANgeon Escape* that was designed and developed for the purposes of this thesis. Although the use of a trained GAN model proved to be suitable in generating novel output levels for the game, meeting specific characteristics and design objectives was restricted by this approach. The solution to this drawback proved to be the application of the latent space exploration of the GAN in order to find the most suitable

latent vector as input to the generative model. The above proposal was achieved by using the evolutionary algorithm CMA-ES following the method introduced in [4] and applied in [5] by Volz et al. for the generation of *Super Mario Bros* levels. In particular, levels of the game *GANgeon Escape* were encoded as 2D vectors, where each integer value was mapped to a specific tile component of the game. After the creation of a level, a handcrafted fitness function extracts specific elements of interest within a level and the CMA-ES starts to explore the input space of the GAN. This process is continuously executed until a suitable candidate is found and a level with the desired characteristics is generated.

The experimental results of having player models with a predetermined player performance and actual players testing the game, showed that the approach followed was capable of generating novel content personalised to each player. Even with a limited dataset on which the GAN was trained, the presented methods were successfully applied. The created levels proved to have an adjusted difficulty scale based on the player performance providing both a challenge and a satisfaction factor for everyone. The outcomes of this thesis are directly applicable to PCG and game design at large, but they are extendable to domains, such as architecture and interactive art.

1.2 Structure of the thesis

In this section we outline the structure of this thesis:

- In Chapter 2, the needed background information for this thesis is presented together with an overview of the implementation mechanics. The structure of a GAN and a CMA-ES model is provided and correlated to the field of AI and Games.
- In Chapter 3, the creation of the 2D platformer game for this thesis, *GANgeon Escape*, is presented together with its mechanics and gameplay logic.
- In Chapter 4, we describe in detail the implementation process of the proposed approach.
- In Chapter 5, the results derived from testing our approach and their analysis are displayed in form of graphs and tables.
- In Chapter 6, an overall conclusion and the summary of the thesis is given pointing out the limitations and the future work of this project.

Chapter 2

Background

In this chapter we present the theoretical background of the methods used in this thesis together with applied examples on other popular game domains. The specific topics that will be further analysed and discussed in the upcoming sections are:

- AI and Games
- Generative Adversarial Networks (GANs)
- Latent Variable Evolution (LVE)
- Covariance Matrix Adaptation Evolution Strategy (CMA-ES)
- Procedural Content Generation
- Unity Game Engine

2.1 AI and Games

Artificial intelligence is closely linked with the field of game development and the relationship between them has been a long one. Yannakakis and Togelius [2] briefly describe how classic board games like Chess and Checkers, with very simple programming requirements and fast simulations, have been the first domains for AI research.

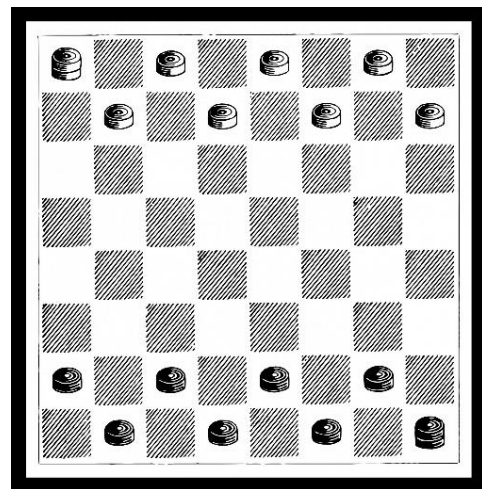
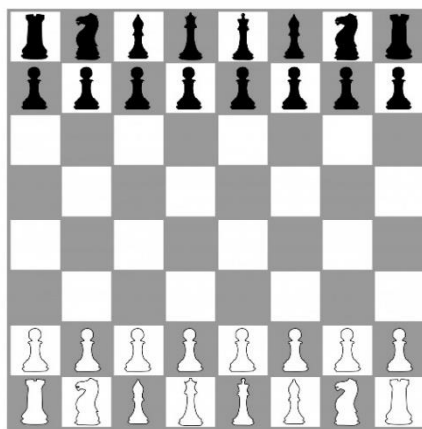


Figure 2.1.1: Chess and Checkers, the first domains for AI research

Chess specifically was the board game with the most AI implementations that tried for years to simulate a human-like play style or even a better one that could play against humans and beat them. A huge milestone in the above approach took place in 1997 when Garry Kasparov, the grandmaster of chess, was beaten by IBM's Deep Blue [6]. A chess machine with complex evaluation functions and a very effective analysis of a Grandmaster game dataset.

Over the next years more and more research has been carried out on applying AI methods to games that are not utilizing turn-based mechanics like the most boardgames. In 2014 the next big milestone was achieved with the Google DeepMind algorithm that was able to play video games on the classic Atari 2600 console. It did not stop there though, since with the capabilities of the always evolving computer hardware, even more complex and modern games were targeted by AI researchers. The reason and the way of using AI have also evolved, from the traditional approach of having agents playing a game, to far more creative and sophisticated methods. Procedural content generation, analysing player experience and behaviour, difficulty adjustment and game evaluation are some of the most popular approaches for increasing the potential of a game.

Games with a growing fanbase require even more content to keep player satisfied for longer play sessions and time in general. AI plays an immense role when it comes to generating game content like audio, gameplay, design and of course the visuals. As a result, this automated process takes much less effort and time than having developers manually handcrafting each asset and aspect of their game. In addition to that, shipping an algorithm that can generate countless variations of game content by just (re)using build-in assets, makes the final product small in size and efficiently playable on all devices.

One great example of using AI in games with the goal of procedurally generating worlds is the grid-based platforming game *Spelunky* (Mossmouth, LLC, 2013). A *Spelunky* level can be represented as a 4x4 grid that is made of 16 different rooms as seen in Figure¹ 2.1.2. Each room has its own unique features with possible openings on each direction to let the player navigate through the level. Randomly placed enemies and traps also provide a great variety of challenges that a player must overcome while keeping the playthrough as interesting as possible. The last thing to do after a full set of 16 rooms is generated, is to find a suitable

¹ <http://tinysubversions.com/spelunkyGen/>

solution path for the level. For the player to beat the level, exploring the solution path is necessary and part of the huge fun factor the game provides.

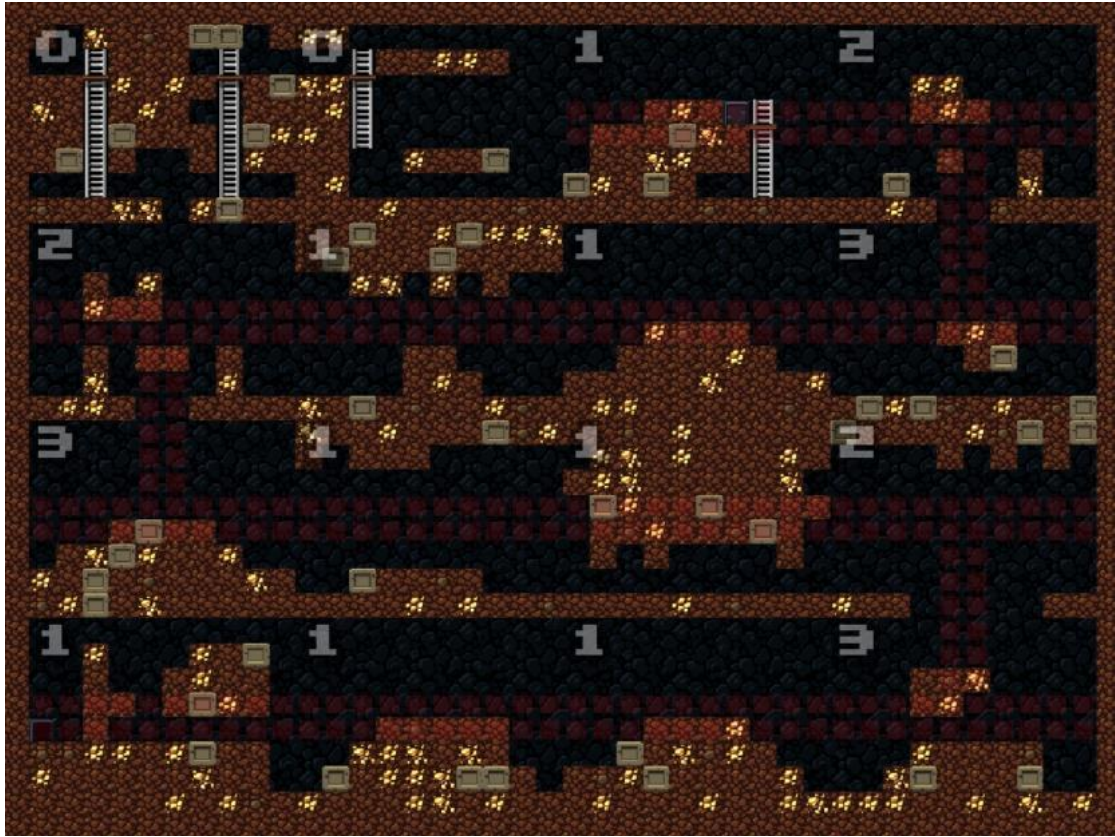


Figure 2.1.2: A generated level from the game Spelunky obtained from the Spelunky Generator Lessons by Darius Kazemi

The above-described approach of generating levels for a grid-based game will be used as a baseline for the game developed by the author specifically for this thesis. *Spelunky* has been a great inspiration for *GANgeon Escape* that will be described and analysed further in chapter 3.

Finally, it needs to be mentioned that AI in games not only serves for a better gameplay experience from the view of a player, but it also contributes to minimizing the effort, time and struggle a developer faces while creating and improving the game. Data and analytics that derive from players, the game itself and the way it is played can be used to enhance the marketing strategies, the overall game production and provide a better understanding about the future of the game and its potential in the market. As a result, having access to such AI-

informed decisions from the early stages of development can contribute to better and improved games.

2.2 Generative Adversarial Networks

Generative Adversarial Networks is a framework of two neural networks in which a generative model G and a discriminative model D are trained simultaneously. This was successfully established and described by Goodfellow et al. [1]. The overall objective of a GAN is to reproduce data based on a dataset on which it is trained. Both the generator (G) and the discriminator (D) compete like in a turn-based game where G creates artificial samples of an image and D differentiates between the generated samples and the real dataset. G takes a fixed-length random noise vector to generate the samples while D has a binary outcome of real or fake. Fake is the result image that comes from G whereas the real output derives from the provided training dataset.

The abovementioned process is started by providing a fixed batch of generated samples together with real images to the discriminator. That way D is consistently updated and learns to classify every sample as real or fake. This binary outcome is then sent to the generator which creates a new set of samples in order to outsmart the discriminator. On every round both G and D are getting better in providing more convincing results and improve their performance. In other words, the two models are competing against each other in a game where G tries to generate samples as close as possible to the real dataset and D tries to identify and distinguish real from fake. This exact architecture of the generative model is presented in Figure 2.2.1. In game theory this approach is also known as a zero-sum game² where a win of one player contributes to a loss of the other. The mathematical expression of this is shown in (2.1) which is a minimax equation where our generator has the task of minimizing the result and our discriminator to increase it.

$$\min_G \max_D \mathbb{E}_{\mathbf{x} \sim p_r} \log[D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z} \log[1 - D(G(\mathbf{z}))]$$

² https://en.wikipedia.org/wiki/Zero-sum_game

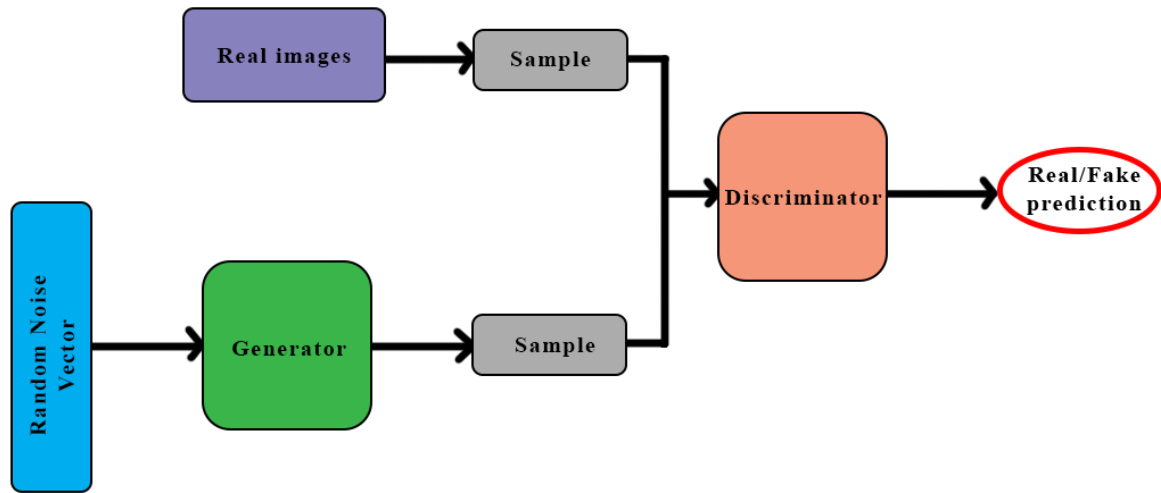


Figure 2.2.1: Overview of a GAN structure

Ultimately, after the training process is completed, both G and D are able to perform their tasks with high accuracy. At that point, the discriminator can be discarded since the main goal is to end up with a generator that takes a fixed-length latent space vector to generate samples as close as possible to the original dataset we provided. This generator should produce outputs that are accepted as valid samples with characteristics such as the real images, but even so, there could be a great variety of different features from which only the wanted ones should be accepted. Previous studies have shown that a conditioned generation of samples, is achieved by searching for a latent space vector that best matches certain requirements.

2.3 Latent Variable Evolution

Generating samples that feature only the required characteristics, can be accomplished by exploring the latent space of features encoded by the GAN. A recent study by Bontrager et al. [4] presented the implementation of latent variable evolution (LVE) as a technique of generating finger prints by training a GAN on a real fingerprint dataset. This method proves the practical approach of searching the latent space of the generator for instances that meet the given requirements. LVE has also been explored by Schrum et al. [7] using tools that allowed

an interactive latent space exploration for GAN models that generate levels in video games. The output levels were optimised and created with attributes that could be changed and specifically selected by editing the latent variables.

A similar contribution by Volz et al. [5] for generating game levels with desired features, uses a fitness-based approach in the investigation of latent GAN vectors. In their research, the structure of a Mario level is carefully encoded based on its tiles where each tile type is mapped to a unique integer. The GAN is trained on a dataset of encoded Mario levels and produces new playable levels with characteristics that derive from latent space exploration.

Experiments on LVE were also carried out with the purpose of generating DOOM levels [8]. The approach includes a previously trained GAN that accepts an input vector of an encoded DOOM level. The generator network of the GAN can therefore be used to generate an image-based representation of a level that is mapped into a vector of real numbers. The approaches described in [4, 5, 8] combine the LVE with the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) which is an evolutionary algorithm capable of producing vectors of real numbers. The latent space input of the generator is explored via CMA-ES until the best matching vector is found to create a level with the wanted features as explained in Section 2.4. (CMA-ES).

2.4 Covariance Matrix Adaptation Evolution Strategy

In recent years evolutionary algorithms are widely used in combination with neural networks. As already mentioned, the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [9] is utilized in finding the best matching latent space input of the trained generative model which is then able to generate the optimized samples as shown in Figure 2.4.1. Prior studies [10] have shown that CMA performs a component analysis of past mutation steps in order to shape the new mutation distribution. A task that is efficiently applied in optimizing non-linear and non-convex problems. In the field of game development, a handcrafted fitness function is needed to evaluate the overall score of the generated game level based on its features. These features can be static and therefore obtained just by the level representation without having the need of using agents that first play the game and then score it based on their calculations.

The work conducted by Volz et al. [5] implements the above mentioned fitness function in combination with CMA-ES and achieves the creation of Mario levels with optimized static features. The result is a segment of a level which consists of a certain amount of ground tiles, tiles above a specified height and enemies. Another study [8] effectively explores the input vector of the GAN using the evolutionary algorithm CMA-ES by maximizing a provided fitness function. Each input vector creates an image-based representation of a DOOM level which is then evaluated based on its features and ranked by its fitness. This fitness value is also responsible in updating the parameters of the CMA-ES so that a new iteration can start over generating another population of encoded levels.

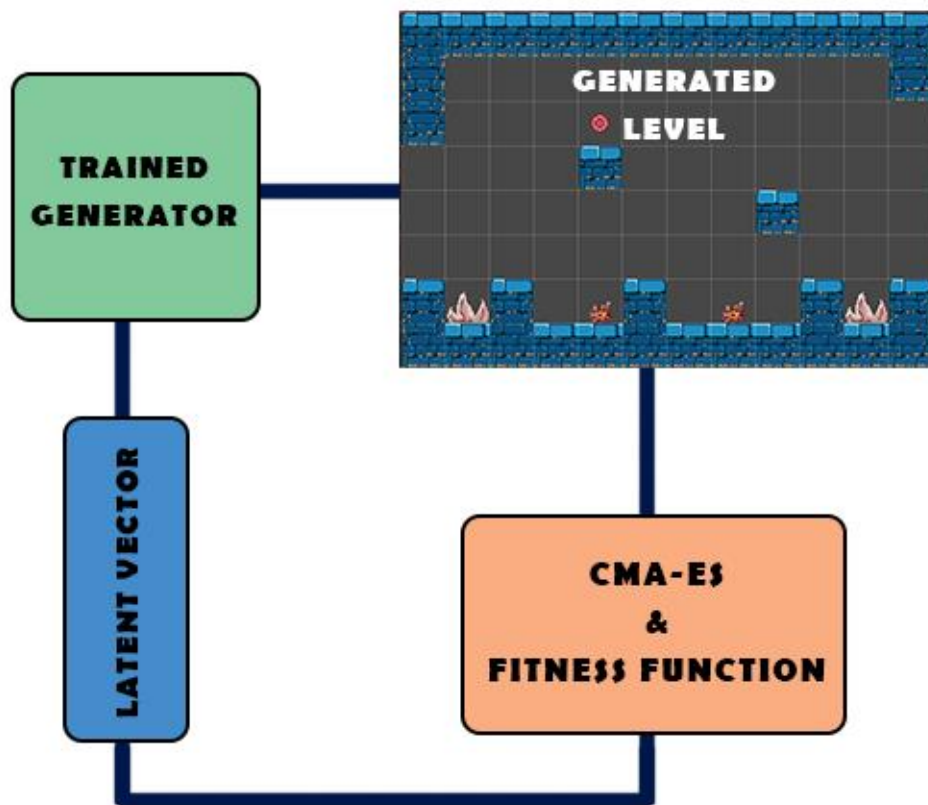


Figure 2.4.1: Overview of the latent space exploration with CMA-ES

2.5 Procedural Content Generation

In the field of game development, Procedural Content Generation (PCG) can be described as an automated process for generating non premade game content. Prior research by Yannakakis and Togelius [2] is exploring the various ways of implementing generated content that can be categorised in level design, game design, visuals, audio, narrative and gameplay mechanics. One of the most prominent applications of PCG is the generation of levels that are capable of meeting both the aesthetic and functional conditions. For instance, the playability of a level or even a whole game can be objectively measured and is part of the functional characteristics of the generated content. Other properties on the other hand, such as the graphics and the design can only be evaluated subjectively [2].

A series of recent studies have indicated that PCG is split into two important approaches of generating randomized game content. The first one is the search-based method (SB-PCG) [11] which utilizes evolutionary algorithms as their search mechanism. Each generation of generated samples is evaluated by a handcrafted fitness function that is able to score all samples based on how close to our desired needs their characteristics are. This process starts over with a new generation of candidates that are modified based on the best scored samples of the previous iteration. As a result, the evolutionary algorithm will eventually create content of high quality that can be compared to human crafted examples. The above approach is widely used in a great variety of game genres such as platformer games for the creation of playable and fun levels [12], Role Playing Games for generating player missions [13], first-person shooters for providing interesting map designs [14] and of course the generation of dungeons [15–17]

Machine Learning Procedural Content Generation (PCGML) is another recent field of study where game content is generated via machine learning models [3]. These models are firstly trained on a dataset that represents content of a specific game and afterwards used to generate new game content of high quality. There have been numerous studies to investigate the use of GANs while approaching PCGML implementations [5, 8] making it possible to control the outcome content by making use of Latent Variable Evolution (LVE) as described in 2.X. The most prominent domains for applying the abovementioned technique are *Super Mario Bros* [5], *The Legend of Zelda* [18] and the first-person shooter *DOOM* [8].

2.6 Unity Game Engine

Unity, developed by Unity Technologies, is a cross-platform game engine used to develop interactive content such as architectural visualizations, animations, interactive simulations and mostly games in both 2D and 3D. The editor itself is supported on Windows, macOS and the Linux platform. Even though Unity runs only on three platforms, the developed applications can be exported and build for more than 19 different platforms³, including desktop, consoles and mobile.

Unity was first announced and released in 2005 and has since been growing into one of the most popular game development engines with more that 60% of all games being developed in the Unity editor according to Unity CEO John Riccitiello. Out of all available platforms, the mobile game market is the fastest growing with an expected revenue of \$76.7 billion by the end of 2020.

The ease of use of the Unity engine and the plethora of online forums, videos and documentations with tutorials and step by step guides allow new developers to effortlessly create their own first game. The scripting environment is based upon the JavaScript and C# programming languages with build in libraries that allow a quick and efficient use of the provided components. Therefore, the Unity engine is not only used by big companies that develop games with high development and marketing budgets. Independent developers and smaller studios make also use of the beginner friendly Unity environment and are even able to compete with the big companies on every platform.

The game created specifically for this thesis is developed in Unity and exported for the Windows platform. The game itself and its development process will be further described in Chapter 3.

³ [https://en.wikipedia.org/wiki/Unity_\(game_engine\)](https://en.wikipedia.org/wiki/Unity_(game_engine))

Chapter 3

GANgeon Escape

The game developed in Unity for this paper is *GANgeon Escape* and in this section, the game concept, the mechanics and the overall functionality are briefly described and presented. The technique used for training the GAN on the dataset that was created for *GANgeon Escape* could be applied to any game within the same genre that provides a sufficient amount of levels. The game itself has taken a lot of inspiration from the *Spelunky*⁴ (Mossmouth, LLC, 2013) franchise but has its unique challenges and playstyle.

3.1 Game Concept

GANgeon Escape is a 2D platformer game with cave exploration mechanics where the goal is to escape from the dungeon in time before lava covers the whole level burning everything in its path. The main character, which the player is controlling, is an Imp (Figure 3.1.1) that can jump, climb and slide on walls, hit enemies and gather diamonds. All of that through levels that consists of 9 rooms in a 3x3 grid.



Figure 3.1.1: Imp, the main game character

⁴ <https://spelunkyworld.com/original.html>

At the beginning of the level, the player starts in one of the bottom rooms and starts as soon as possible to find the critical path that will lead him to the portal in one of the top rooms. The portal is the gate that will allow the Imp to continue on the next level that will have a difficulty scale which will be adjusted based on the player performance in the previous level. During the exploration through the rooms, the player will face enemies and traps and when he gets too close to them, he loses one of his three life points. In addition to that, lava starts to raise from the bottom of the level and after a specific duration it will reach the top of the level giving the player no room to escape. In every next level, the number of enemies, traps, the amount of solid walls and the time it takes for the lava to reach the top are adjusted in a way that provides a straightforward game progression and experience. Examples of generated levels will be presented in Chapter 5 together with the experiments which led to their creation.

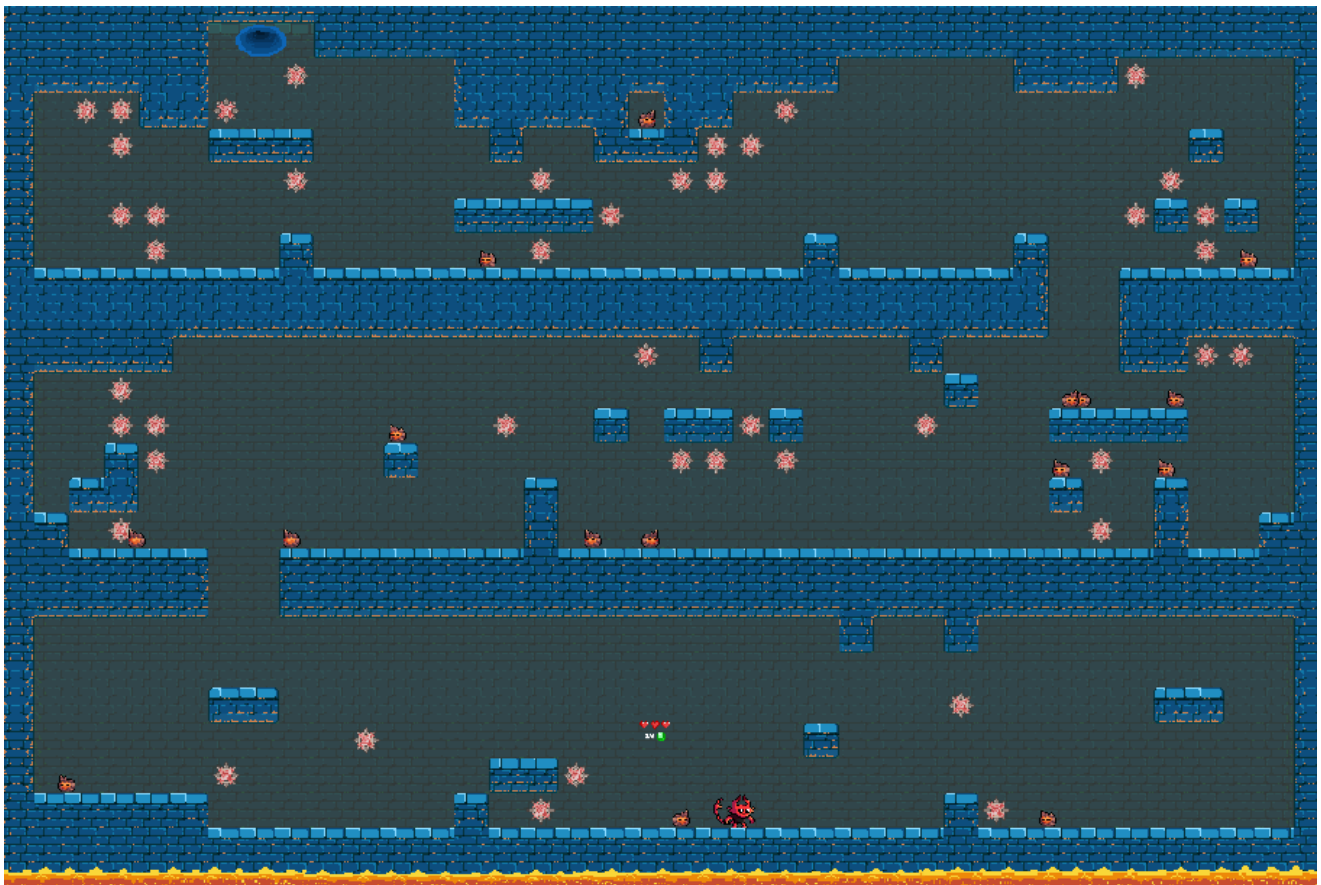


Figure 3.1.2: Screenshot from a fully generated level in Unity

3.2 Game Components

The main components of the game beside the main character are as abovementioned, the solid blocks, the spikes, the enemies and the coins. Figure 3.2.1 shows the sprite representation of each object exactly as it is used in *GANgeon Escape*. The solid blocks serve as physical obstacles that can either block a path entirely or help the player in reaching higher places by jumping on top of them. The spikes are placed all around the level and are static objects that are not affected by gravity. Enemies on the other hand can move horizontally and change path whenever they reach a block or the end of a platform. Spikes and enemies reduce the life of the player by one whenever he gets in touch with them, but the Imp character is also able to perform an attack animation which when used correctly destroys either a spike or an enemy. Coins are the only beneficial item in the game which the player can find by exploring all rooms in the level even if they are not part of the critical path.

Tile type	Sprite
Solid Block	
Spikes	
Enemy	
Coin	

Figure 3.2.1: The sprite representation of each tile type in the game

In addition to the main components, there are two more very important objects in the game. The first one is the lava sprite which is placed at the bottom part of every level the player begins to play. Right after the game begins, the lava object starts to move upwards at a consistent speed which increases depending on which level the player is currently on. Once the lava touches the player, the game is immediately over since all life points are reduced to zero and the level start over for another try. The other object is the portal, which the player must

find and enter in order to complete the level and trigger the player performance evaluation function that runs in the background. The portal is automatically spawned in one of the top three rooms where the critical path ends and it can be either effortlessly reached or much more challenging based on the difficulty of its room. Both these objects can be seen in Figure 3.2.2.

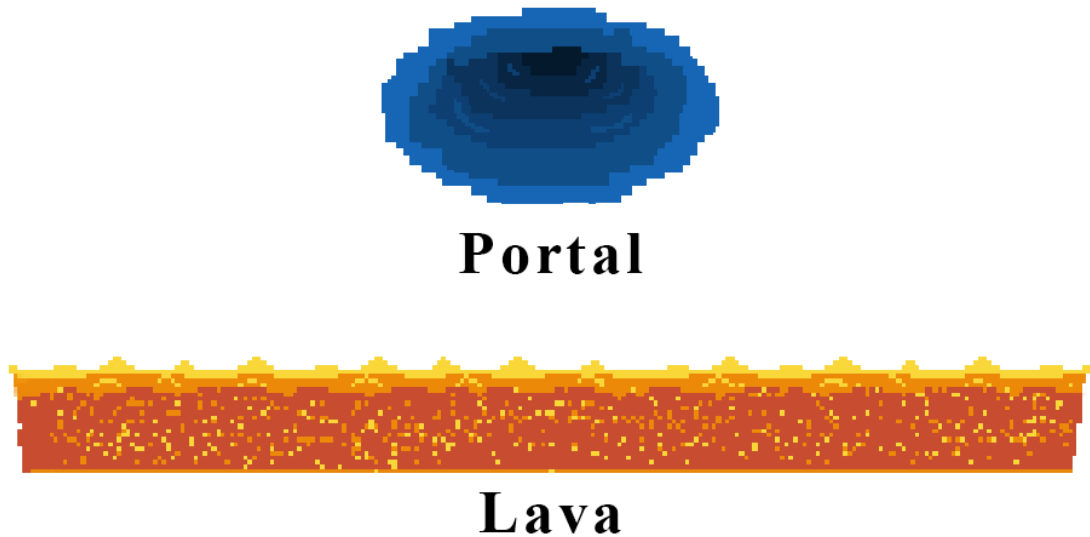


Figure 3.2.2: The portal asset which the player needs to find in order to win and the lava object which ends the game if touched

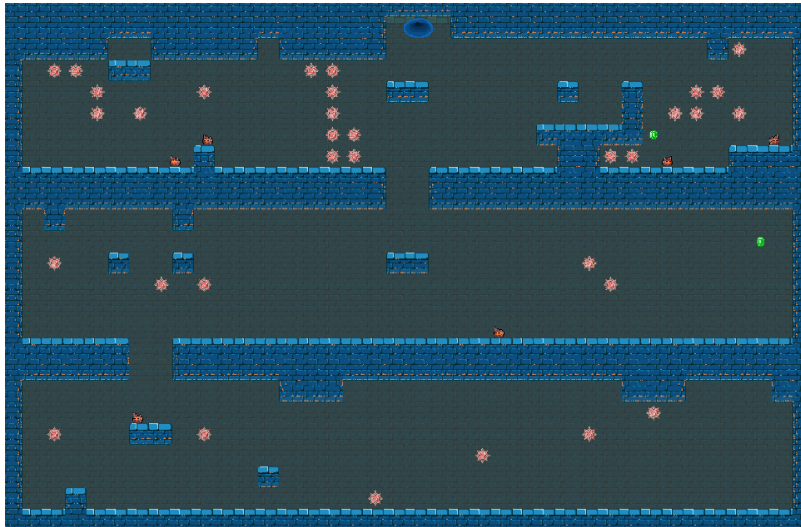
3.3 The Level Layout

Each level of *GANgeon Escape* consists of 9 different rooms that are generated with the trained generative model used in this thesis. Their size in tiles is exact 12x8 These rooms are categorized under three difficulty levels:

- Easy difficulty, for beginner and new players
- Medium difficulty, for players with a little gameplay experience
- Hard difficulty, for the most experienced players

More specifically, the difficulty level is closely linked with the total amount of enemies, spikes and the solid block distribution of the room. The higher the number of these features, the more challenging it is for the player to proceed to the next room. Below are examples of all level difficulties presented together with their total amount of spikes, enemies and solid blocks. The

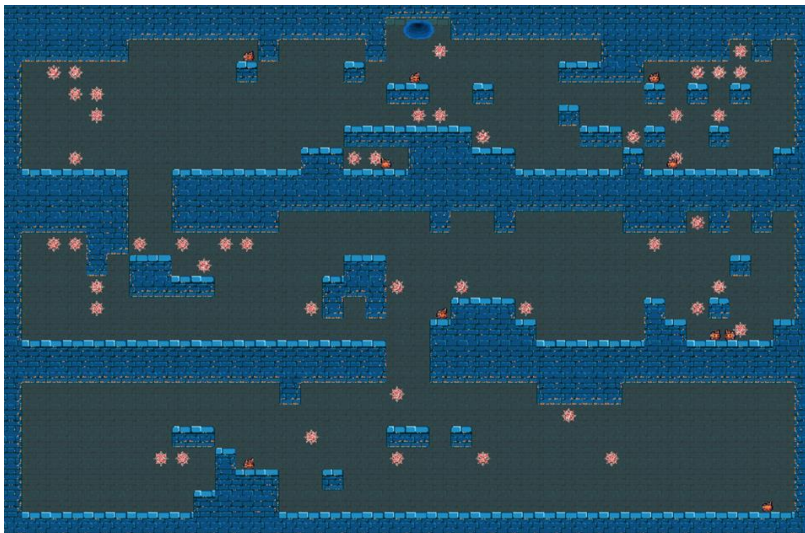
algorithm that computes the level difficulty by sending the above values to the trained GAN will be further explored in Chapter 4.



EASY LEVEL

SPIKES:	33
ENEMIES:	6
BLOCKS:	262

Figure 3.3.1: A representation of an “Easy” level generated by the GAN



MEDIUM LEVEL

SPIKES:	46
ENEMIES:	10
BLOCKS:	330

Figure 3.3.2: A representation of a “Medium” level generated by the GAN



HARD LEVEL

SPIKES: **56**
ENEMIES: **19**
BLOCKS: **360**

Figure 3.3.3: A representation of a “Difficult” level generated by the GAN

Moving from one room to another is possible in all direction as long as an opening is connecting them, through which the player can pass. By default, all rooms are created with an opening to the left and right. The decision for that implementation was carefully planned and afterwards tested, giving the player the total freedom of horizontal movement through the whole level. Vertical openings are only available in rooms of the critical path that are placed on top of each other and are created in script in runtime after all rooms are placed correctly and the critical path is calculated. For calculating the critical path, a function runs at the beginning of the game and chooses a random room from the bottom line which will be the starting point. From there, randomly again, the function takes into consideration all available neighbour rooms and continues in creating the path. Each direction has a fixed probability of how likely it is to place a room based on that:

- Place a room to the left, $p = 2/5$
- Place a room to the right, $p = 2/5$
- Place a room above, $p = 1/5$

These numbers indicate that it is much more likely to move horizontally than moving upwards. A more interesting level with deeper exploration elements is provided which provide a satisfying playthrough. The critical path algorithm comes to an end when it reaches the top line of rooms and tries from there to move one more step up. In the exact room in which that happens the portal is spawned and the room is the final one in the critical path. Figure 3.3.4 shows a complete level with the path the player needs to follow in order to complete it.

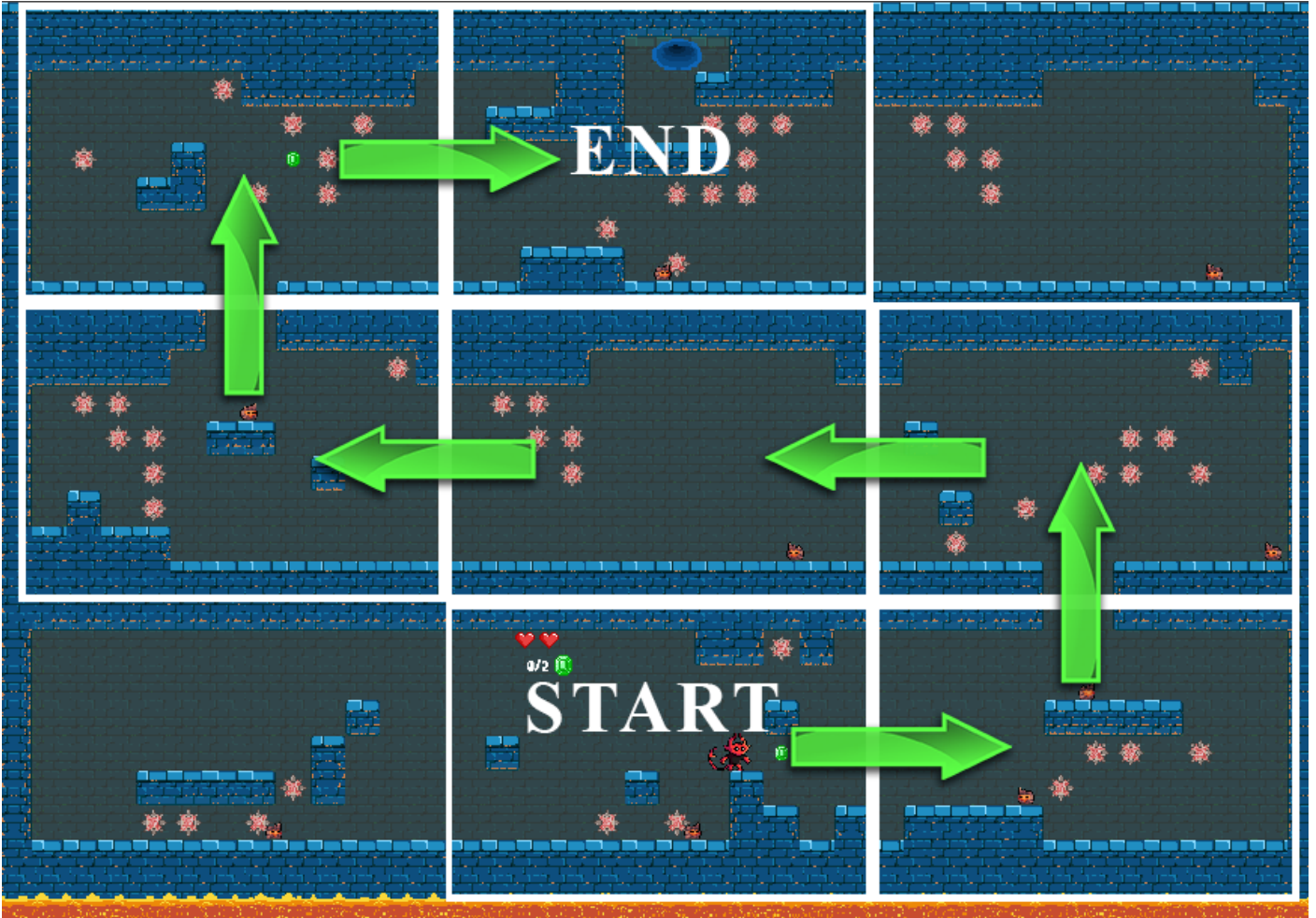


Figure 3.3.4: The critical path which the player needs to find in order to complete the level

3.4 Character Movement

The main character of the game can be moved around by using either the standard left and right arrow keys or by clicking the A and D buttons of the keyboard for moving left and right accordingly. He has also the ability of jumping by pressing the Spacebar and the height of the jump itself can be controlled by holding the spacebar for a longer time until the maximum force is reached. When jumping against a wall, the Imp character can either slide down at a slower speed than simply falling or a wall jump can be performed. The wall jump allows the player to land on harder-to-reach places avoiding any interaction with enemies or spikes allowing him to collect all coins. Pressing the X button will perform a quick attack animation which can be used to defeat both the spike objects and the enemies. Figure 3.4.1 shows how a wall jump can be performed effectively to increase the chances of beating the level.

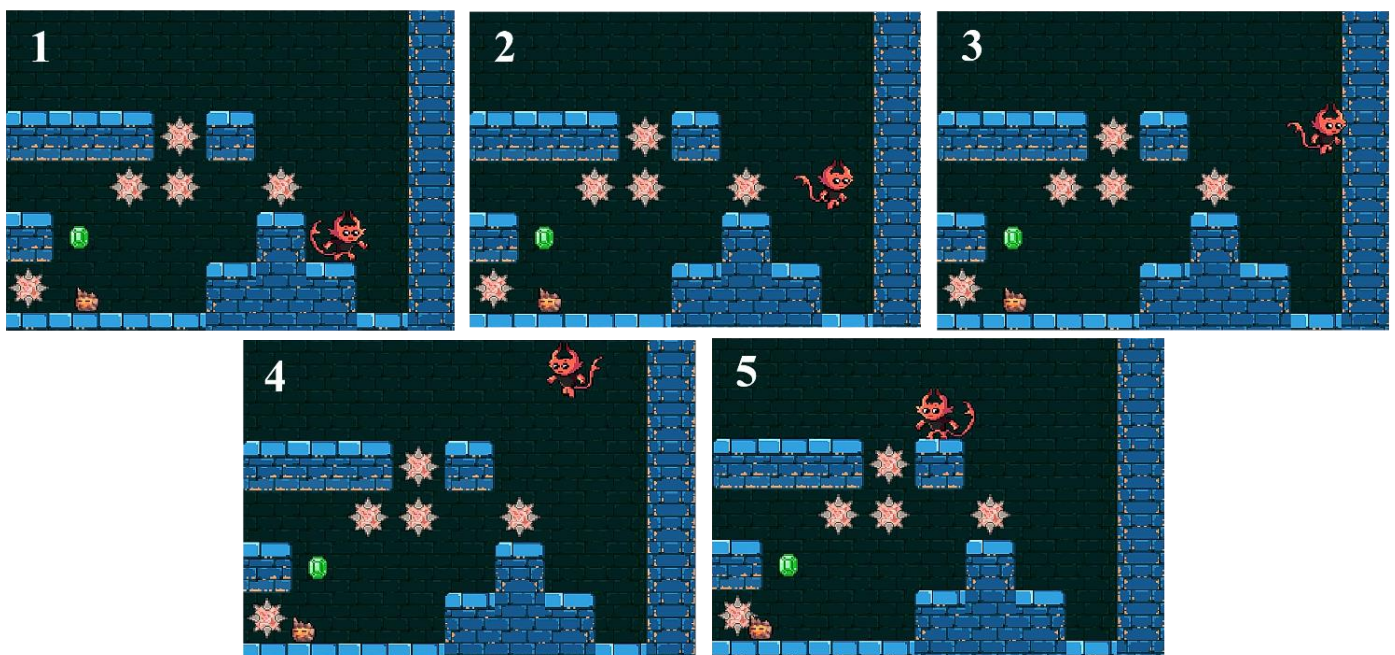


Figure 3.4.1: Performing a wall jump helps in reaching difficult locations and avoiding obstacles

Chapter 4

Implementation

This section will provide an overview of the overall implementation structure and the steps that were taken to achieve the desired results and features of the already described game in chapter 3.

The main goal was to create a personalised 2D platformer game inspired by *Spelunky* where each level is created on a difficulty scale based on how well the player did on his previously played level. Several variables were taken into consideration and scaled accordingly to calculate the next best outcome in order to provide an enhanced player experience and an overall satisfaction throughout a play session. The approach that was followed in this thesis consists of the following stages:

- Creating a handcrafted dataset
- Setting up the structure of the GAN
- Training the GAN model
- LVE to create levels with specific difficulty
- Evaluating the outcome

4.1 Creating a handcrafted dataset

The dataset was the first and most important step, since it represents the structure and the design of all levels and the game in general. After all game assets were done, they were implemented into Unity in order to be used for the creation of the dataset. Each level of the game consists of 9 independent rooms that are afterwards connected horizontally or vertically. From these connections the horizontal openings were the most important ones and it was decided from early development that from each room the player could have the possibility of going left and right. The up and down openings would be implemented at the very end and from inside Unity after the level creation.

The whole dataset consists of 100 different variations of a single room with various difficulty scales and patterns. When designing the room, it was necessary to keep the design and layout consistent for the most part. Specifically, the top and bottom lines were fully covered

with solid blocks and openings to the left and right were always provided. Figure 4.1.1 displays some of the rooms that are part of the dataset.

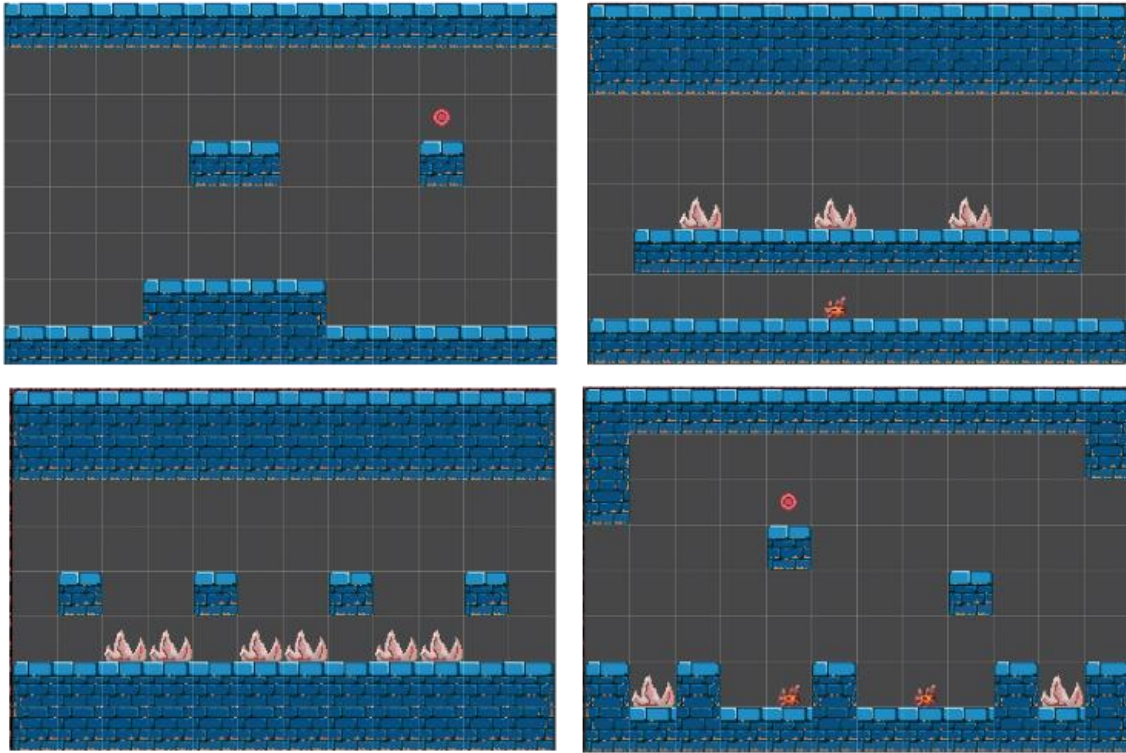


Figure 4.1.1: Different variations of rooms that are available in the dataset

Since the game is tile based where each asset takes the exact space as one tile on the screen, the level encoding method was straightforward with integers being mapped to each tile type (Figure 4.1.2). It needs to be mentioned that the integer value that represents the solid blocks is always 1 despite the fact that there are multiple variations of the same tile (ground, platform, wall). This was achieved by implementing the rule tiles within unity, where each solid block variation had its own unique rule. Whenever the right criteria of the rules were met, the correct variation was applied and took care of a seamless platforming environment. In Figure 4.1.3, it is shown how the abovementioned rules change the tile sprite accordingly choosing from preassigned variations.





Tile type	Identity	Sprite
Empty	0	
Solid Block	1	
Spikes	2	
Enemy	3	
Coin	4	

Figure 4.1.2: Each tile type is mapped to a unique integer value



Figure 4.1.3: The rule tiles used within Unity to create a variety of wall tiles using only 1 integer id value for the mapping

Since every tile is mapped to a unique integer, the final encoded version of a room is a 12 (wide) x 8 (high) 2D array that holds all the needed data. In order for the training dataset, that consists of 100 2D representations of rooms to be used as input to the discriminator model, it was first converted into a one-hot encoded vector. Afterwards, the one-hot vector was reshaped to fit a 16x16 image by padding each side of the array with zero values. These dimensions are then used as input to the discriminator and are also the same as the generators output.

4.2 Setting up the structure of the GAN

The GAN model used for generating the game content consists of two connected neural networks. One is called the generator (G) and the other one is called discriminator (D) [1]. D is trained in identifying which of its input is a real sample and which was a product of the generator. G is trained to provide content that resembles real data as close as possible. Therefore, the network is called generative adversarial because of the fact that both G and D try to compete with each other with the aim of getting better in executing their task.

The discriminator used in this thesis takes a three-dimensional input and outputs a one-bit binary integer. Since its main task is to identify if the input is real or generated, the one-bit output has the value 1 for 'real' and 0 for 'fake'. This conversion is achieved by passing the input vector sequentially through hidden layers that reduce the vectors dimensions but manage to keep its data intact. The architecture specifically consists of 3 Convolutional – LeakyReLU layers that compress and finally flatten the input vector.

The generator on the other hand, takes a random noise latent vector input which is firstly projected and reshaped. Then it is passed through 3 Convolutional – ReLU layers so that each time the width and the height get increased. The hidden layers here work as the opposite of the discriminator and form the output data in a way that can be used as input to the discriminator. Simultaneously, the channels keep on reducing and by the end of all hidden layers a 16 x 16 image is generated with 5 output channels. The overall architecture of the GAN that shows exactly the hidden layers and how they are set up can be seen in Figure 4.2.1.

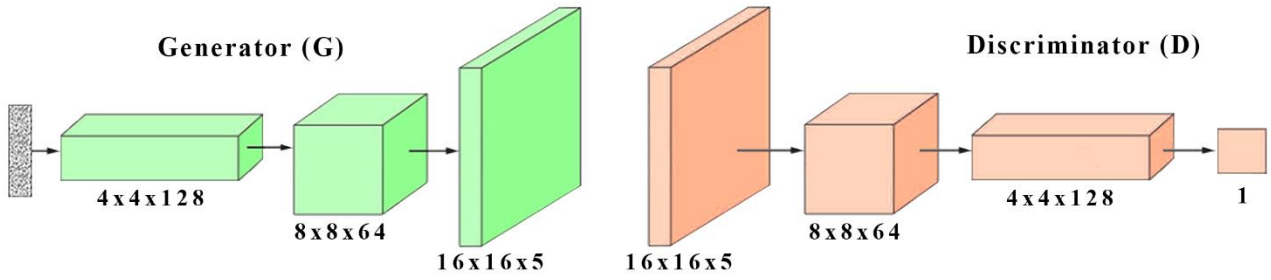


Figure 4.2.1: The architecture of the GAN with the hidden convolutional layers

4.3 Training the GAN model

In this section, the training process of the Generative Adversarial Network will be briefly described starting from the training parameters that were used during the process.

Training parameters:

1. Batch size = 16
2. Noise vector = 16
3. Learning rate = 0.00005
4. Epochs = 2000

The batch size is equal to 16 and is responsible for handling how many samples from the training dataset will be used together during each iteration. The next parameter, noise vector, has a size of 16 since this is also the input size of the generator. The learning rate value defines the speed of network training. Lastly the epochs, is the total number of training iterations that are performed on the whole training set. All the above numbers and values were found to be the most suitable for training the GAN with the best possible outcome result. For updating the weights of both the generator and discriminator, the optimization algorithm called Adam (Adaptive Moment Estimation) is used with the beta1 coefficient value set to 0.5 rather than the default 0.9. The binary cross-entropy function was used during training.

After initializing G and D together with their starting weights, the models are given training samples inside of the training loop and their weights are updated to minimize the loss function described earlier. Both models have their own training function which handles the results of each epoch and reassigns the weights and losses. The discriminator training function

takes in two parameters, a real sample and the optimizer variable. The next step inside of the function is to get real and fake labels at the size of the batch which are then filtered through the discriminator. After the filtering, the losses for the real and fake outputs and labels are calculated and added together in order to be returned. Based on that total loss all parameters of D are updated and the next epoch can start over. For the generator the training function has a lot in common with the previous one but takes only the optimizer as an input. Afterwards, a fake sample is generated and passed through the discriminator with a real label with the aim of convincing D that real data was passed. The loss value is then returned and the generator parameters are also updated accordingly. One training loop as described for D and G can be seen in Figure 4.3.1 where the generator creates a fake sample out of noise and passes it together with a real sample to the discriminator who is then responsible for predicting the outcome.

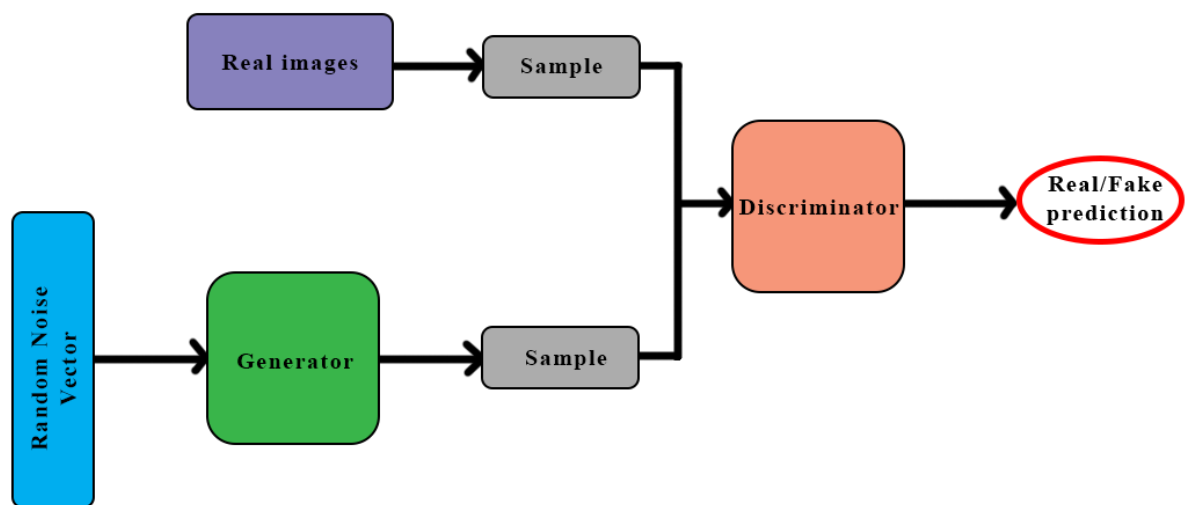


Figure 4.3.1: A training loop for the Generator and Discriminator

For the actual training, a basic for-loop is used that iterates through each epoch that we assigned in the training variables. Each epoch has another iteration through all the batches until the whole dataset is used. For every batch, the beforementioned training functions for both models are called and a simultaneously training process for G and D takes places. For each epoch and at the end of it the losses, the real scores and the fake scores are appended to corresponding arrays and are also printed out in the console. Finally, after all iterations, the trained generator is saved as a file and can be used later for the creation of game content. The

discriminator on the other hand can be discarded since its purpose is complete after the training process.

Finally, to generate a new room for the game a 16 bit random noise vector is passed to the trained generator which then reshapes it to a 16x16 image with 5 channels. This image should eventually look like a room with characteristics of the samples provided with the training dataset.

4.4 LVE to create levels with specific difficulty

The latent variable evolution approach was accomplished by exploring the latent space of features encoded by the GAN. The evolutionary algorithm that proved to be the most promising for this approach was the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [9] because of its capability of evolving vectors of real numbers. As described in chapter 2, CMA-ES finds the best matching latent space input of the trained generative model which is then able to generate the optimized samples based on the developer's needs.

The main goal of utilizing this mechanic was to generate levels whose difficulty was calculated and adjusted based on the performance the player scored in his previous level playthrough. This allows for a more personalized gameplay experience and an overall progression through the levels. The characteristics that are optimized in each room are static features like traps, enemies and the ground distribution but the data needed to sculp the players performance is more complex. Therefore, two separate fitness functions were used. One for analysing how well the player performed in the previous level and one for the evolutionary algorithm in order to adjust the static features of the next level.

The variables needed to compute the player's performance are:

1. t = Level completion time in seconds
2. c = Number of coins gathered
3. l = Lives lost
4. s = Number of spikes destroyed by the player
5. e = Number of enemies destroyed by the player

For each of the above variables, a performance score is calculated which is divided by the total amount that the specific variable can reach. The total value for variables like time and lives is a constant and was found through testing, whereas the total value for static features like coins, enemies and spikes changes for every generated level and is counted as soon as the level is generated:

1. $t_s = 1 - \frac{t}{t_{total}}$
2. $c_s = \frac{c}{c_{total}}$
3. $l_s = 1 - \frac{l}{l_{total}}$
4. $s_s = \frac{s}{s_{total}}$
5. $e_s = \frac{e}{e_{total}}$

This approach ensures that the score values are always scaled within the range [0,1] which is also wanted for the final player performance value. Therefore, a last computation takes place where all scores are added together and then divided by the total amount of scores that were used. The player performance, p , is calculated as follows:

$$p = \frac{t_s + c_s + l_s + s_s + e_s}{5}$$

The player performance variable lies between 0 and 1: 0 indicates a very poor performance and 1 an excellent player performance. It needs to be mentioned that in case a player is not able to complete a level, the next generated level will be based on the previous p value reduced by 0.33 with the minimum score being 0.1. This method gives the player the opportunity to play a significantly easier level where he can adapt to the game's needs.

As in the player performance function, the number of enemies and traps of each individual room are significant in calculating the overall fitness of the room. In addition to them, the total number of blocks is also a part of the computation because of the fact that, only these 3 elements are static features that are explored in the latent space through every iteration.

1. s_r = Total number of spikes in the room over the maximum possible amount
2. e_r = Total number of enemies in the room over the maximum possible amount
3. b_r = Total amount of solid blocks in the room over the maximum possible amount

Calculating the final fitness score, that will be used by the CMA-ES in finding the best matching latent space input of the trained generative model, is achieved by utilizing both the player performance value and the above static feature scores:

$$f = \left| p - \frac{s_r + e_r}{2} \right| + 0.75b_r$$

When it comes to generating a whole new level, specific steps and calculations take place before placing each room in any of the available room spaces inside of the 3x3 grid layout. A performance score at the higher end of the range [0,1] would mean that the player did very well while playing and that the level itself was relatively easy to complete (easy for the specific player since the same level structure may seem difficult to a new player). Based on this score, the trained GAN begins to create the first 3 rooms which are placed in the middle row of the level grid. These rooms will have the Easy (E) difficulty if the previous player performance was low, the Medium (M) difficulty if the player performance was in the middle of the range and the Hard (H) difficulty in case of a high player performance. After this process is completed, the GAN handles the bottom line by generating 3 rooms with an easier difficulty scale than the middle row since the player starts every level from there. At last, the top 3 rooms are produced and their difficulty is increased in comparison to the middle row. The above structure and all its room combinations together with the difficulty indicators can be shown in Figures 4.4.1 to 4.4.3.

It is important to mention that the generated rooms by the trained GAN are divided into 3 difficulty categories based on their fitness value that is calculated with the fitness method.

- $0 \leq f \leq 0.33$ Easy difficulty
- $0.33 < f \leq 0.66$ Medium difficulty
- $0.66 < f \leq 1$ Hard difficulty

Since both the calculated player performance and the fitness score of the rooms and the whole level in general are in the same range [0,1], a direct comparison can take place where a higher player performance results in a higher room difficulty. This approach will be further analysed and displayed with graphs in chapter 5.

	H	
	M	
	E	

Figure 4.4.1: A level structure where the middle row of the grid is filled with rooms that have the medium (M) difficulty

	H	
	H	
	M	

Figure 4.4.2: A level structure where the middle row of the grid is filled with rooms that have the hard (H) difficulty

	M	
	E	
	E	

Figure 4.4.3: A level structure where the middle row of the grid is filled with rooms that have the easy (E) difficulty

4.5 Evaluating the level as a whole

The evaluation process of the newly generated rooms is accomplished by extracting static characteristics like the total number of traps, enemies and the number of static blocks. These features are utilized by the fitness function that returns the fitness score for each room at the time it is generated. After every iteration, the CMA-ES is evaluating the outcome in order to maximize the provided fitness function and adjusts the input latent vector to the generator. The evaluation task is therefore closely linked to the functionality of the fitness function and the latent space exploration that allows the GAN to generate rooms that contain the desired attributes.

After the whole level is generated with all of its 9 rooms, the level evaluation function is called to output a score in the same range of $[0,1]$ that was used so far for all other scores. To accomplish this task, it was necessary to find out and store the minimum and maximum values of the total spikes, enemies and solid blocks that could appear in a level.

- $s_{max} = 80, s_{min} = 22$
- $e_{max} = 22, e_{min} = 3$
- $b_{max} = 360, b_{min} = 245$

These values were used to compute the individual scores for the whole level and normalise them in the range $[0,1]$ as follows:

- $s_l = \frac{s_{level} - s_{min}}{s_{max} - s_{min}}$
- $e_l = \frac{e_{level} - e_{min}}{e_{max} - e_{min}}$
- $b_l = \frac{b_{level} - b_{min}}{b_{max} - b_{min}}$

The level difficulty, d , is then calculated as the average of these 3 values:

$$d = \frac{s_l + e_l + b_l}{3}$$

Having the difficulty score of the generated level not only helps in categorizing it in the existing difficulties but it is also a very practical and direct approach of comparing it with the player performance so that the trained generator model and the CMA-ES are able to generate the optimized samples.

- $0 \leq d \leq 0.33$ Easy level difficulty
- $0.33 < d \leq 0.66$ Medium level difficulty
- $0.66 < d \leq 1$ Hard level difficulty

Chapter 5

Results

The results of this thesis are derived from having two different methods of testing the generated levels. The first includes a variety of player models which have a predetermined player performance with which consecutive levels are cleared. For the second method, *GANgeon Escape* was given to actual players who started the game from a similar difficulty and played 10 levels in a row. Their performance was measured and analysed in the graphs of Section 5.2 and then compared to the graphs of the prefixed player models.

5.1 Non-agent-based testing

For the non-agent based testing, 5 types of player models were used to generate a series of levels which they hypothetically played and completed based on a certain player performance. The player models that were used are:

1. New player
2. Normal player
3. Professional player
4. New to Professional player
5. Professional to New player

For the first 3 models, a manually assigned player performance was assigned and maintained throughout the experiment. Since it is measured in the range of $[0, 1]$, the “New Player” was given a player performance value of 0.1, the “Normal Player” 0.5 and the “Professional player” 0.9. This manually assigned values were all needed for the GAN and the CMA-ES to generate the levels since every other individual parameter, as described in Section 4.4, is included in the above value and no other computation method is required.

The results for the first 3 player models are presented in the Figures 5.1.1 to 5.1.3. It can be seen that the trained GAN model can generate levels with a difficulty score that is adjusted and wrapped around the player performance. Although all generated levels are designed for the exact same player and his play style, the difficulty is not always the same but it is locked between a range around the player performance. This is the result of having a fitness function

that is entirely based on the level representation where only static features like the spikes, enemies and solid blocks are taken into consideration.

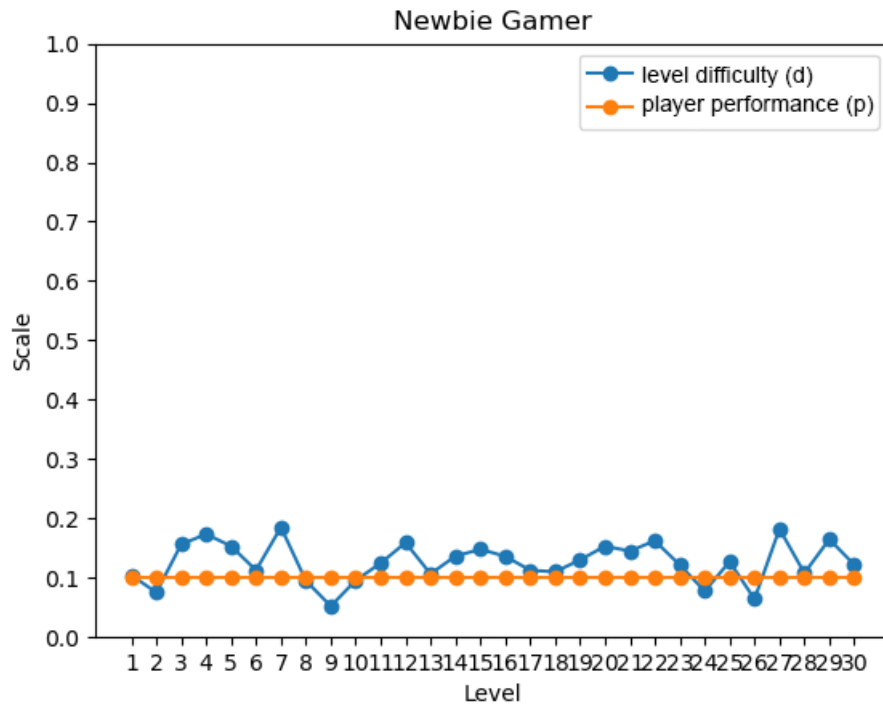


Figure 5.1.1: Optimized level difficulty for a new player. The player performance stays consistent with a value of 0.1 and the level difficulty is adjusted with values close to the player performance.

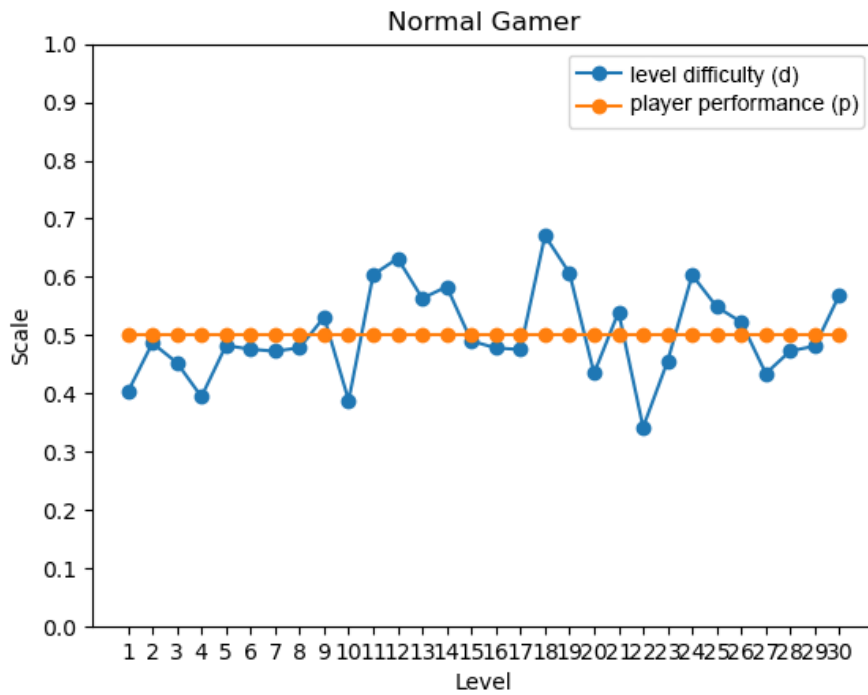


Figure 5.1.2: Optimized level difficulty for a normal player. The player performance stays consistent with a value of 0.5 and the level difficulty is adjusted with values close to the player performance.

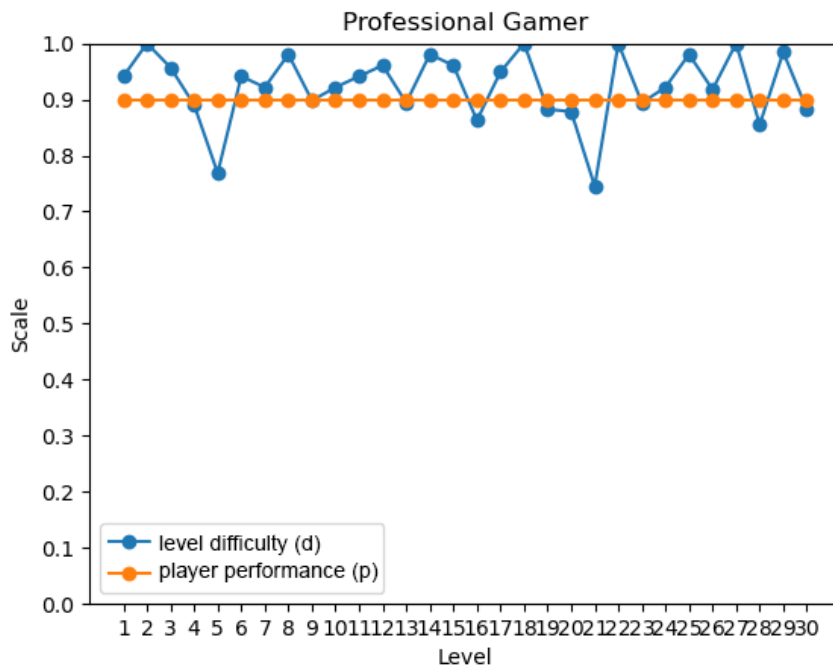


Figure 5.1.3: Optimized level difficulty for a professional player. The player performance stays consistent with a value of 0.9 and the level difficulty is adjusted with values close to the player performance.

For the other 2 player models where the player starts of as a professional turning into a new gamer and the opposite, the player performance is decreased and increased by 0.3 respectively after every level is hypothetically played. Figures 5.1.4 and 5.1.5 show how the level difficulty of each room is adjusted around the player's behavioural pattern. It needs to be mentioned that after each playthrough of 30 levels where the player agent is performing based on a manually assigned and modified performance score, all graph figures follow the same pattern as in the example figures provided but the difficulty of each level can have a score in the variation area around the performance value of the player agent.

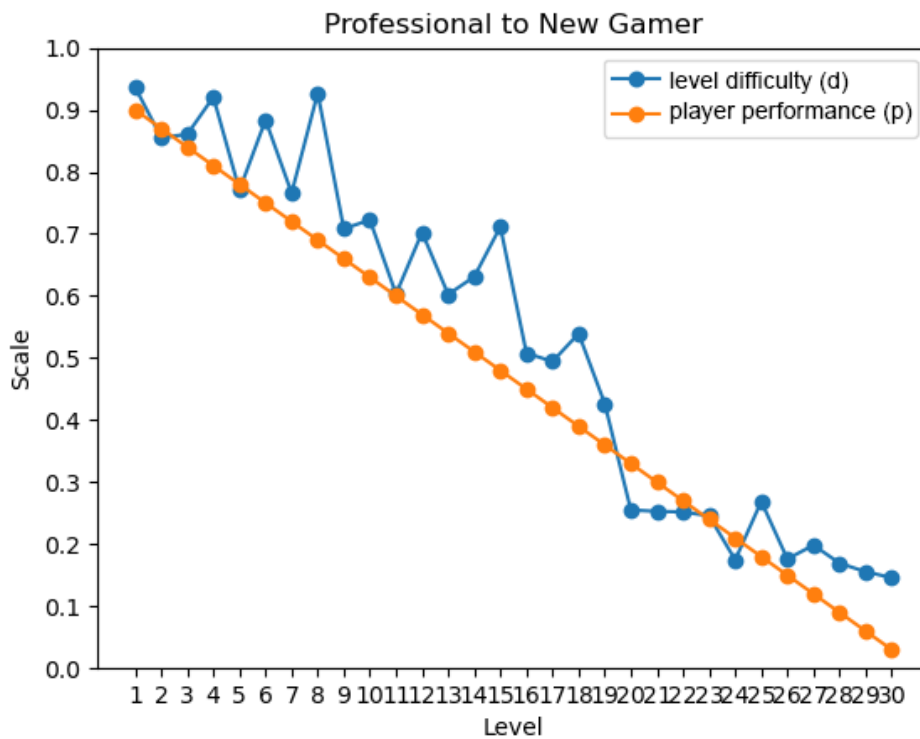


Figure 5.1.4: Optimized level difficulty for a player who starts as professional and turns into a new player. The player performance is decreased after each level by 0.03 and the level difficulty is adjusted with values close to the player performance.

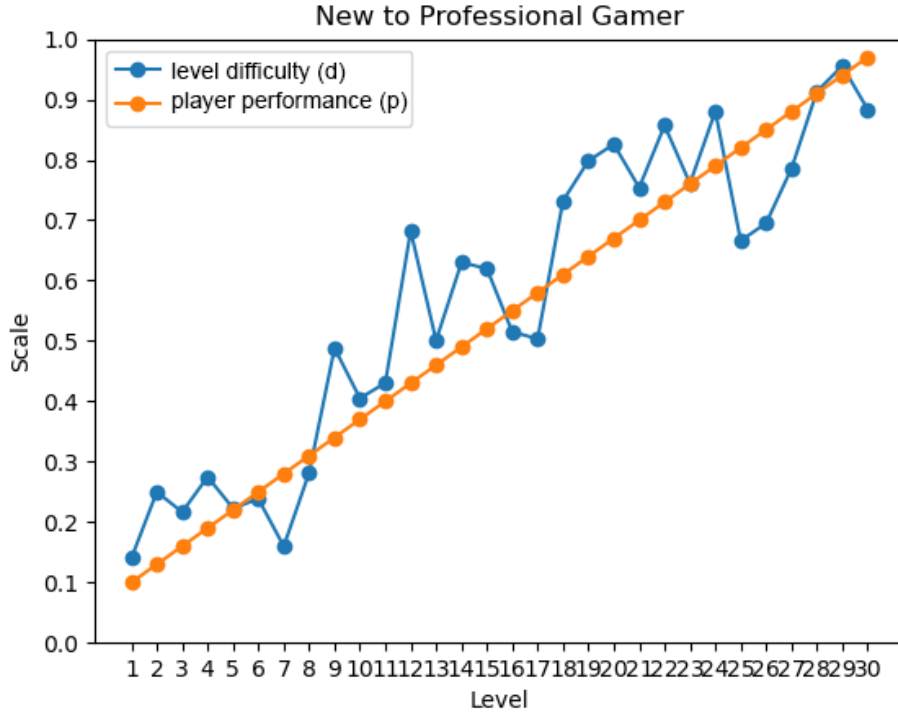


Figure 5.1.5: Optimized level difficulty for a player who starts as new player and turns into a professional. The player performance is increased after each level by 0.03 and the level difficulty is adjusted with values close to the player performance.

5.2 Player testing

Gathering information and statistics from real players who actually played *GANgeon Escape* was an important factor in shaping a better understanding in how the level generation is applied. A total of 10 persons participated in the experiment and played a series of 10 levels starting at almost the same level difficulty. Based on their performance in each level, the GAN generated the next one for them to play. During the process and after every completed level, specific gameplay data was saved for every player and used by the GAN and the CMA-ES to generate levels and for creating graphs and tables that allowed an easier analysis. Specifically, the data needed for the evaluation and the player performance calculation are:

- t = Level completion time in seconds
- c = Number of coins gathered

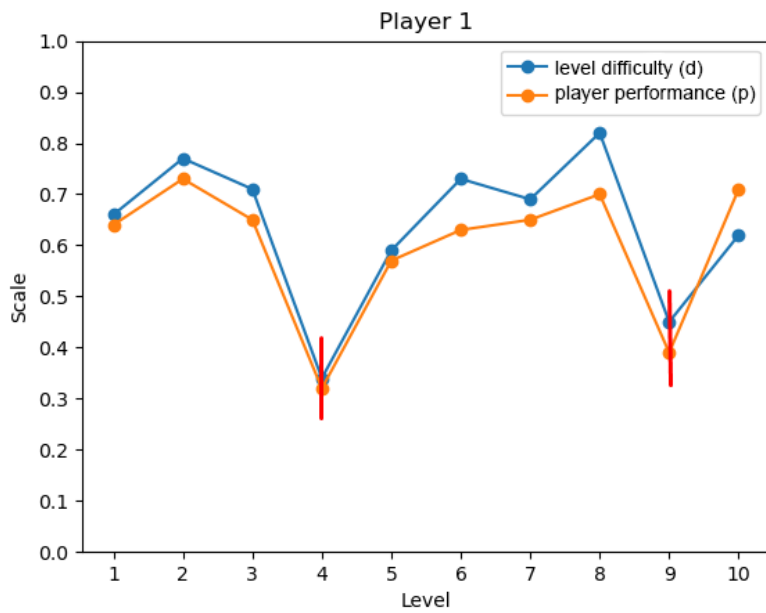
- l = Lives lost
- s = Number of spikes destroyed by the player
- e = Number of enemies destroyed by the player

All of the above variables have a performance score that is calculated when dividing the value by the maximum amount that the specific variable can reach for each level as described in Section 4.4. Then, the player performance p is calculated as the average of the above 5 values.

Overall, it is shown that the trained generative model was able to capture the players needs and generate a significant diversity in levels that also encouraged the players to try out different play styles. From the data analysis, the most noticeable play styles were the following:

- **Speedrunners:** Players who rushed to the portal just to complete the level without gathering all coins or defeating as many enemies and spikes possible.
- **Collectors:** Players who tried and eventually achieved in some levels to collect all coins even risking losing.
- **Hunters:** Players who hunted for all enemies and traps trying to defeat as many as possible. This group of players showed to have more loses than the other groups.

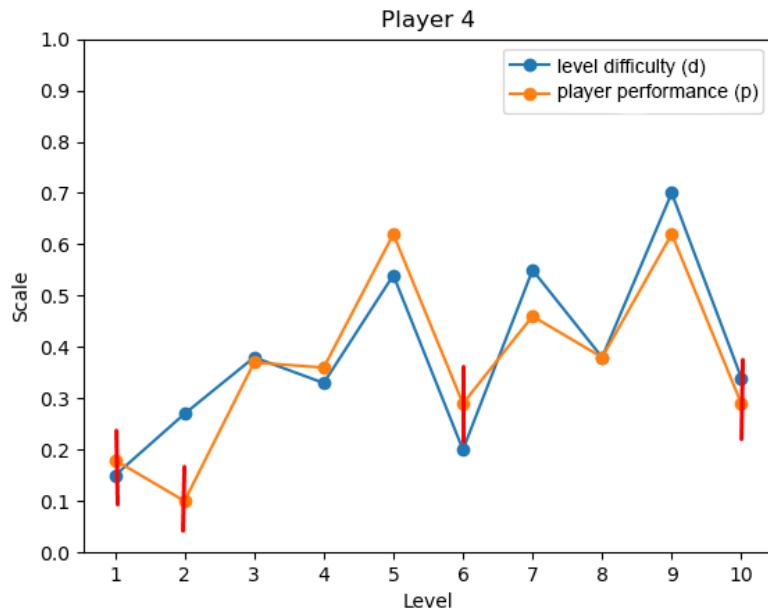
The following graphs show exactly how the level difficulty is adjusted around the player performance of each player for a total of 10 levels. The difference between this approach and having player models which have a predetermined player performance is that actual players may lose at some point and do not complete the level. In this case, a new level is generated reducing the previous player performance by 0.33 and a red line is drawn in the graphs indicating that a player failed to complete the corresponding level. A more in-depth analysis for some of the players' behaviour for the abovementioned levels can be seen in the corresponding tables where additional data is stored.



	Performance	Win/Lose	Level Evaluation
Level 1	0,64	W	0.2
Level 2	0.73	W	0,66
Level 3	0.65	W	0.77
Level 4	0.32	L	0.71
Level 5	0.57	W	0.34
Level 6	0.63	W	0.59
Level 7	0.65	W	0.73
Level 8	0.7	W	0.69
Level 9	0.37	L	0.82
Level 10	0.71	W	0.45
Average	0.597		0.62

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1	7	2	28	5	0	0	0	32
Level 2	14	12	58	18	3	3	0	59
Level 3	17	16	62	27	0	0	1	99
Level 4								
Level 5	6	4	38	18	0	0	2	73
Level 6	12	11	59	31	2	2	2	75
Level 7	20	20	44	26	1	1	2	80
Level 8	10	9	53	30	0	0	1	67
Level 9								
Level 10	7	6	44	24	1	1	1	61
Average	11.62	10	48.25	22.37	0.87	0.87	1.12	68.25

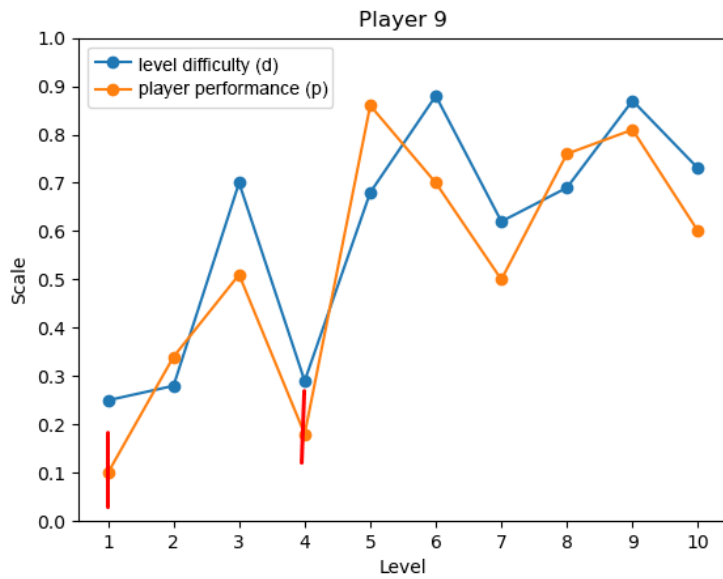
Figure 5.2.1: Data analysis for Player 1



	Performance	Win/Lose	Level Evaluation
Level 1	0.18	L	0.22
Level 2	0.1	L	0.15
Level 3	0.37	W	0.27
Level 4	0.36	W	0.38
Level 5	0.62	W	0.33
Level 6	0.29	L	0.54
Level 7	0.46	W	0.2
Level 8	0.38	W	0.55
Level 9	0.62	W	0.38
Level 10	0.29	L	0.7
Average	0.367		0.34

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1								
Level 2								
Level 3	9	2	27	1	4	2	1	71
Level 4	9	2	52	5	3	2	2	65
Level 5	7	4	39	18	1	1	1	72
Level 6								
Level 7	7	3	30	4	2	1	1	52
Level 8	11	5	57	11	8	4	2	72
Level 9	11	6	49	20	2	2	1	64
Level 10								
Average	9	3.66	42.33	9.83	3.33	2	1.33	66

Figure 5.2.2: Data analysis for Player 4



	Performance	Win/Lose	Level Evaluation
Level 1	0.1	L	0.27
Level 2	0.34	W	0.25
Level 3	0.51	W	0.28
Level 4	0.18	L	0.7
Level 5	0.86	W	0.29
Level 6	0.7	W	0.68
Level 7	0.5	W	0.88
Level 8	0.76	W	0.62
Level 9	0.81	W	0.69
Level 10	0.6	W	0.87
Average	0.536		0.73

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1								
Level 2	8	3	25	3	1	0	1	55
Level 3	6	0	37	6	3	3	1	33
Level 4								
Level 5	7	7	38	25	0	0	0	39
Level 6	10	9	53	30	2	2	1	67
Level 7	22	14	56	24	3	2	2	71
Level 8	15	11	60	34	2	2	0	57
Level 9	20	19	44	32	1	1	0	78
Level 10	22	17	76	42	3	3	2	81
Average	13.75	10	48.62	24.5	1.87	1.62	0.87	60.12

Figure 5.2.3: Data analysis for Player 9

By observing the graphs and the data analysis of each level, it can be concluded that most of the players managed to eventually increase the level difficulty and their abilities to play the game by mastering its mechanics. Even in some cases where players lost the level it was due to the fact that they tried to gather all available coins in the level and defeated as many enemies and spikes as possible. This approach proved to be riskier having a higher possibility of losing. Another group of players on the other hand, who tried to complete the level as fast as possible and also managed to keep most of their lives, scored a high player performance as well since these parameters are equally important as gathering all coins or defeating all enemies.

The most interesting observation is that all players who participated in the experiment scored an average player performance almost equal to the average level difficulty of all their generated levels (Figure 5.2.4). Since these two values are very close to each other, our trained GAN model proved to be capable of creating novel output levels for *GANgeon Escape* that are optimized and adjusted to each players' needs.

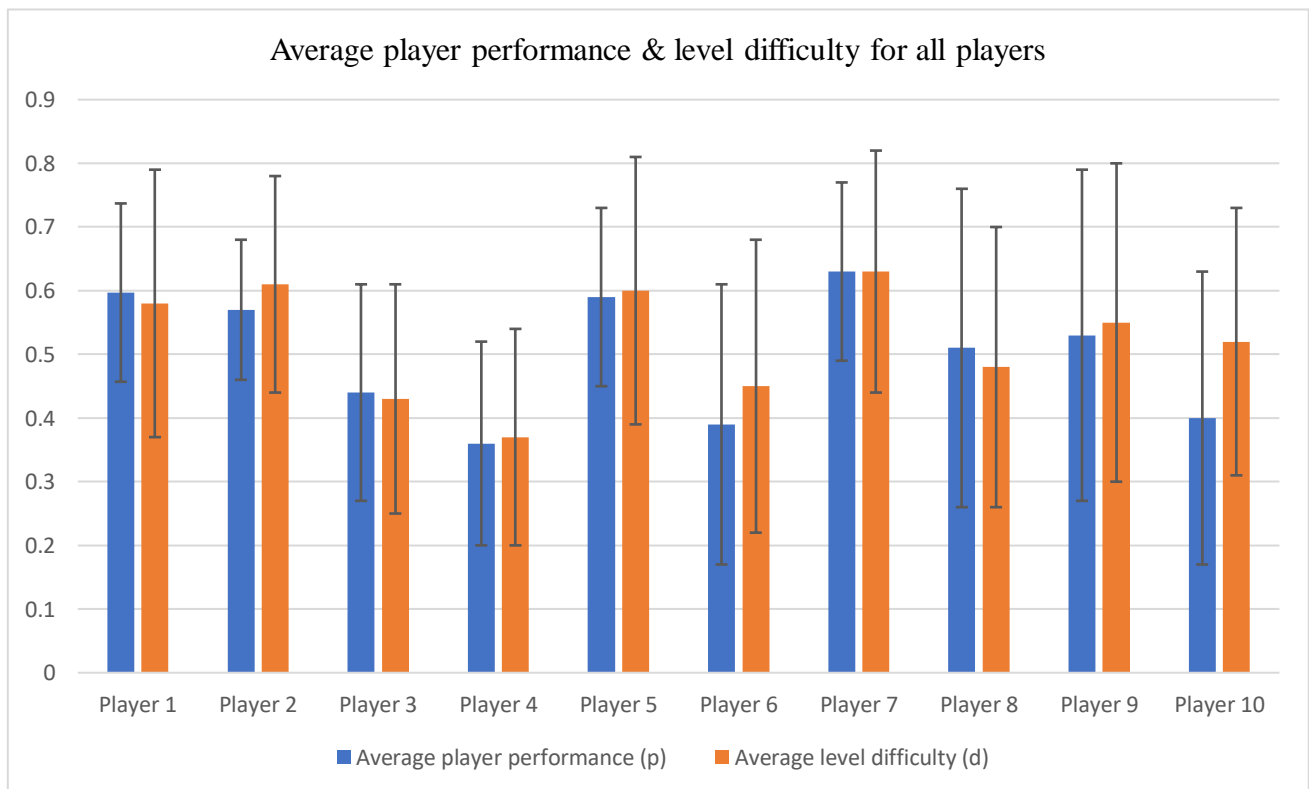


Figure 5.2.4: The average player performance score in comparison to the average level difficulty

Below, a collection of the most important player statistics is displayed in form of a bar chart (Figure 5.2.5). All values are derived by having the average amount of all defeated enemies, defeated spikes and gathered coins across the 10 levels. It can be seen that the majority of the players focused on defeating as many spikes as possible in order to increase their player performance and avoided going after enemies and coins since this action proved to be much more demanding.

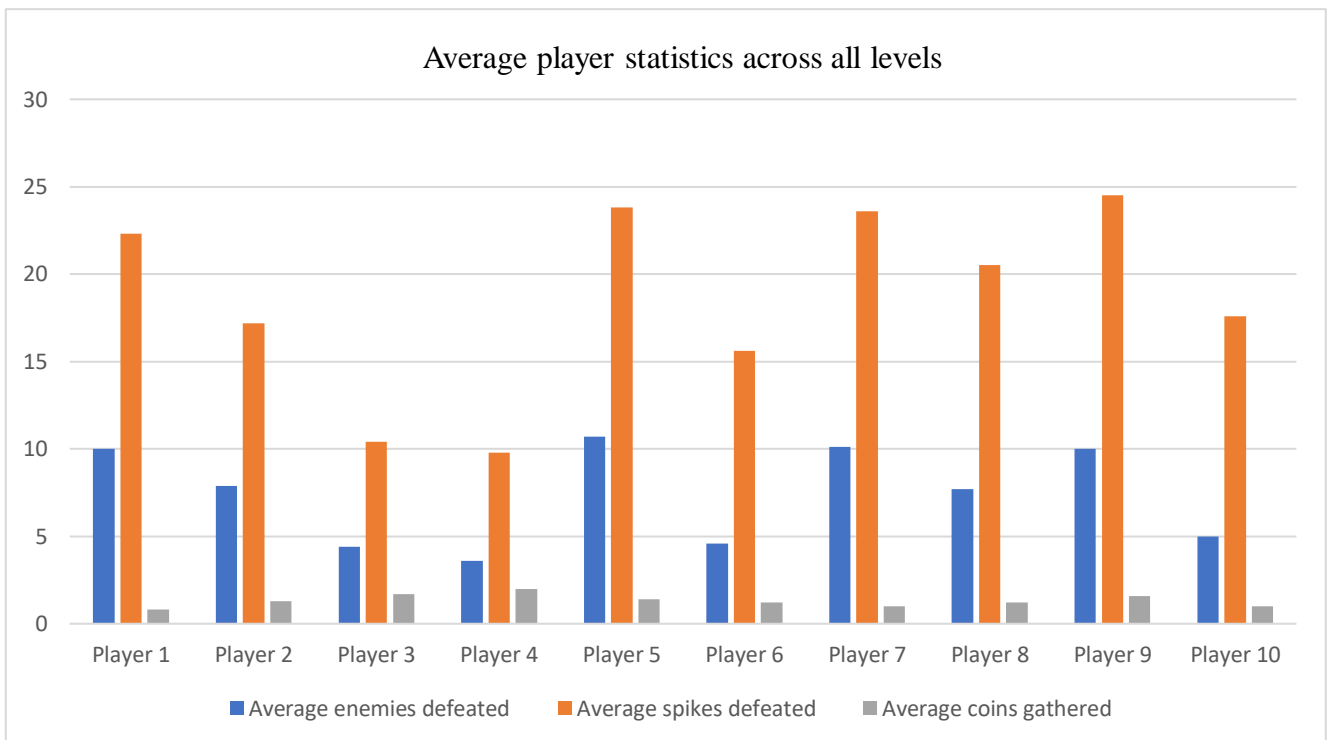


Figure 5.2.5: Some of the most important average player statistic values

Overall, it has been shown with both the non-agent-based testing and the player testing that the trained generative model is capable of creating a great variety of levels whose difficulty scale is calculated based on the player performance. Those levels can be easy to complete with just a few obstacles to overcome or challenging with many enemies, spikes and difficult paths. All of these attributes were explored by the evolutionary algorithm CMA-ES until a level with the desired difficulty score was generated by the GAN.

Chapter 6

Conclusion

6.1 Limitations

Although the implementation of a trained GAN model succeeded in capturing the fundamental structure of a *GANgeon Escape* level, like the approximate amount of enemies, coins and spikes per room and where to place them, some paths could be blocked preventing the player from proceeding. Rooms with such structural issues were fixed after the level generation in code by either adding or removing solid block tiles from specific coordinates in the level grid.

Since the game used for this thesis was not a pre-existing domain but rather a new creation for testing and demonstrating the capabilities of a trained GAN to generate game content, the dataset for the training process could be richer. Not more than 100 encoded room examples were used during training which prevented a bigger variety of level types and patterns. Due to this fact, the obstacles with which the player could interact during a gameplay were also limited but enough for achieving their purpose.

The gameplay aspect which includes the main character's movement mechanics, the enemies patrol system and the overall satisfaction while playing were also limited because of the fact that *GANgeon Escape* was not enough tested by groups of players. Playing the game was limited in just acquiring the experimental results discussed in Chapter 5 that helped in evaluating the models results. Lastly, the generative model used in this thesis was only utilized in creating levels for a specific kind of game in the same genre as *Spelunky* or *Mario* and hence more testing is required in applying the methods used in this thesis on non-grid-based games.

6.2 Future Work

In this section of the thesis, we present potential future improvements that could be implemented to improve the overall functionality and extensibility of the systems that were used.

- Although the custom dataset was proven to be sufficient in order to generate acceptable room samples within a desired difficulty scale, a larger and richer dataset would improve the outcome of the generative model. This would result in a bigger variety of levels providing an even better player experience.
- Having an agents-based testing approach instead of a representation-based testing could also contribute significantly in discovering levels through latent space exploration that are closer to the needs of the player. The fitness function used by the CMA-ES could then take the exact fitness score of the agent and find the most suitable level for the GAN to generate.
- As it was mentioned in the limitations section, the trained generative model used in this thesis was only tested in creating levels for *GANgeon Escape*. To discover the entire capabilities of this system further testing should take place by trying to generate game content for other games in the same genre or even completely unrelated games. The most interesting approach would be to train a GAN on an encoded dataset which derives from a non-grid-based game.

6.3 Summary

This thesis focused on creating new, playable and personalized game content in form of levels for the custom game *GANgeon Escape* that was developed for this system in Unity. A Generative Adversarial Network (GAN) was trained on a handcrafted dataset of possible rooms that when combined, constitute a complete level of the game. The trained generative model takes a latent vector which is not completely random noise but the result of a latent space exploration through the available features. The evolutionary algorithm that proved to be the most promising for this approach was the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) because of its capability of evolving vectors of real numbers. The CMA-ES can evaluate the GANs output by filtering specific static attributes of the level and compute a fitness

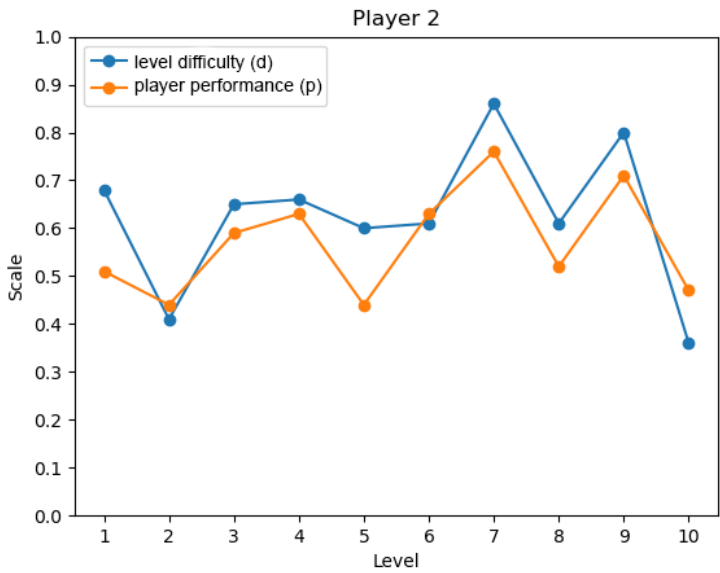
score that needs to be as close as possible to the desired result. Non-agent-based testing and player testing proved the functionalities and capabilities of the system in adjusting the game content accordingly for each player and his performance. In summary, it was shown that a GAN combined with an evolutionary algorithm can generate optimized output levels based on the players performance and wrap the whole gameplay experience around the players style and needs.

References

- [1] I. J. Goodfellow *et al.*, “Generative Adversarial Networks,” Jun. 2014. [Online]. Available: <http://arxiv.org/pdf/1406.2661v1>
- [2] G. N. Yannakakis and J. Togelius, *Artificial Intelligence and Games*, 1st ed. [s.l.]: Springer International Publishing AG, 2018. [Online]. Available: <http://www.springer.com/>
- [3] A. Summerville *et al.*, “Procedural Content Generation via Machine Learning (PCGML),” Feb. 2017. [Online]. Available: <https://arxiv.org/pdf/1702.00539>
- [4] P. Bontrager, A. Roy, J. Togelius, N. Memon, and A. Ross, “DeepMasterPrints: Generating MasterPrints for Dictionary Attacks via Latent Variable Evolution,” May. 2017. [Online]. Available: <http://arxiv.org/pdf/1705.07386v4>
- [5] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. M. Smith, and S. Risi, “Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network,” *CoRR*, abs/1805.00728, 2018.
- [6] M. Campbell, A. Hoane, and F. Hsu, “Deep Blue,” *Artificial Intelligence*, vol. 134, 1-2, pp. 57–83, 2002, doi: 10.1016/S0004-3702(01)00129-1.
- [7] J. Schrum, J. Gutierrez, V. Volz, J. Liu, S. Lucas, and S. Risi, “Interactive Evolution and Exploration Within Latent Level-Design Space of Generative Adversarial Networks,” Apr. 2020. [Online]. Available: <https://dev.arxiv.org/pdf/2004.00151>
- [8] E. Giacomello, P. L. Lanzi, and D. Loiacono, *Searching the Latent Space of a Generative Adversarial Network to Generate DOOM Levels*, 2019.
- [9] N. Hansen, S. Müller, and P. Koumoutsakos, “Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES),” *Evolutionary computation*, vol. 11, pp. 1–18, 2003, doi: 10.1162/106365603321828970.
- [10] N. Hansen and A. Ostermeier, “Completely derandomized self-adaptation in evolution strategies,” *Evolutionary computation*, vol. 9, no. 2, pp. 159–195, 2001, doi: 10.1162/106365601750190398.
- [11] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, “Search-Based Procedural Content Generation: A Taxonomy and Survey,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, pp. 172–186, 2011, doi: 10.1109/TCIAIG.2011.2148116.
- [12] N. Shaker, G. Yannakakis, and J. Togelius, “Towards Automatic Personalized Content Generation for Platform Games,” *Proceedings of the 6th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010*, 2010.
- [13] J. Dormans and S. Bakkes, “Generating Missions and Spaces for Adaptable Play Experiences,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 216–228, 2011, doi: 10.1109/TCIAIG.2011.2149523.
- [14] D. Loiacono and L. Arnaboldi, “Multiobjective Evolutionary Map Design for Cube 2: Sauerbraten,” *IEEE Transactions on Games*, vol. 11, no. 1, pp. 36–47, 2019, doi: 10.1109/TG.2018.2830746.
- [15] M. Mendler and David Adams, “Automatic Generation of Dungeons for Computer Games,” in 2002.
- [16] N. Brewer, “Computerized Dungeons and Randomly Generated Worlds: From Rogue to Minecraft [Scanning Our Past],” *Proceedings of the IEEE*, vol. 105, no. 5, pp. 970–977, 2017, doi: 10.1109/JPROC.2017.2684358.
- [17] R. Linden, R. Lopes, and R. Bidarra, “Procedural Generation of Dungeons,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 6, pp. 78–89, 2014, doi: 10.1109/TCIAIG.2013.2290371.

- [18] A. Summerville and M. Mateas, "Sampling Hyrule: Multi-Technique Probabilistic Level Generation for Action Role Playing Games," in *AIIDE 2015*, 2015.

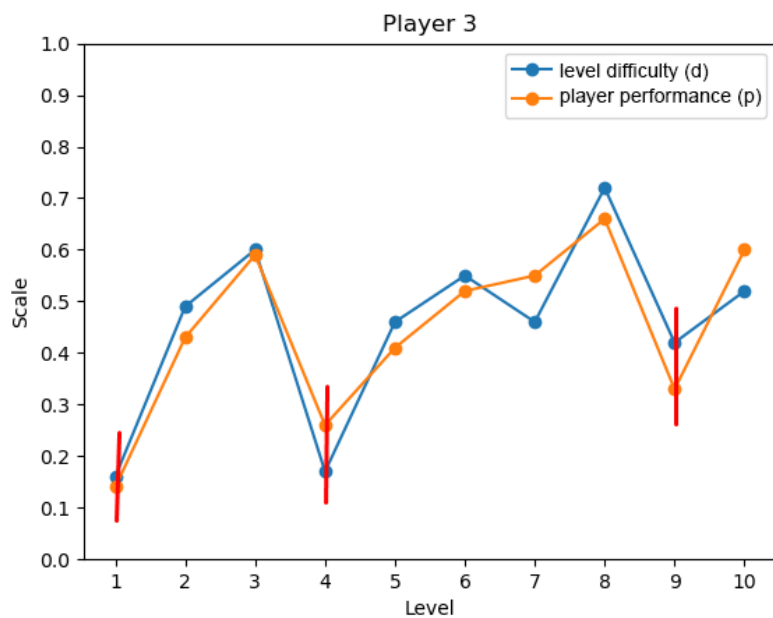
Appendix A – Additional player analysis data



	Performance	Win/Lose	Level Evaluation
Level 1	0.51	W	0.26
Level 2	0.44	W	0.68
Level 3	0.59	W	0.41
Level 4	0.63	W	0.65
Level 5	0.44	W	0.66
Level 6	0.63	W	0.6
Level 7	0.76	W	0.61
Level 8	0.52	W	0.86
Level 9	0.71	W	0.61
Level 10	0.47	W	0.8
Average	0.57		0.61

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1	6	0	37	6	3	3	1	33
Level 2	13	5	53	7	1	0	0	36
Level 3	8	8	49	14	1	0	0	42
Level 4	12	9	57	15	2	1	0	43
Level 5	9	3	64	10	3	1	1	33
Level 6	13	9	54	14	3	3	1	54
Level 7	15	13	54	22	2	2	0	58
Level 8	15	11	61	30	2	1	2	56
Level 9	10	8	50	31	1	1	1	62
Level 10	20	13	53	23	2	1	2	71
Average	12.1	7.9	53.2	17.2	2	1.3	0.8	48.8

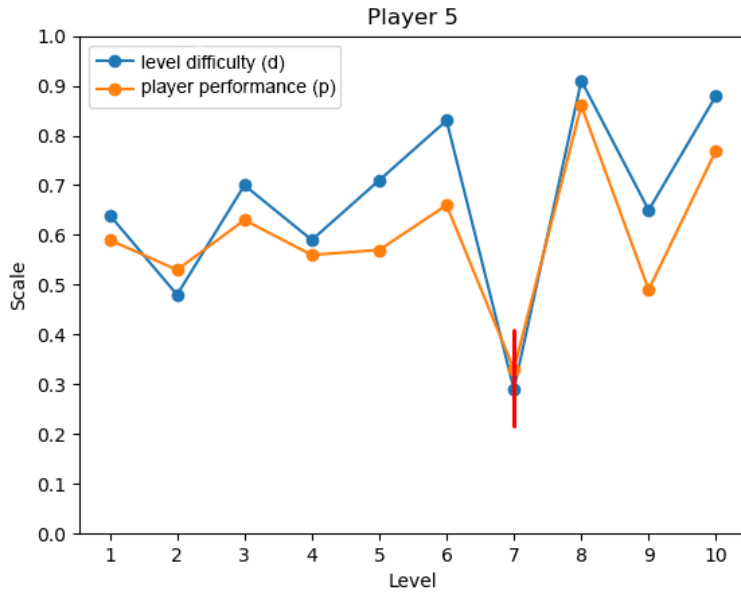
Figure 1: Data analysis for Player 2



	Performance	Win/Lose	Level Evaluation
Level 1	0.1	L	0.27
Level 2	0.43	W	0.16
Level 3	0.59	W	0.49
Level 4	0.26	L	0.6
Level 5	0.41	W	0.17
Level 6	0.52	W	0.46
Level 7	0.55	W	0.55
Level 8	0.66	W	0.46
Level 9	0.33	L	0.72
Level 10	0.6	W	0.42
Average	0.44		0.43

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1								
Level 2	5	2	29	2	5	3	1	70
Level 3	9	6	51	10	2	2	1	71
Level 4								
Level 5	8	2	30	6	1	0	0	46
Level 6	10	5	40	9	4	3	1	64
Level 7	11	5	55	11	1	1	1	66
Level 8	7	3	42	15	0	0	0	60
Level 9								
Level 10	12	8	44	20	4	3	1	62
Average	8.8	4.4	41.5	10.4	2.4	1.7	0.7	62.7

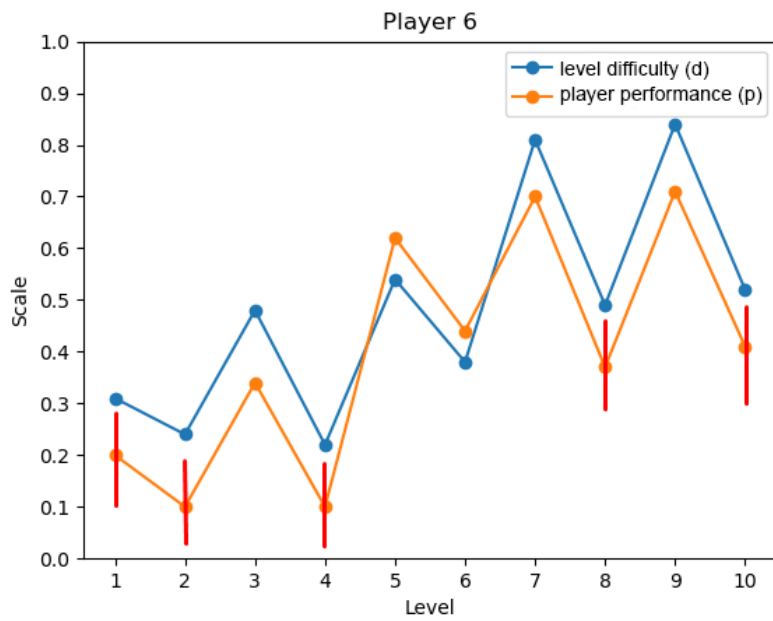
Figure 2: Data analysis for Player 3



	Performance	Win/Lose	Level Evaluation
Level 1	0.59	W	0.23
Level 2	0.53	W	0.64
Level 3	0.63	W	0.48
Level 4	0.56	W	0.7
Level 5	0.57	W	0.59
Level 6	0.66	W	0.71
Level 7	0.33	L	0.83
Level 8	0.86	W	0.29
Level 9	0.49	W	0.91
Level 10	0.77	W	0.65
Average	0.59		0.6

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1	8	4	29	6	0	0	1	48
Level 2	14	12	60	26	2	1	2	56
Level 3	8	5	41	15	1	1	0	62
Level 4	20	14	50	21	2	2	2	81
Level 5	16	13	52	27	3	2	2	59
Level 6	19	13	58	31	1	1	1	73
Level 7								
Level 8	7	7	38	25	1	1	0	39
Level 9	21	16	71	36	4	2	2	80
Level 10	17	13	58	28	3	3	0	48
Average	14.4	10.7	50.7	23.8	1.8	1.4	1.1	60.6

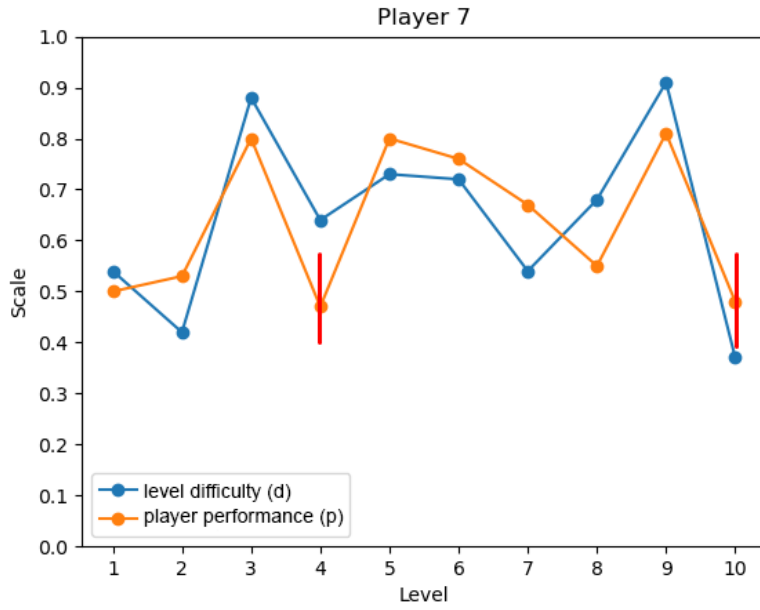
Figure 3: Data analysis for Player 5



	Performance	Win/Lose	Level Evaluation
Level 1	0.2	L	0.19
Level 2	0.1	L	0.31
Level 3	0.34	W	0.24
Level 4	0.1	L	0.48
Level 5	0.62	W	0.22
Level 6	0.44	W	0.54
Level 7	0.7	W	0.38
Level 8	0.37	L	0.81
Level 9	0.71	W	0.49
Level 10	0.41	L	0.84
Average	0.39		0.45

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1								
Level 2								
Level 3	8	3	25	3	1	0	1	55
Level 4								
Level 5	7	4	37	10	1	1	1	51
Level 6	12	6	56	19	2	1	2	59
Level 7	11	5	47	25	1	1	0	58
Level 8								
Level 9	6	5	39	21	3	3	1	68
Level 10								
Average	8.8	4.6	40.8	15.6	1.6	1.2	1	58.2

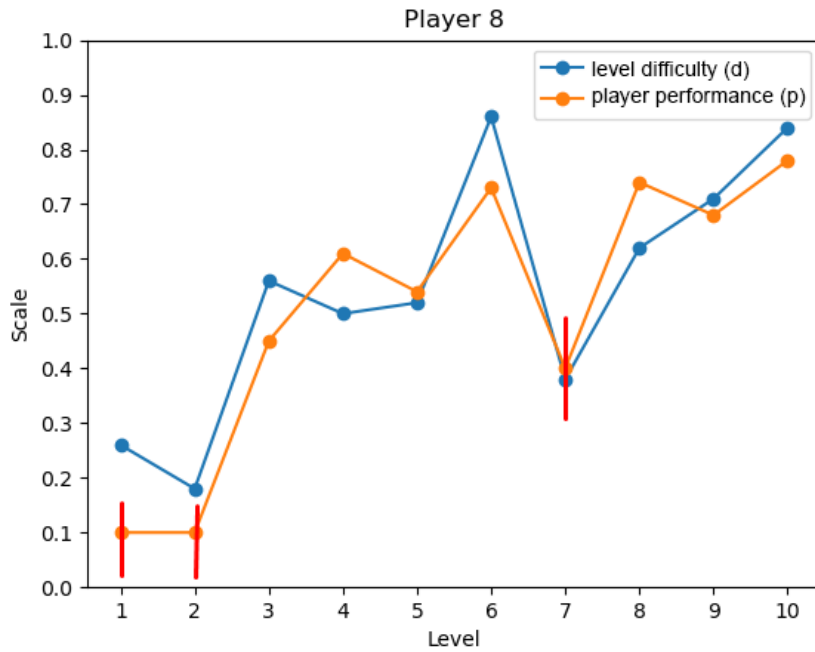
Figure 4: Data analysis for Player 6



	Performance	Win/Lose	Level Evaluation
Level 1	0.5	W	0.26
Level 2	0.53	W	0.54
Level 3	0.8	W	0.42
Level 4	0.47	L	0.88
Level 5	0.8	W	0.64
Level 6	0.76	W	0.73
Level 7	0.67	W	0.72
Level 8	0.55	W	0.54
Level 9	0.81	W	0.68
Level 10	0.48	L	0.91
Average	0.63		0.63

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1	7	3	30	7	2	1	1	39
Level 2	11	6	58	17	3	2	1	62
Level 3	9	8	50	23	0	0	0	42
Level 4								
Level 5	15	13	55	30	2	2	0	46
Level 6	17	13	65	38	1	1	0	69
Level 7	19	14	58	31	0	0	1	74
Level 8	11	5	55	11	1	1	1	66
Level 9	20	19	44	32	1	1	0	78
Level 10								
Average	13.6	10.1	51.8	23.6	1.2	1	0.5	59.5

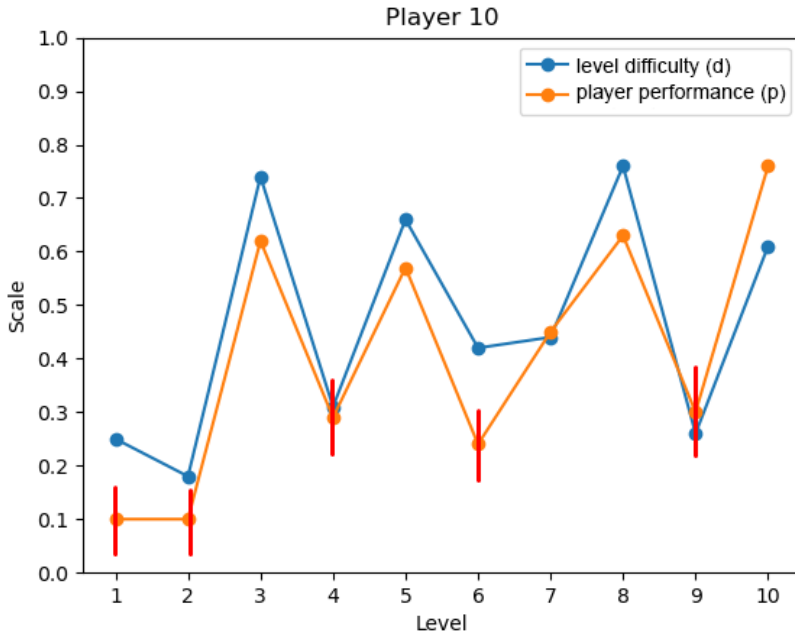
Figure 5: Data analysis for Player 7



	Performance	Win/Lose	Level Evaluation
Level 1	0.1	L	0.22
Level 2	0.1	L	0.26
Level 3	0.45	W	0.18
Level 4	0.61	W	0.56
Level 5	0.54	W	0.5
Level 6	0.73	W	0.52
Level 7	0.4	L	0.86
Level 8	0.74	W	0.38
Level 9	0.68	W	0.62
Level 10	0.78	W	0.71
Average	0.51		0.48

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1								
Level 2								
Level 3	6	1	29	5	1	1	2	59
Level 4	9	5	40	13	0	0	1	54
Level 5	11	6	60	22	5	3	1	58
Level 6	10	7	54	23	2	2	0	58
Level 7								
Level 8	7	7	37	19	0	0	1	59
Level 9	15	12	60	26	3	2	0	61
Level 10	20	16	56	36	1	1	0	64
Average	11.1	7.7	48	20.5	1.7	1.2	0.7	59

Figure 6: Data analysis for Player 8



	Performance	Win/Lose	Level Evaluation
Level 1	0.1	L	0.26
Level 2	0.1	L	0.25
Level 3	0.62	W	0.18
Level 4	0.29	L	0.74
Level 5	0.57	W	0.31
Level 6	0.24	L	0.66
Level 7	0.45	W	0.42
Level 8	0.63	W	0.44
Level 9	0.3	L	0.76
Level 10	0.76	W	0.26
Average	0.4		0.42

	Enemies in the level	Enemies defeated	Spikes in the level	Spikes defeated	Coins in the level	Coins gathered	Lives Lost	Completion time (s)
Level 1								
Level 2								
Level 3	7	4	37	10	0	0	1	52
Level 4								
Level 5	6	4	38	17	1	1	2	72
Level 6								
Level 7	10	6	52	19	2	1	2	67
Level 8	8	4	44	23	1	1	1	65
Level 9								
Level 10	9	7	31	19	2	2	0	71
Average	8	5	40.4	17.6	1.2	1	1.2	65.4

Figure 7: Data analysis for Player 10