Emergent Personalized Content in Video Games

Simone Guggiari
14-932-867

Master's Thesis
April 2019

Dr. Fabio Zünd, Henry Raymond
Prof. Dr. Robert W. Sumner

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Game Technology Center

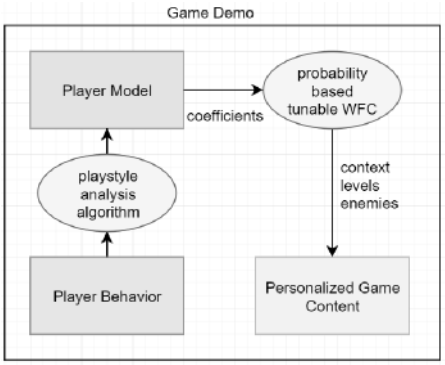
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Game Technology Center

Master Thesis

Emergent Personalized Content in Video Games

Simone Guggiari



Introduction

A huge amount of resources is invested in games regarding content creation. However many games fail to keep the user engaged as the content and challenges proposed do not always match the player skills and personality. Traditionally, games only offer the choice of difficulty which simply isn't expressive enough. The few games that have tried to dynamically adapt the game pace and content to the user have been critically acclaimed and were very well received. However, no general framework for content that is dynamically adapted to the user has ever been developed. This thesis aims at doing just that.

Task Description

The following tasks constitute the project:

- First, a model of the user is developed that can classify the players by analyzing their playstyle in the first levels, and therefore infer which challenges, pacing and layout would be optimal for the user. The model will be based on Bartle's player taxonomy and should be general enough to be applicable to videogames independently of genre.
- Second, an algorithm is developed that uses the player model to create different types of procedural content that is tailored and personalized for the specific user. The QM inspired Wave Function Collapse algorithm is tuned to search a particular probability space determined by coefficients provided by the player model.
- Third, a simple video game is developed that demonstrates the capabilities of the algorithm. The game prototype will be developed alongside the other algorithms to test applicability in an iterative fashion. The content within the game (context, level, enemy behavior) is generated procedurally by the engine and is optimized based on parameters inferred about the user.
- Fourth, a user study is conducted to test how the game adapts to different users and playstyles.

Abstract

Games are interactive experiences by their very nature. This interactivity allows different players to tackle the proposed challenges in their own way. Most games tend, however, to provide static content that all users must experience to progress, without taking into considerations underlying preferences and playing patterns that might be unique to each player.

In this thesis, we explore a general framework that can be used to adapt the game content to the user dynamically. We present an approach that analyzes in-game player behavior and uses this data, combined with established psychological models, to generate procedural, personalized levels, to provide a more engaging and enjoyable experience.

Three parts comprise this thesis: the development of the procedural algorithm that adapts to different players, the implementation of a simple videogame to test our results, and a study to collect data and validate the system. The study shows that catering to different users and adapting the game increases the average level of enjoyment.

Acknowledgements

I want to sincerely thank my supervisors, *Dr. Fabio Zünd* and *Henry Raymond* of the Game Technology Center (GTC) of ETH Zürich for all the time, effort, suggestions, and support they gave me during this thesis. They both followed my progress closely and were invaluable for the completion of this thesis. I couldn't have made it without them.

Henry Raymond was always ready to help out with problems I encountered, both during the development and the writing, and to provide insightful feedback that steered me in the right direction.

Dr. Fabio Zünd was also very supportive and ready to suggest new methods and techniques to try out. From our early brainstorming meetings to the last data analysis ones, he always had effective suggestions on how to tackle every problem.

I would also like to thank *Prof. Dr. Robert W. Sumner* of ETH Zürich for giving me the opportunity to follow this exciting thesis at the Game Technology Center.

I also want to deeply thank all the members of the GTC, including *Dr. Stéphane Magnenat*, *Julia Chatain*, *Violaine Fayolle*, and *Fábio Porfírio* for their feedback and support during the development, and for trying out all the iterations of the game.

Finally, I would like to thank my family, my mother *Michela* for proofreading the thesis and encouragement, my father *Luca* for the support, and my sister *Elissa* for helping spread the word of our user study. I would also like to thank *Maria Eduarda* for her continued support and encouragement.

I would also like to thank every participant that dedicated their time and took part in our study.

Last but not least, I would like to thank you, the reader, for taking the time to read this thesis. I hope the time spent will be fruitful and that you will learn a few things along the way.

Thank you all.

Contents

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Vision . . . 2
- 1.2 Problem Statement . . . 2
- 1.3 Contribution . . . 3
- 1.4 Outline . . . 4

2 Related Work 7
- 2.1 Adaptive Games . . . 7
  - 2.1.1 Research . . . 8
  - 2.1.2 Applications . . . 9
- 2.2 Psychological Models . . . 11
  - 2.2.1 Models Overview . . . 11
  - 2.2.2 Unified Model . . . 13
- 2.3 Wave Function Collapse . . . 14

3 Procedural Level Generation 17
- 3.1 Theory . . . 18
  - 3.1.1 Algorithm Description . . . 18
  - 3.1.2 2D Concepts . . . 19
  - 3.1.3 3D Concepts . . . 20
  - 3.1.4 Algorithm Overview . . . 21
  - 3.1.5 Modules Generation . . . 22
  - 3.1.6 Additional Constraints . . . 25

3.2 Implementation . . . 28
3.2.1 Pseudocode . . . 29
3.2.2 Analysis of Levels . . . 30
3.3 Results . . . 32
3.3.1 Time Analysis . . . 32
3.3.2 Performance . . . 32
3.3.3 Sample Levels . . . 33
3.4 Adaptation . . . 33
3.4.1 Player Encoding . . . 34

4 Game Demo **37**
4.1 Design . . . 37
4.1.1 Constraints . . . 38
4.1.2 Game Mechanics . . . 38
4.2 Implementation . . . 38
4.2.1 General Framework . . . 38
4.2.2 Online Build . . . 41
4.3 Results . . . 42
4.3.1 Final Game . . . 42

5 User Study **45**
5.1 Design . . . 46
5.1.1 Data Collected . . . 46
5.2 Execution . . . 47
5.2.1 Technical Details . . . 48
5.3 Results . . . 49
5.3.1 Overview of Statistical Methods . . . 49
5.3.2 First Study . . . 51
5.3.3 Second Study . . . 53
5.3.4 Final Study . . . 53
5.3.5 Rating Analysis . . . 54

6 Conclusion **61**
6.1 Reflection . . . 61
6.2 Future Work . . . 62
6.3 Outlook . . . 62

Bibliography **63**

List of Figures

1.1 Method Diagram 4

2.1 Infinite Mario 8
2.2 Stress Modulation Coefficients 10
2.3 Survivor Intensity Plot 10
2.4 Bartle's Taxonomy 12
2.5 Maslow's Hierarchy 13
2.6 WFC Bitmaps 14
2.7 3D WFC Applications 15

3.1 WFC Patches 19
3.2 Dihedral Group 20
3.3 Symmetry Groups 20
3.4 Embedded Mesh 21
3.5 Octahedral Group 22
3.6 Generation Scripts 23
3.7 Generation Pipeline 23
3.8 Sample Tilesets 24
3.9 Sample Outputs 24
3.10 Rotation Problems 25
3.11 Hash Computation 26
3.12 Level Skeleton 27
3.13 Normals Problems 29
3.14 Neighborhood Tiles 31
3.15 Sample Levels 33

4.1 Game Architecture 39
4.2 Early Prototype 40

4.3 Texture Gradient 42
4.4 Game Props 42
4.5 Game Tiles 43
4.6 Final Game Screenshot 44

5.1 Star Rating Interface 48
5.2 First Study Charts 52
5.3 Early Cross Correlation 53
5.4 User-Type Pie Chart 54
5.5 User-Type Distribution 54
5.6 Rating Histogram 55
5.7 ANOVA Results 57
5.8 Gameplay Cross Correlation 58
5.9 User Cross Correlation 59

List of Tables

3.1 Execution Time 32

5.1 First Study Correlation 52
5.2 Normal Distribution Fit 55
5.3 ANOVA Table Results 56

1

Introduction

We live among data. Everywhere we go, every service we use, data is collected about us. Have you searched for a particular item? Here are ten locations where you can buy it. Liked a movie? Here are other films that users with similar tastes liked. Also, how about trying that take-away place that recently opened on your way from work to home, at 19:14 when you usually leave? Open your favorite streaming service, and be overwhelmed by a selection of cinematography tailored just for you. Alternatively, visit your favorite online shopping service to discover which items the service thinks you might like best and are most likely to buy.

Data is collected, processed and used at an ever-increasingly fast rate.

Most of it is used to understand our behavior patterns and to either create value for the platform collecting it or to directly sell us more.

Data collection also happens in gaming. Recommender systems are used to suggest games we are more likely to buy based on previous purchasing decisions. Companies secretly guard statistics over customers' purchasing patterns and behavior metrics, such as time spent on a particular title and login times. Most of the data is used for monetization, to understand which items are more likely to be bought to maximize the total turnout. Data is also used by companies to gain insights about their user base, understand their customers' needs and therefore be able to produce other, even more appealing products.

Less frequently, data is used to the players' advantage, to directly improve their experience. Based on the commercial success of titles doing precisely that, we propose that focusing on the refinement of the player's unique experience by using their data is more effective, both from the user's perspective as well as the company's.

1.1 Vision

By analyzing how users respond to specific changes in the game, we could focus on adapting and customizing the game. Moreover, instead of blindly using massives amount of data, we could find an underlying model that represents most of our user base accurately. We could leverage such information, together with selected metrics, to indeed tune the whole game experience.

Starting a new game shows an example of a typical encountered problem. Developers do not know how skilled you are, so there is no way for them to tune the game to provide the best possible experience for you. Sometimes, the games provide a difficulty screen, but due to its coarseness, this doesn't offer nearly enough flexibility. You are forced to estimate which among the provided settings could yield the most enjoyable experience, without knowing much about the game. If the difficulty is set too low, you'll find the game boring and put it on the side in favor of a more challenging one. If the difficulty is too high, you'll quickly grow frustrated with the game and most likely give up.

Furthermore, there might be some aspects of the game you enjoy, whereas others feel like a chore. For example, you might love exploring new worlds and maps to discover the story, but you might find the necessary killing of enemies to accumulate skill points and advance to the next level tedious. Alternatively, you might love the visceral gameplay provided by frenetic action in a racing game but would like to avoid all the additional car tuning between missions. Not to mention a medieval *RTS* game in which you love maneuvering your troops across snowy mountains with narrow passages, but dread the flat desert levels that have none of those characteristics. Wouldn't it be better to have more of the type of levels you enjoy?

Imagine a game that could understand your tastes, and could therefore provide more content you like. While playing your favorite *RPG*, the game could detect that you enjoy the quests provided by a specific character, and generate more in-depth ones. The game could understand you enjoy playing as a mage class, and adapt the levels and challenges to better suit a druid rather than a fighter.

What about an *FPS* that understands your preferred playstyle and generated levels and mission in line with your likings? Imagine going through two long, grueling levels in which you're forced to shoot at everything that moves while progressing through a narrow building, but were then offered a stealth mission on a much more open map. The game could understand you enjoyed the latter more, and modify the subsequent levels, making them more open and allowing you to take a sneakier approach.

Such adaptations would lead to a unique experience tailored just for you, that nobody else could encounter. Games would feel much more personal and immersive. You would influence the game content presented to you by how you approach every single level.

1.2 Problem Statement

Since game adaptation is such a broad concept, we focus on a particular aspect of personalized content: *levels*. Together with game mechanics, levels are one of the videogame aspects that have the largest influence on how a game is played. Even with the same mechanics, different

level designs and layouts can completely change a game. We can turn a game focused on close-quarter combat and short playtimes to one focused on long-range engagements or one focused on stealth. By changing the placement of resources, open fields, obstacles, and narrow passages, as well as enemies, it is possible to change a game entirely and modulate how it is played.

Usually, game development is an iterative process, in which designers start working on an idea, build a prototype, and get feedback for it. They then steer in the direction suggested by the results of the feedback, producing a new iteration. They then test once more the resulting game, always collecting feedback and making the appropriate adjustments suggested by the test group. This process helps to understand what works well and what does not. Testers have thus a direct effect on the final game, based on their tastes and feedback. However, this introduces bias, depending on the sample of testers. Ideally, the game should be flexible enough to take into account possible variations across players and cater to each of them by either changing the content, the pacing, or the difficulty. There is no universally accepted model of player behavior, but it is easy to see how the whole development process could be improved if there were one. A unified model could help designers to take into consideration the different demographics of players. The game could be made variable across different dimensions and adaptable to the users directly while playing. The roles would reverse: the players would not be selecting the games anymore, but rather the games would adapt to the players.

Problem Definition The goal of this thesis is to:

DEVELOP AN ALGORITHM EMBEDDED WITHIN A VIDEOGAME THAT ADAPTS TO THE USERS BY COLLECTING INFORMATION ABOUT THEIR PLAYSTYLE AND USES THIS INFORMATION TO GENERATE PROCEDURAL, PERSONALIZED LEVELS.

We further want to show that this method increases the enjoyment of players. We show this fact by performing a user study. With the data collected from the user study, we provide additional insights about player behavior, design choices, possible causes and future work in the field.

We also research what the most descriptive models that allow us to understand the different audiences of our game are. We try to categorize game mechanics and game genres across these models, providing a general framework that can be used to better understand which elements should be present in a game, and how to balance and tune them to cater to different players. We show that many popular games fit closely into these models and that they include mechanics for each of the different playstyles. These mechanics allow the gameplay to take different directions for different players. We theorize that this could be one of the reasons behind their swift growth in audience.

1.3 Contribution

The contributions of this thesis are the following:

1. We specify a method to collect data about the players and categorize them.

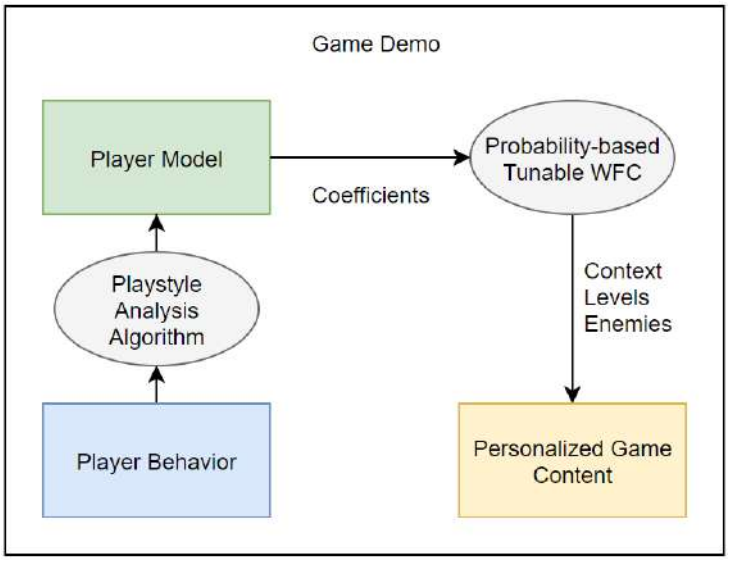


Figure 1.1: *The diagram showcasing the different steps and components of this thesis.*

2. We provide a general method to map user types to parameters required to generate personalized content.
3. We provide details on how to implement a general purpose, 3D level generation algorithm with these parameters.
4. We show that catering to different users with our particular method provides higher enjoyment on average.
5. We provide suggestions on how to incorporate these insights into future work and games.

1.4 Outline

The diagram in Figure 1.1 explains the overall method:

The succession of steps in our system is the following:

1. When playing a particular game, users exhibit *player behavior* depending on both the game and their playstyle and personality.
2. We define a *Playstyle Analysis Algorithm*, that observes and records this behavior using appropriate metrics.
3. This data is used to build a *Player Model* that categorizes players using coefficients, describing the degree of membership to multiple categories.
4. We use this model to tune *weights* and *coefficients* in our Procedural Content Generation (PCG) algorithm. Specifically, we use a type of generator called Wave Function Collapse (WFC).
5. The PCG algorithm is run and produces different results based on these weights. We give

the different levels generated to the user to play. Users have therefore a direct effect on the content presented to them.

6. The system runs as a whole inside a *game demo*, that provides a seamless experience for the player.
7. Users can give *feedback* on enjoyment for a particular piece of content to further tune the algorithm.

We treat each of these points in the corresponding chapter. We conclude that, in our specific case, with our game, parameters and generative algorithm, users report a higher enjoyment when we adapt the content rather than when we do not.

The thesis is structured as follows: in Chapter 2, we look at research done in the field of adapting game content to the user. We look at both academic and industry techniques. We further introduce psychological models that we are going to use to categorize players in the adaptation part of our algorithm. We also introduce the generative algorithm used in our method.

In Chapter 3, we go into detail on how we generate the procedural levels, which are the main form of personalized content we provide. We look at design constraints that have to be satisfied, how we can generalize our algorithm to work in 3D and with any tileset, and how to dynamically tune its weights based on user behavior. We show the results of the final generation algorithm working in-game. The generative algorithm is kept as general as possible, to allow the generation of any level embeddable in a 3D lattice.

In Chapter 4, we look at the developed game demo, discussing the implemented mechanics and the decisions that shaped it. We look at different design and iterations, analyzing why some were better suited for our purposes than others.

In Chapter 5, we present the process we used to conduct the study and collect data of player-behavior and measure the performance of our method. We analyze data and provide results about the viability of our method. We also provide some interesting insights we obtained through the data, that could be helpful for future work.

In Chapter 6, we discuss the overall results of this thesis, what worked well, what we could adjust and what could be improved. We provide suggestions for other applications and future work. We conclude with an outlook on what could come next.

2

Related Work

There is research exploring the field of games that change based on how the user behaves. The body of work is however minimal, and most of the techniques discussed are developed and integrated directly into videogames. Furthermore, most of the research done is in the field of dynamic difficulty adaptation, and not of content customization. The papers that take into consideration the player's behavior to procedurally generate levels usually feed all the collected metrics into a neural network. No approach has tried to use player models as a suitable representation of users to generate adaptive levels. Using models to provide personalized content is the focus of this work. In the following sections, we provide examples of games that change depending on user performance. We also look at papers that either analyze player behavior or adapt procedural content to the user. We then delve into some of the topics that are useful in the following sections, such as player models and generative algorithms.

2.1 Adaptive Games

One way of adapting games to different users is by changing the game's *difficulty*, either before the game starts or dynamically during play. Developers can tune the difficulty by adjusting progression curves and parameters relative to the gameplay, such as spawn rates, health, damage, and ammunition. In its most basic form, asking a user for the level of difficulty at the beginning of the game is a way of custoizing the game.

Adapting the Artificial Intelligence (AI) is also a common method to adjust the difficulty [22]. Notable games that do this are *Black and White* from Lionhead Studios, Capcom's *Resident Evil 4*, and Kojima's *Metal Gear Solid 5*. In the latter, for example, the enemies start to wear helmets if the player uses snipers often.

Adapting AI to the player's behavior could, however, result in more frustrating gameplay if not done correctly. Players could feel the AI is cheating if it appears to know too much or play too

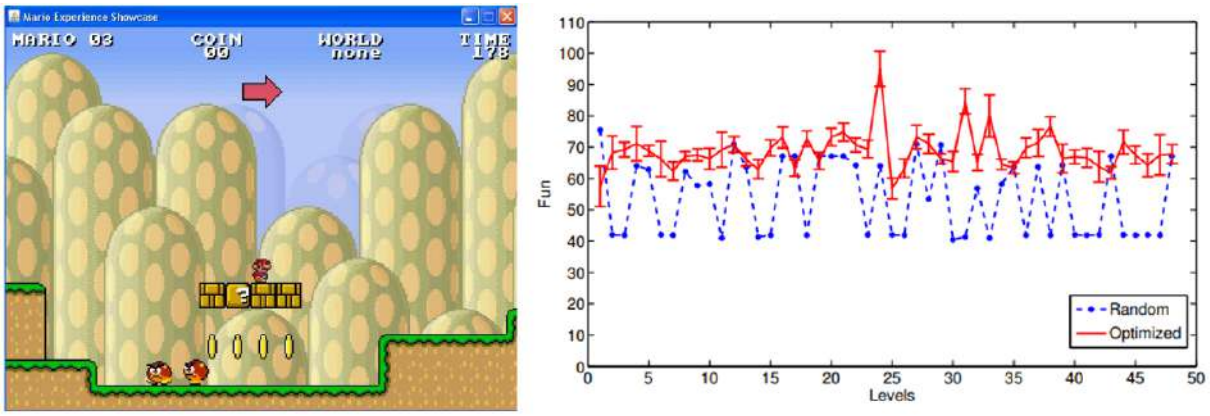


Figure 2.1: *The Infinite Mario Bros. game and a comparison of random and optimized levels for fun. Notice how almost all of the optimized levels achieve a higher score for fun. Figure from [24].*

well. It is usually more enjoyable to have the upper hand, and a beatable AI serves this purpose well.

When changing difficulty, the content presented to the player is however always the same. This adaptation differs from adjusting the content, in which different levels, scenarios or items are provided to each users.

2.1.1 Research

Shaker, Yannakakis, and Togelius [24] showed how to automatically generate personalized procedural levels for platformer games. They were able to predict the player's experience based on several recorded metrics. They split metrics into emotional states, such as fun, frustration, and challenge, and analyzed which features out of a set of 58 had the most significant impact. Users played variations of a clone of Nintendo's classic platformer *Super Mario Bros.* while having their in-game actions recorded. After every level, the authors asked users to rate the level. They then trained an Multi-Layer Perceptron (MLP) to predict emotional states based on level parameters, achieving a high prediction accuracy (64% to 85%). Based on this, they were able to compute which level metrics to tweak to achieve high values in each of the considered emotional states. Figure 2.1 shows the game used and the results.

Other research done in the field of adaptive games tried to maximize player satisfaction [32]. The authors define an adaptive mechanism that modulates parameters of a physical dancing game on an interactive platform. Machine Learning (ML) and gradient ascent are used to steer the parameters of the game towards higher entertainment for the players. There is also further research in how to map game levels to players [21], on evolving the procedural generation of tracks in racing games based on player performance [28], and on how to design adaptive game mechanics that will lead to emergent gameplay [25]. Additional research into *experience-driven* PCG [33] suggests that by categorizing player experience via affective and cognitive modeling, and by adjusting the content in real-time according to user preferences, the experience is positively affected. The paper also provides a taxonomy of the most-used PCG algorithms organized by category.

Müller et al. [9] [10] researched player behavior for the game *Minecraft*. The authors develop tools to visualize how players act and interact, to help designers and server administrators tune the game and modify the game experience. In these papers, the authors investigate a player classification for this particular game, and use the data collected is used to predict player collaboration.

2.1.2 Applications

We now present several examples of practical applications where games adapt to the user.

Rubber-banding One of the most simple is *rubber-banding* in racing games. Enemy racers are artificially slowed down if the player is behind them, depending on the distance. Similarly, they are sped up if the player is far ahead. The effect is similar to connecting the player and the enemy racers with a rubber band. The rationale behind this mechanism is that it provides a more challenging gaming experience: if the player is far behind, he or she should have a possibility of recovery, or the race won't be as exciting. Likewise, a race in which the player gains a large margin in the beginning and then keeps it is not engaging. This method allows the game to keep the player in the optimal difficulty range.

Matchmaking Another way to adapt the game to players is a matchmaking algorithm based on parameters recorded or inferred about the user. The algorithm matches players based on criteria decided by the designers. One traditional example are skill-based matchmaking algorithms. Examples are the Elo ranking system [17], used both in chess and competitive games such as Valve's *Counter-Strike: Global Offensive*. Other skill-based ranking systems exist, such as Microsoft's *TrueSkill* [29]. They both assign coefficients to players, representing the estimated level of skill and eventually the variance or confidence interval. When given new information, such as wins/losses against other players, the coefficients are updated. Wins against players of different skill are weighted differently depending on these coefficients. Results in matches against opponents with greater relative difference in skill carry more weight than against players of similar ability. This strategy provides a system that tends to converge to the underlying skill of each player. Game designers can use such systems to pit players of similar skills against each other, ensuring balanced matches that are more engaging for both sides. Such systems can also be tuned to consider parameters other than skill. For example, players with a particular playstyle can be matched with similar opponents or teamed up with users with a similar playstyle to modulate gameplay.

Left 4 Dead One game that goes a step further in adapting the pacing of the game to the players is Valve's *Left 4 Dead* [3] [30]. The developers want to promote replayability and generate dramatic pacing. To achieve this, they adjust the pacing dynamically to react to how the players are progressing through the game. The designers want to generate spikes of interest at crucial moments to keep the users interested. Action spikes that happen too frequently would exhaust the players, whereas supplying too few would bore them. Their goal is to use *structured unpredictability* to place these spikes, depending on how the users react. To do so, they keep

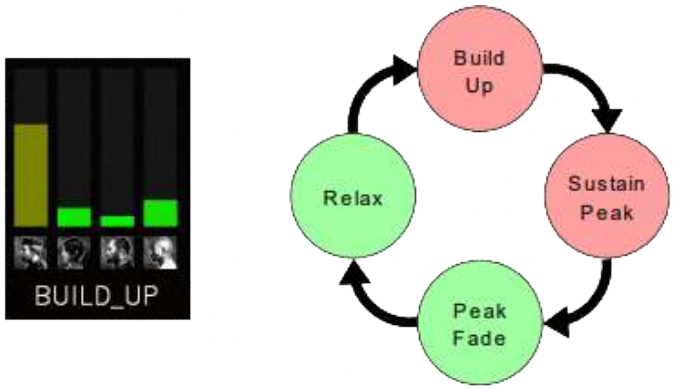


Figure 2.2: *The figure depicts the stress level of four players, as well as the four phases and their progression. Figure from [3].*

Figure 2.3: *A plot of how the stress parameter (survivor intensity) modulates the number of enemies spawned in Left 4 Dead. Figure from [3].*

track of the stress level of each player with a single one-dimensional parameter, as seen in Figure 2.2. This parameter is increased by a set amount when a particular action happens, such as killing an enemy or taking damage. The game then modulates across four phases depending on the value of the parameter: a *build-up* phase, in which the system spawns more enemies, is followed by a *sustain-peak*, in which the stress level maxes out. After that, a *peak-fade* phase reduces the number of enemies and hazards spawned, allowing the stress level to decrease. Once the parameter reaches a minimum value, a *relax* phase follows, in which the system spawns useful pickups and allows the players to recover. After that, the cycle starts over.

The system was widely acclaimed and well received by players. Although it is simple, it allows the game to detect how well or how poorly the players are progressing, and tune spawning parameters that modulate the gameplay to fit the situation. The resulting population size, compared against the desired density, is shown in Figure 2.3.

Other Notable Games There are many more games that change the content shown to the player based upon in-game actions. Examples are Telltale's *The Walking Dead*, in which the story diverges depending on the choices made. Every storyline is however hand-crafted and has, therefore, a high cost associated to it. For this reason, it is not possible to provide too many options or storylines that do not eventually converge. This reason is why PCG is well suited

to generate content. Once the system is in place, the cost of generating new content is very small. Another notable example is Monolith Productions' *Middle-earth: Shadow of Mordor*, in which the procedural Nemesis system simulates the ranks and evolution of roles in the enemy army. By defeating an enemy, another orc takes its place in the command hierarchy, changing the unfolding story. If the orc defeats the player, it is promoted and is harder to defeat in the next encounter. Enemies also acquire traits based on how the player approaches missions.

2.2 Psychological Models

In this section, we look at some models used to categorize players. Some of them apply to any population sample, whereas others are more suited for gamers. We use some of these models when trying to map user behavior to categories and when designing the game mechanics to include in our demo.

All of these models try to categorize players across axes or to put them into bins while uncovering underlying modalities about play and behavior.

2.2.1 Models Overview

Bartle's Taxonomy One of the models we analyze and use the most is *Bartle's Taxonomy* of player types [2]. In his seminal 1996 work, Bartle analyzes typical behavior of players as observed in Multi-User Dungeon (MUD). He discovers they can be categorized based on their in-game actions. He suggests a division of players across two axes: *acting versus interacting* and *players versus world*. The first axis represent the control modality, the second the content. He combines these axes on a two-dimensional plot, obtaining four quadrants. He labels the four resulting categories as follows:

- Killer: players who enjoy interfering and imposing themselves on other players. Their most basic need is that for *power*.
- Achiever: players who focus on doing things to the game, acting on the world. They try to collect status tokens, and their primary drive is that for *safety*.
- Explorer: players who are interested in understanding the game world, and interacting with itld in a more profound, meaningful way. They are most interested in being *knowledgeable*.
- Socializer: players that seek friendship with other players and want to be part of a community. The primary need is *social* contact.

We can represent the four quadrants graphically as shown in Figure 2.4. Stewart and Zenn [27], [34] provide more explanations and in-depth discussions about this model.

Myers-Briggs Another model we use in our tests is *Myers-Briggs's Type Indicator*. It was developed by Cook-Briggs and her daughter Briggs-Myers starting in 1917. It splits personality across four axes:

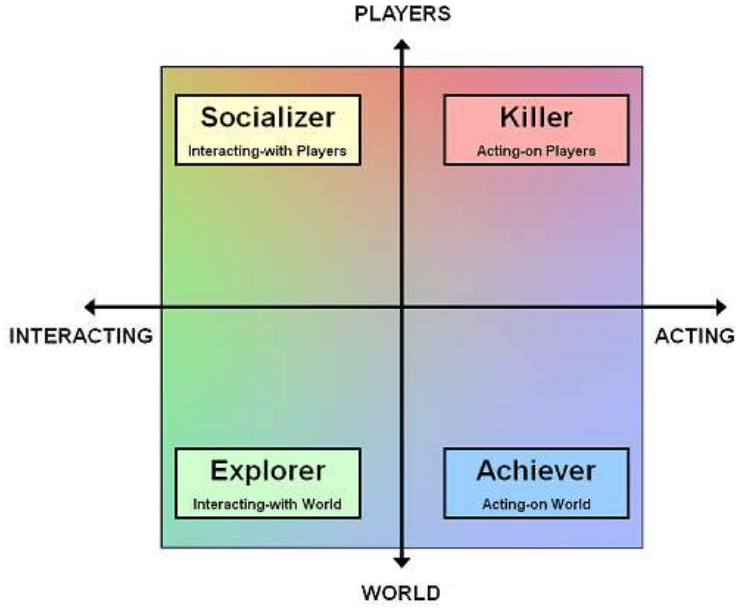


Figure 2.4: *The two axes described by Bartle, with the four resulting quadrant. Figure from [27].*

- What is the primary interaction method? Extraversion-Introversion [E-I]
- How is the information collected? Sensing-Intuition [S-N]
- How are decisions reached? Thinking-Feeling [T-F]
- What is the preference for action? Judging-Perceiving [J-P]

By splitting every subject across the four axes, we obtain 16 different categories, each of which is associated with peculiar characteristics and encoded via the resulting 4 letters representing the categories. Neris [1] offers a good overview on this subject.

Other Player Models There are many other player models suggested in the literature. Most of them closely map to Bartle's Taxonomy. Bart Stewart's article [27] gives an overview of the most influential ones. Included are Keirsey's *Four Temperaments* [14], Caillois and Lazzaro's models [4], as well as the Gamism-Narrativism-Simulation (GNS) and Mechanics-Dynamics-Aesthetics (MDA) frameworks suggested by Ron Edwards [7] and Hunicke, LeBlanc and Zubek respectively. Stewart tries to combine them into a *Unified Model* to explain existing games. Canossa [5] also tries to model player behavior in computer games using a model he calls the *Play-Persona*. Malone [16] theorizes that three factors make games enjoyable: challenge, fantasy, and curiosity. By categorizing players across these dimensions, it is also possible to derive models to encode different players.

Maslow's Hierarchy Maslow's hierarchy of needs is a model of intrinsic behavior motivation. Its structure resembles a pyramid, where blocks at the bottom need to be satisfied before those above will be considered. Figure 2.5 depicts this hierarchy graphically. We can observe parallelisms between the player models previously mentioned and Maslow's hierarchy. If we ig-

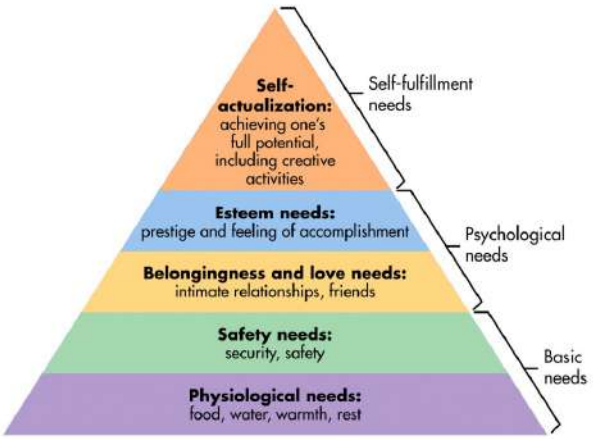


Figure 2.5: *A visual representation of Maslow's hierarchy of needs. At the bottom, basic needs such as physiological ones and safety. They are followed by psychological needs, that include outward-aspects such as relationships, and inwards ones, such as esteem. Finally, self-fulfillment needs are the capstone of this pyramid - image from Wikipedia.*

nore *physiological* needs which do not come much into play in the virtual world of videogames, we see that achievers closely correspond to the safety level of the pyramid. Socializers fit very well with the belongingness and relationships layer. Explorers and killers, although the not very descriptive names, seem to fit into the esteem and self-actualization levels, that require knowledge and action respectively.

The Concept Of Flow Mihály Csíkszentmihályi proposed in 1975 the theory of *Flow* [6], described as a mental state of full immersion in the activity at hand. The concept has been used in videogames to tune challenges to the optimal level. This theory is instrumental in the case of dynamic difficulty adjustment, to ensure the player doesn't feel boredom or frustration, caused by levels of challenge too low or high respectively.

2.2.2 Unified Model

What most of these models seem to suggest, is that there are underlying needs common to all player types, and that, by using these needs, it is possible to understand the audience and provide a better gaming experience. Most of these models also imply that games are a form of need satisfaction, where different players try to fulfill latent wants through play. These models can be leveraged to build many different types of multidimensional representation of game users by using coefficients, each representing the degree of membership to different categories. After thorough analysis, we decided that Bartle's Taxonomy was a general enough framework to provide enough flexibility and prediction power. It is, therefore, the primary model used in this thesis. We use it to both categorize players, as well as predict which game mechanics and elements are more likely to be enjoyed by a particular user.

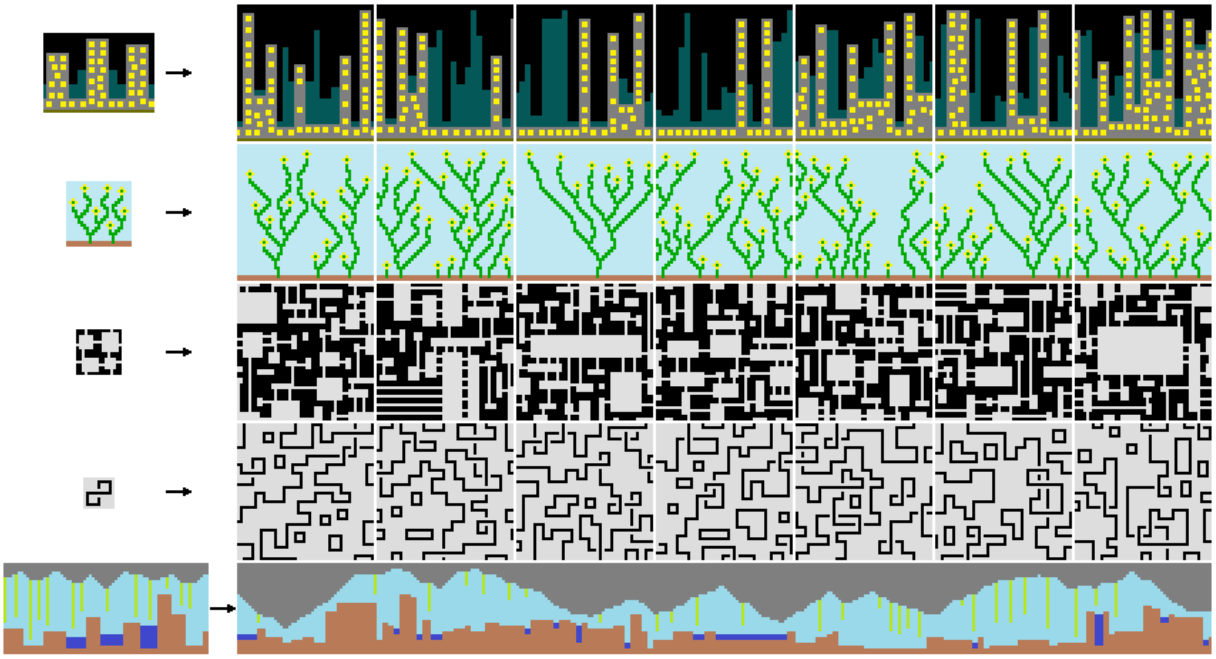


Figure 2.6: *Sample input for the* WFC *algorithm as well as the generated output. In the last example, the reader can see how appropriate input images and distributions can be used to generate output resembling a platformer game.*

2.3 Wave Function Collapse

The WFC is a generative algorithm introduced in 2016 by Russian developer Maxim Gumin under the pseudonym *ExUtumno* [8].This algorithm generates bitmaps that are locally similar to input bitmaps. *Quantum mechanics* were the inspiration for the algorithm, that uses similar ideas of superposition of states (wave function) and local observations of individual states that cause them to collapse to a definite value. Sample inputs and their corresponding outputs can be seen in Figure 2.6.

This algorithm serves a starting point for the development of our personalized procedural level generator. We discuss the inner workings of the algorithm and our implementation in more depth in the next chapter.

The author stated that this work was heavily inspired by Paul Merrell's work on Model Synthesis [20] [19] [18], that explores ways of automatically generating both images and meshes from a small sample of user-defined inputs. The output satisfies various geometric and algebraic constraints that enforce different types of similarities between input, output, and their distribution of features.

The algorithm is relatively new but has already been widely used by the developer community, with ports in most programming languages, efficient implementations of different models [15] and extensions [23]. It has even been used in tech demos and games, such as Brian Bucklew's *Caves of Qud* in its 2D implementation, as well as Oskar Stalberg's *Bad North* [26] in its 3D form. Various online demos were also developed (see Figure 2.7). Researchers investigated the parallelisms between WFC and constraint solving methods [12], as well as similarities with

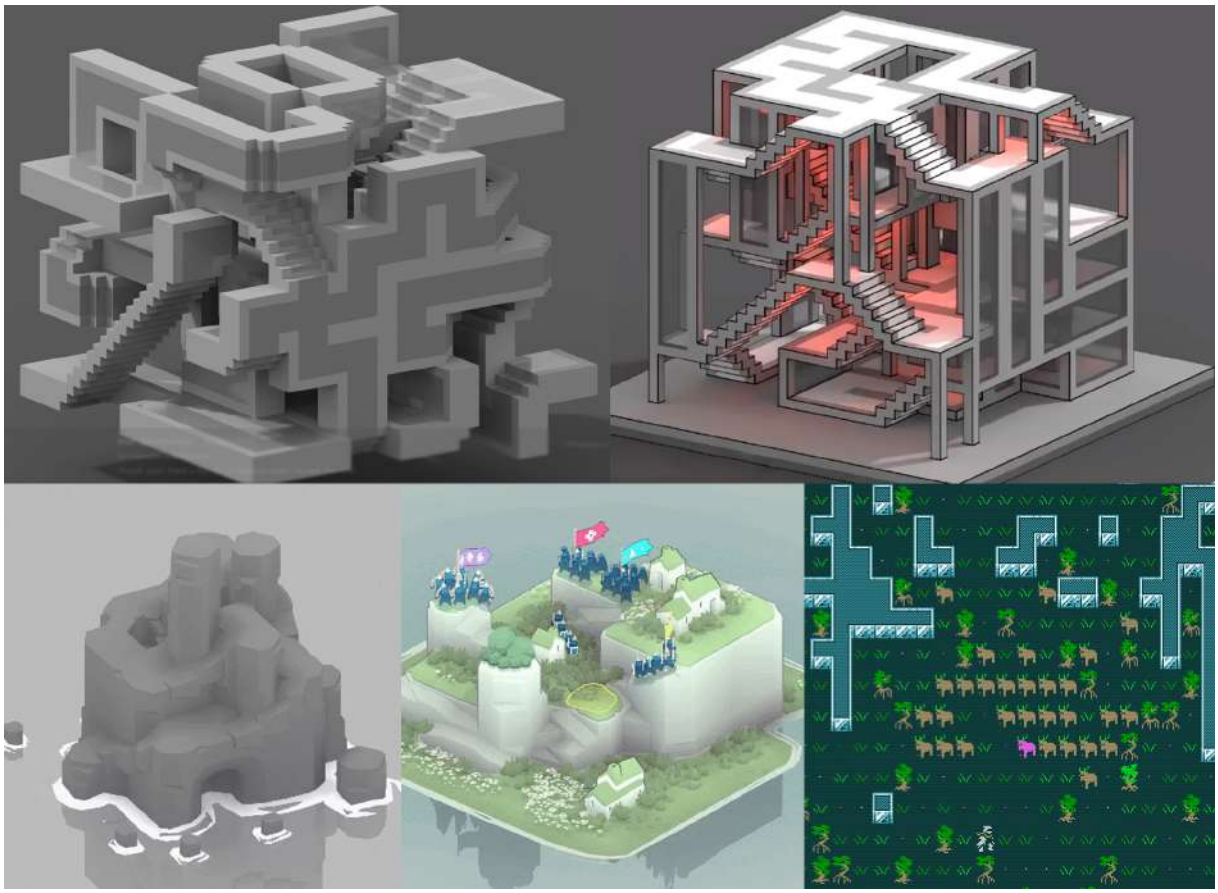


Figure 2.7: *Examples of results obtained with both 2D and 3D* WFC. *Above: Escher-esque structures obtained by combining modules. Below: islands generated for the game Bad North and Caves of Qud.*

Procedural Content Generation via Machine Learning (PCGML) [13]. We use this algorithm to generate and adjust the personalized levels in our game.

3

Procedural Level Generation

In this chapter, we explore how we procedurally generate levels that adapt to the user.
We start with the Theory Section 3.1, where we define the goals and constraints we have to satisfy. We then introduce fundamental concepts relative to the algorithm and discuss the technical design decisions. In the Implementation Section 3.2, we look at how the theoretical concepts can be applied practically in code. We also discuss technical details such as data structures and implementation decisions. We discuss some problems encountered and the solutions found to solve them. In the Results Section 3.3, we show some of the results obtained at different implementation stages. We exhibit the final generated levels, their variability, and discuss playability factors. Finally, in the Adaptation Section 3.4, we discuss how we can adapt the generation process to take user preferences into account.

Overview The following formula summarizes the pipeline for the whole process:

$$X \to P \to W \to L \to S$$

- Players play a level and exhibit behavior. We record their metrics and package it into a data point $x_{i,j} \in X$.
- We use the data X of all players to compute the player model $p_i \in P$ for a particular player i.
- We use this player representation p_i to find a tentative weight vector $w_j \in W$.
- We use the weight vector w_j in our WFC algorithm to generate a level $l_j \in L$.
- Users play this new generated level and provide a rating $s_j \in \mathbb{N}$, used to tune our algorithm further.

The cycle is now complete, and a new data point $x_{i,j+1}$ is collected from this new playthrough. Index i represents a player, index j a round and level.

There are four functions involved in this process, represented by arrows in our formula:

- The model function $\mathcal{C} : X \to P$ maps the recorded round metrics of a player to a player model.
- The adaptation function $\mathcal{F} : P \to W$ maps a player representation to weight parameters.
- The generative algorithm WFC : $W \to L$ uses the weights and the hand-crafted tiles to generate a level.
- The player itself: $\mathcal{R} : L \to S$ plays a level and gives it a rating.

We now look at how we can encode, represent and implement these various functions.

3.1 Theory

The goal of our algorithm is to generate content in the form of levels. These levels must be suitable for our game and adapt to the player. The in-game player behavior should be taken into consideration and influence the generation process.

We therefore need a procedural content generation. The generator should be able to create a wide variety of outputs, such that different users can experience different content. We also need to have control over it, meaning there must be some input that is either adjusted manually or provided by another sub-system. If the algorithm simply relied on a random number generator, the only control we would have over it would be the random seed, which doesn't offer nearly enough expressiveness or personalization. The algorithm should also be fast, and be able to generate levels at runtime without burdening the user with long loading times. After considering all the above goals and constraints, we settled on the *Wave Function Collapse* algorithm (introduced in Chapter 2), that provides a perfect starting point for all our requirements.

3.1.1 Algorithm Description

In its original form, the WFC takes a bitmap picture as input and outputs a bigger image that resembles the input. This output image contains coherent, locally similar patterns to the input image. You could imagine the process as cutting up the input into small squares, containing a few pixels each, duplicating each patch multiple times, and trying to glue them together in a coherent way by looking at the borders and overlap. Figure 3.1 shows an example, in which some patches and their corresponding rotated output are highlighted.

With this algorithm, it is possible to manually define a small input image of desirable features and patterns that are automatically coherently combined. It is therefore well suited to define small local patches of our level, for example, the patterns for streets, obstacles, and buildings, and obtain a bigger level in which features are combined realistically with a specified distribution.

Figure 3.1: *An example of the output generated by* WFC*, in which it is possible to see some of the patches used. Image from ExUtumno [8].*

3.1.2 2D Concepts

Simple and Overlap Models There are two versions of the algorithm: one that works with patches of size $N \times N$ pixels, called the *Overlapping Model*, and one that works with smaller 1×2 patches called the *Simple Tiled Model*. The latter is - as the name implies - simpler, because the only constraints that have to be satisfied are between neighboring patches. This model, however, offers less expressiveness and the generated structures are less complex. The reason is that in this model, the algorithm understands only contact constraints and not constraints between modules that are further apart.

Pixels and Sprites The algorithm can be changed to work not only with pixels but with sprites as well. Instead of using a colored pixel as the smallest unit, we can use a square sprite instead. The input thus cannot be a single image anymore, but we have to specify the individual sprites (patches) that the algorithm uses, as well as the boundary constraints that have to be satisfied between patches.

Modules, Slots, and Tiles To be more precise in our description, we differentiate between the terms tile, module, and slot. *Module* defines a sample tile that can be duplicated and placed at a particular location in our output. *Slot* defines a fixed position in our output, that must be filled by a unique module. We borrow this nomenclature from Oskar Stalberg's talk [26]. We further distinguish the concept of *tile*: tiles are unprocessed modules, usually sample sprites, meshes, or pixel patches, whose adjacency constraints are still unknown. A single tile could be transformed by using reflections and rotations to obtain a particular module, once augmented with adjacency data. Different modules can share the same underlying tile as primitive, but must then have different transformations to be unique.

Rotations and Symmetry Sprites that can be rotated and reflected constitute a *dihedral group*, more precisely the D_4 group. When working with sprites, we can generate 4 rotations for each side, and we can flip the sprite along one axis. Therefore, for every sprite, there are 8 possible generable configurations.

Having to list the same adjacency constraint for the same tiles multiple times based on their rotation and reflection is very time-consuming, so we determine the symmetry for a particular tile and let the algorithm figure out itself the other implied adjacency rules. We can define

Figure 3.2: *A sprite depicting the stop sign shows the Dihedral group D_4.*

Figure 3.3: *The five symmetry groups explained above, with the class letter and sample sprite. Image from [8].*

classes depending on the invariance of the sprite to the D_4 transformations. We give classes descriptive names such that actions of the dihedral group D_4 on the letter representing the tile has the same effect as on the sprite itself. We distinguish the following categories, depending on symmetry along the aligned or diagonal axes:

- X: symmetric along all 4 axes, invariant to all transformations.
- T: symmetric along one aligned axis, invariant to one reflection.
- I: symmetric along both aligned axes, invariant to both reflections, invariant to double rotation.
- L: symmetric along one diagonal axis, invariant to reflection and double rotation.
- \: symmetric along both diagonal axes, invariant both reflections and rotation, invariant to double rotation.

These groups are represented graphically in Figure 3.3.

3.1.3 3D Concepts

Since we generate levels in three dimensions, in this section we discuss how to adapt the general ideas from the previous section to work in 3D as well.

Voxels and Modules

The generalization of a pixel to 3D is the *voxel*. There have been successful ports of the WFC algorithm to work with voxel structures [15]. With this method, it is possible to input a $X \times Y \times Z \in \mathbb{N}^3$ voxel lattice, where each entry is assigned a color, and obtain a bigger voxel lattice output that satisfies the constraints of the original WFC implementation described previously. We can proceed exactly like for the two-dimensional case, by cutting up the input into $N \times N \times N$ blocks and use them to produce the final result. Neighboring blocks have to overlap correctly and appear with a similar frequency as in the input.

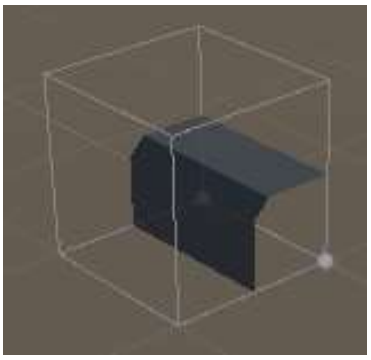


Figure 3.4: *A sample 3D mesh embedded in a unit cube.*

The generalization of a sprite to 3D is a *mesh*. Since in 2D we worked with square sprites, in 3D we work with meshes that fit within a unit cube, as Figure 3.4 shows. We consider triangular meshes whose vertices all fit either within the unit cube or on its boundary.

The aforementioned model is what we use in our generation.

Transformations

In 3D, we start working with the *octahedral symmetry* O_h [31], which represents in how many ways can we rotate and reflect cubes (and therefore, any mesh we embed in them). The dual of the cube is the octahedron, hence the name. We get $6 \times 4 = 24$ transformations of our original cube if we allow quarter-turn rotations about the three principal axes. If we allow reflections as well, we get $24 \times 2 = 48$ possible configurations. With these degrees of freedom, we can get up to 48 different modules to use in our algorithm for each input mesh. Depending on the symmetry of the input tile, we could obtain less, as multiple transformations could map to the same result. Figure 3.5 shows the topology of the octahedral symmetry group.

Our algorithm should be able to generate all the permutations for a given tile, making sure to take into consideration only those who do not produce duplicates due to symmetry. It must, therefore, understand this group structure.

3.1.4 Algorithm Overview

We are looking for a function that takes a set of tiles T, the allowed transformations, the desired output size $S \in \mathbb{N}^3$ and a probability distribution over the tiles, and generates a *coherent* level by combining the provided tiles.

The pipeline for our algorithm is the following:

- MODULE CREATION: in this phase, we generate all the modules from the provided tileset.
- WFC: the WFC algorithm runs on the generated modules.
- WEIGHT ADAPTER: a weight adaptation subsystem can tweak the weights that the algorithm uses at runtime to influence the generation outcome.

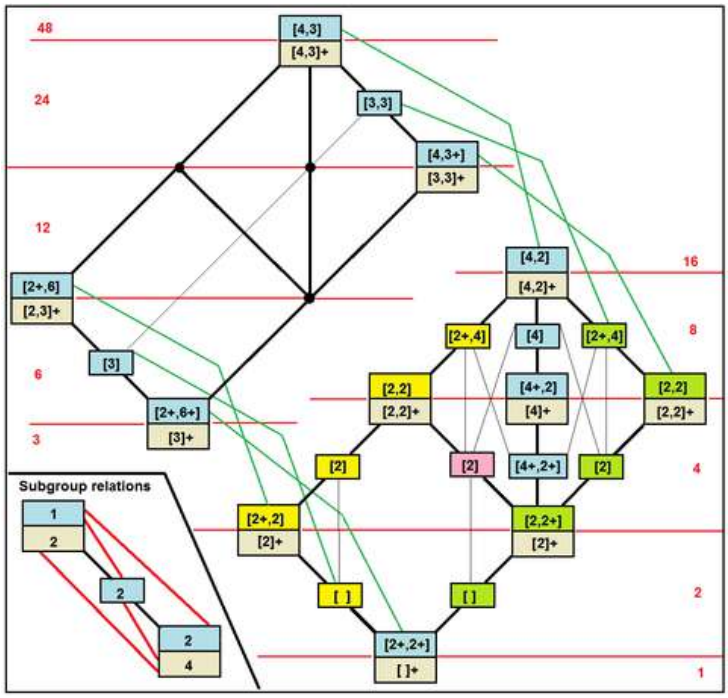


Figure 3.5: *Octahedral subgroups and their relationships are shown using Coxeter notation. Image from Wikipedia [31].*

Figure 3.6 shows the Unity interface for these three scripts, in which we can modulate the inputs. Figure 3.7 shows the chart of data flowing across these components, taking into account developers and players.

3.1.5 Modules Generation

To allow our algorithm to be as general as possible, we do not want to have to annotate tiles with adjacency constraints. We want the algorithm to understand on its own which tiles can be placed next to which others. Depending on the application, we might want to allow rotation about some axes but restrict it about others. Automatic adjacency recognition enables us to provide any tileset and directly obtain the appropriate modules, labeled and transformed.

With this system, we can generate *any* level whose structure follows a regular grid. Not a single line of code must change. It is only necessary to provide the appropriate tileset, specify which transformations are allowed (for example, excluding all rotations except about the vertical Y axis to preserve floors and ceiling structure) and the approximate tile distribution.

The algorithm computes side constraints via hashing to determine which tiles can be placed next to each other. It further generates all the permutations of the input tiles based on the allowed transformations. It keeps track of which objects must be instantiated for each tile. It also stores the weights associated with each tile and automatically normalizes them based on the number of module configurations for each tile. If we can transform a tile in multiple ways to obtain different modules, we do not want this fact to increase its probability of being instantiated, as there are now more modules per tile.

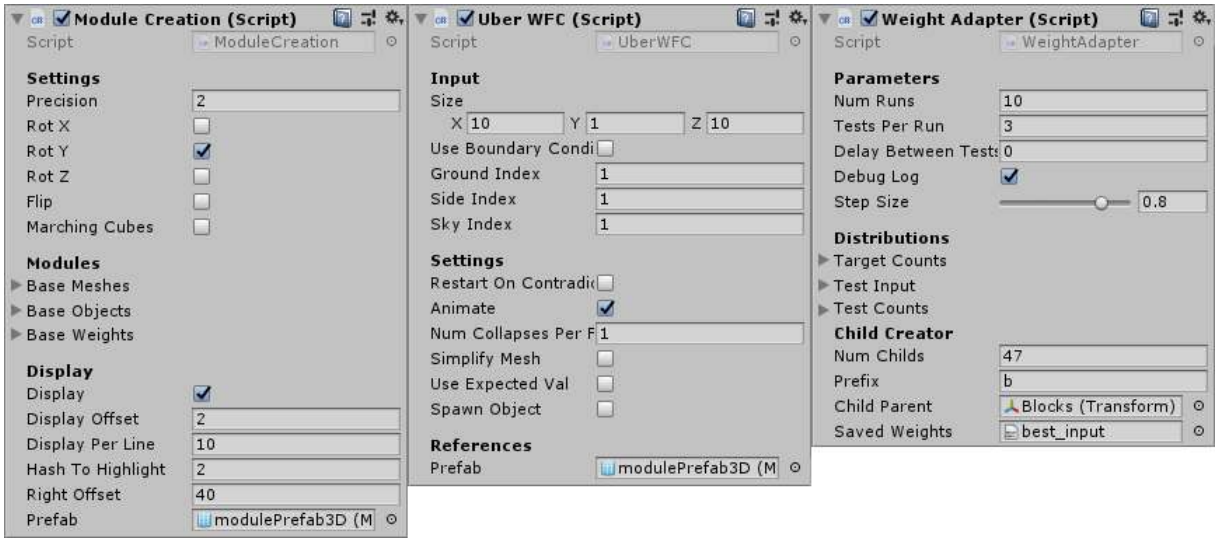


Figure 3.6: *The three main scripts in our level generation pipeline.*

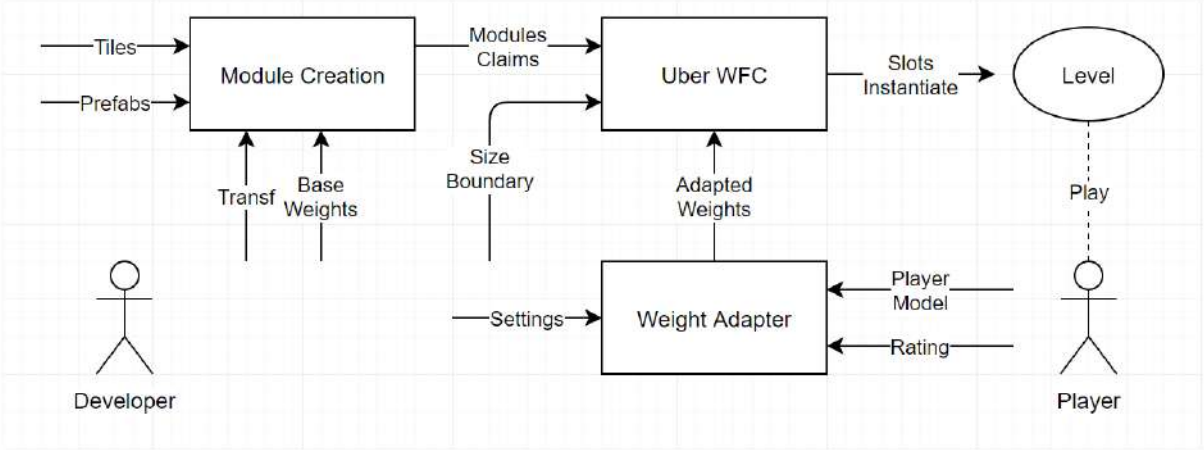


Figure 3.7: *The flowchart explaining the different steps, input, and output of the different subsystems. The developer generates a suitable set of tiles, possibly associated with a set of prefabs to instantiate at runtime, specifies a default probability distribution and the allowed transformations. The* MODULE CREATION *component produces all the necessary modules and slots and figures out all the adjacency constraints via hashing. The* WFC ALGORITHM *takes the modules and claims, as well as output size and possibly boundary constraints to be satisfied, and generates a suitable assignment per slot with the given weights. The level is generated via prefab instantiation, and the player plays it. Based upon the player model inferred from the recorded data and the rating, the* WEIGHT ADAPTER *system tunes the weights trying to maximize future ratings. The* WFC *algorithm is run to generate the next level with the adapted weights.*

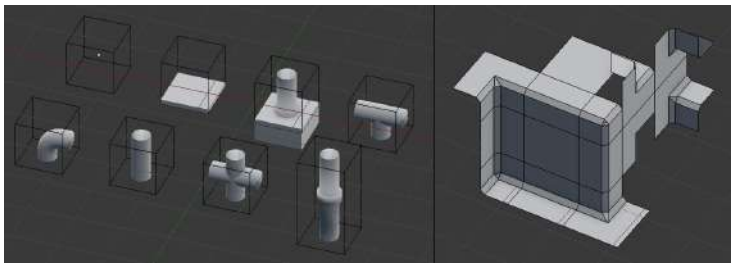


Figure 3.8: *Two completely different sets of tiles as input. Left: pipes, used to generate complex plumbing patterns. Right: island tileset, used to generate - not surprisingly - islands. The two tilesets also differ in the fact that the island tileset represents marching cubes, whereas the pipe one does not.*

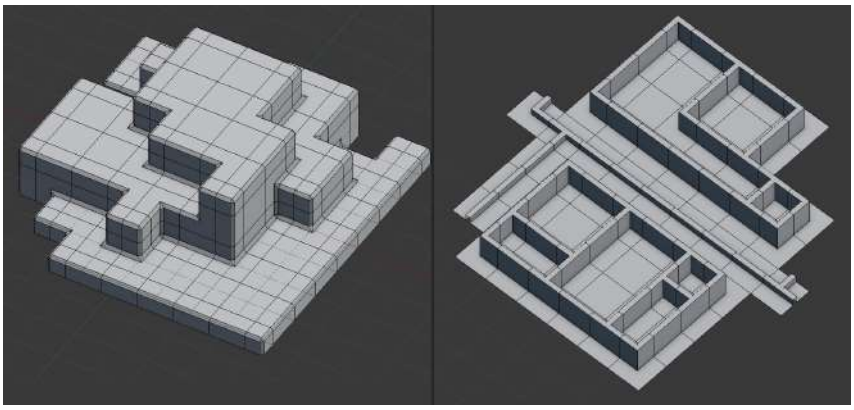


Figure 3.9: *Some outputs from the algorithm, showcasing a generated island and the interior of a building. The algorithm is the same; we only change the input tileset. The algorithm can generate any tileable level and is therefore extremely versatile and flexible.*

Input/Output We provide a 3D tileset to our Module Creation algorithm, and we specify along which axes we allow rotations. We also specify if the algorithm can flip tiles when generating modules. We provide objects to instantiate for each tile - if different from the underlying mesh representation - and also specify default weights for each tile.

We can further specify debugging parameters to highlight what the algorithm is doing behind the scenes, such as displaying the intermediate modules and highlighting specific hashes.

There are up to 48 possible modules generable from a particular tile. However, depending on symmetry, there could be duplicate configurations. We must detect all valid transformations that produce unique configurations. We need a function $f : (T, I) \rightarrow M$ that given a set of tiles T in the form of triangular meshes and a set of values I indicating the allowed transformations, produces a set of modules M, where each module is a pair $(t \in T, a \in \mathbb{R}^{3\times4})$. The set of modules must be both complete and only contain unique elements. $\mathbb{R}^{3\times4}$ is the set of 3D transformations.

We can see that the naive idea of assigning an id to each side to represent adjacency doesn't work as soon as we introduce transformations. We can rotate a side about its normal, and while still maintaining the same position and side id, the actual adjacency configuration would change. Figure 3.10 presents this problem.

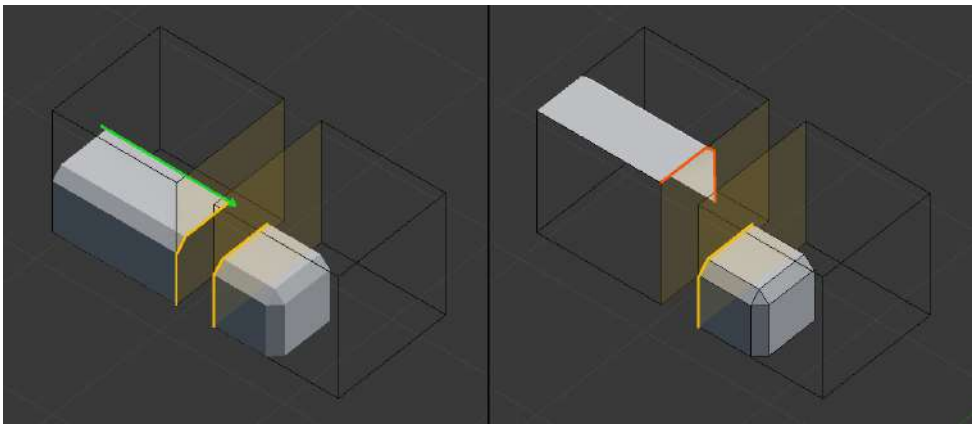


Figure 3.10: *The problem with assigning an id to a side. By fixing an unique id to a face to represent an adjacency class and then performing rotation along the face normal, the id remains the same (illustrated as the yellow face), but the adjacency is destroyed (red boundary).*

The solution we implemented is first to perform the rotation of the modules, and compute the unique per-face hashes after. A graphic depiction of this method, as well as an in-depth explanation of hash precomputation, are in Figure 3.11.

Appearance

To reduce the load of the algorithm, we generate low-poly versions of our tiles, that are meant to only convey to the algorithm the adjacency constraints. We then model higher resolution tiles that are textured, and can even be composed of multiple meshes and colliders. We assign one or more of these high-resolution tiles to each low-resolution polygonal mesh used by the algorithm. When the generation process is complete, instead of instantiating the low-poly version of the tiles stored with each module, we instead pick one of the available high-resolution ones. We then transform them appropriately to ensure consistency. It is possible to provide multiple high-resolution prefabs for the same low poly representation if the structures match. The algorithm picks one of the available prefabs randomly to enhance level variety. Figure **??** shows an example of a level as seen from the algorithm and the user, with the low poly and high-resolution meshes respectively.

3.1.6 Additional Constraints

Border Constraints

Since our levels are of finite size $S = X \times Y \times Z$, we can specify boundary constraints to make sure that only a specific subset of appropriate tiles are placed on the border. For example, we might want the ground to only contain some tiles, or the four lateral borders of the rectangular cuboid not to intersect any mesh. The user can conveniently specify these constraints via the algorithm input, and the constraints are set up in the beginning before running the algorithm. With this feature, we can, for example, generate islands surrounded by water or finite structures that don't extend against the border, such as buildings. The algorithm removes the indices of

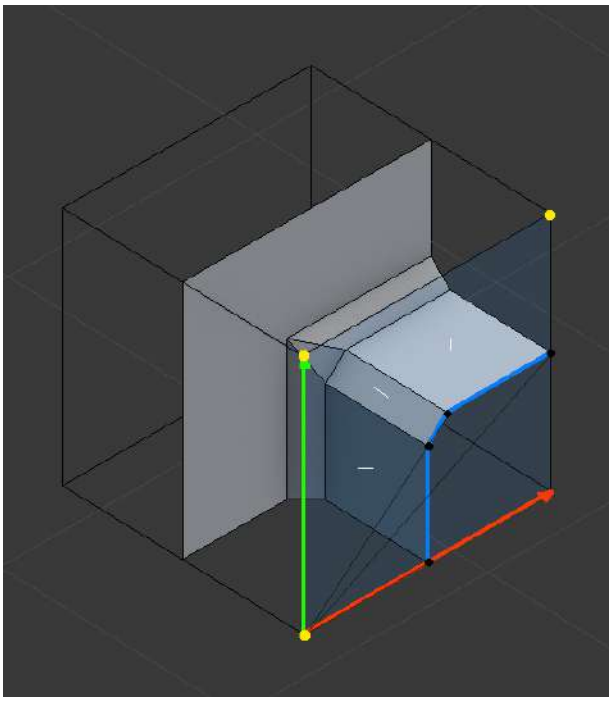


Figure 3.11: *The method used to compute side-hashes, visualized. For each side on a given cube (light blue), we detect vertices (black dots) and segments (blue lines) lying on the boundary. We project them onto the side plane with origin at the smallest corner coordinate (red and green axes) to obtain normalized, 2D coordinates in the range* $[0, 1]$*. We further compute for each corner of the cube if it is inside our outside the full composed mesh, using the face normals (white) closest to the corner. The alignment direction of the normal with the vector from corner to the cube center tells us whether the corner is inside or outside. If the corner is outside, it is marked (yellow). The set of vertices, segments, and corners is normalized (sorted and rounded) to ensure the same hash for different sets representing the same boundary, and compensate for numerical errors. The numeric hash value is mapped via a dictionary to the next unused natural number to allow our algorithm to work with side hashes in the range* $[0, H[$*, where* H *is the number of unique hashes. This also enables additional memory savings and optimizations by using smaller integer formats instead of the fully-fledged integer type.*

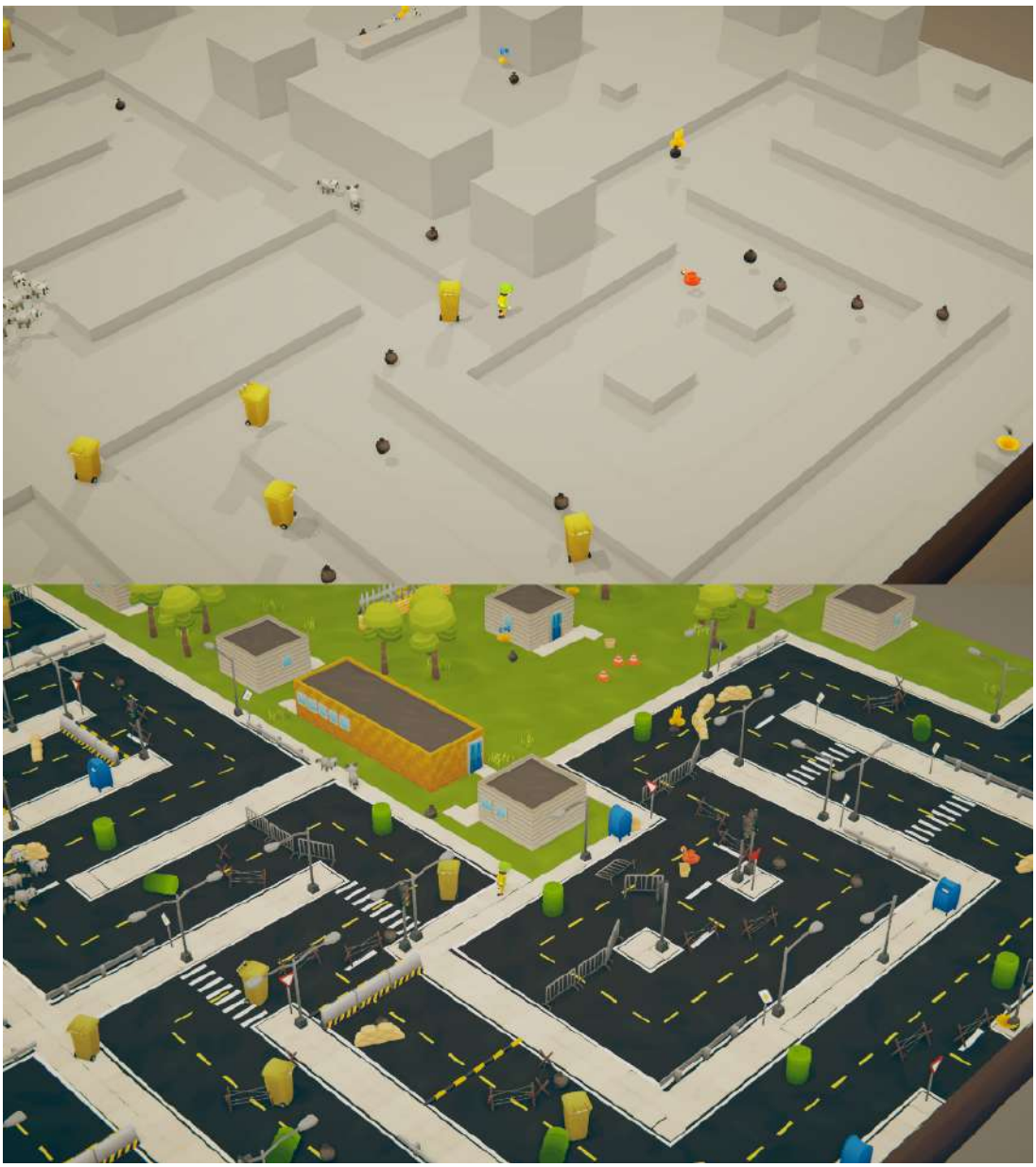


Figure 3.12: *A view of the underlying tiles representation as seen by the algorithm (above) when generating a given level (below).*

the disallowed modules at the specified positions. It is also possible to force the algorithm to place a given module in a certain slot. We achieve this result by removing all the other modules and indices from that slot, forcing the algorithm to collapse this slot as the first choice.

Multi-Tiles

One major change we made to the WFC algorithm is that we allow *multi-tiles* to be part of the input tiles. These are particular tiles that do not fit into a single unit cube. Having the algorithm work with multi-tiles expands the space of generable levels, and doesn't constrain it only to use features that fit within a single unit. Their use enables us to include bigger features, such as buildings, that give a more captivating and interesting structure to our levels.

Using multi-tiles implies that the smallest unit in our algorithm is not the module anymore. Previously, a single slot could contain exactly one module, arbitrarily rotated, and it was thus possible to store an index for each slot indicating the appropriate module.

We now need a smaller unit of computation other than the module: we, therefore, introduce the concept of *claim*. A claim is a unit cube element that makes up a module. Modules can now be of arbitrary integer size and are composed of one or multiple claims. The modules store tile mesh, object, and transformation information, whereas claims store adjacency constraints.

The use of claims changes the structure of the algorithm fundamentally, as many assumptions regarding the mapping of modules to slots are no longer valid. Furthermore, additional boundary constraints must be taken into account.Propagating information does not happen linearly anymore, since a module collapse could simultaneously fill multiple slots. We refer the interested reader to the code implementation to learn more.

Trivia We went through four versions of the algorithm before it satisfied all our constraints. We had WFC, BETTER WFC, SUPER WFC, and UBER WFC, which is the final version used in the software.

3.2 Implementation

In this section, we discuss some of the implementation details of the algorithm.

Module Creation In the implementation of the module creation system, we iterate over all available transformations and compute the resulting mesh for a given tile. We then quantize the local position of all vertices to avoid rounding errors. Every vertex is inside the unit cube, therefore has values in the $[0, 1]$ range. The quantization that yielded the best results uses values in the $[0, 1]$ range, with increments of 0.01. An implementation in which we were comparing every resulting mesh with all previously computed ones would be costly, with a runtime of $O(m^2v^2)$ where m is the number of modules and v is the number of vertices per module. For every module, we would need to check every other module and compare every vertex across the two modules. This time could be brought down to $O(m^2v \log v)$ by pre-computing sorted lists of

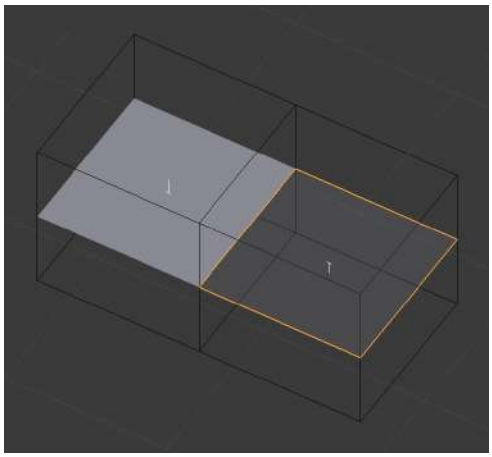


Figure 3.13: *The problem of computing hashes without taking into consideration normals. Meshes that have the same boundary are placed together although the result is undesirable.*

vertices for every module. We can further reduce the time to $O(mv)$ by using hashes. For every module, we can compute a custom hash based on the mesh. Meshes are stored using indexed triangle lists, meaning there are many ways to store the same mesh. We need to ensure that meshes that look the same but have different underlying representation have the same hash. To achieve this goal we can run a preprocessing step on each mesh to sort vertices and faces. Since we work with tiles that can potentially represent marching cubes, we must also pay attention to distinguish in which direction each normal is facing. If we do not, planes that represent ceilings or floors would be considered as being the same, leading to incorrect results, as Figure 3.13 shows.

To solve this problem, for each of the 8 corners of the cube embedding the mesh we compute if it is inside or outside the volume that the triangular mesh defines. We do this by checking whether the vector from the corner towards the center of the module is facing the same direction as the closest normal. If the dot product between the two is > 0, we can then mark the cube vertex as being inside; otherwise, it is outside.

Once we generate all the modules and augment them with the proper information, we are ready to run the WFC algorithm.

3.2.1 Pseudocode

The overall structure of the algorithm is very similar to what Maxim Gumin describes in his original WFC implementation. We don't go into detail on how WFC works, and instead focus on the changes we performed. We give here an overview of the pseudocode and refer the interested reader to Gumin's GitHub [8] or the code for the full implementation.

1. Read the input tiles, augment them with the allowed transformations and prepare the module objects.
2. Preprocess the modules to compute the adjacency constraints and the side hashes. For any given hash, store the number of modules that have it.

3. Initialize the *wave* multidimensional array, representing the allowed modules at each slot.
    a) For efficiency, do not store which modules are allowed at each slot, and instead which indices are allowed on each slot boundary. Store how many modules satisfy these constraints as an integer per slot.
4. Compute boundary constraints, by removing all disallowed modules and indexes from our wave array.
5. Precompute the entropy for each slot based on the probability distribution.
6. Iterate until a contradiction happens or the algorithm completes:
    a) Find the slow with the minimum entropy. If none is available, contradiction happened, and the algorithm terminates. Use a min-heap for efficiency.
    b) Compute the observation at the given slot with a probability proportional to the weight of the modules. Collapse this slot to the found module.
    c) Propagate the collapse information by decreasing the number of indices for all removed modules' side hash indices on all 6 sides. If on a boundary this number reaches 0, propagate this information change to that neighbor slot. The reduction of available hash affects the neighbors since no other module is available with a given side hash.
    d) If one slot is reduced to a single available module, collapse this slot recursively as well. All changes are stored on a stack to reduce algorithm times.
7. If all slots collapse without any contradiction, instantiate the appropriate prefabs. Otherwise, restart the algorithm generation.

3.2.2 Analysis of Levels

One big part of this project was to design a set of tiles that would generate interesting and variated levels. Although both the WFC algorithm and our weight adaptation system work with any tileset, a proper tileset is required to produce compelling results. We went through many different iterations of our tiles before we settled on the final version. The requirements were the following:

- The tileset should be versatile enough to produce levels that look and play differently.
- The levels generated should be well suited for playability. All areas should be accessible, and navigation should be possible for both players and enemies.
- Levels should be coherent, without any visible artifact or anomaly.
- The number of tiles should be limited, preferably under 30, to avoid a complexity blowup and longer generation times.

We had multiple iterations of the set of tiles, to ensure the levels satisfied the above constraints. Some of the tileset iterations are shown in Figure 3.14. The final set is composed of 14 base tiles.

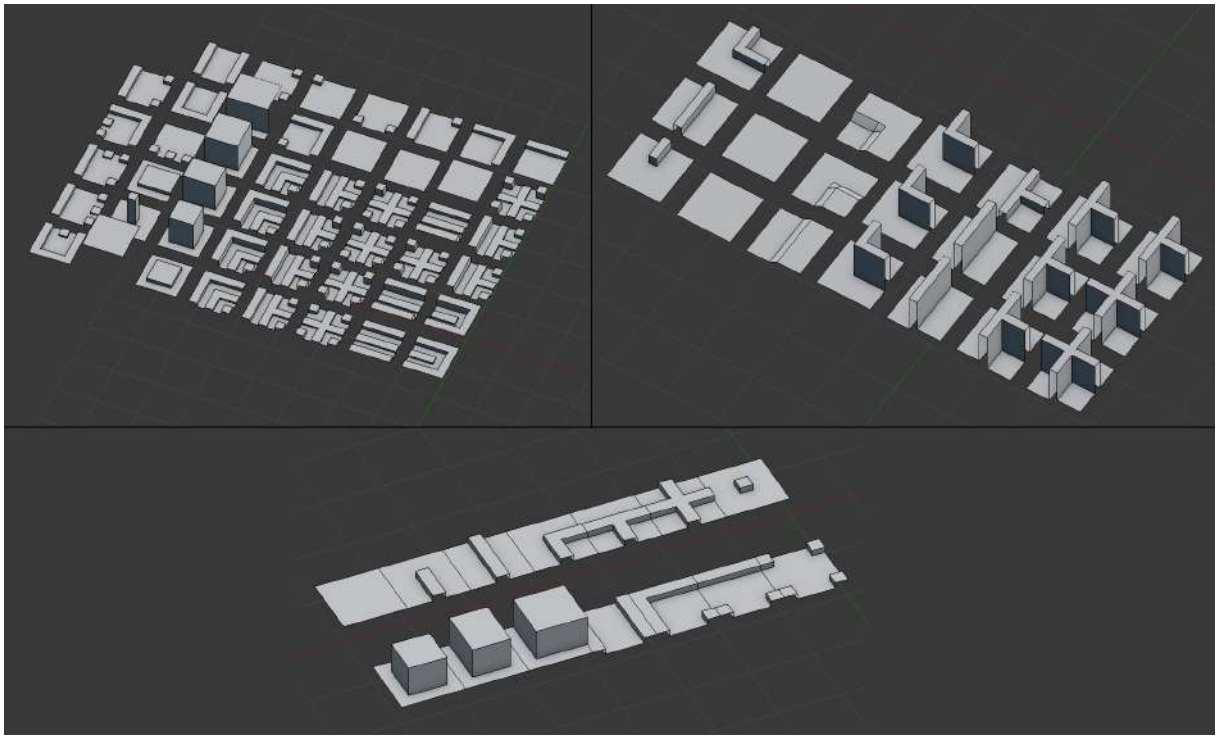


Figure 3.14: *Multiple iterations of the tileset used to express the city neighborhood layouts. Left: very expressive tileset that allows the creation of streets, parking, buildings, parks, and fences. Such a tileset is not very well suited to ensure playable levels due to the high number of features. Right: tileset used to express building interiors, with features for walls, doors, and windows. The produced results exhibit similar problems of playability as the previous tileset. Bottom: final tileset used in the game, with features for buildings, streets, and fences.*

Table 3.1: *The time in seconds required for our algorithm to generate levels of different sizes. The algorithm works fast enough for levels of size up to $20'000$ tiles. Above that loading times impact the experience. For our purpose, this is more than enough.*

size	μ (s)
$10^3 = 1k$	0.068
$27^3 = 20k$	4.72
$40^3 = 64k$	48.2

3.3 Results

3.3.1 Time Analysis

In our algorithm, $M - 1$ modules are iteratively removed from a grid of S slots, in which each slot can initially contain any module. Due to this fact and to the cost of information propagation across slots, the algorithm needs at least $\Omega(MS)$ time. The naive implementation of iterative exclusion could cost up to $O(M^2S^2c)$ if for every removal we iterate over all remaining possibilities to check suitability. The cost c represents the cost of comparing modules.

The factor D represents the number of adjacency constraint to be satisfied per module ($D = 4$ in 2D for squares, and $D = 6$ in 3D for cubes). This factor is always present and impacts the overall hidden constants. By precomputing modules constraints, as well as efficient constraint propagation, we can reduce the computation time to $O(SID+MI)$. I represents the number of different adjacency constraint to be satisfied. I is also the number of unique side-hashes *indices*, as these concepts are the same. The number I is always $\leq MD$ since in the worst case every module has all D unique sides. In practice, however, I is much smaller than M $I \ll M$, and doesn't increase linearly with M since modules can be added with the same border constraint as previous ones. Our algorithm time scales therefore linearly with the number of slots and constraints and is thus very efficient. If $S > M$ as it's usually the case, the number of modules has a minimal initial cost in the generation and is asymptotically negligible.

3.3.2 Performance

The algorithm can generate levels very quickly. The levels we generate in-game are of size $10 \times 10 \times 1$, and the algorithm needs an average of $53ms$ to run from start to finish, including memory allocation and game object instantiation. The times measured had average and standard deviation of $\mu = 53, \sigma = 23$ respectively on an Intel Quad-Core i7-6700 @ 3.40 GHz. We tested the generation for other sizes of levels. Table 3.1 reports the speed results.

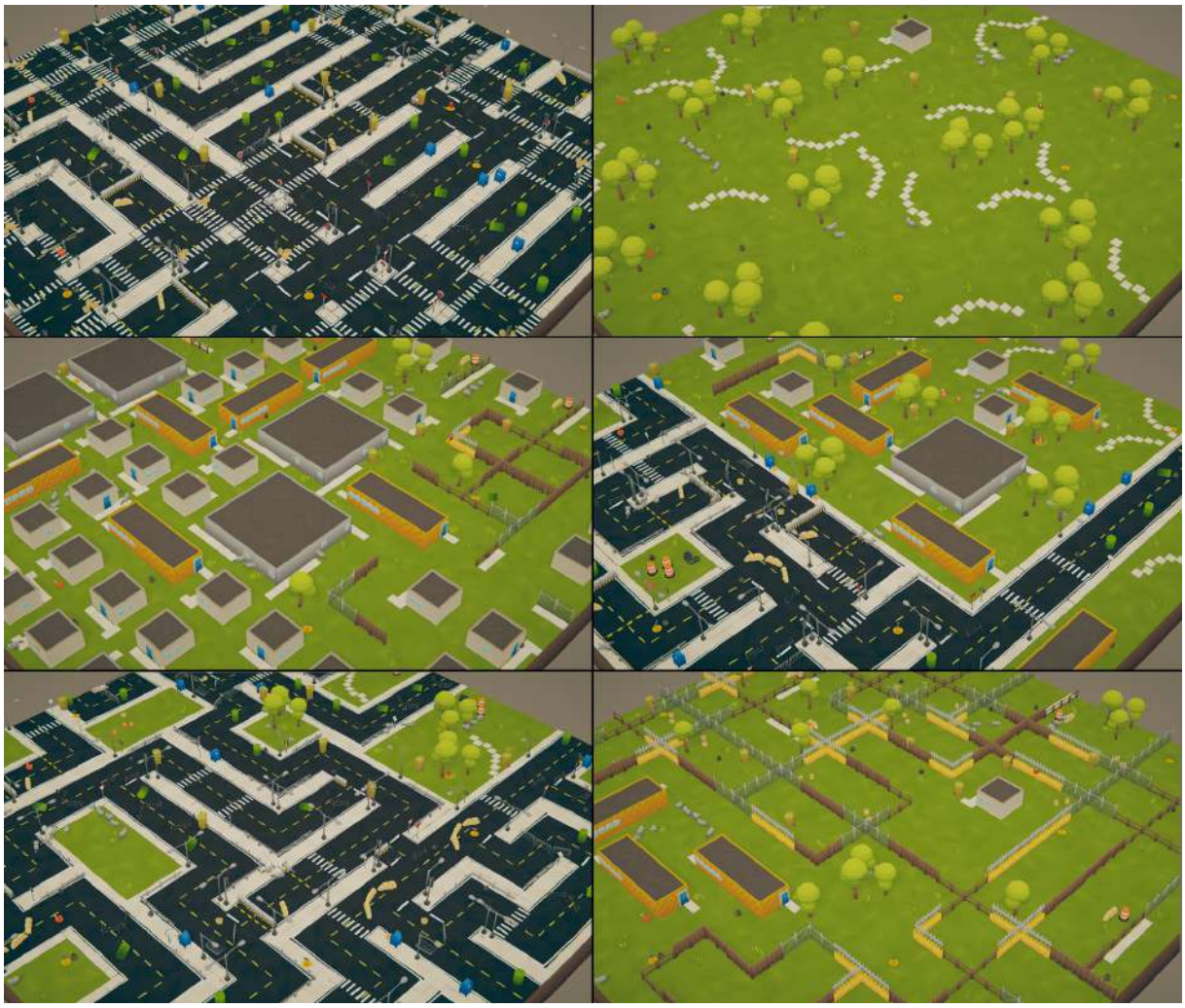


Figure 3.15: *Some variations of levels generated by the algorithm. By changing weights of different modules, it's possible to adjust the layout of streets and paths, the amount and placement of buildings, or the creation of wide empty areas. The resulting gameplay varies from open encounters to close combat in cluttered environments.*

3.3.3 Sample Levels

We show in Figure 3.15 some of the levels generated by the complete system, with weights manually tweaked to showcase different scenarios.

3.4 Adaptation

We need a way for external models to influence the generation. We decided to tune the weights of the WFC algorithm since it is possible to obtain a wide variety of outputs by only changing the probabilities of placing the individual tiles.

3.4.1 Player Encoding

As we saw in Chapter 2, some systems encode players with a single one-dimensional parameter, such as skill or stress. A lot of manual work is required in these systems to ensure that the coefficients are balanced and capture what the designers intended. On the other side of the spectrum, some systems encode players using 10 or more parameters, where a ML algorithm is fed all the recorded metrics in their raw, unfiltered form. We decided that we wanted to try an approach somewhat in the middle: with more expressiveness than a single parameter, but more control than just using all the recorded metrics. We decided that the best way to be consistent with the models introduced in the previous chapter, especially Bartle's, 4 parameters would be optimal. In the final version of the game, we focus on a few key metrics that allow us to distinguish players precisely. These metrics provide us the most information about the playstyle, as they are the ones that vary the most. They are:

- k: number of kills
- a: amount of items collected
- e: percentage of level explored
- s: number of deposits searched

For each round r_j played by a given player p_i we collect a data point $x_{i,j} = (k, a, e, s)$. The point can be generalized as $x_{i,j} \in \mathbb{R}^d$, where d is the number of dimensions. For each coefficient of $x_{i,j}$, we can compute the mean μ and standard deviation σ of the whole set of data points X, as well as μ_i and σ_i for the subset $X_i \subseteq X$ for a given player. Note that these quantities are vectors, and we can consider each coefficient individually. It is now possible to normalize the coefficients as

$$p_i = (\{-c, \frac{\mu_i - \mu}{\sigma}, c\} + c)/2c$$

where the operator $\{a, b, c\}$ is the median operator, which effectively clamps b to the range $[a, c]$ if $a \leq c$. This formula represents how much a particular player deviates from the average of all other players. Players that deviate by $-c$ or more standard deviations from μ obtain a value of 0. A value of 1 represents a positive deviation of $c\sigma$ or more, and a value of 0.5 represents no deviation. This model assumes that the measured parameters follow a normal distribution. A reasonable assumption based on the fact that the design of the game forces players to perform a certain average amount of each of the available actions, and that players' agency allows them to decide how much to deviate from that amount based on their preferences. We usually set $c = 3.3$. With normal distributions, this implies that no more than 0.096% of the players lies outside the range $[-c, c]$ and needs clamping.

With this system in place, we obtain a compact representation for the characteristics of our player. The more observation we collect, the more the representation tends to the true latent unobserved mean μ_i^* due to the central limit theorem. Therefore, so will p_i. The more the players play, the more confident we can be about the coefficients. The system is simple enough to be easily implemented and, as we show in Chapter 5, works relatively well. We, therefore, represent a particular player with a point $p_i \in P$.

We now need a mapping $\mathcal{F} : P \to W$, where $p \in P$ is a data point representing a player in our model, and $w \in W$ is a point in a d-dimensional space $\mathbb{R}^d$ representing the coefficients to feed to the generative algorithm. $\mathcal{F}$ is the algorithm that decides the level parameters for a particular player. The dimensionality d of $w \in W$ can be any value, and in our case, we have $d = 16$, where the first 14 parameters represent the weights fed to the WFC, whereas the latter 2 are coefficients for our spawners. The spawners generate enemies and items, and each coefficient modulates the expected time between two consecutive spawns and the maximum amount of allowed instances.

WFC is deterministic, and so we can simplify $W \to L \to S$ to $W \to S$. We see a single player as a function $\mathcal{R}_i : W \to S$ that for each level generated with parameters $w \in W$ assigns a rating value s in a set range, representing the overall enjoyment of that particular player for that level. We have $s \in S = \{1, ..., 10\}$ representing the number of half-stars given in-game. A rating of 1 means very little enjoyment, 10 represents the maximum. The rating function for all players is specified as follows: $\mathcal{R} : P \times W \to S$, that for a given player and level assigns a rating.

We make a few simplifying assumptions:

- We assume that players that exhibit similar behavior enjoy similar levels.
- We assume that the image of the rating function $\mathcal{R}$ is linear. Specifically, we assume that if $\mathcal{R}_i(w_1) = s_1, \mathcal{R}_i(w_2) = s_2$ then $\mathcal{R}_i(\frac{w_1+w_2}{2}) = \frac{s_1+s_2}{2}$.

These simplifying assumptions might not be entirely accurate in all cases but provide a good starting point that works well in practice. We have no way of distinguishing players other than from their behavior, so without the first assumption, our system should map the same input to different outputs to maximize enjoyment, which is not feasible. The second assumption states that if a player liked a level with a certain amount of feature one, and didn't like it with another amount of the same features, he or she will most likely enjoy it an average amount if we provide an average amount of that feature. This assumption works well in practice since, with a small enough granularity, any function can be approximated by a piecewise linear function.

The problem of the adaptation algorithm $\mathcal{F}$ is the following: given a particular p_i representing a player, find

$$w^* = \text{argmax}_{w \in W}(\mathcal{R}(p_i, w)) = \mathcal{F}(p_i)$$

We are looking for the generation parameters that yield the highest possible enjoyment for the player. This problem could be easily solved if we could compute $\mathcal{R}^{-1}$. By computing $\mathcal{R}^{-1}(p_i, 10)$, where 10 is the maximum score value, we could directly obtain the best level parameters. However, our function is unknown and not invertible.

We now have a space-searching problem. We need to efficiently search the space W for a given player p_i to find one of the elements that provide the maximum enjoyment. We leverage data of other users to do that.

The main idea is the following: look at users with a similar representation. We can now use the assumption that similar users enjoy similar levels. Weight the found users within a certain distance d_{max} in P proportionally to how close they are. For existing users, we already collected data about which levels were the most enjoyable. For each round played, we know the player model, the weight parameters and the rating obtained ($p_j \to w(p_j) = w_j \to r(w(p_j)) = s_j$). We can use the assumption of linearity of the ratings to compute an estimation of what could be

the first level parameters. Once we obtain this first estimate w_1, we generate the corresponding level l_1 and supply it to the player.

$$w_1 = \frac{1}{norm} \sum_{p_j \in N(p_i)} \delta(p_i, p_j) \cdot r(w(p_j)) \cdot w(p_j)$$

We look at the neighborhood N of p_i, and weight the found points depending on distance and rating. $\delta(,)$ is a continuous, monotonically decreasing function that assigns a value of 1 when the points are the same, and 0 when they are distant d_{max} or more. We therefore only consider the points with distance $[0, d_{max}]$. This function must not be linear. We use the cosine curve in the range $[0, \pi]$, appropriately scaled. $r(w(p_j))$ returns the rating given by the user for the given paramers $w(p_j)$. We effectively weight positive results more to steer w_1 in that direction. $norm = \sum_{p_j \in N(p_i)} \delta(p_i, p_j) \cdot r(w(p_j))$ is used to normalize the weights in the correct range.

The player plays the level and provides us a rating s_1 for it. We now have more data to be leveraged for future users, and we can continue to explore the space since the first estimate, most likely was not w^*, the optimal weights. The rating of the user influences our searching strategy: if the rating is positive, we use a tiny step size η, since we believe that we are close to the optimum due to the already high rating. If the rating is negative, we use a much higher step size, η, when computing the next weight, since we are most likely away from one of the maximum values and we want to reach it more quickly.

At this point, we have a collection of points X of rounds played. For each one, we have a data point $p_{i,j}$ representing the player model we had at the beginning of that round. This fact doesn't hold for the first level, in which we have no data about the players as they didn't play yet. Because we filter out the first two rounds played, this isn't an issue. We also have for each round a corresponding point w_j representing the level generation parameters, and the rating s_j. We can use this information to estimate which changes yield a positive increase in the rating score.

We can compute the gradient $\Delta \in \mathbb{R}^d$ for the direction in which we estimate the enjoyment increases and compute the next weight via

$$w_{j+1} = w_j + \eta \cdot \Delta$$

We now generate a new level using these new parameters, using the function **WFC!** (**WFC!**) as previously described.

Finally, the user rates the level. We treat the function $\mathcal{R}$ as a black box since we can think of the player as an external rating system.

4

Game Demo

In this chapter, we present the game developed during the thesis. The goal of the game is to provide a framework for the generative algorithm to run in to evaluate its performance. It further allowed us to have full control over the game mechanics and embed appropriate analytics where we saw fit. We begin with a design section, in which we discuss the constraints that our game had to satisfy. We also discuss the design choices for the various game mechanics, settings and game architecture to allow seamless integration with our algorithm.

In the implementation section, we briefly discuss some important details about the development that allowed the game to run directly in an online browser without causing performance issues.

Finally, in the results section, we look at the complete game, what's possible to do in it and discuss some possible improvements.

4.1 Design

Many games feature an avatar for the player to control directly. They often have either a free, lateral or top-down camera to frame the action. They also frequently feature levels to explore, items to collect and enemies to fight. We decided tha we should follow this formula, to keep the results of this thesis as applicable as possible to a wide variety of games. A game implementing the most common features would allow us to generalize the results to a bigger set of games instead of distilling results specific to a particular subset of games. We, therefore, decided to design a simple third-person top-down game in which the player can move, interact with objects, loot and collect items, as well as fight and shoot.

4.1.1 Constraints

There are several constraints that our game demo should satisfy:

- The game should be general enough to apply the results obtained in this particular instance to a broader set of games.
- The game should feature multiple game mechanics, to allow users to choose which actions to execute and still be able to beat the levels.
- There should be at least one game mechanic suited for every main player trait we discussed in the previous chapters. Multiple game mechanics per playstyle would be desirable, however, at least one is required.
- The game mechanics should be balanced. Although a completely balanced game is hard to achieve, the presence of a clear dominant strategy would bias the results toward a particular playstyle.
- The game should be accessible enough to allow both expert players as well as newcomers to play it long enough to collect relevant data.
- The levels should be short enough to allow the algorithm to adapt after every single round and to allow us to collect more data points per user in a fixed amount of time.
- The game should support the generation of procedural content that could be modulated by our algorithm to influence the player experience.
- The game should be polished enough to appeal to a wide enough audience to collect more data.
- The game should be simple enough to be implementable in a short amount of time by a single developer.

4.1.2 Game Mechanics

We decided that the game should feature at least one primary game mechanic to appeal to all player types. We decided that we should include a fighting mechanic to cater to the *killer* player type, as well as a collecting mechanic for the *achiever* player type. We also decided to include unique hidden to discover items to appeal to the *explorer* type. We decided not to include any social mechanics as that was impractical, requiring a multi-player environment.

4.2 Implementation

4.2.1 General Framework

In the first iteration of the implementation of the game, we didn't know which game mechanics were better suited for our goal, as well as which metrics we should record. We, therefore, decided to develop a very general *framework* for our game that provided some of the most

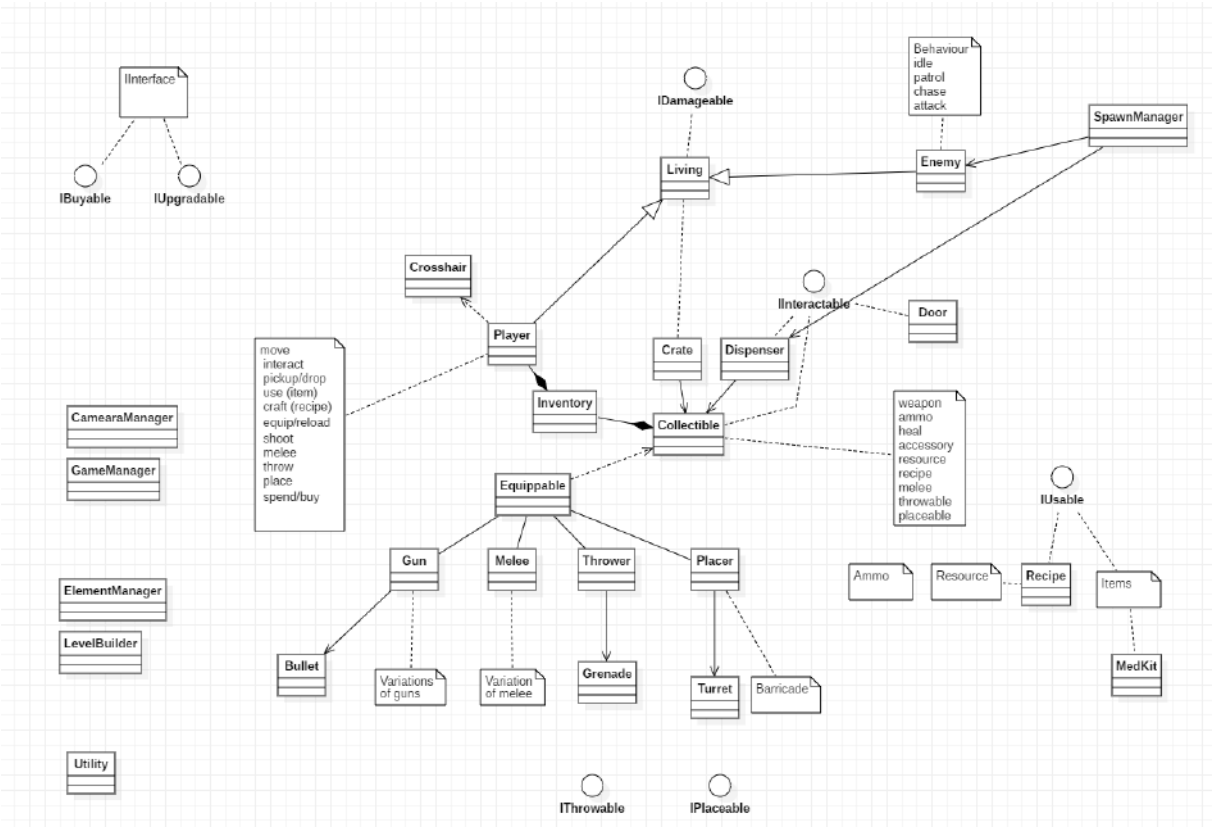


Figure 4.1: *The UML architecture of the game, with a high-level view of the components. The player class has all the components that allow it to perform all in-game actions. Equippable items are used to shoot, melee, throw and place objects in the game with a common interface. Enemies exhibit different state behaviors as depicted. A spawner instantiates them and manages their population. All collectible items share the same interface to work in the inventory.*

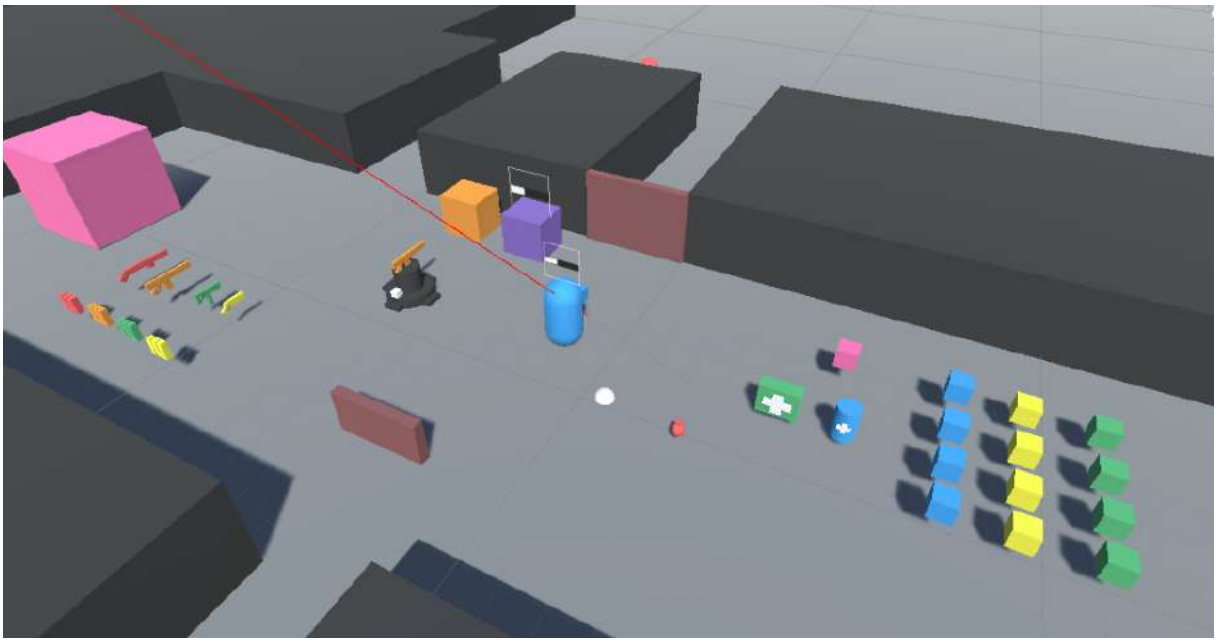


Figure 4.2: *A screenshot of the early prototype. The player, in blue, can navigate the environment, interact with the objects present, collect, craft and use items, as well as fight enemies.*

common features available in action-exploration games. Such a framework would allow us to rapidly iterate over different designs to test applicability without having to start coding a new game demo from scratch each time. By using this framework, a multitude of different third-person action games can be coded in a brief amount of time, using the provided features and a subset of the building blocks and components provided. This approach revealed instrumental for rapid game iteration during our development. A screenshot of this prototype is shown in Figure 4.2.

We implemented a system for the character to navigate the environment, that allowed him to walk, run and vault over objects, to explore the level. There is a looting system that allows users to interact and search various environmental objects to discover items. We also implemented a simple inventory mechanic that lets users collect weapons, ammunition and other useful items such as medkits, powerups, and melee weapons. The inventory has a fixed capacity to force the users to decide which items they should carry around. We implemented a fighting system that allows players to shoot, brawl and throw grenades and other objects such as bottles. A crafting system is also present in which users can collect resources, such as wood, iron, and plastic, and combine them via recipes to create ammunition, throwables and healing items. We also implemented a rudimentary AI system, in which the enemies were able to navigate the level independently using *nav-meshes*. The AI is state-based and uses a Finite-State Machine (FSM). It is composed of four states: *patrol*, *chase*, *investigate* and *attack*. Enemies patrol the level, ensuring a uniform distribution of agents across the whole level. When they detect a player, either by sight or sound, such as a weapon fired, they go into chase mode, trying to reach the player as long as it is visible. If they are close enough, they attack the player. If the player manages to run away and break line of sight, enemies investigate the last known location. After some time, they return to regular patrolling. In case enemies are in the way of the player, he or she can distract them by throwing an object and causing the enemies to investigate the impact

area. Enemies also have a simple communication system that allows them to share information with spatially close enemies. If one of them sights the player, it informs close-by agents such that they all can start chasing.

The game was rough but allowed us to start recording early the behavior of users. We quickly observed where difficulties lied and what we had to change. The framework contained too many mechanics for the players to grasp in a limited amount of time and this meant the players would spend the majority of the time trying to understand the game rather than playing it and evaluating the generated levels. The looting mechanics and inventory were too complicated and unintuitive. Furthermore, the game was very imbalanced as there had been very little playtesting. To beat a level, users had to reach a particular location, and this meant that some completed every level in little time by speeding through the level without performing any action. Others were unsure of what the goal was so they just wandered around trying out every possible action. The results we collected were thus dependent on what the users understood rather than their playstyle. We also had to normalize the results to ensure uniformity about the data. The rough look of the game was also a factor in deterring some players from trying it out, reducing the amount of data collected.

After these observations, we decided to apport the following modifications to our final game:

- The game should be simplified to allow users to understand all the mechanics in the first two minutes.
- The game should be more clear such that users quickly understand the goal.
- Each level should be playable for a fixed amount of time.
- The inventory should be removed to simplify the game further.
- More playtest is required to ensure a balanced game.
- The game should have a more polished and appealing look.

4.2.2 Online Build

To ensure a smooth testing experience for players across a wide variety of machines, we decided to deploy the game online to remove the download and install steps. This deployment strategy, however, came with limitations. Most of all, the memory limit for the online build. We solved this limitation by using three techniques:

- Dramatic reduction of draw calls and memory usage by using a single 128×128 pixel texture for every mesh and object in the game.
- Reduction of dynamic memory allocation by using *pools*.
- Reduction of memory used by the generative algorithm by implementing a fixed-size stack optimized to work only on chunks of the whole level at a time.

Every mesh is specially unwrapped to fully utilize the color palette encoded as multiple gradients in a single png image, shown in Figure 4.3. The game shader interpolates the quantized gradient to ensure a smooth game look. This method dramatically reduces the amount of mem-

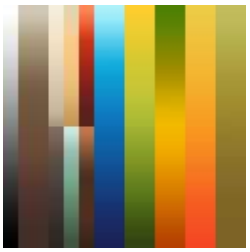


Figure 4.3: *The single* 128×128 *texture used for every model.*

Figure 4.4: *All the props, weapons, characters and items featured in the game.*

ory used by the game and the draw calls for all object prefabs (Figure 4.4).

To avoid expensive memory allocations in the online browser during play, when instantiating bullets, collectibles, and new enemies, we use *pools*. The basic idea is to allocate a large enough chunk of memory at the beginning, to store all the objects needed during execution. When we need to instantiate a new object, instead of allocating a new piece of memory and building a new instance, we utilize an unused object in the pool. The use of pools means we have an upper limit on the number of objects present at a given time on screen, but this isn't a problem in practice as we can reuse the oldest object.

For the WFC, we use a similar method to reduce memory needs when computing the collapse of the different modules. The level is generated in chunks, and when a subsequent chunk is generated, border constraints of the previous chunk are taken into consideration to ensure an overall consistent result.

4.3 Results

We here shortly present the final developed game with all the features and changes discussed above.

4.3.1 Final Game

The final version of the game is a much simpler and polished version of our early prototype, with many features stripped off due to the changes mentioned above. The goal is to score the most points within the allotted time. Users play in short rounds, each lasting two minutes. Upon

Figure 4.5: *All the* 14 *basic tiles used in the game, each with the corresponding set of prefabs to instantiate.*

round completion, the game awards a number of stars to the players proportional to their score. The players can earn points by killing enemies, collecting hidden items and exploring the level. Users can interact with bins to find equipment and collectibles. Enemies are peaceful unless attacked. This mechanic is in place not to force players to kill enemies if they do not want to, and instead focus on the actions they prefer. The player can explore the level to find powerups such as extra health, life or ammunition. A screenshot of the final game in action can be seen in Figure 4.6.

Figure 4.6: *Two screenshots of the final game in which all of the game elements and features are present.*

5

User Study

The goal of this thesis is to show that by catering to user differences and personalizing the content they encounter while playing, their enjoyment of the game increases. Of course, enjoyment or *"fun"* is a subjective aspect, so a study is needed to assess how well our method performs.

We wanted to test our method and game with a large enough sample of subjects to obtain reliable results. Such a sample would allow us to obtain insights into how players behave in our game and which traits are most prominent. The study also allows us to see if there is correlation between what the users report as behavioral traits and what they were performing in the game. Positive correlation, either between our system and an increase in the given ratings or between reported traits and in-game behavior is desirable and would indicate that further, more in-depth research in this emerging field is required.

Execution We executed three studies during the duration of this thesis:

1. The first one was performed after three months of development, to get an early assessment of our method and game.
2. The second one was performed with a small sample of people a week before the final study, to get early feedback and adjust the game, algorithm and study to obtain the best possible results.
3. The third and final study was performed across two weeks of the last month of this thesis. This study is the main one in which we collected the most data with the final version of our game and algorithm.

In the following sections, we discuss what the goals of the different studies were, how we designed them, and the results that we collected and extracted from the data.

5.1 Design

All the studies were designed to be completed as quickly as possible to ensure we would get enough participants. A higher number of participants means a more representative sample of players. They should last between 15 and 20 minutes in total. During this time we should collect both behavioral data as well as gameplay data. Behavioral data inferred from psychological tests allows us to draw connections between player personality and in-game behavior. Gameplay data allows us to analyze what the players do and what are the connections with overall enjoyment. We should also find a suitable way to measure how much a given player is enjoying a session.

5.1.1 Data Collected

As mentioned, we are interested in two types of data: behavioral data the users provide about themselves in a self-evaluation environment, and the data collected automatically by our game during gameplay.

Self-evaluation Data To have a baseline from which we can categorize our players and understand them better, we ask them to evaluate their gameplay preferences directly. We, therefore, ask them to fill out existing questionnaires used to infer the player's preferred playstyle and personality. We used two questionnaires:

- Bartle's Taxonomy questionnaire: used to determine which Bartle traits are more prominent in the user.
- Myers-Briggs personality questionnaire: used to understand how the users perceives the world, how they form decisions and the main interaction modes.

Gameplay Data We also want to record the actions performed by the players during the play session. We should detect and store the main events happening and log them for further analyzing and filtering. Understanding how players behave in-game allows us to infer which game features are more suited for a given player. Our system automatically logs all relevant events.

Data Filtering To ensure consistent data, we decided to filter it based on the following parameters:

- The first two rounds of every player are culled. Players had to get used to the game and were most likely still learning the basics in the first few rounds. The focus during these initial levels is on learning instead of evaluating how much they enjoy a particular level. Furthermore, it is harder to evaluate a level without any baseline reference. Starting from round 3, players have experienced the game a little, and it is easier for them to evaluate levels compared to previous ones.
- Levels with 0 stars. Since we didn't force participants to rate every level, if they ignored

the rating or forgot to do it, the default one is 0 stars. We cull these results, but keep all other, from half star up to 5.

Considerations We expected a very high variance in subjective ratings of enjoyment across levels, which in some context could be a problem since users that tend to evaluate everything more positively or negatively could skew the overall results. However, we aren't primarily interested in the rating of the game *per se*, but in the rating of the algorithm personalizing levels. We decided to provide multiple rounds such that what the *relative* difference in scores between rounds, and not their *absolute* values would be more prominent. Therefore, no matter if a participant tends to rate every level more highly or negatively, the relative scores between rounds still provide insightful information.

5.2 Execution

Recruitment We recruited participants via an email campaign and social media posts (twitter, discord, Facebook) and asked to take part in the study. Most of them were either affiliated with the GTC, the Computer Graphics Laboratory (CGL), ETH or friends, family and gamers recruited online.

Upon clicking the provided link, they were redirected to an online *Google Form* in which the purpose of the study was explained, without going into detail to avoid bias. We gave the participants an anonymity disclaimer that also explained how we would record and use their data. After accepting, they were asked to provide an anonymous username formed by combining letters from names of family members, to ensure full anonymity. The alias is formed by the first two letters of the mother's name merged with the last two letters of the grandfather's name. The questionnaire also asked how frequently they played videogames, to ensure that the sample of the population questioned was representative of a typical gaming population.

Questionnaire We redirected participants to an online questionnaire to categorize them according to Bartle's taxonomy [2]. The test contained 30 questions with two options each to choose from. Each choice was polarizing the subjects into one of the four categories described by Bartle. Upon completion, the participants obtain four percentual scores, each representing the degree of membership to each of the four categories (Killer, Achiever, Explorer, Socializer). The sum is normalized to 200%, meaning that a score of 50% in any category is considered neutral. Each category can range from 0% to 100%. Each player gets assigned the category with the highest percentage. They were asked to report their scores to continue.

Gameplay Afterward, users were redirected to a version of our game playable directly in the browser. This way, users could complete the study quickly, without having to download anything, that would slow down the process. The percentage of users completing the study would also increase since installing software could deter some users that would not finish the study. Online deployment allows us to avoid most machine-specific technical issues.

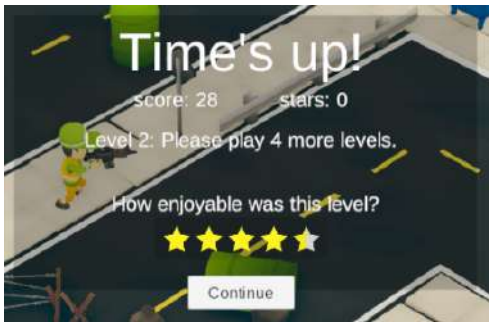


Figure 5.1: *The interface presented to the player at the end of every round to evaluate the level.*

Users were asked to play 6 rounds of the demo. Each round lasts exactly 2 minutes, to ensure that the data collected is uniform for all players and doesn't need to be normalized to take into account different playing times.

At the end of every level, users are given a score based on their performance and are asked to rate the level of enjoyment of the level just played on a scale 1 to 5 stars, with half-star increments. Figure 5.1

We immediately ask each user to evaluate the level to ensure that the memory of the level just played is fresh and is not mixed with future runs of the game. Upon every round completion, the data of the user, together with the level data and score are automatically uploaded to our server via an online database. This way, if players are not able to play all 6 levels, or stop early, we still get data for every single round completed.

5.2.1 Technical Details

To be able to record data from user playthroughs, we developed a simple general-purpose online logging system. With this system, it is possible to embed function calls in the middle of relevant gameplay code to automatically record an event, package it with additional metadata and send it to an online server that logs it into either a database or a simple text file. The function LOGONLINE(STRING, PARAM) is given a representative string for the event type, and any number of parameters of any base type (bool, int, float, string). The data is packaged together with the username, the timestamp and the frame count since the game started. The data is then sent to the online server via an HTTP POST request, in which all data is stored as a key-value pair in the request body. On the server side, a PHP script gets the request, decodes it, and then either stores each key in the appropriate database column, or generates a CSV line to be added to a simple TXT file for storage.

With this method, we could conveniently record any relevant event such as kills, collections, actions, and exploration progress The system also allows us to change which events are logged and add new ones without changing the code, further preventing code bloat.

At the end of every round, we package the most important data and send it to a collective database. The data format for each entry is the following:

(TIMESTAMP, USERNAME, LEVEL, DURATION, SCORE, PERSONALIZATION,

KILLS, COLLECTED, EXPLORATION, SOCIALIZED, RATING)

We send information about when the round was played and by whom, and round information, including duration in seconds and the score. We also send if the algorithm for personalization was on or off. We then send a summary of the actions performed by the user, including the number of kills, collected objects, and the percentage of the map explored. Since our game doesn't feature any socialization mechanics, the default socialized value is 0 for each entry. We then report the rating given by the user at the end, on a scale of $0 - 10$ half stars.

The code uses C# *reflection* to serialize every field of any given object automatically. The system can analyze source-code variable names and use them to build the serialized object. Therefore, if we want to log more parameters, we merely have to declare a new variable in our class definition. The system then takes care of automatically updating its value, reading it, serializing it, and adding it to the HTTP request body. The server-side automatically detects a new data format and adds an appropriate column for it.

This system saved us invaluable time while trying to identify which parameters were best suited to be logged in our game.

Each user is given a database or file since we log thousands of events. The final summary of each round, stored as a single entry per round, is instead stored in a shared database.

5.3 Results

In this section, we discuss the results obtained from the different studies.

We recall that the goal of the study is to show that, in the context of our game, personalization of content leads to higher enjoyment of the overall experience. We further explore other questions with the data collected, such as different correlation measures between measured parameters and insights that these provide. We start with a brief refresher of the statistical methods we use.

We then go over the initial study, the insights it gave us and the modifications we decided to apport to the game, the algorithm and future iterations of the study.

Afterwards, we go into detail with the results of the second and third studies, analyzing the data with different statistical methods, and provide the assumptions, interpretation of results and conclusions.

5.3.1 Overview of Statistical Methods

In our data analysis, we utilized well known statistical methods to prove or disprove our hypotheses. For the non-technical reader, or for the reader that wants a brief and quick refresher, we here give a short introduction to each method. We state what each method is used for, for which type of data it is best suited, the underlying assumptions, and typical results. We mainly

state what the method computes and what is the idea behind it. We do not go into detail with formulas and refer the reader to specific resources [11] to learn more. Feel free to skip this section if you're already familiar with these statistical methods.

Structure In most of our analysis, we follow this structure:

1. We start by defining the hypotheses: the *null hypothesis* H_0 and the *alternative hypothesis* H_A.
2. When appropriate, we state the α value if different from the standard value of 0.05.
3. We state the decision rule (one or two-tailed).
4. We provide the computed value of the test statistic.
5. We compare the value to decide if it is significant and if we should accept or reject our hypothesis.
6. We interpret and explain the results.

Median, Average, Variance, Standard Deviation For every set of data recorded, we provide the median, the average, the variance, and the standard deviation. These indicate where the $50th$ percentile of the data resides, the mean of the population and how much a sample tends to variate from the average.

P-value For most statistical tests, the *p-value* tells us the probability of obtaining a particular result. Small values indicate a low probability of obtaining the result by chance and is therefore desirable. We often compare this value to our α cutoff value to determine if the result is significant and should, therefore, be taken into consideration or not.

One vs. two-Tailed For certain tests, we can specify if the result should be *one-tailed* or *two-tailed*. One-tailed results are best suited when we want to show that the difference between the two variables is directional (e.g., a positive increase). Two-tailed results are instead well suited for when we want to show that the variables differ in any direction (either positively or negatively).

T-Test Computing the difference of means between two sets of data only tells us so much. The T-test compares two sets of data to determine if there is a significant difference in their means, while also taking into consideration how much the data varies. It is a ratio of signal to noise, where the signal is the difference in means (a high difference implies much signal), whereas considerable variance in the data means much noise. A high ratio, greater than 1, indicates that there is more signal than noise and that the difference in means is more prominent, whereas a low ratio indicates a lot of noise. The extended amount of noise could be causing the difference in means rather than underlying differences in the distribution. We usually compare the t value to a critical value to determine if we should accept or reject our null hypothesis.

The assumption underlying the T-test is that the data is normally distributed, with similar variance, and a similar number of points per set. It is also well suited for when we are unaware of the standard deviation of the population.

ANOVA The Analysis of Variance, as the name suggests, is used to look at the variance and estimate what could be causing it. If we have different test conditions (treatments), we can analyze the variance *within* the groups, as well as *between* groups. Large variance within groups indicates underlying differences in the population, whereas large variance between groups indicates differences in the conditions. The F value is the ratio among 'between' and 'within' variance. A high value indicates the variance comes mostly from different groups and therefore we reject the null hypothesis that states there is no significant variance in the means between conditions. The p value is the probability that a particular F value happens. ANOVA can be used with an arbitrarily large amount of variables, comparing all the variances across them. The t-test is a particular case of ANOVA.

Covariance and Correlation The covariance measures how much two random variables vary together. A high value means that as one variable increases, we can expect the other to do as well, and vice-versa. A substantial negative value indicates an inversely proportional relationship between the variables. A value close to 0 indicates that the realization of one variable doesn't help us predicting the realization of the other. We usually normalize the covariance to obtain the correlation, by dividing it with the root of the variances. The correlation between variables is a value in the range $[-1, 1]$, where absolute values close to 1 indicate a strong correlation, whereas values close to 0 indicate non-relatedness of the data.

It is possible to compute a correlation matrix, where each series of data is cross-correlated with each other series, giving us an upper triangular matrix with variances on the diagonal.

It is important to note that a significant correlation does not mean causation.

Independent vs. Dependent Variables We distinguish between *independent* and *dependent* variables. Independent variables are changed or controlled in our experiments, whereas dependent variables are those tested and measured. In particular, independent variables are those we set ourselves in our game and system, whereas player behavior is usually dependent, measured variables.

5.3.2 First Study

We had ten participants in the first iteration of our study. We asked them to fill out the questionnaires, and we recorded they playthrough in the early game prototype. The personalization algorithm wasn't running yet, and we just tested the gameplay, as discussed in Chapter 4. We could, however, analyze how the recorded metrics compared to the psychological models. Table 5.1 reports the correlation results obtained in this first study, and Figure 5.2 shows their plots.

Table 5.1: *The Pearson correlation coefficients obtained in the first study, as well as the corresponding p-value. No result is significant, so we cannot infer anything from the data.*

	Pearson correlation	p-value
killer	0.2919	0.1568
achiever	0.1997	0.3385
explorer	0.2663	0.4402

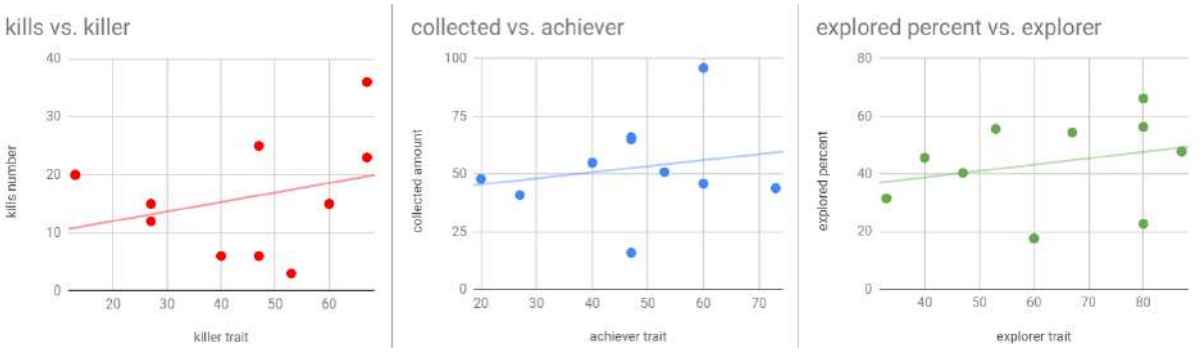


Figure 5.2: *The results obtained in the first study. Each chart plots the degree of membership to a given category against the number of corresponding actions in the game. For example, for each user, their Killer percentage trait is plotted against the number of kills. All three plots show a very weak correlation, but due to the small number of data collected, none of the results is significant.*

	user	level	score	wfc	kills	collecte	explore	rating
user		-0.03	-0.07	0.01	-0.08	0.14	-0.04	-0.11
level			0.56	0.06	0.57	-0.22	-0.16	0.29
score				0.25	1.00	-0.30	-0.05	0.52
wfc					0.26	-0.32	-0.09	0.42
kills						-0.37	-0.11	0.49
collected							0.60	0.10
explored								0.25
rating								

Figure 5.3: *Preliminary results for the small sample of data collected. Green values indicate a strong positive correlation, red results strong negative correlation, white represent non-statistically-significant correlation.*

5.3.3 Second Study

Before running the full user study, we asked a small sample of people to try out the final version of the game and questionnaire, to ensure everything was working as intended. Cross-correlating all the data collected showed some interesting results, although with small sample size. These results are reported in Figure 5.3.

5.3.4 Final Study

In the final study, we had a total of 96 participants, who took part in our survey and gameplay. They played a combined 570 rounds of our game.

Filtering and Processing We had to remove users that either didn't complete the study or provided incorrect or incomplete information. We removed all unnamed users, users who didn't participate in both questionnaire and gameplay, users whose alias didn't match in the questionnaire and gameplay, users who played less than 4 rounds. We also removed users who participated more than once. For questionnaire-only statistics, we still kept users who didn't have the required amount of rounds in the gameplay section.

Users whose alias didn't perfectly match (maximum of 1 character mismatch) but whose results had very similar timestamps (within 30 minutes of each other) were manually linked. Aliases were trimmed to 4 characters and normalized only to allow lowercase alphabetic characters. Users were assigned a random id to ensure complete anonymity and avoid any unwanted correlation.

We removed users whose reported questionnaire results didn't sum up to 200% if the difference was greater than 2%, and normalized their result if they were off by less than that.

After the filtering process, we had a total of 71 participants and 464 rounds for our analysis. Figure 5.4 shows the users categorized per dominant Bartle trait, and Figure 5.5 shows the distribution of trait percentages.

From Figure 5.4, we can already see some results worth mentioning: an overwhelming majority of the players sampled resulted explorers, with a percentage of 66.7%. One of three facts could explain this observation: the test is biased and unable to capture the dominant trait, the sample

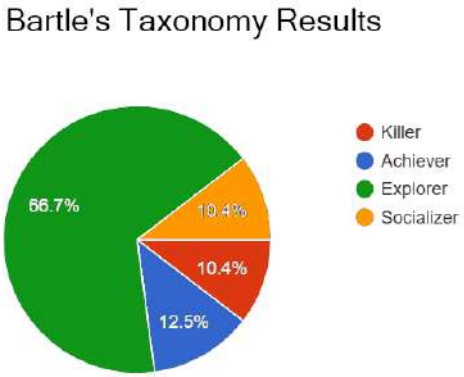


Figure 5.4: *Results for the percentages of measured users in each category.*

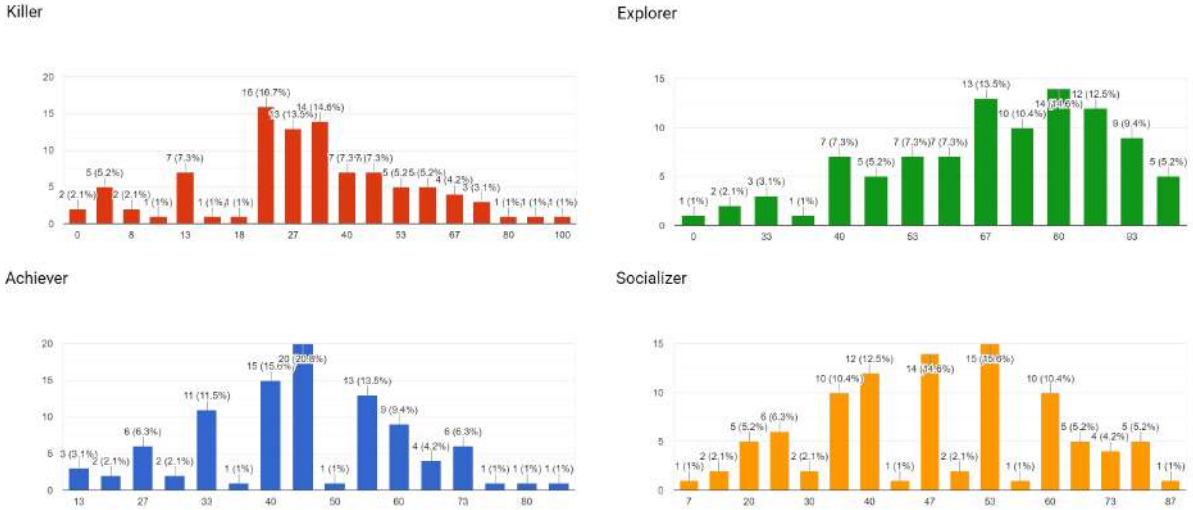


Figure 5.5: *Results for the distributions across the four measured categories in the questionnaire.*

of players polled differs from an average population, or the traits aren't uniformly distributed.

5.3.5 Rating Analysis

We now present the results of the final user study.

Probability Fit We first analyze if there is a significant difference between the ratings of personalized and not personalized levels.

We want to show that there is a positive, significant increase between the two ratings, and thus that the observed data come from two distributions with different mean. We set H_0 as the hypothesis of *no* significant positive increase between the means, and H_A as the alternative of positive increase.

During the execution of the test, we had an independent control variable that was either set *on* or *off* with a 50% probability each. This variable controls whether we use tuned coefficients to

distribution	μ	σ
off	6.1441	2.3954
on	6.7554	2.2422

Table 5.2: *Mean μ and standard deviation σ for fitted normal distributions of positive and negative treatment ratings.*

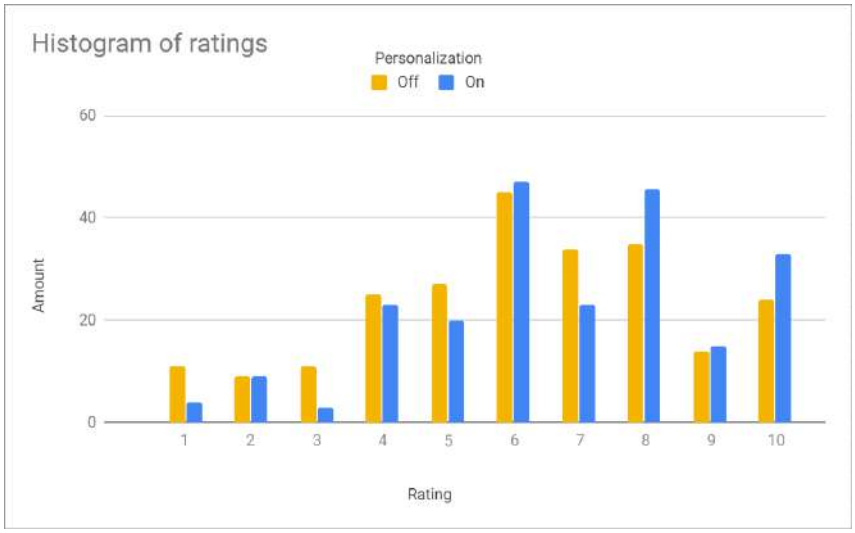


Figure 5.6: *A histogram showcasing the ratings given. In yellow, ratings for when the level is generated randomly. In blue, ratings for when the personalization is on.*

generate our level, or if we do not.

To avoid bias between random parameters and those that are adjusted but not personalized, we take the unpersonalized samples from a pool of samples other users saw. We want to ensure we measure how much users prefer levels that are tuned to themselves rather than levels tuned to the general population sample.

For each sample, we get the dependent variable, a rating provided by the user at the end of the level. We split the ratings into two treatments, one for adaptation on and one off.

We can analyze the distributions and see that empirically they fit very well into a normal distribution. We can imagine subjects having an underlying preference, and deviating from that preference depending on the particular level. By fitting a normal distribution to ratings of both treatments, we can observe that they have a very similar mean and deviation. We can further observe that the mean for the positive treatment is slightly higher, of about 0.6113. The positive treatment also has a slightly lower variance and deviation. We can expect this result since for personalized level we try to discover the underlying preference, whereas for non-personalized we pick a level uniformly at random, increasing the variance in ratings. Table 5.2 shows the value for the fitted normal distributions. Figure 5.6 shows the histogram of ratings for the two treatments.

Source	SS	df	MS	F	p
Groups	39.15	1	39.1492	7.33	0.007
Error	2468.15	462	5.3423		
Total	2507.3	463			

Table 5.3: *Numeric values for the one way ANOVA test on ratings separated into adapted and non-adapted treatments.*

T-Test We then run a one-tailed two-sample unequal variance T-test. We want to show a *positive* increase in enjoyment and not an arbitrary difference in scores. Furthermore, by setting the independent variable randomly, a single user might be provided with a different amount of personalized and non-personalized levels. A consequence of this assignment is that the total number of ratings for each treatment is most likely different.

One assumption that has to be satisfied is that the data must follow a normal distribution. We showed this fact empirically.

The T-test p-value is 0.0078, which is smaller than our $\alpha = 0.05$, meaning that we reject our null hypothesis H_0 and accept the alternate hypothesis H_A.
We, therefore, conclude that there is a significant positive difference in the means of the underlying distributions. We have thus proven the central claim of this thesis.

ANOVA We can perform ANOVA (Figure 5.7) on the data to obtain the same conclusion, however with more detailed intermediate results. We can see in Table 5.3 the results for the ANOVA test. SS is the sum of squares, df are the degrees of freedom. MS is the mean squared error. The F value is the ratio between the mean squared errors. Finally, the p value is the probability that the test statistic would be greater than the computed statistic. A small p-value indicates that the mean differences are significant, that holds in our case. The conclusion is the same as with the t-test, specifically that our treatment increases enjoyment.

Mann-Whitney U test Both the ANOVA and the T-test work assuming continuous, normally distributed data. We decided to test our results with *Mann-Whitney's U test*, that doesn't make those assumptions, to ensure correctness. The only assumption this test makes is that the samples should be independent. We fulfill this requirement since the independent variable (treatment on or off) is chosen $\overset{iid}{\sim}$ Ber(0.5), and the dependent rating is only dependent on the current level played.

The results we get for this test are $p = 0.0058$. We can, therefore, reject H_0, the hypothesis that the samples come from the same underlying distribution. We have to remember that this test is two-tailed, meaning we test for a difference, and not a positive difference, hence the smaller p-value.

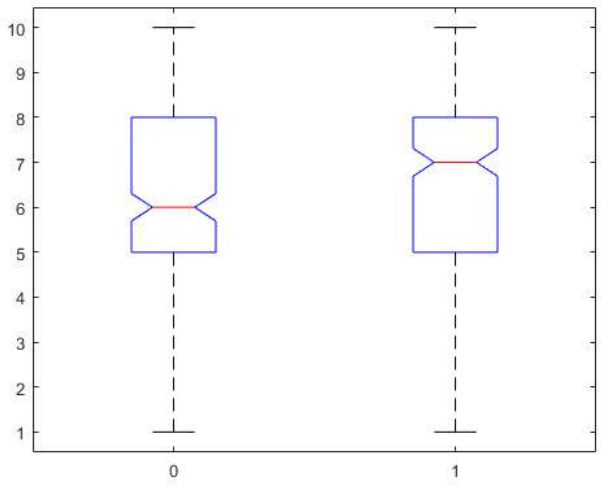


Figure 5.7: *Plots for the two treatments, ratings when the adaptation is on and off respectively. The plot shows that the ratings in both treatments ranged from* 1 *to* 10*, and that the middle* 50^{th} *percentile is comprised of scores between* 5 *and* 8 *in both treatments. The median lies to* 6 *for the uncustomized treatment and* 7 *for the customized one. The dents in the graph indicate the confidence interval.*

Correlation Matrices

We further analyze the collected data to see which insights it gives us on our method and possible future work. We cross-correlate every data column collected that agrees on dimensionality. We use *Kendall's tau* correlation coefficient. Significant values for which the associated p-value is smaller than our 0.05 threshold are marked red. By cross-correlating so much data, we expect some of the results we discuss to be *false positives*. The choice of our α-value considers this implicitly.

The first matrix, shown in Figure 5.8, shows the correlated results for each group of data collected. We consider a single two minutes round as a data point. We collected 464 datapoints for this table.

We now analyze the significant values in our table:

- USER ID: we can see that for the user id row, we do not have any significant correlation. We can expect this since we randomly generate and assign user ids.
- LEVEL AND SCORE: we can see a positive correlation between level and score. A possible explanation is that players familiarize themselves with the gameplay mechanics in the first few rounds, and after that they are more efficient. We noticed that however, users could not increase their score dramatically. This fact suggests that the learning curve isn't too steep and that there aren't too many ways to improve in the game.
- LEVEL AND ACTIONS: we can see that as the game progresses, users tend to focus less on killing and more on collecting and exploring (negative correlation for the former, and positive for the latter two). Users learn a dominant strategy and discover that they can achieve a higher score by focusing more on collecting (and therefore also exploring the

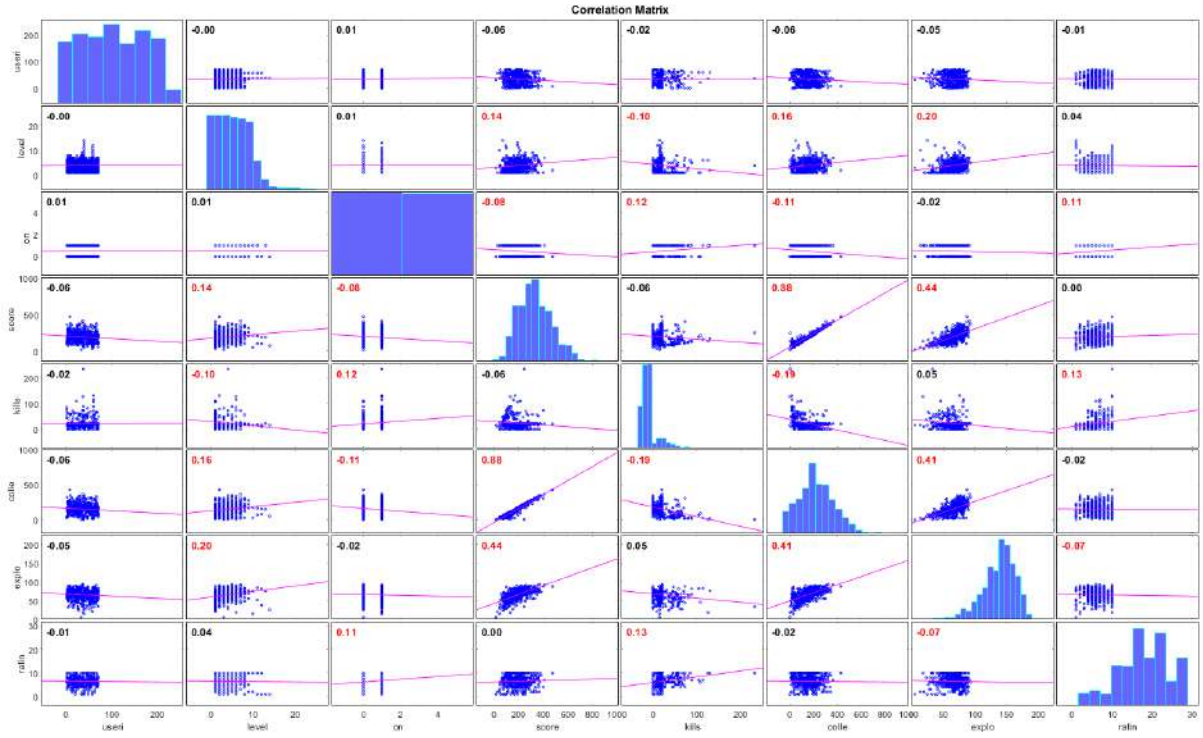


Figure 5.8: *Kendall's cross-correlation of gameplay data*

level). In our early tests, we had the opposite result. Users learned that killing yielded the most points. We slightly tweaked the mechanics to incentivize exploration and collection after that, which is a possible cause for these results. A balanced game in which there is no dominant strategy but multiple equally valid ways to achieve the score would be optimal.

- PERSONALIZATION: since in this row the values for personalization are logical and not numerical we ignore the results. We run categorical tests for this data.
- SCORE: we can notice a very high correlation between score and number of collected items, and between score and exploration. A possible reason is that users discovered this as being the easiest way to earn points and thus focused on this strategy. Exploration doesn't yield as many points as collecting, but it is necessary to find loot.
- KILLS: we can see a negative correlation between kills and collected amount. This correlation indicates that when users focus on killing or collecting, they have less time to focus on the other. We can expect these results since the round time is limited.
- KILLS AND RATING: a positive correlation between kills and rating indicates that users tend to rate the level higher when they spent more time defeating enemies. This suggests is that this type of action contributes to the overall enjoyment, by considering our sample in aggregate.
- EXPLORATION AND RATING: a slightly negative correlation between the percentage of level explored and rating indicates that users seem to enjoy the experience less when they explore more. Although the levels are procedurally generated, there is a finite amount of

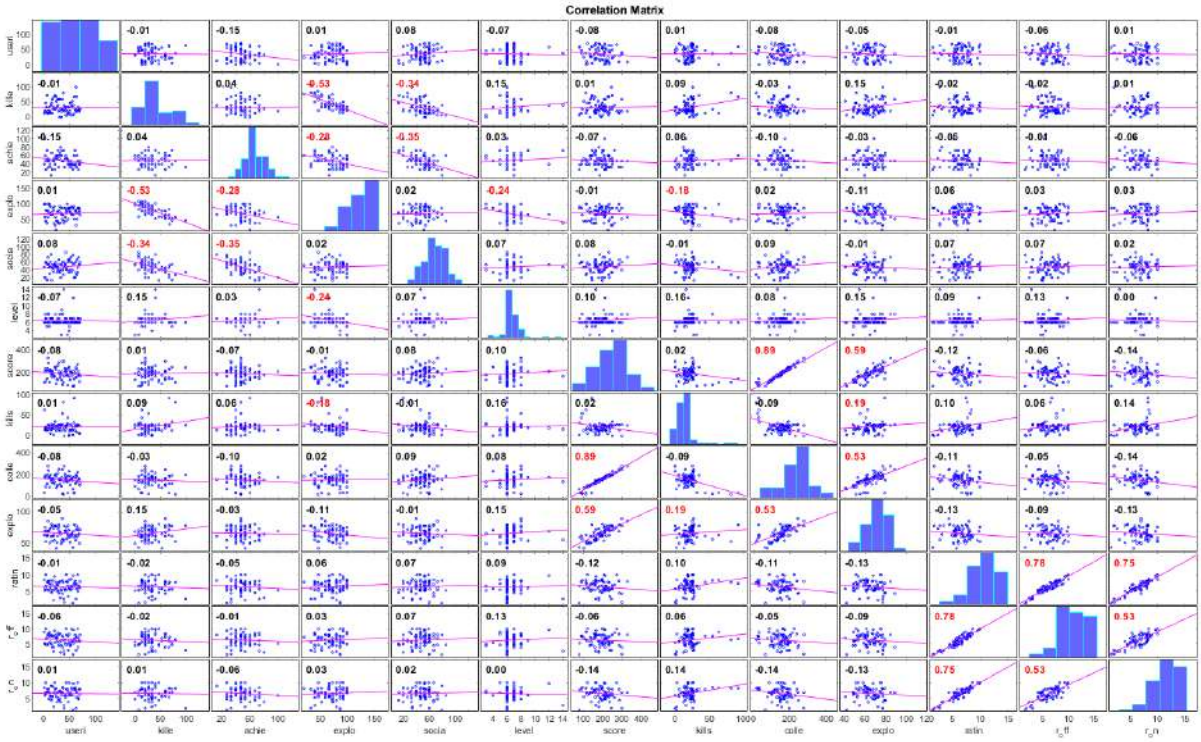


Figure 5.9: *Kendall's cross correlation of user data*

basic blocks, which might be the reason why.

Shortcomings

We here suggest things that we could have improved in our study design. We feel that, from the study design point, it would have been preferable to only focus on one single independent and one dependent variable, to show results between the two. We should have only tested enjoyment against the two personalization treatments. The recruitment process should also be kept in mind. By reaching out to different population samples, such as academics and online gamers, we could be getting different amounts of effort in the reporting of results. Considering the overall population of gamers, this could be seen as a strength, instead of limiting ourselves to an academic field only.

6

Conclusion

In this chapter, we provide a short reflection on the overall thesis and results, suggest some further possible applications and future work and provide an outlook on what could come next.

6.1 Reflection

During the development of this thesis, we discovered an approach on how to personalize the generation of levels based on the exhibited player-behavior. We achieved this by combining established psychological models with innovative general-purpose generative algorithms. We developed a way to bridge between the two dynamically. We found a significant correlation between the adaptation and positive player enjoyment reported via rating. This result confirms our hypothesis that, by taking into consideration different player types during development and providing different game scenarios, we can improve the overall experience. This result holds in our specific case, and we are confident that, thanks to the generality of the models and algorithm, this result applies to other cases as well.

During the research phase of the thesis, we discovered insightful models to express behavior and map the gameplay mechanics to different player tastes. We learned compelling similarities between multiple player models, which suggests an underlying latent unified model. We also analyzed and learned from different frameworks that look at play and the needs they fulfill. We were also able to create a general taxonomy of game mechanics mapped to user traits and preferences.

The results confirmed our initial hypothesis, and we hope that this result motivates further research in the area of personalized and adaptive games.

On the improvement side, we believe that using our method on existing games, instead of developing our own, could have provided several advantages. These include more time available

to fine tune the algorithm parameters, and a bigger data set to leverage in the analysis phase. These changes would have saved us several iterations spent on finding the most suitable game mechanics and fine-tuning them. On the other hand, developing our own game allowed finer control of the data collected and the mechanics implemented to suit the study better.

We could have balanced better the game we developed. As the game design plays such a significant role in the enjoyment of the players, a well thought and structured game has a tremendous impact on the scores given. We saw early on that the game was incentivizing one specific aspect of gameplay, and even after further balancing, there still was a strategy preferred by most players. This fact skewed our results in favor of this playstyle and reduced the variability in the generated levels.

6.2 Future Work

The future work regarding this project can take different directions. Using more complex underlying player models could yield better results. It is also possible to take into consideration the temporary emotional state of players, such as joy, anger, and excitement, to better estimate the latent model independently of small behavior variations due to these factors. We could apport changes to the amount and type of metrics collected, trying to find the metrics that describe the model most accurately. We could develop genre-dependent models, to take into consideration different populations of gamers that tend to be attracted to different types of games, and therefore influence the results of the algorithm. It is also possible to explore other applications of these insights. In this thesis, we focused on level generation, but future work could try to adapt the story, the items, the enemies or any other element commonly found in video games. Comparisons between the results for each of these categories could help steer the research towards the area that has the most effect on player enjoyment.

The game used in this thesis could be extended to provide more replayability and depth. Having users play for more extended sessions could yield different results, more accurate due to the higher availability of data. The use of our method on existing games, to adapt and modulate existing features could also yield impressive results. Using an established game would have the advantage of a more significant user base and therefore more data to analyze and leverage.

Further research questions could address any of the points mentioned above, as well as the practical applicability of these research models into commercial titles to reach a wide audience.

6.3 Outlook

This thesis contributes to the field of procedural content generation algorithms that adapt to the user. We hope that in the future, similar methods will be more widespread than they are now. We further hope that this thesis serves as a starting point for further research in the field. By looking at the trend in technology-related areas, we can see that personalization of content is a big topic with high interest from both users and developers. We believe that in the future we will see more games that embrace this concept to improve the overall player experience.

Bibliography

[1] NERIS Analytics. 16 personalities. https://www.16personalities.com/.

[2] Richard Bartle. http://mud.co.uk/richard/hcds.htm.

[3] Michael Booth. The ai systems of left 4 dead.

[4] Roger Caillois. *Man, Play, and Games*. University of Illinois Press, 2001.

[5] Alessandro Canossa. Play-persona: Modeling player behaviour in computer games. 06 2009.

[6] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. 01 1990.

[7] Ron Edwards. Gamism: Step on up. http://www.indie-rpgs.com/articles/21/, 2003.

[8] ExUtumno. Wave function collapse github repository.

[9] Stephan Mueller Barbara Solenthaler Robert W. Sumner Markus Gross. Heapcraft: interactive data exploration and visualization tools for understanding and influencing player behavior in minecraft. In *MIG*, 2015.

[10] Stephan Mueller Barbara Solenthaler Robert W. Sumner Markus Gross. Heapcraft: Quantifying and predicting collaboration in minecraft. In *Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'15, pages 156–162, 2015.

[11] Gareth James and Witten. *An Introduction to Statistical Learning*. Springer Texts in Statistics. Springer, 2013.

[12] Isaac Karth and Adam M. Smith. Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 68:1–68:10, New York, NY, USA, 2017. ACM.

[13] Isaac Karth and Adam M. Smith. Addressing the fundamental tension of PCGML with discriminative learning. *CoRR*, abs/1809.04432, 2018.

[14] David Keirsey. The four keirsey temperaments. https://www.keirsey.com/, 1970.

[15] Sylvain Lefebvre. Voxmodsynth. https://github.com/sylefeb/VoxModSynth, July 2017.

[16] Thomas W. Malone. Toward a theory of intrinsically motivating instruction. *Cognitive Science*, 5(4):333 – 369, 1981.

[17] Albyn C. Jones Mark Glickman. Rating the chess rating system. http://www.glicko.net/research/chance.pdf.

[18] P. Merrell and D. Manocha. Model synthesis: A general procedural modeling algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 17(6):715–728, June 2011.

[19] Paul Merrell. *Model Synthesis*. PhD thesis, University of North Carolina at Chapel Hill, 2009.

[20] Paul Merrell and Dinesh Manocha. Continuous model synthesis. *ACM Trans. Graph.*, 27(5):158:1–158:7, December 2008.

[21] Joseph C. Osborn, Adam Summerville, and Michael Mateas. Automatic mapping of NES games with mappy. *CoRR*, abs/1707.03908, 2017.

[22] Marc Ponsen Pieter Spronck. Adaptive game ai with dynamic scripting. *Institute for Knowledge and Agent Technology*, 2005.

[23] Hugo Scurti and Clark Verbrugge. Generating paths with wfc. In *AIIDE*, 2018.

[24] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards automatic personalized content generation for platform games. 12 2010.

[25] Dean Smith. Emergent mechanic design for video games with procedural content, 2016.

[26] Oskar Stalberg. Epc2018 - wave function collapse in bad north. https://www.youtube.com/watch?v=0bcZb-SsnrA, 2018.

[27] Bart Stewart. Personality and play styles: A unified model, 2011.

[28] J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 252–259, April 2007.

[29] Yordan Zaykov Tom Minka. Trueskill ranking system. https://www.microsoft.com/en-us/research/project/trueskill-ranking-system/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Ftrueskill%2F, 2005.

[30] Fandom L4D Wiki. Left 4 dead wiki: The director. https://left4dead.fandom.com/wiki/The_Director, 2010.

[31] Wikipedia. Octahedral symmetry.

[32] G. N. Yannakakis and J. Hallam. Real-time game adaptation for optimizing player satisfaction. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(2):121–133, June 2009.

[33] G. N. Yannakakis and J. Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2(3):147–161, July 2011.

[34] Jacqueline Zenn. Understanding your audience – bartle player taxonomy. `https://gameanalytics.com/blog/understanding-your-audience-bartle-player-taxonomy.html`, November 2017.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

EMERGENT PERSONALIZED CONTENT IN VIDEO GAMES

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
GUGGIARI	SIMONE

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, 15/04/19	14-901-660 SIMONE

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.