HOCHSCHULE
ANSBACH

Fakultät Medien

Making Procedurally Generated Levels in Top-Down Games Seem More Handcrafted

Bachelorarbeit im Studiengang Visualisierung und Interaktion in digitalen Medien

vorgelegt von

Danny Karim Sehili

Matrikelnummer 00154863

Erstgutachter: Prof. Florian Machill

Zweitgutachter: Prof. Dr.-Ing. Helmut Roderus

Declaration

„I confirm that I have written this thesis unaided and without using sources other than those listed and that this thesis has never been submitted to another examination authority and accepted as part of an examination achievement, neither in this form nor in a similar form. All content that was taken from a third party either verbatim or in substance has been acknowledged as such."

Ort, Datum _____ Unterschrift _____

Abstract

This bachelor thesis investigates methods to enhance the handcrafted quality of procedurally generated levels in top-down games, focusing on dungeons. The study begins with an overview of procedural generation in general. It follows with an analysis of handcrafted levels, identifying key elements that contribute to their quality. Procedural generation methods are then explored to find techniques that can replicate these characteristics. Based on the findings, a mixed-initiative level generator is developed, incorporating the identified elements through utilizing the procedural generation methods. The generator's output is evaluated for its similarity to handcrafted levels and the presence of handcrafted characteristics. The results demonstrate that the proposed approach successfully enhances the handcrafted quality of procedurally generated top-down levels, contributing to more engaging gameplay experiences.

Contents

List of Figures

Chapter 1

Introduction

In the world of video games, it is hard not to come across the term "procedural content generation" (PCG). One of the first uses of PCG was in 1980 in the game *Rogue* (Epyx, 1980) (Hendrikx et al., 2013, p. 14), where a dungeon had been randomly generated (Green, 2016, p. 16). Green (2016) explains that the concept of procedural generation itself does not necessarily involve randomness, but the use of randomness is usually expected to vary the generated results (p. 2). The concept of procedural generation is not only used in a variety of contexts, but also describes a lot of different processes (Shaker et al., 2016). The term describes something that has been created by one or multiple algorithms, without having been there before (Green, 2016, p. 1). Including terrain, characters, architecture, stories, objects like weapons (Togelius et al., 2010) and even textures, sounds and animations (Green, 2016, pp. 12-13), almost everything seems possible to be procedurally generated. Taking procedural content generation to an extreme, you could even create a generator that creates an entire game, with procedurally generated mechanics and rules (Kate Compton, 2017). Recent research in the field of PCG also does not only utilize machine learning (Gutierrez and Schrum, 2020), but also trains them to cooperate with human designers (Zhou and Guzdial, 2021) to be even more efficient in generating content.

With PCG being heavily reliant randomness however, it is conclusive that the level design itself might lack intention, concept, and even meaning in general. Short and Adams (2017) state that a game having this exact issue is *No Man's Sky* (Hello Games, Sony Interactive Entertainment, 2016) (p. 8), which creates entire universes with planets and even the living beings inhabiting them. However, this strong reliance on procedural content generation resulted in a lack of gameplay (Short and Adams, 2017, p. 8).

To find a solution to this problem, the aim of this research is to find a way to create a procedural level generator that generates levels that appear as if they had been made by hand or could even be mistaken as such, resulting in levels possessing the qualities of good level design, which will be done in the context of games in the top-down view. To achieve this, ways to control the randomness of the generation process, which Green (2016) claim is the key to procedural generation that is based on randomness (p. 10), will be researched. In the field of PCG, much research has been done focusing on

improving different qualities of generated levels (Dormans, 2010; Short and Adams, 2017; Totten, 2017), but not much research has been done about generating levels with the goal of them being mistaken for a handcrafted one. This thesis will aim to combine the existing research to achieve that goal.

1.1 Top-Down Games

The top-down view is a camera orientation that shows players the game world from above their avatar (Totten, 2014, p. 151). Totten (2014) describe it as "looking at a building plan" (Totten, 2014, p. 151). This separates the view of the player and their character, different from other perspectives, like the first-person view or third-person view. Totten (2014) mention a main difference that results from this separation, which is the ability of games using the top-down view to show the player things their avatar cannot see, for example if a wall is blocking the view (p. 152). Totten add that the top-down view can effectively produce gameplay that, first, is based on players needing to adjust their position to line up their attacks with their target (p. 152) and, second, is less demanding of the player's ability to react fast (p. 152).
The top-down has been around for quite a long time and has already been used in famous games like *Pong* (Atari, 1972), which came out in 1972. It has been popular for many RPG games like *Earth Bound* (Nintendo, 1994) or *The Legend of Zelda*. The top-down perspective is also the main domain of games of the roguelike genre, a genre where the player explores a randomly generated dungeon, starting inside a new dungeon every time they die (Epyx, 1980; Devolver Digital, 2016; Edmund McMillen, 2011), thus tieing the top-down view to the dungeon level-type. The reason why most roguelike games are displayed in top-down view is that the game *Rogue* (Epyx, 1980), which defined the entire roguelike genre (Green, 2016, p. 16), was also a game in top-down view.

1.2 Procedural Content Generation in Top Down Games

Rogue (Epyx, 1980) not only inspired a genre around navigating the player through a dungeon in top down view, but also the use of procedurally generating the dungeons in this genre. Since then, numerous roguelike games, like *Diablo* (Blizzard Entertainment, 1996) or *The Binding of Isaac* (Edmund McMillen, 2011) have been made (Green, 2016, p. 16), closely following Rogue's style of sending the player into a randomly generated dungeon displayed in a top-down view. Due to this, procedural level generation seems to not only be tightly connected to roguelike games in general but also to the top-down dungeon space.

1.3 Thesis Structure

To achieve the goal of creating seemingly handcrafted top-down levels, in chapter 2, it will first be made research about the field of procedural content generation in general, to become familiar with the topic. This will cover properties of procedural generators, common methods to generate content and the strengths and weaknesses of using procedural generation.
In chapter 3, it will be defined what makes a level handcrafted or generated, first covering the structure that is found in handcrafted dungeons. Additionally, it will be taken a look at level design patterns and techniques commonly used in professional level design of handcrafted games, presenting ones that would be useful to generate seemingly handcrafted levels.
In chapter 4, it will be taken a look at procedural generation methods and games that allow enough control to implemented the findings of chapter 3, or are already implementing them in their generation.
Chapter 5 will cover the generator that has been conceptualized and created, resulting from the combination of the research done in chapter 3 and chapter 4, presenting the concept of the generator, the methods that have been used and the level design patterns that have been implemented in the generation process and the conceptualized top-down game the generator is based on.
Chapter 6 will present an example level that has been generated with the level generator, subsequently covering the evaluation of that level through moderated user tests and discussing its results.

Chapter 2

An Overview of Procedural Content Generation in Video Games

2.1 Generator Properties

The topic of procedural content generation is vast. This makes it important to distinguish between the several different types of procedural generation concepts that can be implemented in a piece of software. To get a better understanding of procedural generation and the ways it can be used, it is necessary to become familiar with the terminology used within the field and the different types of procedural generators it describes.

2.1.1 Online or Offline

Togelius et al. (2010) explain how the use of a generator, or the generator itself, can be classified as online or offline. They further explain that when talking about online generation in video games, one is talking about a game that will generate new content at runtime, for example, by creating new levels or worlds every time a new game is started. Examples for this would be the roguelike games *The Binding of Isaac* (Edmund McMillen, 2011) or *Enter the Gungeon* (Devolver Digital, 2016). Like their genre already suggests, in both of these games, one explores the depths of a generated dungeon, with a new dungeon being generated every time a new playthrough is started. Other game genres, like the more sandbox-esque open-world survival games like *Minecraft* (Mojang, 2008) or *Terraria* (Re-Logic 505 Games, 2011) that generate entire worlds at runtime, are also excellent examples of the use of online generation.
Offline generation, on the other hand, creates the content not during the game itself, but instead during its development (Togelius et al., 2010). Short and Adams (2017) mention that developers can use offline generation to generate larger segments of a huge travellable map (p. 4). They add that, additionally, these generated map segments can be modified and rearranged by human developers by hand (p. 4).

2.1.2 Seeds or Paramater Vectors

According to Togelius et al. (2010), another property used to differentiate between procedural generation algorithms, is the scope in which their output can be manipulated. They state that there are two different approaches. One approach, they describe, is based on the concept of being solely controlled by one input value, which is referred to as a seed. These seeds, most of the time, will be represented by a number or a string of letters (Short and Adams, 2017, p. 273). Although dependent on the algorithm, in many of them, using the same seed in multiple separate processes of generation, will always give the exact same output (Green, 2016, pp. 7-8). Coming back to the examples for online generation, games like *Minecraft* (Mojang, 2008) or *The Binding of Isaac* (Edmund McMillen, 2011) not only generate their content based on seeds, but also let their players enter one to replay or share worlds.
Togelius et al. (2010) describe that the second popular approach used to controlling a generators output are parameter vectors, which are vectors that allow multiple values for different properties affecting the generation process.

2.1.3 Stochastic or Deterministic

Togelius et al. (2010) report that another impactful trait you can categorize procedural generation algorithms by is whether they are stochastic or deterministic. They explain that while deterministic generators return the same content when using the same input values, the output of a stochastic one will still differ, despite the use of identical input values.
As already mentioned in section 1, most generators are based on randomness. However, with games like Minecraft (Mojang, 2008), this randomness only comes from the randomly chosen seed, while the seed itself produces the same outcome every time, making the generator deterministic.

2.1.4 Constructive or Generate-and-test

Togelius et al. (2010) make a distinction not only by how a generator generates but also by how often they generate. They explain that first, there is constructive generation, which will generate content only once. They deliver good content right away, which leads to not only shorter but also very consistent generation times (Shaker et al., 2016, p. 31).
Togelius et al. (2010) further explain that there is also generate-and-test generation. As the name suggests, these generators do not only generate but also evaluate the generated content (Togelius et al., 2010). Togelius et al. (2010) state that this happens with the usage of some sort of evaluation function, which will rate the generated content by specified criteria and determine if it is of acceptable quality. If it is not, the content will be discarded and a new instance will be generated. This process is repeated until

the generator creates an instance that is labeled as being of high enough quality by the evaluation function.

2.1.5 Automatic or Mixed-Initiative

The traditional, fully automatic use of PCG limits designers in their control about the generated outcome, only giving control about input parameters (Shaker et al., 2016, p. 10). Shaker et al. (2016) explain that at some point a new approach, called "mixed-initiative", has been developed, which involves designers much more, allowing algorithm and human to work together more efficiently (p. 10). An example of this is the mixed-initiative tool *Tanagra*, which generates a 2D-platformer level around sections manually designed by a human (Smith et al., 2010).

2.1.6 Generic or Adaptive

Shaker et al. (2016) describe that there are adaptive generators that take the player's behaviour into account, subsequently adapting the generation process, which generic generators do not (pp. 8-9). They mention that adaptive generators can be used to adjust the difficulty or generate content that the player seems to prefer (pp. 8-9).

2.2 Procedural Generation Techniques

Since PCG has been around for quite a while, it accordingly had lots of time to develop since its first use (Shaker et al., 2016; Short and Adams, 2017). In this section, it will now be taken a closer look at common generation techniques.

2.2.1 Search-Based

Also known as optimization-based (Sturtevant et al., 2015), search-based procedural content generation is a type of generation of the generate-and-test category (Togelius et al., 2010).
Togelius et al. (2010) describes two defining properties of a search-based approach. First, contrary to the general concept of generate-and-test, search-based PCG does not ultimately accept or reject the generated instances, but instead give them fitness-level. This fitness-level works as a grade and is determined by a fitness function. Second, every new generated instance is dependent of the fitness-grade assigned to its predecessor, with the goal of producing content ascending in fitness. For the successful production of content instances that possess the necessary features, it is typical to use an evolutionary algorithm (Short and Adams, 2017).

2.2.2 Grammars

Formal grammars are another popular tool for generating content. Even though they even though they were originally introduced in the field of linguistics, they also found

their use in other fields (Dormans, 2010).
Dormans (2010) explains that, to set up a complete grammar, one needs an alphabet and a set of rules. The grammar's alphabet consists of symbols. The rules of the grammar define rewrite operations, which consist of a left-hand side (LHS) and a right-hand side (RHS). Both sides of the rule describe a pattern of symbols. With this information, grammar-rules replace a found pattern that appears in the LHS of a rule with the corresponding RHS of the rule. Further, Dormans (2010) explains that symbols are divided into two categories, terminals and non-terminals. Non-terminal symbols are symbols that are defined on the LHS of a rule. If a rule, however, produces a symbol, not contained in any rule's LHS, it will never be replaced, thus being labeled as a terminal. While the original linguistic grammars generated strings, in the field of level generation grammars popularly use the domain of graphs and shapes, conveniently called "graph grammars" and "shape grammars" (Dormans, 2010; Williams, 2015, p. 16-17). The concept of these two grammar types will be covered in more detail in 2.2.3.
Shaker et al. (2016) clarify that there are two main categorizations when it comes to grammars, being whether they are either deterministic or nondeterministic and whether they apply rewriting operations in a sequence, or in a parallel manner (p. 75). They explain that a grammar qualifies as deterministic, if every symbol has only one corresponding rule, causing the generator to always return the same result, when given the same input (p. 75). Nondeterministic grammars on the other hand, have multiple possible rules able to be applied to one symbol, choosing a rule either randomly or by specified, or even generated, contingency (p. 75). While sequential rewriting limits a grammar to only apply one rule every iteration, parallel rewriting allows it to apply multiple rewrite operations simultaneously (p. 75-76).
Sturtevant et al. (2015) mention that grammar's can also be used in combination with a generate-and-test approach. They add that grammars allow a high degree of control over separate parts in the generated content, but tend to result in over- or undergeneration or a repetitive output.
Dormans (2010) heavily emphasizes the importance of implementing recursive rules when wanting to generate a bigger variety of results.

2.2.3 Constructive

As already introduced in Section 2.1, constructive generation is the faster and more consistent counterpart to generate-and-test generation. They are commonly used to generate dungeons (Shaker et al., 2016, p. 31) thus making them relevant to this research. In addition to being fast and consistent, constructive algorithms are also easy to implement (Shaker et al., 2016, p. 31). However, constructive generators also come with the downside of a lack of control (Shaker et al., 2016, p. 32). The different types of constructive techniques will be covered in this section.

Space Partitioning

Shaker et al. (2016), explain how space partitioning, is the process of creating a level by taking the level space and repetitively splitting it into multiple cells (p. 33). In the for level generation most popular space partitioning method, binary space partitioning (BSP), every cell emerging from the previous iteration is being split into two new ones (p. 33). This process will be repeated a desirable amount of times, with cells being stored in a BSP Tree (p. 33). This BSP Tree is useful for fast collision detection or raycasting (Shaker et al., 2016, p. 33) and considered as a practical representation of the level itself (Williams, 2015, p. 9). In the example of generating a dungeon, after the process of splitting the level area into cells is finished, rooms will now be placed inside them (Williams, 2015, p. 11-12). Lastly, these rooms will be connected with corridors and the dungeon generation is finished (Williams, 2015, p. 11-12).
An advantage of using this technique, is that, by the nature of the algorithm, it is impossible for the generator to create overlaps. Additionally, the generated instance will always have a very organized and structured appearance (Shaker et al., 2016, p. 34).

Agent-Based

When referring to agent-based generation, one describes the creation of a room-sequence by using an agent that digs tunnels through a defined level space (Shaker et al., 2016, p. 38-39). Compared to the previous approach of partitioning space, agent-based generation has a rather organic and imperfect appearance, when compared to the cleaner and more geometric one of the space partitioning method (Shaker et al., 2016, p. 38-39).
Shaker et al. (2016) describe how agents are NPCs that can be programmed to build the level when being thrown into the level space (p. 38-39). Based on this programmed behavior, the end result can vary quite a bit (p. 38-39). As an example, they specify two types of agents (p. 38-41). While one can be extremely reliant on randomness, the other can be given more intelligence, to keep track of the level structure as a whole and have foresight of what comes next (p. 38-41). When comparing the output of these two different agents, the level created by the first agent will be fairly unstructured and disorganized, including the overlapping of rooms and dead-ends (p. 38-41). The level created by the latter one however, will be the complete opposite: the ability to have foresight about the generation allows the agent to avoid the mistakes being made by the first agent, getting rid of overlapping rooms and dead ends (p. 38-41).
Nevertheless, Shaker et al. (2016) mention that this approach has another problem: the high probability of the predefined level space not being used to its entirety (p. 38-41). This can happen if the agent reaches a point where his intelligence will not allow him to place another room or keep digging, because it would otherwise result in a dead-end or the overlap of two or more rooms (Shaker et al., 2016, p. 38-41).

Cellular Automata

Much like space partitioning, cellular automata also works with cells (Johnson et al., 2010). Different from the splitting of cells with space partitioning, here, cells get eliminated in every iteration (Williams, 2015, p. 18).
Johnson et al. (2010) introduce an approach to use this algorithm in the context of level generation. First, they define cells as a point in a grid. These cells can be either of type floor or type rock. Additionally, each cell has its own neighborhood value, which is calculated by the amount of rock cells that are surrounding it. Moreover, there is a transition threshold to be defined that determines a cell as "floor" or as "rock", based on its neighborhood value.
The algorithm of Johnson et al. (2010) starts off by randomly distributing a defined amount of rock-cells across a grid with only floor-cells. Then, when actually starting the algorithm, each iteration checks for the neighborhood value of each cell and decided whether or not to transition it to its counter-state, based on its neighborhood value.
After finishing the generation after a fixed amount of iterations, the result is a realistic and organic cave-like structure. Johnson et al. explain that this algorithm is not very taxing on the game's performance, which makes it appropriate for usage at runtime.

Grammar-Based

Grammars, which have already been introduced in section 2.2.2, can also be used as a base for a constructive algorithm (Shaker et al., 2016, pp. 45-46). When generating a dungeon with generative grammars, it is popular to resort to graph or shape grammars (Williams, 2015; Dormans, 2010).
Dormans (2010) states that graph grammars are a structure, composed of edges and nodes. He explains that, if the in the grammar specified rules detect a pattern of nodes and edges defined as a nonterminal symbol, this structure will be replaced by a new one, being specified in the right-hand side of the rule. To keep track of the nodes during the rewrite operation, grammar rules specify a number for every single one (Dormans, 2010). Williams (2015), explains that graph grammars are limited to generate the structure and relations between nodes, not the dungeon space itself (p. 17).
Dormans (2010), uses graph grammars to generate a dungeon's mission, which will later be assisted by the dungeons space. They define the symbols of the grammars alphabet with tasks that the player has to take on.
The concept of shape grammars is very similar to the one of graph grammars, the difference being that instead of edges and nodes, a shape grammar's alphabet consists of smaller shapes that, when combined, create a larger one (Williams, 2015, p. 17). Dormans (2010), used shape grammars to accommodate the mission structure that was generated beforehand.

2.2.4 Constraint-Based

Totten (2017) describes the constraint-based method as a declarative approach (pp. 167-168). They state that constrain-based generation uses a constraint solver to return a desired result, based on specified strict constraints. They add that the time a constraint-based generator takes to generate content is heavily dependent of the problem the constraint solver needs to solve and the amount of constraint that has been defined. With more constraints, there will also be a better chance of the generator returning content of desired quality.

2.3 Strengths and Weaknesses of PCG

2.3.1 Strengths of PCG

According to Totten (2017), one of the main advantages of using procedural content generation is the seemingly infinite amounts of content that can be provided by the player, giving them a new challenge every time they play, thus increasing the replayability of the game compared to handcrafted games (p. 160, pp. 219-220). They explain that there are multiple other benefits resulting from this, like the challenge that comes from being faced with new levels every time, making it impossible to know what comes next or to memorize any patterns, as opposed to handcrafted games (pp. 220-221). Addtionally, they mention that comes from the almost endless amount of content is the exploration and discovery value that comes with it, since, when compared to handcrafted levels, the player will never have encountered every single gameplay scenario the game can provide (pp. 221-222). Another advantage that comes with the use of procedural systems is their ability to adjust themselves, for example adapting to the player (2.1.6) (Totten, 2017, p. 160).
Green (2016) state that the generation also leads to the ability to create larger games in general (p. 13). They explain that this is due to two reasons. First, since generating a level is significantly faster than building a level by hand, one is capable of creating larger worlds much faster with PCG (p. 13). The second reason is that "if a game utilizes procedural generation correctly" (Green, 2016, p. 13), the size of a generated world knows no limits (p. 13). This comes from the fact that procedural generation systems do not need to save the patterns of objects, but only the objects themselves, leaving the combination of them to an algorithm (Green, 2016, p. 10)

2.3.2 Weaknesses of PCG

Green (2016) claims that one disadvantage of PCG is the generation of content that can feel repetitive (p. 15), as it can be the case for constructive or grammar-based methods (Sturtevant et al., 2015). Green (2016) also mention that PCG can be computationally intensive, making it necessary to always consider which hardware is capable of running the generation algorithm (p. 15).

Short and Adams (2017), compare PCG to a painting brush that the painter may hold closer to the end of its handle, which results in less control over the brush strokes (p. 18). Due to this lack of control, one can never assure that the generation will always go as expected and might include undesired and broken results that might only occur extremely rarely (Short and Adams, 2017, p. 6). Lastly, procedural systems can be implemented on a large and complex scale, making them very difficult to debug (Totten, 2017, p. 163).

Chapter 3

Defining What Makes a Level Handcrafted

To achieve the goal of generating levels that seem handcrafted, it is necessary to get an understanding of what can contribute to a level's handcraftedness. This segment will analyze handcrafted levels for their characteristics. First, it will be taken a look at two analyses (Dormans, 2010; Stout, 2012) done of dungeons in games from the *The Legend of Zelda* (Nintendo, 1986-2023) series. All titles from this series will be referred to as "Zelda games". Second, it will be researched what the overall structure of a well-designed level looks like and what level design patterns are common to use. For the sake of simplicity, all findings resulting from this research will be referred to as "humanization techniques", meaning that they improve the handcraftedness of a level.

3.1 Dungeons in The Legend of Zelda

To get an overview of how level designers build dungeons in handcrafted games, this section will present two analyses done of dungeons in two different Zelda games.

3.1.1 Dormans' Analysis

Dormans (2010) presents a detailed analysis of the structure of a dungeon in the game *The Legend of Zelda: Twilight Princess* (Nintendo, 2006), which can be applied to most dungeon-like levels in action adventure games. In their study, they found that a dungeon can be split into two different structures: mission and space (Figure 3.1). While missions define the sequence of tasks necessary to be completed to finish the level, space gives the mission a structure to be mapped on. Dormans emphasizes that mission and space are independent of each other and that one mission can be used for several spaces, just like many different missions can be applied to the same space. They also present a formula that is supposed to mirror the key stages of a dungeon's structure, found in many Zelda games. It consists of four stages, representing the journey of the player and his development throughout the dungeon.
The four stages are as follows:

1. kihon stage: acquire an ability in an isolated space
2. kihon-kata stage: repeat and master the learned ability

3. kata stage: learn about the combination of multiple abilities

4. kumite stage: test the player's skills in a boss fight

Dormans (2010) add that the analyzed Forest Temple additionally follows a Hollywood film-esque structure, where the player gets introduced to the adventure by a character. Then, after defeating the mid-level boss, they obtain an item which sets the start to the dungeon's final segment.
It is important to note that this obtained item also acts as a key to multiple locks inside the dungeon, ascending the form of simple doors (Dormans, 2010). Locks and keys not being restricted to the principle of inserting a fitting key into the correct lock to open a door but instead being embodied by items is a very common design principle when it comes to dungeons in action-adventure games (Dormans, 2011).
Although the analysis done by Dormans (2010) is about a dungeon in a third person game, this formula is seen in all entries within the series, such as *The Legend of Zelda* (Nintendo, 1986), *The Legend of Zelda: Link's Awakening* (Nintendo, 1993) and *The Legend of Zelda: A Link to the Past* (Nintendo, 1991), which are all played in top-down view.

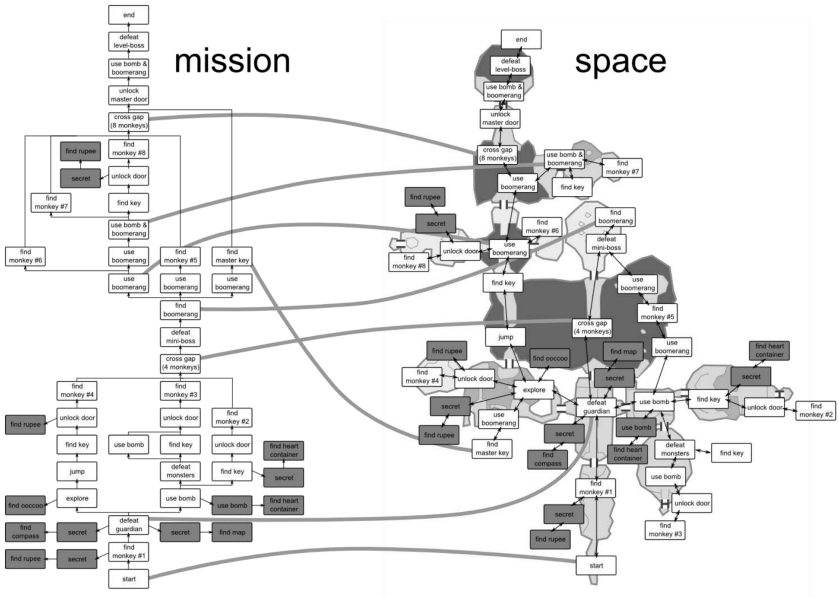


Figure 3.1: Mission and Space (Dormans, 2010)

3.1.2 Stout's Analysis

Stout (2012) did an analysis similar to Dormans. They, however, focused on dungeons from the top-down game *The Legend of Zelda* (Nintendo, 1986). While presenting similar findings to Dormans concerning the training of the player, they also focused on the difficulty and branching of the dungeon. They report that dungeons always have a linear critical path, which is the path the player needs to take to progress. However, optional rooms of exploration have been used to hide the linearity of the

critical path. Concerning difficulty, they state that the difficulty of rooms on the critical path increases from the beginning to the end of the dungeon.

3.2 Level Flow

To get a better and more detailed understanding of the general structure and pacing of game levels, it has been analyzed how level designers compose a level or game in terms of suspense and other things that can make players engaged.
Chen (2007) explains that flow (or pacing) is a key concept for designing game experiences. Their research is based on Csikszentmihalyi (1990), who defined these key components of flow:

1. Challenge
2. Goals that are clear to the player
3. Direct feedback
4. Distraction from everyday life
5. The feeling of control
6. Losing self-consciousness
7. Altering the sense of time

Chen (2007) explains a concept called the "flow zone", in which modern games are trying to keep the player in. The flow zone is the desired state of optimal balancing between challenge and player ability, keeping the player from getting too overwhelmed or too bored.

3.2.1 Difficulty Scaling

Feil and Scattergood (2005) divide the flow of a game into three different acts. In the beginning, the player will be taught how to play the game, giving them easier challenges that make them feel skilled (pp. 14-16). In the middle of the game, which they describe as the main part, is where the game should ramp up in difficulty, adapting to the improvement of the player's skill (pp. 16-17). It is explained that this rise in difficulty should happen gradually (p. 17). They also mention the concept of player growth, expanding the player character's strength by adding or enhancing abilities (p. 17). Last, there is the finale (pp. 17-19). The finale provides challenges consisting of many challenges the player faced before, to test everything they learned on their journey.
Feil and Scattergood (2005) also mention that the increase in difficulty can be implemented through the supply of resources like health or ammo and the amount and difficulty of enemies placed (p. 108).

This is the same difficulty scaling mentioned in the analysis done by Stout (2012).

3.2.2 Difficulty Breathing

In addition to difficulty scaling, Feil and Scattergood (2005) also state that when applying difficulty scaling to an area inside the game by making areas more difficult towards the end, it is important to alternate between moments where the player can relax before moving on to the next, more difficult challenge (pp. 107-108). In this thesis, this concept will be referred to as difficulty breathing.

3.2.3 Objective-Challenge-Reward

A level commonly consists of objectives, challenges, and rewards, first giving players an objective, then letting them face a challenge, and when successfully beating the challenge, giving them a reward (Feil and Scattergood, 2005, pp. 119-136). When designing the game's or level's flow, these three concepts appear to be essential.

Objectives

The objective tells players about their next destination, giving them a place to go and things to do (Feil and Scattergood (2005), p 126). This ultimately provides a goal for the gameplay, making objectives important for implementing the flow element of setting clear goals the player can follow. One can categorize these goals as long-term or short-term, short-term goals being smaller tasks for the player like defeating enemies or finding items, and long-term goals representing a bigger mission that consists of multiple short-term goals (Totten, 2014, p. 260).

Challenges

Feil and Scattergood (2005) explain that challenges are what stand between the player and successfully pursuing the objective (p. 10). Challenges can be implemented in many different ways, including giving players a time limit to achieve something, testing how far the player can go before getting defeated, or giving the player limited resources that they have to use carefully to overcome the challenge (Feil and Scattergood, 2005, p. 10).

Rewards

Totten (2014) explain that rewards give the player motivation to face challenges or even play the game at all, and sometimes they can become the objective themselves (p. 259). Tekinbas and Zimmerman (2003) define four different types of rewards (chapter 24, pp. 18-19):

- Reward of Glory: Do not affect gameplay but rather represent the player's achievement in the game.

- Rewards of Sustenance: Resources to help the player through challenges, like healing items or items that increase damage.
- Rewards of Access: Can be used once to grant the player access to new areas.
- Rewards of Facility: Grant the player new abilities or expand already existing abilities.

Totten (2014) explains that rewards can also take the shape of gamespaces (pp. 246-250). It is stated that while reward vaults contain reward items, there are other reward spaces, like rewarding vistas, that reward the player with great views or visuals, or meditative spaces that reward the player with calm, relaxing gameplay (pp. 246-249). They also mention narrative spaces, which reward the player with progression in the story (pp. 249-250). It is emphasized that rewarding vistas, meditative spaces, and narrative spaces also reward the player with a moment of relaxation (pp. 246-250), maintaining a good flow.

3.3 Patterns in Level Design

In this segment, it will be covered what techniques, structures, and overall design patterns professional level designers use and the reasoning behind them. This research focused on patterns that also have a strong presence of a designer's intent to be used to achieve the goal of making a level actually seem as if it was made by one. Additionally, dungeons of the top-down RPG game *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) have been analyzed, finding representations of these level design techniques within its level design.

3.3.1 Gamespaces

Totten (2014) presents multiple gamespaces that can make up a level's overall structure (Totten, 2014, pp. 112-125). First, he presents three basic size types: narrow space, intimate space, and prospect space (pp. 118-125). While narrow spaces are only slightly larger than the player, prospect spaces are areas very big and open, leaving the player without guard against attacks (pp. 122-123). Intimate spaces are neither small, nor big and instead have a size and structure in which the character can comfortably use his movement abilities to move around in (pp. 120-122). Totten also mentions other gamespaces like labyrinths (p. 113) and mazes (p. 114), which will play an important role in sections 3.3.3 and 3.3.4.
Totten (2014) emphasizes the importance of the combination of these different gamespaces to create interesting dynamics (pp. 125-130). An important concept he covers is the alternation between prospect spaces and refuge spaces (p. 210-221). Refuge spaces are being described as intimate spaces that provide cover and a view onto a prospect

space (p. 211). Alternating between these two spaces creates an interesting relationship between spaces that are safe or unsafe throughout a level, thus creating "emotional experiences" (Totten, 2014, p. 210) for the player (pp. 210-212). This represents the flow concept of difficulty breathing, where the player is in a more comfortable space. Many examples of prospects and refuges can be seen when looking at a map of a dungeon in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) (Figure 3.2).

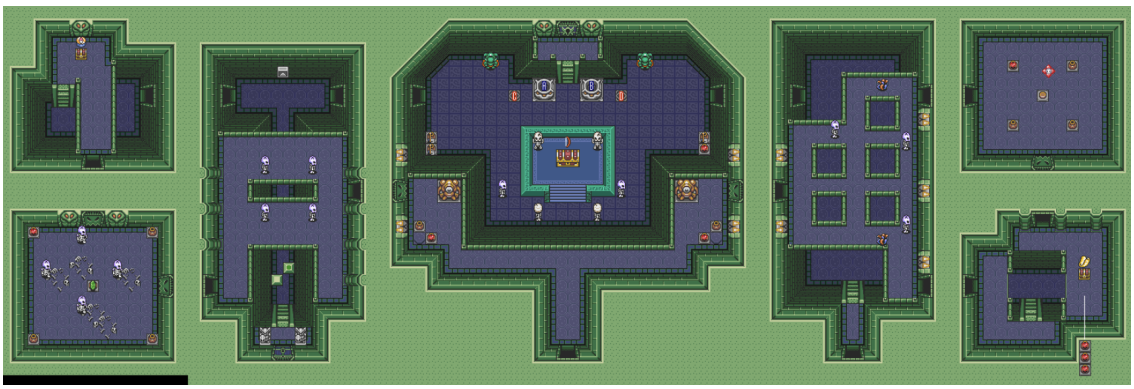


Figure 3.2: Prospects and refuges in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) (Rick N. Bruns, 2019b)

3.3.2 Encouraging Exploration

Totten (2014) explain that another common technique in level design is the encouragement of exploration (pp. 243-245). They explain that one way to do this is through zen views (pp. 251-252). Zen views use the sight of the player to show them a reward while not directly showing them how to get there, thus encouraging them to explore the level to find a way (pp. 251-252). In top-down games, this can be implemented easily because the player is not actually limited to the avatar's view but sees the entire level from above (section 1.1). Under these circumstances, one can easily show players a reward in a room hidden behind a wall while still being visible to the player but only accessible by entering the room through a different entrance (Totten, 2014, p. 244). *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) uses this technique to hint at a big chest the player can see but not reach while traversing a bridge (Figure 3.3). This gives the player a hint to explore the area around the chest to get to it. The chest mentioned is the same chest the player reaches at the end of the maze mentioned in the previous section.

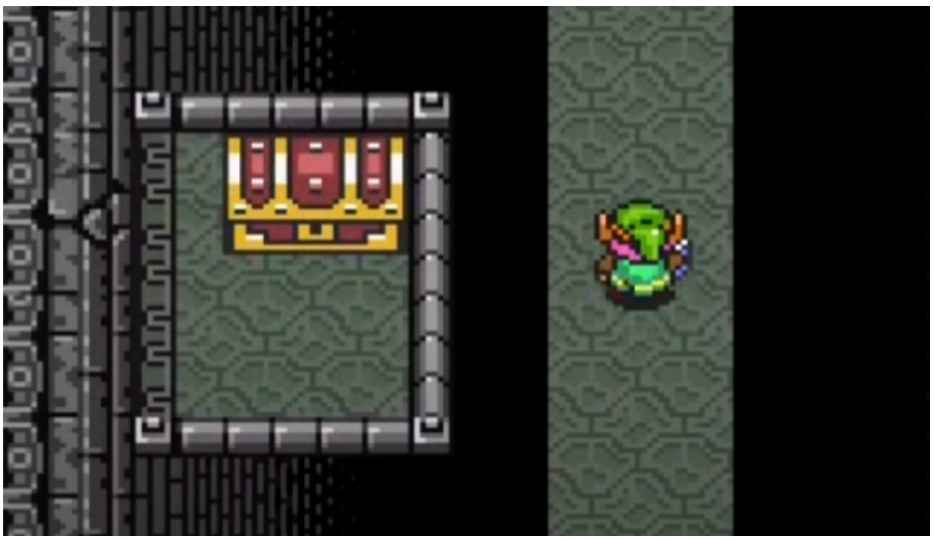


Figure 3.3: A dark maze with invincible enemies in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) (SweetJohnnyCage - Narrated Guides & Walkthroughs, 2020)

3.3.3 Using Player Emotion

Totten (2014) explains how level designers can make use of the player's emotion (pp. 114-116, pp. 208-210). They explain that this can be done by combining narrow architecture, like mazes, with very strong enemies (pp. 114-116). They also describe how another way to introduce drama in a level, especially in games where players have to use their resources carefully, is by taking advantage of the player's low amount of resources by letting them face difficult challenges that are hard to estimate, for example because they include hidden enemies, which creates stressful gameplay (pp. 208-210). The dark palace in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) uses a small, dark maze where the placer has to quickly reach the next door before getting hit by invincible enemies roaming around the maze (Figure 3.4).

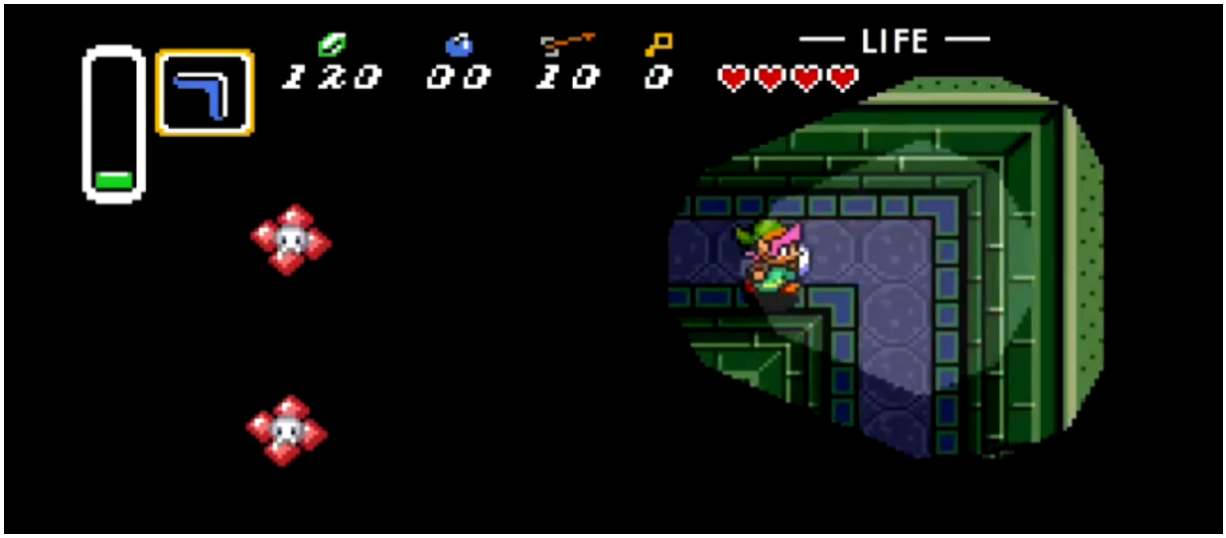


Figure 3.4: A dark maze with invincible enemies in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) (SweetJohnnyCage - Narrated Guides & Walkthroughs, 2020)

(a) (b)

Figure 3.5: The player gets rewarded after finding their way out of the maze in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) (SweetJohnnyCage - Narrated Guides & Walkthroughs, 2020)

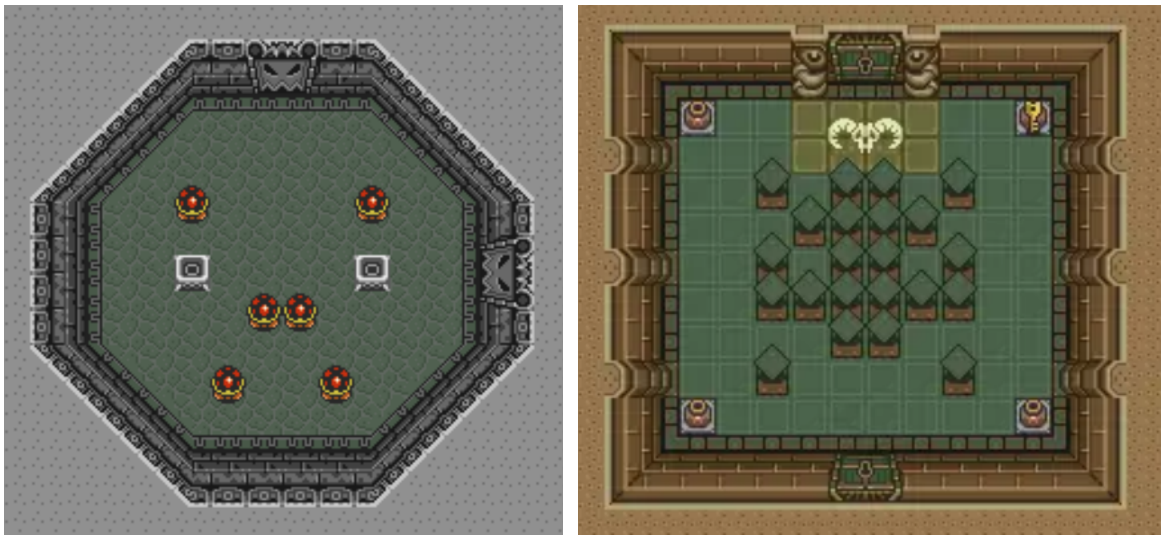
3.3.4 Balancing Risk and Reward

Risk is a common tool to implement in level design used to contribute to the player experience's excitement (Totten, 2014, p. 194). McMillen (2010) explains how, originally introduced in arcade games, risk-reward is a concept rewarding players for taking a higher amount of risk than usual. They state that this risk is oftentimes completely optional, which is why a level should reward taking that risk accordingly. Typically, the risk players take is losing health (McMillen, 2010; Totten, 2014, pp. 208-210).
Totten (2014) explains how the example of creating dramatic experiences by taking advantage of a player's vulnerable status and hidden enemies in section 3.3.3, can also be used as a risk-reward scenario (pp. 208-210). This is due to the challenges unpredictability, putting players, even when possessing a high amount of resources, at risk of losing those resources in order to achieve the challenge's rewards (pp. 208-210). They also mention that the uncertainty of gamespaces like mazes also serves as a good base for risk and reward (p. 114, p. 210)
A good representation of this can be found in the maze of *The Legend of Zelda: A Link to the Past*'s (Nintendo, 1991) Dark Palace. Here, the player wanders through a dark maze with many branches (Figure 3.5). However, the risk the player takes to explore the maze gets rewarded accordingly with a key and an item necessary to progress, with the item even rewarding them with a new weapon.

3.3.5 Object Structure & Object Purpose

Since procedural generation relies a lot on randomness, the random distribution of objects across the level would result in rather chaotic, unstructured, and even meaningless patterns. This is the exact opposite of what one can find when looking at handcrafted levels. When looking at dungeons in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) for example, it is easy to make out symmetrical, structured, clean, and generally just visually appealing patterns. This also applies to the level's architecture,

which includes mostly symmetrical shapes for rooms (Figure 3.6).



(a) (b)

Figure 3.6: Symmetric object structures in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) (Rick N. Bruns, 2019d; Rick N. Bruns, 2019c)

Feil and Scattergood (2005) stress the importance of having enemies have a reason to be where they are (p. 101). They explain that this reason could, for example, be to protect something the player might want to get or to reach something the player needs to defend (p. 101). This fulfills the concept of objective-challenge-reward, putting a challenge between the player and the reward, in this case leaving it optional to the player if they want make it their objective to obtain that reward. With random generation in PCG, this is also something that only has a possibility of happening and will not be consistently created.

Regarding object purpose, many patterns where enemies have the purpose to protect something from the player can be found in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) (Figure 3.7).

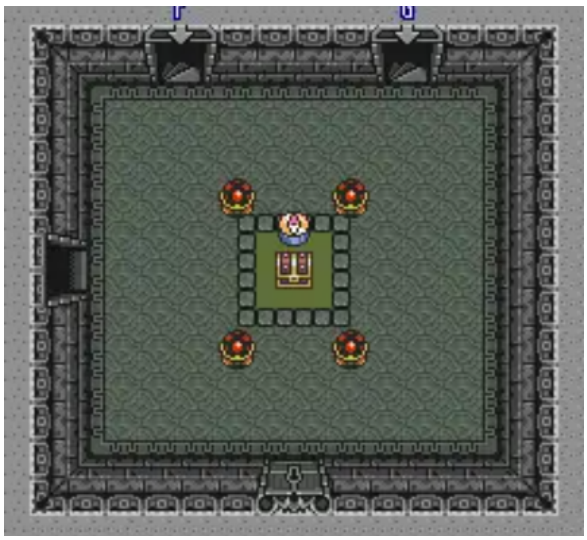


Figure 3.7: Enemies protecting items from the player in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991) (Rick N. Bruns, 2019d)

3.3.6 Giving the Player Hints

Totten (2014) describes that by using repeating patterns, a level can indirectly tell the player about what is coming next (p. 46-48). They explain that this concept can give players not only information about difficult challenges that are coming up but also give them time to plan and prepare (p. 46-47). Totten give the example of *Mega Man* (Capcom, 1987), which uses an empty corridor that is placed right before the boss fight every time, which makes the player recognize them as an indicator for a boss fight (p. 47).

Additionally, transitions like these can also give the player a hint by their nature alone without being repeated, using game objects to signalize the entrance to a more dangerous area (Totten, 2014, p. 142). This could also be applied to the empty boss corridors used in Mega Man (Capcom, 1987), since they are so distinct from the rest of the level that one could suspect a special event is coming up even on the first encounter with the pattern.

McMillen (2010) explains that it is also common to use collectable items to guide players where to go and where rewards could be hidden.

An example of this can also be found in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991), where the room connected to the room in which the boss is located, has a narrow path to the boss room door, guarded by enemies (Figure 3.8). This gives players the feeling that the are entering the throne room of the dungeon's strongest enemy. Additionally, on the floor tiles in front of the boss room door, players can find a symbol on the ground signalizing that this is the boss room. This symbol is repeatedly used throughout the game.

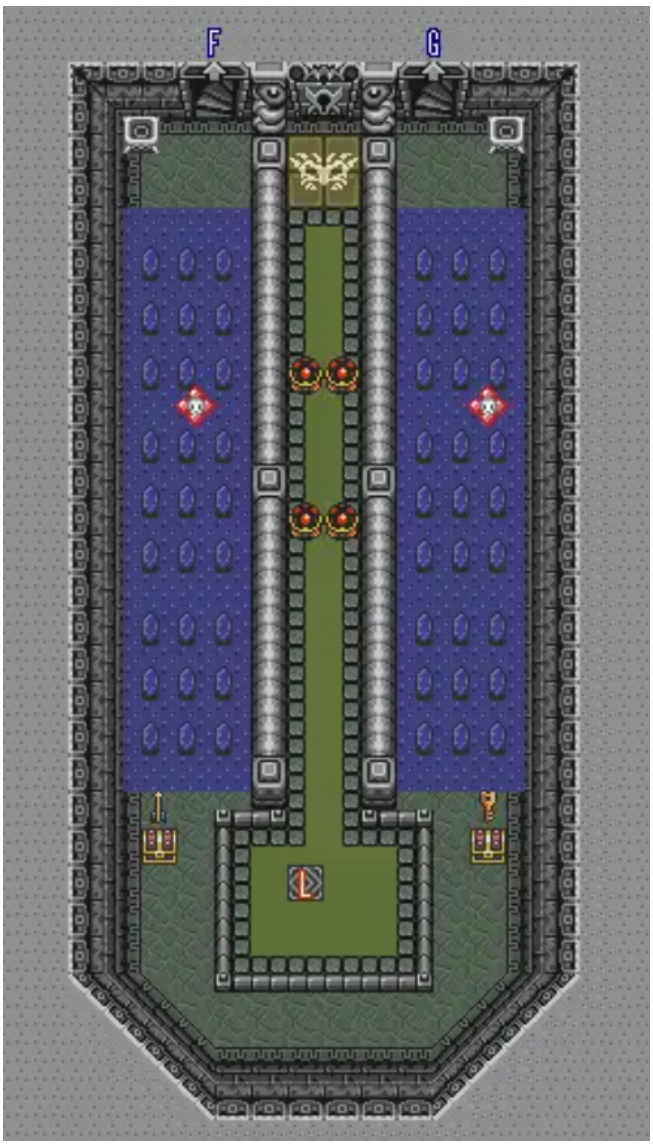


Figure 3.8: A dark maze with invincible enemies in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991)(Rick N. Bruns, 2019d)

Chapter 4

Procedural Generation Methods to Generate Seemingly Handcrafted Levels

This chapter will present research done about PCG techniques and approaches that provide the necessary control over the generator's output and make it possible to design a level to a high enough extent that implementing the established humanization techniques is possible in a procedural system.
All promising methods that were found during research fall into the category of mixed-initiative generation. This makes sense, since, when looking at the types of generators (section 2.1), mixed-initiative ones seem to provide the most control over the generation, since the designer is directly involved in the process. This allows for strong restrictions that lead the generator to behave in a desired way but can still provide the variety of randomness in procedural generation (Totten, 2017; Short and Adams, 2017; Mawhorter and Mateas, 2010; Smith et al., 2010). This makes them favorable for the goal of implementing certain design patterns into a procedural generation process.

4.1 Vertical Handcrafted Integration

In their analysis about handcrafted integration in procedurally generated games, Totten (2017) came to the conclusion of two different types of integrating handmade content: vertical integration and horizontal integration (p. 227). Totten (2017) describes vertical integration as a handmade sequence, continuously progressing through the entire playthrough, instead of just through a single level, establishing purpose and coherence (p. 238). In the analysed games, this is being done by integrating handmade narratives or puzzles (p.238).

4.1.1 Case Study: Faster Than Light

In his analysis, Totten (2017) explains, how the game *FTL: Faster Than Light* (Subset Games, 2012) (*FTL*) makes use of a handcrafted narrative supporting the game's otherwise generated levels (pp. 227-229). In this game, the player navigates a his spaceship to escape the enemy that is chasing him, moving through a map entirely made up of nodes, each node being a certain event (p. 228). While the majority of these nodes are procedurally generated, some nodes contain handmade quests and are laid out in a se-

quence specified by hand (p. 228). While the handmade instances are optional, player are animated to still play through them, by, when beating the challenges provided by these nodes, receiving higher rewards compared to the rest of the nodes (p. 228). Totten explains that the incorporation of this handmade narrative not only introduces variety to the procedural structure, but also makes separate playthroughs distinct and unique (p. 229, p. 230).

4.1.2 Case Study: Spelunky's Secret Puzzle

Similar to *FTL: Faster Than Light* (Subset Games, 2012), *Spelunky* (Mossmouth, 2008) also uses vertical integration to create a narrative that accommodates the playthrough (Totten, 2017, pp. 230-231). This handmade narrative, however, is less of a story like *FTL*'s and instead a secret puzzle (Totten, 2017, pp. 230-231) that the player might not even come across in the first place (Mossmouth, 2008). Like *FTL: Faster Than Light*, this secret quest is not necessary to complete the playthrough and consists of certain items (pp. 230-231). If players wants to enter a secret path towards the end of the level and get a special ending, it is mandatory for them to collect all of the items, since they act as keys to reach this secret path. Spelunky also incorporates a different sort of narrative by scripting different biomes, like ice and jungle, which change every every four levels, each containing different objects and enemies (p. 231) (Mossmouth, 2008). Totten (2017) explains that the use of this handmade narrative leads to a unique challenge for players wanting to ascend the default difficulty of Spelunky and counters the complexity of generating it procedurally (pp. 230-231).

4.1.3 Game Forge

Hartsook et al. (2011) present another mixed initiative method, focusing on the story of procedurally generated games. Their system, *Game Forge*, uses a sequence of plot points, which represent the order in which a player has to perform different tasks like defeating enemies, buying or selling items, or interacting with certain NPCs, as well as defining where players have to perform that action. They explain that this plot point sequence can be generated, but can also be designed by a human. Based on this predefined story, the game world will be generated, taking player preferences into account (Hartsook et al., 2011).

4.1.4 Conclusion

FTL: Faster Than Light (Subset Games, 2012) and *Spelunky* (Mossmouth, 2008) show useful ways of how one can effectively string together a series of procedurally generated levels to accommodate a human-designed, precise, and complex mission, still keeping the qualities of PCG's replay and exploration value.
Game Forge is a more general representation of the vertical handcrafted integration done in Mossmouth (2008) and Subset Games (2012), allowing for control of the se-

quence of events the player encounters and ultimately giving the opportunity to design the progression of a game.
Vertical handcrafted integration seem to be a powerful tool when trying to generate a certain type of progression and narrative throughout the game, making it a promising tool to implement flow (section 3.2) in a generated level.

4.2 Horizontal Handcrafted Integration

When integrating handcrafted content in a horizontal manner, it is more about the interchangeability between the generated content and its handmade counterparts (Totten, 2017 p. 238),

4.2.1 Case Study: Dungeon Crawl Stone Soup

Dungeon Crawl Stone Soup barbs, 2006 is a roguelike game combining PCG and handmade content in a very interesting way. In this game, levels are procedurally generated for the most part, but not entirely (Totten, 2017). Totten (2017) explains how *DCSS* (Dungeon Crawl Stone Soup) uses handmade instances, called "vaults", to introduce variety and character into the visual appearance and gameplay flow of the otherwise fully procedurally generated level (p. 229-230). These vaults provide more difficult and well designed and planned gameplay experiences (p. 229). To balance the risk players will take trying to conquer vaults, when successful, they will also be given bigger rewards (p 229-230). Because of their specific, symmetrical structure or special objects, already at the start of a level, the player can immediately identify the handcrafted vaults (pp. 229-230) and Totten gives the idea that hiding the obviousness of this may be not even desired in the first place (p. 236). Similar to Faster Than Light, even though the exploration of vaults is not necessary to progress (p. 230), players are encouraged to explore the handcrafted instances by being given the chance to obtain greater rewards (Totten, 2017, p.230).

4.2.2 Case Study: Spelunky's Dungeon Patterns

In addition to vertical handmade integration, *Spelunky* (Mossmouth, 2008) also implements handcrafted content in the horizontal manner (Totten, 2017). Totten, 2017 describes how *Spelunky* uses handmade rooms that define its structure on a macro level to generate dungeons (pp. 230-231). These rooms are later being procedurally modified on the micro level to introduce more variety and counter pattern repetition (pp. 230-231). *Spelunky* also features something called "level feelings", which are levels that restrict the generator to generate special events and structures (Mossmouth, 2008), or dominantly use a certain enemy type (Totten, 2017, pp. 230-231).

4.2.3 Case Study: Dungeonmans

Dungeonmans (Adventurepro Games LLC, 2014) is a game using handcrafted integration to design multiple distinct gamespaces like "graveyards, fallen castles, crypts, friendly towns, mighty towers, and the good old-fashioned dungeon" (Short and Adams, 2017, p. 63). For dungeons, it is explained that to create the playing field, handcrafted rooms are being placed in a procedural manner (p. 66). The room data is stored inside text files containing a rectangular seed defined by ASCII characters, with the first line defining the dimensions of the seed (pp. 63-64) (Figure 4.1). It is emphasized that although the seeds are rectangular, the rooms themselves can still be nonrectangular, leading to their goal of producing more interesting results (p. 63). Short and Adams (2017) explain that an "area gen code" is used to fill the rooms with loot, enemies, and external rooms, which is why they are not included in the room seeds (p. 64). For dungeons, the purpose of the handcrafted rooms is to provide spaces where combat is fun, hence why they use "a mixture of large areas and small, twisting paths and grand hallways, tight clusters of rooms and open arenas with cover" (Short and Adams, 2017, p. 66).

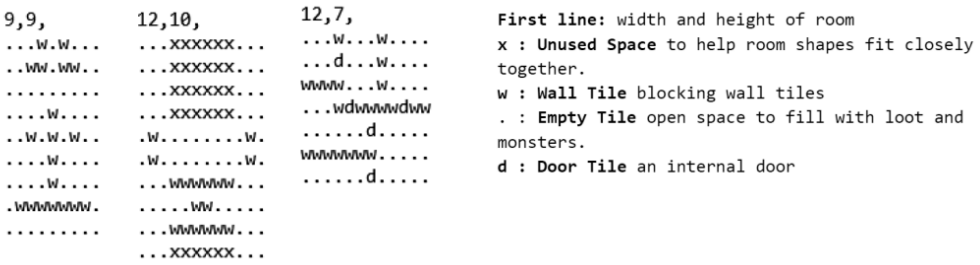


Figure 4.1: Seed data for dungeons in *Dungeonmans* (Adventurepro Games LLC, 2014) (Short and Adams, 2017, p. 64)

To generate a level, Short and Adams (2017) describe that one specifies the level space as empty and define a maximum room count (p. 64). Then, a random room is being inside the level space and the dungeon is being grown from there by continuing the placement of rooms with random orientations (Short and Adams, 2017, pp. 64-66). Short and Adams (2017) do not recommend changing the shape of rooms that otherwise would not fit, explaining that it would lose the concept and idea defining the room in the first place (p. 65). After the overall structure of the dungeon has been generated, they also implement some extra doors to avoid backtracking by creating cyclic paths between rooms (p. 66), moving away from a branching nature, similar to like Unexplored does. Additionally, they also pick out doors on walls that are too long and empty and widen them to make it more difficult for the player to use tight room openings to their advantage in combat against enemies (p. 66).

Short and Adams (2017) also give insight on how the generation of crypt areas works, this time with all room seeds having the same dimensions of 7 x 7 and holding exits to the north, east, south, or west (p. 66). In their overall style, crypts aim for an entirely different structure, the goal being more narrow with some open areas to give the player space to breathe (p. 66). Different from the dungeon seeds, crypt seeds also provide information about loot and also statues exclusive to crypt areas, adding to the darker atmosphere (p. 66) (Figure 4.2). Crypt rooms also have another big difference from the dungeon ones, this time having open sides, enabling them to be combined to create larger shapes (p. 66). This works by sorting rooms into lists based on their exit direction, one for each direction (p. 66). To combine rooms, the generator picks out a room with an open exit and then chooses a room with an opposing exit direction to merge it with (p. 67). A further distinction from dungeons is that crypts also feature different enemy types, making effective use of the level's narrow nature by using multiple enemies to chase the player and enemies that can control the player (Short and Adams, 2017, p. 68).

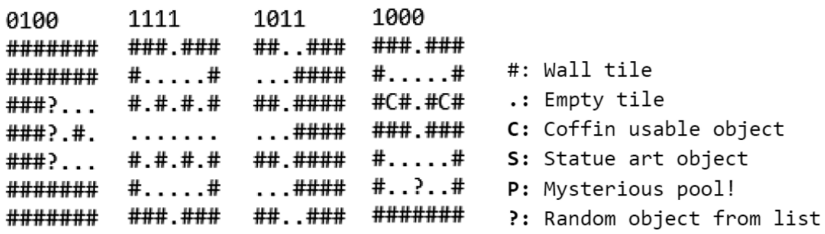


Figure 4.2: Seed data for crypts in *Dungeonmans* (Adventurepro Games LLC, 2014) (Short and Adams, 2017, p. 67)

4.2.4 Conclusion

Dungeon Crawl Stone Soup (barbs, 2006) shows the implementation of a lot of the previously mentioned level design patterns. Vaults create a balance between risk and reward and Totten (2017) explains that when placing objects like items and enemies, the game also takes into account what area the player is in and how deeply into the dungeon it is located (p. 229), incorporating the concept of biomes and difficulty progression. Additionally, Totten also explains that *DCSS* uses different patterns between playthroughs when generating content to make them more distinct (p. 229). Placing large elements entirely created and designed by hand into a level, like *DCSS* does with vaults, is a simple and effective technique to contribute to the handcrafted appearance of the level as a whole, effectively balancing out the randomness of the procedurally generated content.

Spelunky (Mossmouth, 2008) presents an approach to effectively balance the randomness of PCG and the design of handcrafted instances, by blending the two, creating rooms that follow a handmade macro structure, but, because of the procedural variation for only specific room tiles, will rarely be identical. The implementation of level

feelings also contributes a lot to the levels it is used in, following a challenge designed by hand, strongly constraining the level generator to follow that planned out challenge. The approach used in *Dungeonmans* (Adventurepro Games LLC, 2014) seems effective in providing the ability to design different areas with the alternation between different gamespaces, which, shows high potential of providing the necessary ability to implement many of the humanization techniques. It also proves to successfully generate completely different structures as a whole by providing many opportunities to implement different level design, which becomes clear when comparing dungeons with crypts. The implementation of cycles also adds a lot to a handcrafted appearance, since these are also found a lot in handcrafted levels, as stated by Short and Adams (2017) (p. 84).
Horizontal handcrafted integration appears to be a tool with high value. As seen in the examples, it is capable of providing control over the space and, more importantly, the patterns it generates. When considering the goal of implementing the analyzed level design patterns (3.3, this seems to be an effective way to achieve it.

4.3 Grammars

Dormans Dormans (2010) presents how to utilize the results of their analysis of action adventure game dungeons, which has been covered in section 3.1.1, within the process of procedural dungeon generation. They start off by first explaining the utilization of generating dungeons in a more abstract way, introducing a simple alphabet and rules for a dungeon grammar and give an example that represents the structure of the analyzed Forest Dungeon in this form (section 3.1.1). The grammar starts with the symbol of the dungeon as a whole, getting replaced by the symbols ”obstacle” and ”treasure”. The obstacle will then be replaced by patterns of symbols, containing typical dungeon segments like ”key”, ”lock”, ”monster”, ”treasure” and ”room”. According to Dormans, this approach can produce something boring or too short, but it is stressed that this problem does not lie within the grammar itself but in the rules that make it up.
Dormans (2010) later gives a detailed explanation of the generation process of the actual dungeon. Based on their previous statement that mission and space are to be separated, they go on to explain how one can generate missions and spaces in separate approaches, using graph grammars for mission generation and shape grammars for space generation. They have both already been introduced in Section 2.2.3, but will be explained in more detail in the following segment.

4.3.1 Mission Generation

Dormans (2010) states graph grammars are a good choice when generating missions and is best being done in a nonlinear form. They also stress that mission grammars require a alphabet consisting of distinct tasks that not only challenge but also reward the player. They give a detailed explanation in how they use graph grammars do

generate the dungeon mission. After setting up the grammar's alphabet and rules (Figure 4.3), they explain the iteration steps are as following:

1. Check for nonterminals
2. If a nonterminal pattern has been found, the detected nodes will be numbered after the corresponding rule's LHS.
3. Remove all edges between the nodes.
4. Replace the nodes in the graph with their RHS equivalents
5. Add nodes of the RHS that do not have an equivalent number on the LHS of the rule.
6. Reconnect the nodes with egdes after the applied rule's RHS and remove the numbering.

Dormans adds that one can only add the functionality to remove existing nodes. In addition to the normal edges that are represented by a single arrow, they also introduce double arrows, which represent a "tight coupling" (Dormans, 2010) between nodes, forcing the succeeding room to be placed behind its predecessor, giving the designer more control of the level's branching. This has the important purpose of fulfilling meaningful level design practices like placing keys or rewards behind a challenge or a lock, excluding the chances of these objects being randomly placed before the player even encounters the problems to solve, which would result in poor gameplay quality (Dormans, 2010).

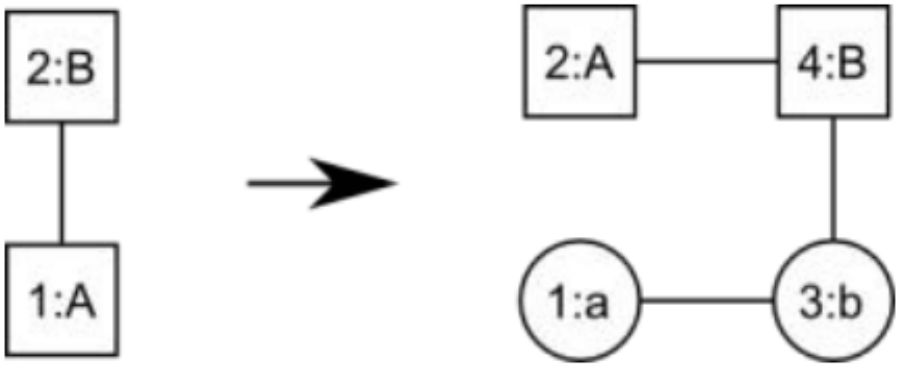


Figure 4.3: Graph grammar rule to generate missions (Dormans, 2010)

4.3.2 Space Generation

Next, Dormans (2010) explain their process of grammar-based space generation, using shape grammars. Similar to the numbering when using graph grammars, they introduce the usage of indicators to help with orientation, like finding the starting point. He establishes a simple grammar (Figure 4.4), possessing an alphabet only containing

the three symbols of “wall”, “open space” and “connection”, defining only the connection as a nonterminal. The mentioned orientation indicator is given to the connection, taking the shape of an arrow on a square and being used to indicate the location and orientation of the replaced connection (Dormans, 2010). Dormans (2010) further explains how during the generation process, these connections will be closed by replacing them with basic dungeon structures like a simple wall or a T-Fork, either closing or growing the structure of the level. In an example, Dormans declares a single connection as a starting symbol and explains that the every iteration, the connection to be replaced was selected at random.

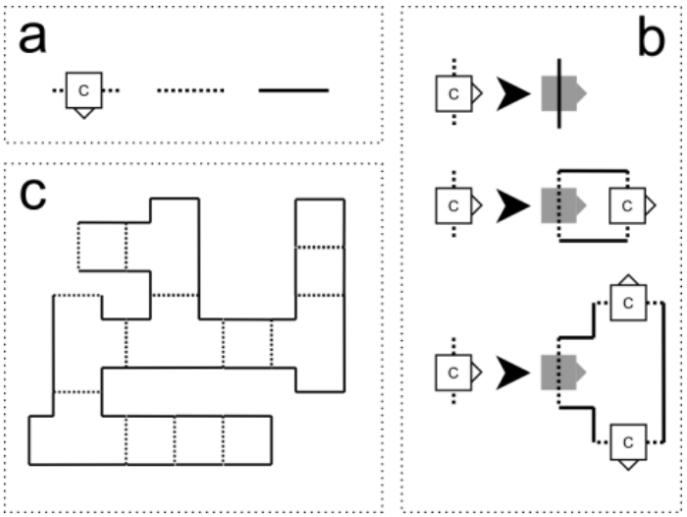


Figure 4.4: Shape grammar alphabet and rules to generate space (Dormans, 2010)

4.3.3 Combining Mission and Space

At the final segment of their study, Dormans (2010) demonstrates the combination of the two generation processes for mission and space. Thy achieved this by attaching mission symbols to the rules of the shape grammar. Their grammar iterates through every symbol in the mission graph generated in advance, filtering out the shape grammar’s rules not coupled to this specific mission symbol and then applies one of them at one of the possible locations the shape could be inserted in, being picked randomly but also based on a relative priority. To realize the before mentioned tight coupling, the generator tracks what shape has been generated for which mission (Dormans, 2010). Dormans (2010) also introduces the dynamic parameters to be used together with shape grammar, to gain more control over the selection of rules. They explain that these parameters will allow the grammar to favor rules that provide a bigger challenge to the player, or switch to a completely different rule set. This function aids in building a progression in the level’s difficulty Dormans, 2010. In addition to this, Dormans (2010) also proposes the use of commands that can be added to a rule and will be run right before the rule is applied. He demonstrates this with a shape grammar rule replacing a connection for a T-fork, tied to the mission symbol “exploration” and associating the

rule with the command “repeat(3)” (Dormans, 2010). In the example Dormans (2010) gives, the exploration symbol comes right after the entrance, which has three different connections, resulting in the placement of three T-forks on the entrance’s connections by technically only applying one rule. He also provides a solution to the problem of the grammar possibly generating a dungeon that has the boss room connected to the entrance or to a room right next to it, leading to a level where the player can just skip all the tasks and directly go to the boss room. Their solution to this, is the usage of a rule command that disables all connections generated up to that point, leaving only connections of newer rooms, further into the mission chain. Similar to tight coupling, this also gives the designer more control about branching inside the level. As stated by Dormans, after the generator iterates through every mission symbol, the grammar can proceed with normal rewriting operations inside the shape grammar domain, with a separate rule set having no associated mission symbols, replacing nonterminals until none are left. Dormans also describes the idea of giving separately generated rooms the ability to reconnect, if specified conditions, like being already well aligned for example, are met.

4.3.4 Lock and Key Mechanisms

Based on the research presented in the previous section, Dormans and Bakkes (2011) practiced further research into the topic, conducting improvements to the approach and also establishing the use of player models, enabling the content generation to adapt to the player’s behavior. Compared to the previous technique, there are two main differences. First, since this time, they focused more on the positioning and order of locks and keys, which led them to break down the mission’s alphabet into something much simpler, only possessing the symbols of “start”, “task”, “lock”, “key” and “goal”. Secondly, space generation was not anymore entirely up to shape grammars, giving the reasoning that they had difficulty “generating spaces for missions which allow multiple paths to converge at the same target”(Dormans and Bakkes, 2011). Their solution to this problem is the prior reorganization of the mission graph into a more organic layout, which serves as the base structure to be accommodated by shape grammars. They state, however, that this technique proved to have difficulties with overlapping connections, but also proposed easy solutions for this problem in the scope of action adventure games, such as the implementation of teleporters or portals for instance. They further explain that after the organic reorganization of the mission graph, task nodes will be replaced with rooms varying in size, containing keys and being connected by locked doors. The next step uses the evolution shape grammars to give these rooms more detail. Concerning locks and keys, they introduce rewrite rules that lead to moving locks towards the front and keys towards the back, ensuring the player comes across the lock first. Another difference to Dormans (2010) approach, is that while he implements the functions of tight coupling and closing off previous connections, to gain

more control about branching, there is no need for this here, since the mission graph already determines the final branching of the level.

4.3.5 Cyclic Generation

Short and Adams (2017) describe the approach taken for *Unexplored* (Ludomotion, 2017), which is a top-down dungeon crawler. They describe a method that takes their previous research about grammars for level generation and the lock and key mechanisms of Dormans and Bakkes (2011) one step further. Joris Dormans (2016) states that, for *Unexplored*, a lot of inspiration has been drawn from Zelda games and that, for this game, the goal was to recreate the feel of their thought-out level design in a procedural generator.

According to Short and Adams (2017), *Unexplored* combines different design patterns to create a level "that feels consistent and creates an exploration challenge beyond a series of individual monsters and hazards" (Short and Adams, 2017, p. 84), again focusing on the design of locks and keys (pp. 87-93), as already previously done by Dormans and Bakkes (2011). Joris Dormans (2016) explains that for *Unexplored*, their goal was to make a level generator that feels handcrafted. They tried to achieve this by developing a level generator capable of generating the advanced lock and key mechanisms Zelda games, and they was heavily inspired by the franchise as a whole (Joris Dormans, 2016).

In the approach used to generated levels for *Unexplored*, Short and Adams (2017) introduce the usage of cycles, which uses two routes to connect the level's start and goal, different from the traditional branching structure using only one route (pp. 84-86). Short and Adams claim cycles to be more effective than branching, stating that they contribute to the level flow and give the player multiple solutions to beat a level (pp. 84-86). It is explained that the implementation of cycles knows multiple ways, dependent on the difference between the two paths (p. 84).

Unexplored implements many different key and lock types that are procedurally generated, directly following many of lock-key concepts found in Zelda games (Short and Adams, 2017, pp. 91-93). Short and Adams (2017) state that for keys, they differentiate between "single purpose" and "multipurpose", "particular" and "nonparticular", "consumable" and "persistent", and keys that are stationary and keys that the player can pick up (pp. 91-93). Locks are distinguished by their safety ("conditional", "dangerous", or "uncertain"), their permanence ("permanent", "reversible", "temporary" or "collapsing"), and their symmetry ("valves" or "asymmetrical") (Short and Adams, 2017, pp. 91-92).

Short and Adams (2017) further describe how to integrate these different types of keys and locks in the generation process, in *Unexplored*'s generator, they can be tagged with a type (pp. 91-93). The same works for room nodes, which could also be of different types (p. 89). To realize these design patterns inside the generative grammar, grammar

rules with specific rewrite operations introduced (Figure: 4.5) (pp. 87-90). These grammars create and expand cycles and then incorporate special level design patterns based on the different lock and key types and other ones like patrolling monsters or a longer safer and a shorter more dangerous path (pp. 87-90). Some patterns even consider what the player can see at certain points, hinting at keys without providing a direct path to them, for example (pp. 87-90).

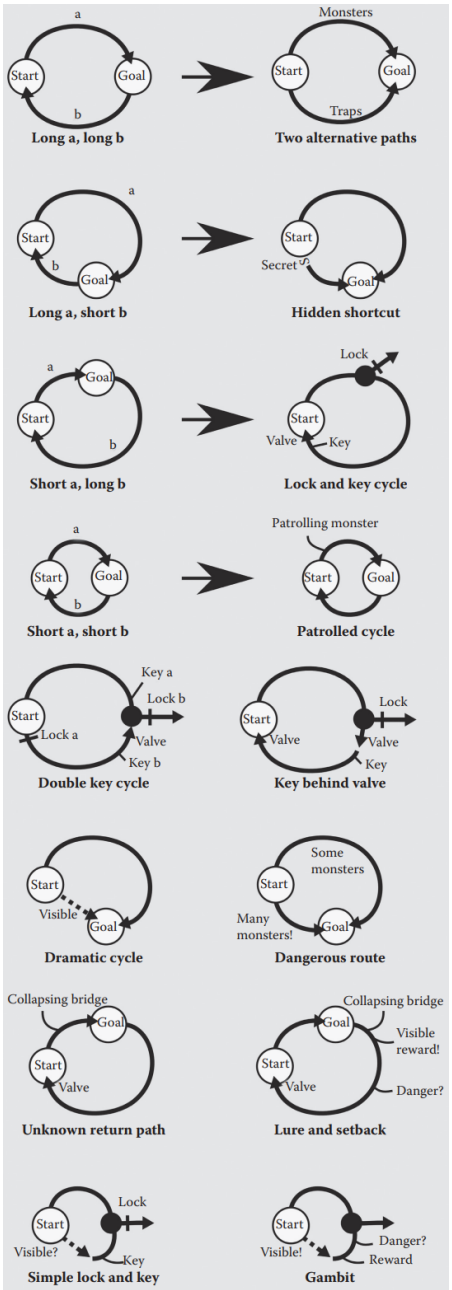


Figure 4.5: Graph grammar rules used in unexplored (Short and Adams, 2017, p. 90)

For the mission generation, Short and Adams (2017) are using graph grammars like the previous approaches (83-90). To make the transition from the generated graph into space easier, the graph will be rearranged along a two-dimensional grid beforehand (p. 94), something already considered by Dormans and Bakkes (2011). Short and Adams explain that all rewrite operations will be performed on that graph, with nodes storing

their room information (p. 94). To translate the mission graph into the dungeon space, a low-resolution tilemap closely following the graph structure will be created, followed by the creation of a high-resolution tilemap to introduce more details to the dungeon (pp. 94-95). After that, other transformation rules will be used for decoration, room expansion, and the incorporation of additional features (pp. 94-95). This last procedure is very similar to how Dormans and Bakkes (2011) used shape grammars. Shape grammars in particular, however, are not directly mentioned by Short and Adams (2017).

4.3.6 Conclusion

With the goal to create levels that feel handcrafted, combined with the heavy inspiration Zelda games, Dormans (2010) developed a procedural generation technique that uses grammars, together with their research about dungeons in action-adventure games, to their full potential and generates structures very close to the quality of the dungeons found in Zelda games, which becomes very clear when looking at the countless techniques and patterns that *Unexplored* (Ludomotion, 2017) is able to generate.
Short and Adams (2017) also state that cycles are found in many different handcrafted dungeon levels (p. 84) giving *Unexplored*'s implementation of other popular design patterns featured in handcrafted games a fitting environment to simulate handcraftedness. *Unexplored*'s usage of many different keys and lock types, some of which are more abstract, takes important design principles from dungeons in Zelda games, which are of very high value when trying to generate similar structures. Another really important thing to note is that the implementation of rules that include showing the player the key without showing the way, is a direct representation of zen views (section 3.3.2), used to encourage the player to explore the dungeon space, inside a procedural system. These grammar approaches seem to be an effective tool when it comes to implementing the previously established level design patterns in procedurally generated levels. Being able to place rewards like keys behind a task (Dormans, 2010) incorporates an objective-challenge-reward structure and being able to force locks being encountered before keys (Dormans and Bakkes) not only incorporates a objective-challenge-reward structure, but also encourages the player to explore the level. Additionally, the ability of switching between rule sets to implement a scaling in difficulty (Dormans, 2010) gives important control over the level flow. Shape grammars seem highly customizable, making them a useful a tool to implement different gamespaces and alternate between them. The high customizability of shape grammars and the possibility to design them by hand, together with the the high control that comes from grammar rules (Dormans and Bakkes, 2011), also seem useful when trying to get more organized object placement and implementing object purpose. Furthermore, the high control that comes with grammar rules should also prove practical for implementing a good balance of risk and reward. Generally, Dormans and Bakkes (2011) emphasize that, with the right rules,

"the level design principles that drove the design of the Forest Temple level can be easily codified" (Dormans and Bakkes, 2011), further implying that this method is a useful tool to generated seemingly handcrafted levels.

Chapter 5

Generator Concept

Based on the research about handcrafted levels and the research about suitable procedural generation methods, a generator expected to generate seemingly handcrafted levels has been created. The generator mainly follows the grammar approach presented by Dormans (2010). Numerous other approaches like the one used in *Dungeonmans* (Adventurepro Games LLC, 2014), as presented by Short and Adams (2017), or the one presented by Hartsook et al. (2011), have also been used. The result is a mixed-initiative tool with a high degree of handcrafted integration. More specifically, the generator combines the analyzed grammar approaches (4.3) with the approach of *Dungeonmans* (Adventurepro Games LLC, 2014) (Short and Adams, 2017) and uses the concept of *Game Forge* (Hartsook et al., 2011) to connect mission and space along a defined narrative, as seen in the examples of *Spelunky* (Mossmouth, 2008) (section 4.1.2) and *FTL: Faster Than Light* (Subset Games, 2012) (4.1.1).

5.1 Grid Generation

First, it was relevant to implement a grid system. Since the game would be only consisting of elements distributed a 2D grid, the grid has to cover the X and Y axis of the game world. Each grid-tile is 100 units long and wide. To place objects in a grid, dependent on their position inside it, objects have been placed in steps, 100 unit large, towards the worlds x and y directions.

5.2 Level Seed

After the implementation of the grid system, there was a need for information that the grid system can use to distribute objects. Inspired by *Dungeonmans* (Adventurepro Games LLC, 2014), the level generator follows an ASCII approach. The entire level information, including every tiles position and object type, can be represented by a seed. This seed will always rectangular. With this functionality, one can save the generated level as a text file and enter it to build the level without generation. In the level seed, each character represents one tile, the character representing the object inside the level.

5.3 Grammar Generation

The generator is largely based on the grammar-based approaches analysed in section 4.3, following the original approach of Dormans (2010) closely. However, within the scope of this thesis, graph grammar generation has not been implemented. Instead, to enable more opportunities in terms of control, an approach similar to the method presented by Hartsook et al. (2011) has been used, where designers can define a sequence of missions in order.

5.4 Alphabet

The grammar's alphabet consists of the many object types able to be placed inside the level. Additionally to only the corresponding ASCII character, the alphabet object has also been given the type of tile the ASCII character represents and the blueprint actor class representing the object in the final level when built. For example, tiles could be "wall", "floor", "enemy", "key".
It is important to note that not all tile types correspond to object types in the final level. Some tile types are only relevant for the level generation process and not objects being placed in the level. These tiles would be "start", "padding", "connection", and "closed connection". The "start" tile is responsible for the spawn of the player, marking the start position in the level. Tiles of the "padding" type only serve the purpose of making seeds that represent nonrectangular structures rectangular. This provides the opportunity to define nonrectangular shapes in the design process, leading to much more control in the design process, similar to *Dungeonmans* (Adventurepro Games LLC, 2014). Connections function the same way the connections in the approach described by Dormans (2010), serving as anchor points new rooms can attach to. A closed connection is a connection that is not enabled, thus being excluded from being chosen to attach a new room to the structure. The purpose of this further explained in section 5.7.

5.5 Rules

Grammar rules consist of one LHS and one or multiple RHS. Each LHS and RHS is presented by a pattern object. These pattern objects hold multiple types of data:

- Tile type
- Seed
- Seed dimensions
- Tiles information

- Numbering seed

This data is relevant information for the generator to properly apply grammar rules to the seed.

5.5.1 Pattern Seeds

The pattern seed holds the information about the actual object that would be placed inside the level. This is the tile-pattern that will replace the previous pattern inside the seed, when chosen to be applied
This seed will always be rectangular. To guarantee a rectangular seed while still defining a nonrectangular pattern, one can use the previously mentioned "padding" tiles, which are not objects to be placed and only exist to make the seed rectangular. This is necessary to provide relevant functions, like locating characters inside the seed at a certain position or converting the index of a character inside the seed to be represented as a grid slot, providing the column and row in which the desired character can be found inside the seed.

5.5.2 Seed Dimensions

The seed dimension of a pattern holds information about the number of columns and rows the rectangular seed consists of. This is essential to calculate grid slot information, since seeds are not kept in there rectangular form, but will be transformed into "straight seeds", getting rid of all newline characters, making them more compatible with the generation process.

5.5.3 Tiles Information

A pattern also holds information about all the tiles it contains. Every tile object possesses multiple properties:

- Tile type: The type of the tile.
- Character: The ASCII character it holds.
- Grammar numbering: The tile's number in the numbering seed. Only the key character will be numbered. This helps finding the key character of a rule by searching for the tile with "1" as the grammar numbering.
- Index in string: The index of the tile inside the seed.
- Position in grid (grid slot): The position of the tile as a grid slot, saying in what row and column of the pattern seed the tile is located.
- Relative position to key character: The position of the tile, relative to the pattern's key character.

5.5.4 Numbering Seeds

Numbering seeds are essential for the pattern detection process. They have the exact same dimensions as the pattern seed, but mark one character as the key character. This is done by using the character "1" as tile, instead of the character that would represent the object itself. This numbering process is similar to the node numbering mentioned by Dormans (2010). By marking one tile as the key characters, patterns will provide a starting point all the other tiles can refer to, helping with coordinating the pattern structure when detecting a pattern, inserting a pattern, and checking if the pattern would fit into the seed in the first place.

5.6 Seed Setup

Before the generation process starts, all specified rules will be prepared in a setup process, preparing the rule's patterns for the generation process and calculating necessary information. This process will first calculate the seed dimensions, using the newline characters in the seed. Then the seed will be "straightened", getting rid of all newline characters to make it readable for grammar generation. Afterwards, every tile's tile information, as explained in section 5.5.3, will be calculated and saved in the rule. For the tiles information, it will also be calculated what relative position tiles have to the seed's key character. This will be important for pattern detection, replacement and insertion. Additionally to the calculation of all the rule's information, the setup process will also create versions of every rule rotated by 90°, 180°, and 270°. This makes rule's be independent of the rotation they have been designed in, practically making them able to be rotated so they fit with any of the four angles.
This setup process makes it easy to design rules, only giving the designer the task to define a seed and the seed's key character with the numbering seed. Since the seed will be transformed to a straight seed automatically, designers can design the seed as rectangular, which represents how it would look inside the level.

5.7 Mission

As stated in section 5.3, the procedural generation of a mission using graph grammars has not been implemented. Mission do, however, exist inside the generator and can be specified. Instead of being based on graph grammars, for missions, it has instead been taken inspiration from Hartsook et al. (2011). One can build a mission by defining a sequence of mission symbols. With this, the end result will not be much different compared to Dormans (2010) approach, since the mission there, even though it originally is a graph, also results in a sequence of symbols. This makes the only difference that missions will not be procedurally generated, but instead entirely crafted by hand.

Similar to tiles, there are two different types of mission symbols, symbols that represent rooms in the dungeon and symbols that only exist for generation purposes.

5.7.1 Room Mission Symbols

Room mission symbols are responsible for building the dungeon's architecture. All of these symbols have a rule set tied to them, the rule set being created and set by the designer. The designer can define multiple different mission symbols that will tell the generator to place rooms defined in the rule set that is tied to the mission symbol. For example, mission symbols could be "entrance", "loot", "enemies", "key", "item", or "goal". The designer can model multiple rooms in the rule sets tied to these mission symbols, which will then be randomly chosen to be placed.

5.7.2 Generator Mission Symbols

Compared to room mission symbols, generator mission symbols work different. They exist to provide more control over the branching of the dungeon and order of rooms. They also give the designer the opportunity to specify in what order the player should come across certain rooms. This is important when wanting to implement the level design pattern of encouraging exploration (section 3.3.2).
First, there is the "close all connections except last room" symbol. As mentioned in section 4.3, this is an idea already considered by Dormans (2010), presented as a method to prevent "accidently connecting the final room to a room near the entrance" (Dormans, 2010). Closing all connections except the ones in the last room assures that every placed room from that point on will be only reachable by traversing the room placed before the symbol was used. In addition to that, there is the "toggle connections" symbol. This symbol can open or close connections of a specified room. The designer can specify the target room in the mission symbol itself. Closed connections will not be able to connect a new room to the dungeon. Tight coupling, as proposed by Dormans (2010) and covered in section 4.3, has also been implemented in the generator. For this generator, however, its functionality has been expanded, allowing designer to specify which target room the room should be tightly coupled to. This allows the designer to force a room to connect to a connection of the last placed room.
In addition to the branching settings, mission symbols also have a value that defines how often a room should be evolved.

5.8 Room Evolution

After the dungeon rooms have been generated, the room evolution process starts. This process applies as separate set of evolution rules to each room.

5.9 Rules Application

When the generation has been initialized, an empty seed with a connection symbol at the center will be generated. The dimensions of the seed are set in the generator's settings. This connections symbol will serve as the starting point for the first placed room to connect to. To build the dungeon, the generator will go through the specified mission sequence. For each mission symbol in the mission sequence, one grammar iteration will be ran. Each grammar iteration, the generator will search for rules that can be applied to the level. This is done by first getting every slot in the seed that holds a character appearing as the key character in the LHS of a rule. Then, for each slot, it will be checked which of these rules hold a pattern on their LHS that it is actually present in the seed at that exact spot. These rules will be saved.
Subsequently, for every rule, it will be checked if its RHS's will be able to be inserted in the seed without overlapping other tiles. This will exclude tiles of the pattern about to be replaced. For every rule, fitting RHS will be saved as candidates.
Lastly, to apply a rule to the generation, one random rule will be selected. If the selected rule has multiple RHS that are candidates, one random RHS will be chosen. With the application of the rule, every seed tile included in the rule's LHS will be replaced by the chosen RHS. This process can not only add, but also remove tiles.

5.9.1 Pattern Detection

To detect a LHS pattern inside the seed, an algorithm has been written using the relative position of tiles that is stored in the pattern's tile data and was calculated beforehand. In this process, tiles in the pattern will be compared to tiles in the seed, relative to the seed slot that is currently checked. If every tile is identical, the LHS is marked as found inside seed.

5.9.2 Replacement Check

To replace a pattern, it will be first checked if the necessary space is available. This process ignores all empty space tiles and tiles that belong to the pattern that would have to be replaced, if the rule would be chosen. The algorithm iterates through every tile that would be added to a slot not already possessed by previous pattern, checking it would place a tile on another tile that belongs to a different pattern. If that is the case for any tile, the RHS of this rule will be discarded and not be marked as a replacement candidate.

5.9.3 Pattern Replacement

After every rule that includes a LHS found in the seed and an RHS able to fit into the seed, has been found , a random rule will be applied. To replace the pattern, the pattern's relative tile information will be used again, inserting the RHS of the

chosen pattern to the level seed. To provide the ability to change tiles of pre-exising patterns, instead of only adding new ones, tiles of the previous pattern (LHS) will also be overwritten.
For room placement, all newly placed tiles will be saved as an additional pattern, only storing the room. This is needed for the grammar evolution inside the separate rooms.

5.10 Level Building

After the seed has been generated, the level will be build using the associated objects to be placed along the previously mentioned grid.

Chapter 6

The Humanized Level

This chapter will cover how the humanization techniques inhabiting level has been generated. This level will be referred to as the ”humanized level” (Figure 6.4). To build the 3D representation of the level, assets from different asset packs have been used (Miguel Lobo, 2019; Rick N. Bruns, 2023; Rick N. Bruns, 2019a).

6.1 Game Concept

To conceptualize the generator, it was evident that a game concept was required. Beyond the fact that one can only generate a level when a game concept is present, furthermore, to evaluate, it was also mandatory to be a level able to be played.

Since research already focused on dungeons and more importantly the structure of those found in Zelda games, the game concept to generate a level for, would also be very close to the gameplay of a Zelda game. Largely based on the analyses presented in section 3.1, the game concept would consist of multiple key aspects that define the gameplay of Zelda games. Around these key aspects, a small game called “Sword Quest” has been conceptualized.

6.1.1 Game Dimension

The game is three dimensional. Although top down games originated in a two dimensional space (Epyx, 1980; Nintendo, 1986), a top down game is only defined by its top down camera view.
The fact that the previously done research about level design was largely based on *The Legend of Zelda: A Link to the Past* Nintendo, 1991, a 2D game, will also not be problematic, since the analysed level design techniques are independent of the dimensions they are used in. Another reason why the shift in dimension will not affect the research, is that, while games like *The Legend of Zelda: A Link to the Past* are essentially 2D, they do, in fact, simulate a third dimension in 2D space. This makes the translation of the research done on 2D games to 3D unproblematic. An example for this is *The Legend of Zelda: Link's Awakening* (Nintendo, 2019), which is a 3D remake of *The Legend of Zelda: Link's Awakening* (Nintendo, 1993) for the game, a

game that was originally 2D.
It is important to note that, to keep experiments simple, the level to be generated will not make use of the third dimension in terms of gameplay. Despite the game using 3D assets, it will still function the exact same as a 2D game, being strictly projected on a 2D grid. While some objects, like walls, will be higher than others to model the architectural space, they are non-interactable and only for visual purposes.

6.1.2 Camera Perspective

Since this research is about top down games, the view of the game would also be in the same perspective. It is to note that the game view is not entirely top down, but instead tilted by a small degree, to get a better view of the 3D models making up the level, making identifying them much easier.

6.1.3 Player Character

The character's movement was held simple. When looking at the character abilities featured in *The Legend of Zelda:A Link to the Past* (Nintendo, 1991) the character's abilities are mainly provided by the collected items, expanding the characters abilities a lot. Without any collected items however, the character can only walk (Nintendo, 1991). Since the generated level would follow this very same structure of expanding the character abilities by using items, the basic movement abilities are only walking.
The player character has five health points and can be damaged. It the player health reaches zero, he will die and respawn at the beginning of the level with restored health points. The death of the player will not reset the level.

6.1.4 Locks and Keys

Since locks and keys were a big part of the research done in this thesis (Dormans and Bakkes, 2011)(Ludomotion, 2017), they will also be included in the game's mechanics. They will work in a simple way, where player just has to walk through the key to pick it up. It will then be automatically be added to the player's inventory, automatically unlocking the locked door if he stands next to it. Only one key and one locked door will be present in the level.

6.1.5 Items

Like already mentioned, items play an significant in a Nintendo, 1986-2023 dungeon. Dormans (2010) presented the importance of items, by emphasizing that they can not only be used as a weapon, but as a key as well. Since this generator has the goal to implement this concept into the generated level, the player will be able to pick up an item. This item will be a tool for players to defend themselves, but also will be mandatory to be used to progress through the level. As soon as players acquire the item, they will gain the ability to shoot projectiles. This projectile can be used as

many times as the player desired. This is, however, restricted to an ability cooldown, so the player can only use the ability in a specified frequency.

6.1.6 Enemies

While traveling through the dungeon, the player will encounter enemies. These enemies are stationary and, similar to the player, also have the ability to shoot projectiles. They will only shoot if the player is near them, within a specific range. If a projectile hits the player, they will lose one health point. Additionally, player will also lose one health point, if they touch an enemy. Enemies have only one life point and, when hit by the player's projectile, die and get destroyed immediately.

6.1.7 Rewards

To provide some type of additional, but optional, loot to the player, they can collect potions. These potions will restore health that has been lost in enemy combat. One potion will restore one health point. Potions are only found in reward-rooms. These rooms exist to reward players that take a risk by exploring the level more than one needs to, inhabiting potions to heal the player by a high amount.

6.1.8 Boss

To finish the level, players have to beat the dungeon boss. This is an enemy very similar to the ones distributed across the dungeon, with more health points and also stronger and bigger attacks. If the boss is defeated, it will drop an escape portal that, when entered, makes the player finish the level.

6.2 Level Concept

This section will cover how the level for Sword Quest has been designed, talking about the alphabet and rules that have been set up and how they were used.

6.2.1 Alphabet

Apart from the default generator tiles, the grammar alphabet consists of architectural tiles like ”ground”, ”wall”, ”door”, ”lock”, ”hole”, ”fence”, and ”pillar”. The empty space tile has been used for walls, filling out everything where nothing has been placed, effectively closing rooms. Walls are not traversable. The separate wall tiles were actually not used but were originally planned to be walls that could not be overwritten, which is not the case for empty space tiles. Door tiles use the same objects that ground tiles did but serve the purpose of still being able to identify connections between rooms after rooms have been generated. This is important for the room evolution. The lock tile represents a locked door that can only be opened when players possess the key. Holes are walls that represent a hole in the floor, acting the same as walls. Fences

and pillars are non-traversable tiles that can be placed within a room. Fences use an event that automatically rotates them correctly after the level generation process is finished. If the algorithm detects a corner in the fences, that fence gets replaced with a pillar tile, which is actually not placed during the generation process. It is important to note that connections also use the wall object, which effectively closes rooms with open connections remaining after the generation process is finished.
As for non-architectural tiles, there are “enemy”, “key”, “ability”, “potion”, “boss”, “potential enemy”, “potential potion”, and “goal”. Key and ability tiles are only placed once in a level, representing the key to the locked door and the ability players can defend themselves with, which is represented by a sword. Enemy tiles represent the enemies attacking the player, while potion tiles represent potions that heal the player when collected. The potential enemy and potential potion tiles serve the purpose of not fully representing an enemy or potion placed in the level but can be evolved into one during room evolution. Lastly, the boss tile represents the boss necessary to defeat, which leaves behind the goal that the player needs to collect to beat the level.

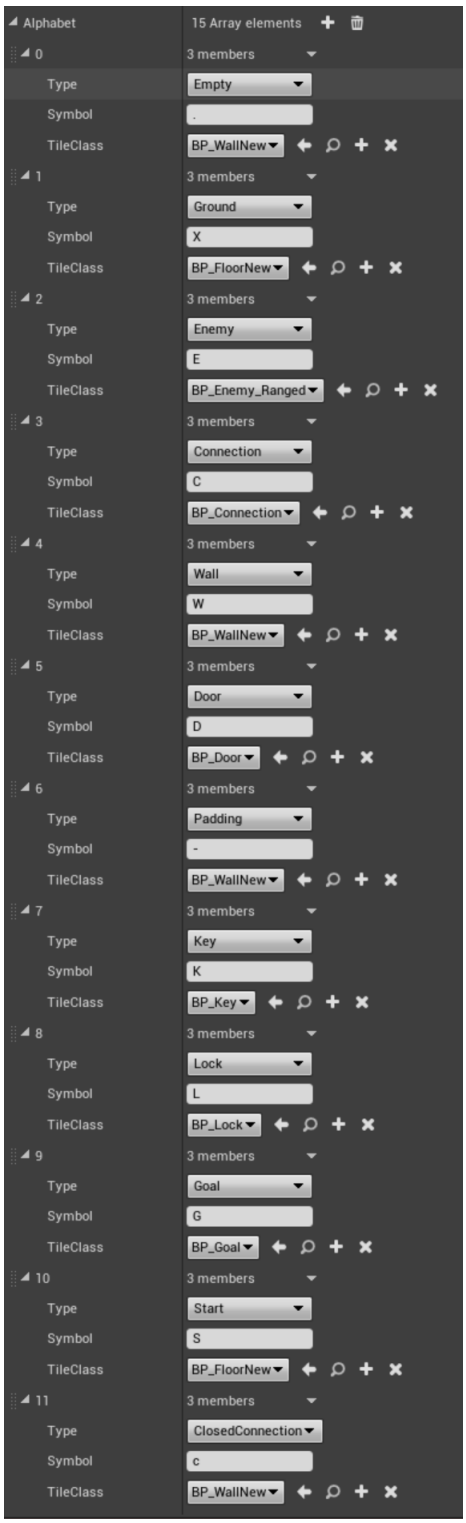


Figure 6.1: Part of the designed grammar alphabet

6.2.2 Mission Symbols

For mission symbols, which, except for the default ones used to steer generation, represent rooms, there are “entrance”, “key”, “ability”, “small exploration”, “big exploration”, “maze”, “enemy lock”, “room after enemy lock”, “reward room”, “lock”, and “boss”. The entrance contains the “player start” tile and is a neutral room where the player can safely spawn. The exploration rooms are rooms containing enemies and

potions. The key and ability rooms contain their associated items. The enemy lock room is a room where enemies stand in the way of the player, stopping them from progressing to the next section. The “room after enemy lock” room is located right behind the enemies that block the door and contains potions. Mazes are rooms where enemies and potions are located at the ends of its branches and where players have to find their way to the next room. The lock room is a neutral room containing nothing but a locked door, which leads to the boss room, where players have to fight the boss. Lastly, there is the reward room, which is a room containing only potions. All rooms are identical in shape and size, except for the maze, big exploration, and boss rooms.

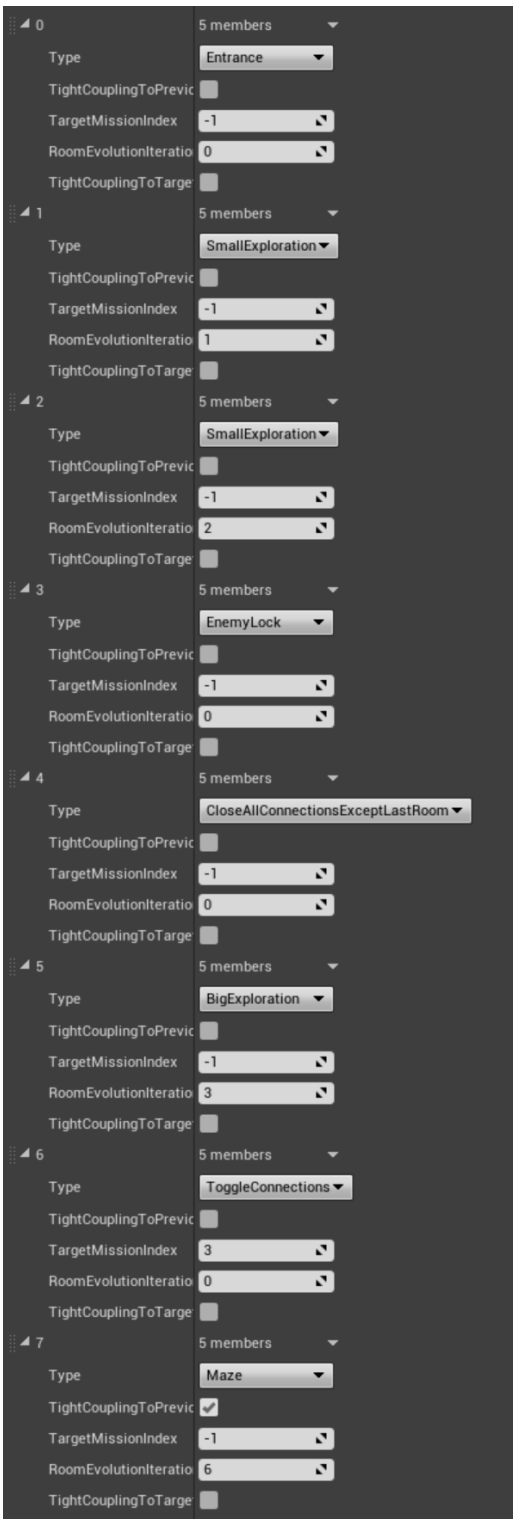


Figure 6.2: Part of the designed mission sequence

6.2.3 Rules

Room rules mainly have a single connection symbol in their LHS so they can attach to a room with open connections. An exception to this is the boss room, which can only connect to a lock tile. The RHS of their rules containt rooms in different variations, only differing in the objects they contain and not in size. For example, exploration rooms differ in the positions of the enemies and potions that will later be evolved and

potions are not included in every exploration room. The maze uses the potential enemy and potential potion to increase variety in the entirely predesigned maze structure. The reward room and "room after enemy lock" room rules contain different variations containing random potion patterns and amounts of potions.
The room evolution rules contain patterns that will evolve enemies and potions. Evolving these will always result in a pattern that contains more enemies than before. One of these rules uses fences and pillars to guard the enemies inside them. They also contain rules to evolve the key and the ability to place enemies around them. The door tile has been used to create rules that place and evolve enemies blocking it.

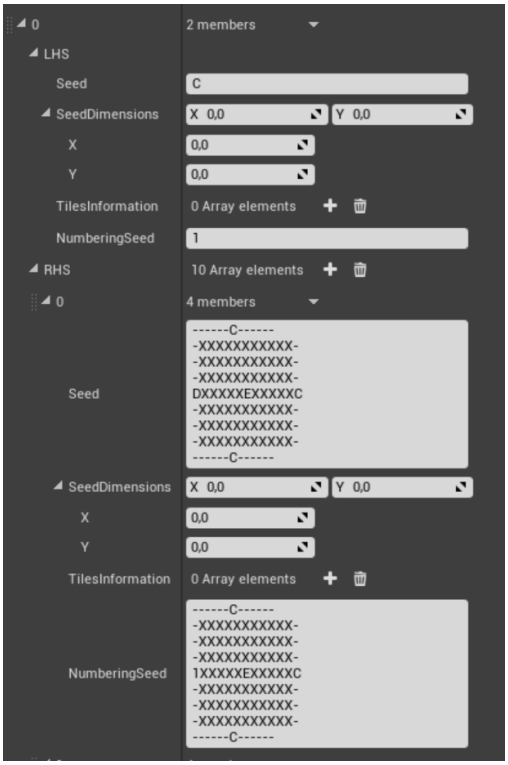


Figure 6.3: A rule to place a small exploration room

6.2.4 Level Runthrough

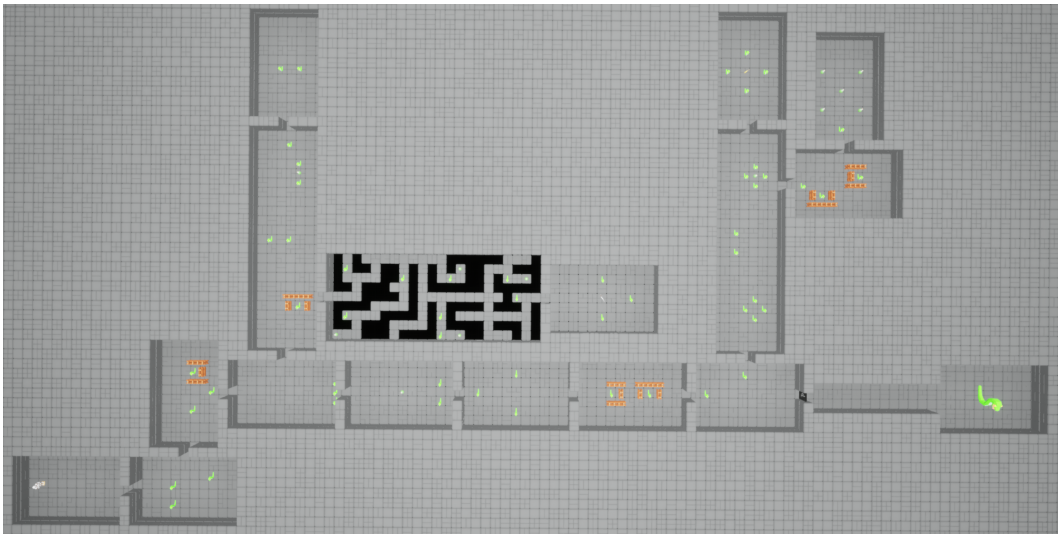


Figure 6.4: Humanized level

The player will spawn at the entrance. From here, they are free to explore the dungeon. At the beginning, players can not defend themselves and have to dodge the enemies' projectile attacks. The first key destination is the enemy lock room, where enemies are blocking the way to the next room. This plans to encourage the player to look for something that will make him get past the enemies. This will make players pass a bigger room with more enemies, making it more dangerous than other rooms. Afterwards, they will come across the maze (Figure 6.5). In this maze, they need to figure out the way to the next room, while being attacked by enemies. The maze randomly places enemies and health potions at the ends of its branches, which results in some hearts being accessible with the risk of being attacked by other enemies and some potions being protected by enemies, making them impossible to reach yet. After the player finds the exit of the maze to the next room, they reach the room where the sword is placed (Figure 6.5), which gives them the ability to shoot projectiles as well, which they can use to defend themselves against enemies. The sword itself is also guarded by enemies, making it a challenge to acquire it. After acquiring the sword, the player can now make their way back to the enemy lock room to defeat the enemies blocking the way. On their way back through the maze, they have the opportunity to recover health points by defeating the enemies blocking a health potion on the other maze branches.

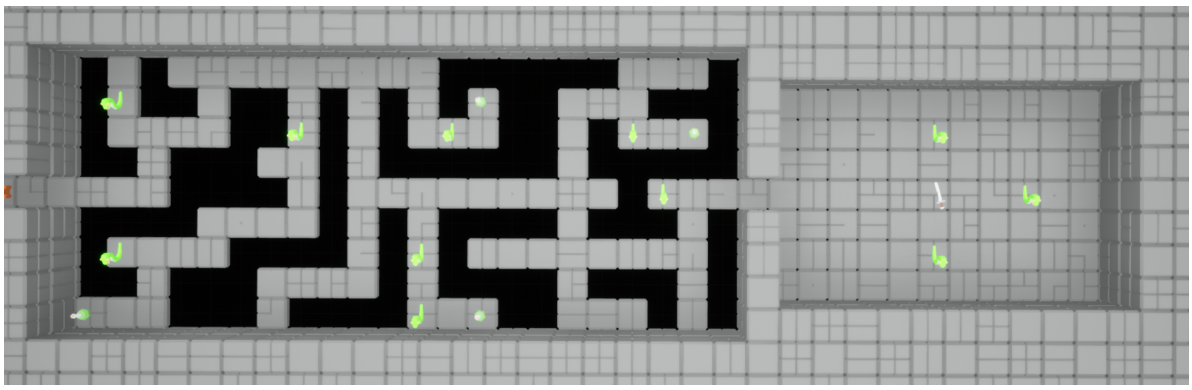


Figure 6.5: Maze that leads to the sword

After defeating the enemies blocking the path, they can enter the second segment of the dungeon and get a random amount of health potions to collect. Next, players will reach the room with a locked door, encouraging them to search the dungeon for the key. Like in the first dungeon segment, the player must now again cross a big exploration room to find the key. This big exploration room, however, will have a reward room attached to an exploration room. This will reward players for taking the risk of proceeding through these rooms. Another room directly attached to the big exploration room space will include the key, which is protected by enemies, similar to the sword (Figure 6.6). After acquiring the key, the player can go back to the locked door and go through it.

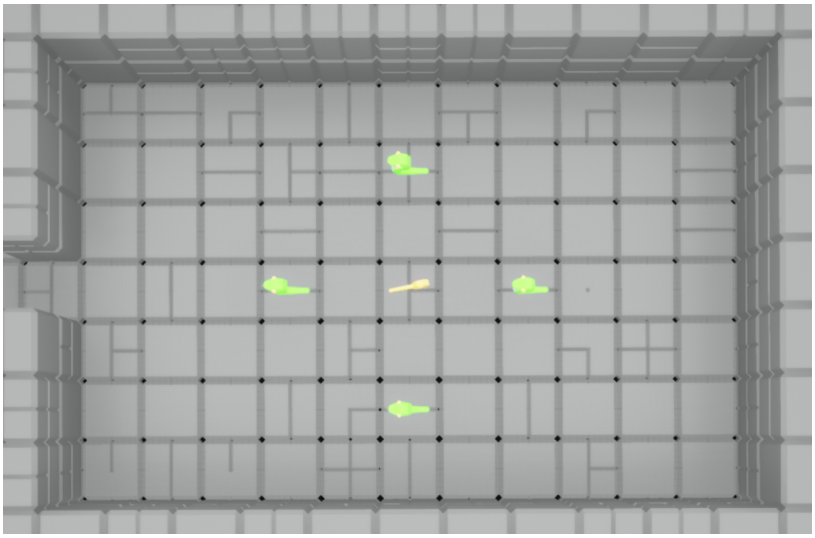


Figure 6.6: Key, protected by enemies

They will then enter the boss corridor with the boss room attached at its end (Figure 6.7). After defeating the boss, it will spawn the level goal, and the player can complete the level. The entire dungeon structure has exploration rooms randomly placed in parts of the dungeon, giving the level a higher exploration value and to providing the possible scenario of players not walking in the right direction the first time.

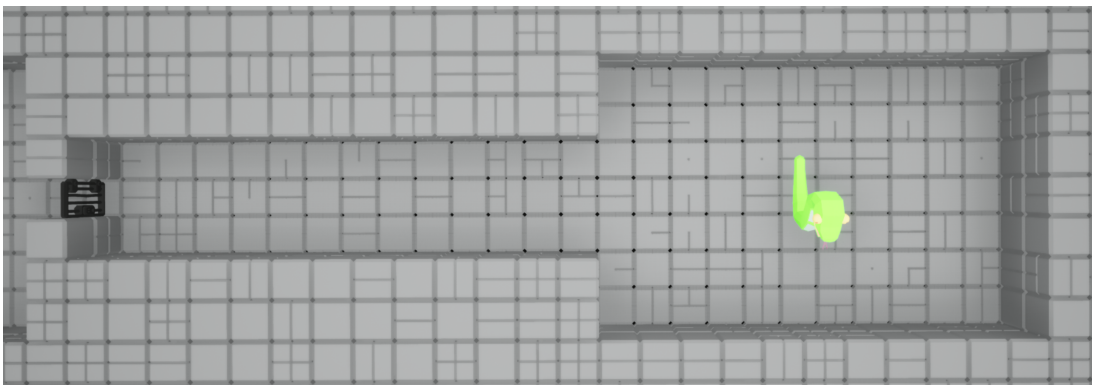


Figure 6.7: Boss room

To design the level, the branching structure of the dungeon was first sketched on paper as a graph. This branching structure mainly represented the segments of the dungeon that have been generated with the bottleneck rooms, which were the "room after enemy lock room" that would start the next segment. Optional exploration rooms have also been included in the graph, but their actual branching would not be determined. It was only the number of rooms per section that was specified.

6.2.5 Applied Humanization Techniques

Many of the researched humanization techniques have been tried to be applied through the design of the level. This segment will present how this has been done.

Gamespaces

Many different gamespaces that were mentioned in section 3.3.1 have been utilized. Most of the rooms are narrow refuges. The big exploration rooms represent big prospect spaces. As already mentioned, a maze has also been included in the level.

Critical Path

It has been tried to achieve a similar branching nature as the one presented in the analysis done by Stout (2012). The goal was to have critical path with optional rooms. For this, the previously mentioned generator's branching tools were used.

Kihon-Structure

A structure similar to the kihon-structure Dormans (2010) found in their analysis (section 3.1.1) has been recreated using the generator's branching tools. This structure also lets the player first encounter the enemy lock room, which they can only pass after acquiring the sword that lets him defeat the enemies blocking the way. With this, the concept of using items as keys has been implemented. This indirectly teaches him how to use the weapon. The boss serves the purpose of testing how well the player learned to use the weapon.

Increasing Difficulty

Additionally, an increase in difficulty has been implemented. This works through the room evolution, which gradually adds more enemies with every evolution step. With this, one can specify an increase in difficulty by increasing the evolution steps of rooms towards the end.

Difficulty Breathing

Difficulty breathing has been implemented in the form of gamespaces. Big exploration rooms, which represent prospect spaces, contain more enemies than small exploration rooms, which represent refuges. Additionally, the maze is also supposed to be a more difficult segment than the rest of the dungeon.

Encouraging Exploration

Unfortunately, since the functionality of reconnecting rooms like presented by Dormans (2010) could not be implemented within the scope of this project, zen views, as discussed in section 3.3.2, could not be implemented in the generated level. However, the concept of ensuring that the player comes across the lock before the key, as presented by Dormans and Bakkes (2011), has been used as a substitute, since it essentially has the same effect of encouraging the player to explore. This is done by hinting to the player that a reward, in this case the sword or key, is somewhere in the level.

Objective-Challenge-Reward

Structures of objective-challenge-reward have also been implemented. This mainly revolved around the sword and the key, which reward the player with progress or a new ability. In the example of the sword, players are first given an objective by encouraging them to explore the dungeon through the enemy lock, then faced with a challenge, which is the maze, and lastly rewarded with the sword.

Object Purpose & Object Structure

Rules have been set up in such a way that many of them serve the purpose of protecting something. The previously mentioned rules that make enemies block doors stand in the player's way of progressing to the next room. Enemies that evolve from potions will also always be positioned in a way that they protect that potion.
Room rules have generally been set up in a tidy structure and evolution rules always evolve in a symmetrical manner.

Balancing Risk and Reward

In the dungeon, players rarely get something without a challenge, which creates a balance between risk and reward. Since potions are mostly protected by enemies, players always take the risk of losing health when trying to acquire them. Additionally, players risk losing health when exploring the branches of the maze or trying to acquire the potion at their ends. The reward room also creates a balance between risk and reward by rewarding players for exploring an area of the dungeon where no key is found.

Player Emotion

Similar to the technique mentioned in section 3.3.3, the maze here also serves the role of using player emotion when player's lost much health beforehand. Similar to the example in *The Legend of Zelda: A Link to the Past* (Nintendo, 1991), here, players also get rewarded with a new tool when finding their way through the maze.

Giving the Player Hints

Giving the player a hint has been implemented in a similar manner to the boss corridors mentioned in section 3.3.6. Here, the boss room also has a long corridor signalizing that a strong enemy might be at the end, giving players time to prepare.

6.3 Evaluation

6.3.1 Non-humanized Level

Additionally to the humanized level, a level without any humanization techniques has been created (Figure: 6.8). This has been done by creating a separate set of rules, generating everything almost entirely random. Also, the alternation in gamespaces has been completely left out, only using rectangular rooms identical in size. Since the level was randomized and there was no enemy lock, to let players not be able to enter the boss room by finding the key before the sword, the functionality has been added that the locked door only opens when players possess both items. The purpose of this level is to serve as a negative example, compared to the humanized level, assuring more accurate evaluation results.

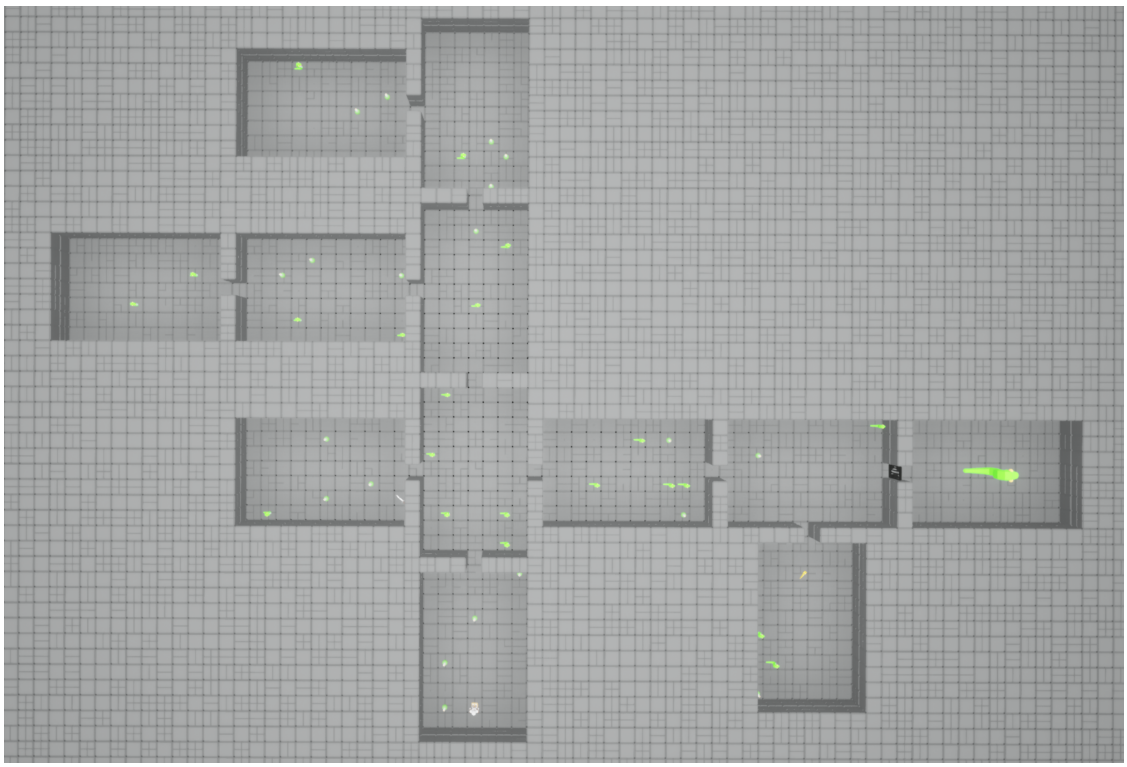


Figure 6.8: Non-humanized level

6.3.2 Evaluation Method

For evaluation, testers between the ages of 20 and 28 were invited into a room providing a PC and the playable levels, so every test would be done under the same circumstances. 8 of the testers were male, 2 were female. Since many of the humanization techniques were advanced level design methods, all testers were required to have experience playing video games. Beforehand, the rules and controls, of the game have been explained. Testers could use the “W”, “A”, “S”, and “D” keys to walk around and the spacebar to attack as soon as they had collected the sword. To evaluate the handcraftedness of the generated level, testers were given both levels to play.
Evaluated were the following aspects:

- Flow
- Balance of risk and reward
- Player Emotion
- Encouragement to explore
- Structure and symmetry
- Enemy purpose
- Giving the player hints
- Resemblance to dungeons from Zelda games

- If testers thought the level was created by a human or an algorithm

To evaluate the flow of the level, questions have been asked regarding clear goals, if they ever felt lost, difficulty scaling, difficulty breathing, and if the structure of objective-challenge-reward was present. To evaluate the balance of risk and reward, testers were asked how rewarded they felt for taking certain risks, including the risk taken when exploring the dungeon. For player emotion, players have been asked how they felt when wandering through the maze. The questioning regarding encouraging exploration focused on if players encountered something encouraging them to explore the dungeon, like the enemy lock and the locked door. Structure and symmetry were evaluated by how symmetrical and structured testers perceived the enemy and potion patterns. Regarding enemy purpose, testers were asked if they felt like enemies stood in their way of getting where they wanted. Testers were asked how surprised they were by the boss when going through the locked door. The use of gamespaces was not evaluated individually, but rather through the aspects of difficulty breathing and player emotion, which it results in.
Additionally to the other questions, testers were also asked two more general questions. First, they were asked if they have played a Zelda game. If they had, they were asked how closely they think the humanized level resembled a dungeon from *The Legend of Zelda*. Lastly, testers were asked if they ultimately think whether the level had been created by a human or if it was generated. All questions have been asked after the completion of each level.
It was originally planned to let testers first play the non-humanized level. However, testers would expect the boss after playing the first level, when encountering the locked door in the second level. This would result in inaccurate evaluation results. To avoid this, 5 of the 10 testers have been given the humanized level first. The other 5 testers were given the non-humanized level to play first.

6.3.3 Evaluation Results

In all categories, the humanization techniques implemented in the level were much more recognized by the testers in the humanized level than in the non-humanized level. However, 3 of the 10 testers have stated that they have not seen many differences in the object structure and symmetry. Testers felt much more rewarded for taking risks when playing the humanized level than when playing the non-humanized level. 9 of the 10 players were encouraged to explore further when meeting the enemy lock and the locked door in the humanized level. They also understood what to do, which shows that players got given clear goals. Out of the 10 testers, only 2 claimed that they encountered things encouraging them to explore. 5 of the 10 testers felt lost in the non-humanized level at some point. For the humanized level, not one of the testers stated that they ever felt lost. Regarding the boss, 3 of the 5 testers that were to first play the humanized level had a feeling they were to encounter the boss. On the other

hand, every single one of the testers who were first given the non-humanized level was surprised that the boss appeared and felt unprepared. Regarding player emotion, 4 of the 10 testers stated that they were nervous to die when encountering the maze. Testers stated they felt an increase in difficulty and also noticed difficulty breathing in the humanized level. Only 2 testers felt this was present in the non-humanized level. A presence of enemies blocking the player was acknowledged by all testers when asked about the humanized level. However, 5 testers agreed that this was also the case for the non-humanized level. Lastly, although many testers were not sure whether or not the non-humanized level was created by a human, all of them were convinced that the humanized level was.
All of the 4 testers that have played a Zelda game before stated that they saw a high resemblance to their dungeons in the humanized level, one person even mentioning a specific one they got reminded of while playing. This was not found to be the case for the non-humanized level.

6.4 Discussion

It is to conclude that the found methods of vertical and horizontal handcrafted integration (section 4.1) and the grammar approaches presented in section 4.3, were efficient in implementing the humanization techniques. Encouraging exploration, balancing risk and reward, giving the player hints, object purpose and using player emotion were found to be very present in the humanized level and lacking in the non-humanized one, which already shows the presence a high handcraftedness. Generally, the presence of flow was also much more perceived in the humanized level. The results also conclude that the humanized level possesses a structure similar to Zelda game dungeons, proving that the structure of Zelda dungeons has been successfully incorporated. When telling testers that the humanized level was created by an algorithm, many were surprised. This validates that the overall level could easily be mistaken for a handcrafted level.
However, when looking at the results of the evaluation, it becomes clear that the handcraftedness of the humanized level was not optimal. This results from some testers perceiving it as having a similar structure to the non-humanized level, which also applies to object purpose. A potential reason for the lack of symmetry and structure could be the enemies blocking doors, which seem to merge with the other enemies in the room, breaking up the structure.
Although the generator generally proved to be efficient in generating levels with remarkable handcraftedness, it is clear that there are still some flaws to be improved. However, it is important to separate the generator from the design of the mission sequence and rules. A lot of problems could stem from design issues that have nothing to do with the generator itself, since the outcome is almost entirely independent of the designer. Dormans and Bakkes (2011) stress that grammar rules are difficult to set up and need a longer time to be effective. Due to this not being possible within the scope

of this thesis, there is still a lot of improvement expected from designing better rules, which could improve the handcraftedness of generated levels much more.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

This thesis followed the goal of creating a generator that can generate top-down levels that seem as they had been made by a human and not by an algorithm. This goal has been followed by defining and analysing characteristics being used in professional level design, to be subsequently implemented these in the generator. Additionally, it was first necessary to research about what kind of generators and methods would allow these characteristics of handcrafted levels to be implemented. The result was a mixed-initiative generator combining numerous of techniques allowing control over different aspects of the generation process, able to implement the researched humanization techniques and even resembling the dungeon structure found in Zelda games games. This thesis presented a successful method to achieve the goal of generating levels that are similar to handcrafted ones.

7.2 Future Work

It would be interesting to see the generator's results when using rules that are more polished and thought through. With that, one could probably improve the generated handcraftedness a lot more. These improved rules could create more complex and interesting patterns. The generator would also benefit from the implementation of more variations in the dungeon rooms. This could also implement architecture within the rooms, which has barely been done in this thesis. A main issue in this generator was the lack of the dungeon's ability to reconnect, which made it difficult to implement zen views and thus could not be evaluated. Solving this issue would open up a lot more possibilities to improve the handcraftedness of levels. This is also a major aspect where the work on this project could continue. The level feelings in *Spelunky* (Mossmouth, 2008) are an interesting technique that the generator would also benefit from. Furthermore, generation times did not turn out to be very short, which is unfavorable for a constructive generator. This problem, however, could be easily solved by moving the code into C++ classes, which are currently almost entirely inside Unreal Engine's blueprints, resulting in the code running much faster. Currently, mission symbols are entirely based on human input. Adding a degree of randomness to these symbols can

contribute to a higher variety in the generated content, without changing the input. There are also many ways one could continue this research. First, one could apply this generator to a bigger game with more content and different enemy types, creating a lot more opportunities to create grammar rules around them, creating different enemy formations, for example. Since, to incorporate the level design patterns, the generator was restricted by hard constraints, it could be evaluated in terms of variety in the generated content and be improved in that regard. A potential way to improve the handcraftedness of the generated levels even more would be to combine the grammar-based generator with a search-based approach, as recommended by Dormans (2010). Applying this generator to a different domain than a top-down game, for example, a third person game, is also something where a lot of research could be done. Researching additional humanization techniques to be implemented would surely result in improved handcraftedness as well.

Bibliography

Adventurepro Games LLC. (2014). Dungeonmans [Steam].

Atari, N. (1972). Pong [Arcade].

barbs. (2006). Dungeon crawl stone soup [PC].

Blizzard Entertainment, B. N. (1996). Diablo [PC].

Capcom, L. (1987). Mega man [Nintendo Entertainment System].

Chen, J. (2007). Flow in games (and everything else). *Communications of the ACM*, *50*(4), 31–34.

Csikszentmihalyi, M. (1990). Flow: The psychology of optimal experinece harper perennial.

Devolver Digital. (2016). Enter the gungeon [Steam].

Dormans, J. (2010). Adventures in level design: Generating missions and spaces for action adventure games. *Proceedings of the 2010 workshop on procedural content generation in games*, 1–8.

Dormans, J. (2011). Level design as model transformation: A strategy for automated content generation. *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, 1–8.

Dormans, J., & Bakkes, S. (2011). Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), 216–228.

Edmund McMillen, H. G. (2011). The binding of isaac [Steam].

Epyx, M., Artificial Intelligence Design. (1980). Rogue [PC].

Feil, J., & Scattergood, M. (2005). *Beginning game level design*. Thomson Course Technology.

Green, D. (2016). *Procedural content generation for c++ game development*. Packt Publishing Ltd.

Gutierrez, J., & Schrum, J. (2020). Generative adversarial network rooms in generative graph grammar dungeons for the legend of zelda. *2020 IEEE Congress on Evolutionary Computation (CEC)*, 1–8.

Hartsook, K., Zook, A., Das, S., & Riedl, M. O. (2011). Toward supporting stories with procedurally generated game worlds. *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, 297–304.

Hello Games, Sony Interactive Entertainment. (2016). No man's sky [Steam].

Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, *9*(1), 1–22.

Johnson, L., Yannakakis, G. N., & Togelius, J. (2010). Cellular automata for real-time generation of infinite cave levels. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 1–4.

Joris Dormans. (2016). *Cyclic dungeon generation*. Youtube. https://youtu.be/c_a3pxIFqh0

Kate Compton. (2017). *Practical procedural generation for everyone*. Youtube. https://youtu.be/WumyfLEa6bU

Ludomotion. (2017). Unexplored [Steam].

Mawhorter, P., & Mateas, M. (2010). Procedural level generation using occupancy-regulated extension. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 351–358.

McMillen, E. (2010). *Analysis: Mcmillen on risk & reward in video games*. Retrieved June 2, 2023, from %5Curl%7Bhttps://www.gamedeveloper.com/pc/analysis-mcmillen-on-risk-reward-in-video-games%7D

Miguel Lobo. (2019). *Low Poly Dungeon Asset Pack*. Retrieved March 28, 2023, from %5Curl%7Bhttps://www.unrealengine.com/marketplace/en-US/product/low-poly-dungeon-asset-pack%7D

Mojang. (2008). Minecraft [PC].

Mossmouth, M. S., Xbox Game Studios. (2008). Spelunky [Xbox 360].

Nintendo. (1986). The legend of zelda [Nintendo Entertainment System].

Nintendo. (1991). The legend of zelda: A linkt to the past [Super Nintendo Entertainment System].

Nintendo. (1993). The legend of zelda: Link's awakening [Nintendo Entertainment System].

Nintendo. (1994). Earthbound [Super Nintendo Entertainment System].

Nintendo. (2006). The legend of zelda: Twilight princess [Nintendo Gamecube].

Nintendo. (2019). The legend of zelda: Link's awakening [Nintendo Switch].

Nintendo. (1986-2023). The legend of zelda.

Re-Logic 505 Games. (2011). Terraria [Steam].

Rick N. Bruns. (2019a). *Free Easy Enemy Pack*. Retrieved May 16, 2023, from %5Curl%7Bhttps://quaternius.itch.io/animated-easy-enemies%7D

Rick N. Bruns. (2019b). *The legend of zelda: A link to the past d02: East palace (pendant 1) labeled map - super nintendo (snes)*. Retrieved May 28, 2023, from %5Curl%7Bhttps://www.snesmaps.com/maps/ZeldaALinkToThePast/Zelda3MapD02.html%7D

Rick N. Bruns. (2019c). *The Legend of Zelda: A Link to the Past D03: Desert Palace (Pendant 2) Labeled Map - Super Nintendo (SNES)*. Retrieved May 28, 2023,

from %5Curl%7Bhttps://www.snesmaps.com/maps/ZeldaALinkToThePast/Zelda3MapD03.html%7D

Rick N. Bruns. (2019d). *The Legend of Zelda: A Link to the Past D06: Dark Palace (Crystal 1) Labeled Map - Super Nintendo (SNES)*. Retrieved May 28, 2023, from %5Curl%7Bhttps://www.snesmaps.com/maps/ZeldaALinkToThePast/Zelda3MapD06.html%7D

Rick N. Bruns. (2023). *Kubes - Kube Dungeon KIT*. Retrieved March 28, 2023, from %5Curl%7Bhttps://assetstore.unity.com/packages/3d/environments/dungeons/kubes-kube-dungeon-kit-246794%7D

Shaker, N., Togelius, J., & Nelson, M. J. (2016). Procedural content generation in games.

Short, T., & Adams, T. (2017). *Procedural generation in game design*. CRC Press.

Smith, G., Whitehead, J., & Mateas, M. (2010). Tanagra: A mixed-initiative level design tool. *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 209–216.

Stout, M. (2012). Learning from the masters: Level design in the legend of zelda [https://www.gamedeveloper.com/design/learning-from-the-masters-level-design-in-i-the-legend-of-zelda-i-, Last accessed on 2023-06-01].

Sturtevant, N., Smith, G., & Togelius, J. (2015). *Making things up: The power and peril of pcg*. Youtube. https://youtu.be/B11RlHZsmGE

Subset Games. (2012). Ftl: Faster than light [Steam].

SweetJohnnyCage - Narrated Guides & Walkthroughs. (2020). *Eastern palace walkthrough - the legend of zelda a link to the past*. Retrieved May 28, 2023, from %5Curl%7Bhttps://youtu.be/eBzen_gghek%7D

Tekinbas, K. S., & Zimmerman, E. (2003). *Rules of play: Game design fundamentals*. MIT press.

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2010). Search-based procedural content generation. *Applications of Evolutionary Computation: EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Istanbul, Turkey, April 7-9, 2010, Proceedings, Part I*, 141–150.

Totten, C. W. (2014). *An architectural approach to level design*. CRC Press.

Totten, C. W. (2017). *Level design: Processes and experiences*. CRC Press.

Williams, N. (2015). An investigation into dungeon generation.

Zhou, Z., & Guzdial, M. (2021). Toward co-creative dungeon generation via transfer learning. *Proceedings of the 16th International Conference on the Foundations of Digital Games*, 1–9.