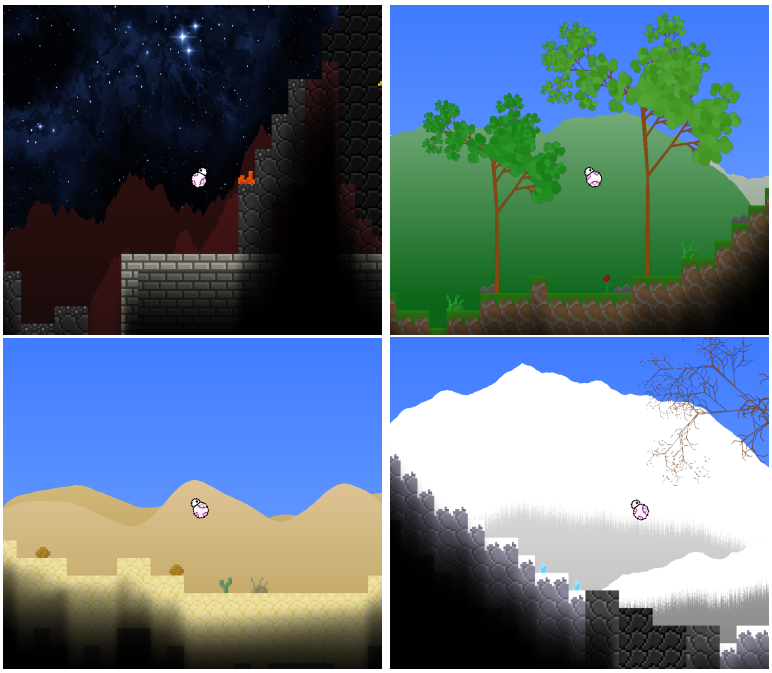


Exploring Procedural Content Generation for a 2D Space Exploration Game

Bachelor of Science Thesis in Computer Science and Engineering

Edvin Broman, Harald Brorsson, Gustav Gränsjö
Emil Hukić, Sabrina Samuelsson, Erik Sänne

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Bachelor's thesis 2018:87

Exploring Procedural Content Generation for a 2D Space Exploration Game

EDVIN BROMAN
HARALD BRORSSON
GUSTAV GRÄNNSJÖ
EMIL HUKIĆ
SABRINA SAMUELSSON
ERIK SÄNNE

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Interaction Design
Chalmers University of Technology
Gothenburg, Sweden 2018

Exploring Procedural Content Generation for a 2D Space Exploration Game

Edvin Broman, Harald Brorsson, Gustav Gränsjö, Emil Hukić, Sabrina Samuelsson, Erik Sänne

Supervisor: Staffan Björk, Department of Computer Science and Engineering
Examiner: Olof Torgersson, Department of Computer Science and Engineering

Bachelor's Thesis 2018:87
Department of Computer Science and Engineering
Division of Interaction Design
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers' Department of Computer Science and Engineering
Gothenburg, Sweden 2018

Abstract

Within video game development, it is common to employ Procedural Content Generation (PCG). PCG is the act of letting a computer instead of a human designer create digital content. An important issue within PCG is how to increase the quality of the content that these generating algorithms produce. This thesis investigates the creation of cohesive 2D game worlds using procedural content generation, aiming to give an impression of what techniques and algorithms are suitable for this domain and how they can be used and modified. This is done through the implementation of a PCG system that generates a galaxy containing a multitude of solar systems populated with planets which the user can explore in a game. Some of the generation techniques explored are: cellular automata for generating cave systems, noise functions for generating heightmaps, L-systems for generating trees, and Markov chains for generating names. The individual systems produce good results overall, and interact moderately well with each other to create a cohesive world. The generated planets are however not particularly visually distinct within a specific planet type. The different generation techniques used are evaluated and reflected upon to give a better understanding of their strengths and weaknesses.

Keywords

Procedural Content Generation, PCG, Level design, Game development

Sammandrag

Inom spelutveckling är det vanligt att använda sig utav Procedurell Generering, även kallat PCG från engelskans 'Procedural Content Generation'. PCG innebär att man låter en dator skapa digitalt innehåll, istället för att låta en mänsklig utvecklare göra det manuellt. En viktig utmaning inom PCG är att upprätthålla kvalitén hos innehållet som algoritmerna genererar. Denna rapport undersöker skapandet av sammanhängande 2D-spelvärldar genom att använda PCG, med målet att ge en uppfattning om vilka tekniker och algoritmer som kan vara passande för detta specifika domänet, samt hur de kan användas och modifieras. Detta görs genom att implementera ett PCG-system som genererar en galax med en stor mängd solsystem med planeter som spelaren kan utforska i spelet. Några tekniker som undersöks är cellulära automater för att generera grottsystem, brusfunktioner för att generera höjddata, L-system för träd och Markovkedjor för namn. De individuella systemen producerar innehåll som till stort anses tillfredsställande, och de interagerar relativt väl med varandra för att skapa en sammanhängande värld, även om den visuella varitaionen inom specifika planettyper kan anses något bristfällig. De olika teknikerna som används utvärderas och reflekteras kring för att ge en bättre förståelse om dess styrkor och svagheter.

Acknowledgements

We would like to thank our supervisor Staffan Björk for his helpful insights and discussions. We would also like to show our appreciation for Fackspråk in general and Calle Carlsson in particular for their help with structuring this thesis.

Group 87

Contents

1

Introduction

This chapter introduces what Procedural Content Generation is, what it is used for, and some of the issues of interest that exist within the field. The purpose of this thesis, as well as more specific goals and delimitations, are also explained.

1.1 Background

Ever since their inception, the potential scope of video games has steadily increased and so has the amount of content that is needed. This situation led to developers trying to find ways to facilitate the creation of said content. Could they get the computers to create content of a quality on par with or exceeding that of human designers, which would potentially save them both time and money?

Procedural Content Generation (PCG) refers to this algorithmic creation of digital content. Developers define algorithms and rules - often including some randomness - which the computer then follows, to output some form of content. PCG has been used for a long time in both video game development and filmmaking; this project's focus will lie on the creation of games. In games, the content generated by PCG can come in many forms, such as level geometry, story, weapon attributes, music, textures and so on [1, p. 1].

The main attraction of PCG is that a computer can create things magnitudes of times faster than a human designer. By leveraging the computational power when creating content, a lot more content can be created within a specific time-frame. A game designer then no longer needs to hand-craft each level, which can be a very time-consuming task. As a result, content can be created more cheaply and efficiently. It also enables games to have a much broader scope than with traditional methods. Once the algorithms are developed, a virtually infinite amount of content can be produced, making it theoretically possible to provide a player with an infinite amount of new experiences.

There are however still substantial difficulties to tackle when designing PCG methods, and many of them can be summed up as to how to be able to produce content of the same quality as hand-crafted content [2, pp. 64-65]. Defining what consti-

tutes high-quality content in games is quite tricky. What factors lead to enjoyable gameplay varies strongly from game to game [1, p. 6], and therefore, it is difficult to define a general set of good qualities for content generation. PCG techniques are merely tools and must be modified and adapted to suit the needs of every individual project to produce the desired result.

In a game where the player can move freely in any direction, it might be acceptable for the levels to include large pitfalls since the player can fly over them. However, in another game where the player has a limited jumping distance, it is of utmost importance that the pits have a maximum width or an alternative way to proceed through the level. Limiting the output space of the generating algorithms in this way to exclude bad content can, however, be a difficult balance to strike. Placing too rigid restrictions may risk good content also being excluded and thus lowers the variation while placing too relaxed restrictions can allow low quality or unplayable content into the result.

This ties into another considerable difficulty regarding PCG: making the content visually distinct and expressive [1, pp. 6-7]. Even if the actual output space of the generation is ample, the players may not always perceive the variation as great [3]. If a lot of the produced content is functionally identical with only minimal parts of the content differing, it will not actually be perceived as being varied.

There have been many games made that heavily use PCG when designing their worlds. Two such games are Terraria [4] and Starbound [5]. Both of these games generate vast, tile-based 2D worlds including both a varying surface and cave systems. The details as to exactly how they generate their worlds is however not readily available to the public, which means that if one is interested in developing something similar, the next best course of action is to start from a published non-specific generation method and start modifying it to fit the specifics of the project. By documenting how this can be done, and the issues faced and solved, future developers might be able to speed up their development and avoid having to deal with similar problems.

1.2 Purpose

The purpose of this project is to, through the use of a diverse selection of procedural content generation techniques, investigate possible ways of generating cohesive but varied environments suitable for exploration and adventure-like side-scrolling games.

1.3 Goal

In order to fully be able to satisfy the purpose, the program that is to be implemented should be a generator for a complete game world. This game world should contain

individual side-scrolling 2D levels with exploration possibilities. It should also be possible to show considerable variation between the 2D levels, and representing the levels as different planets could provide this opportunity. To make the game world cohesive, the planets should be contained in solar systems within a galaxy.

It is important for the levels to be varied and distinct, yet they should still give the impression of belonging to the same world, which is also the reason for having three layers of content. The idea is that each layer should depend on its parent layer; for example, a solar system sets parameters for its planets such as temperature, gravity, how well it is suited for life and so on. The appearance and design of a planet should then vary based on these parameters, which can provide a greater sense of cohesion in the game world.

1.4 Delimitations

To better be able to focus on the study of procedural content generation, it was decided that the number of gameplay elements implemented in the game should be severely limited. The player will not be able to do more than move around and look at the different levels generated; no interactive elements will be included, nor will there be a story to the game and sound and visuals will be kept very simple.

Furthermore, the parameters that influence the generation of a level will be kept to a small amount, due to two primary reasons. The most important reason is to showcase how multiple independent parameters could work together to create a multitude of unique levels. The second reason is simply to limit the scope so that the project can reasonably be finished on time.

Most things in the game will not be generated with the aim of full scientific accuracy. In part, this is because a real galaxy contains many things that are, quite frankly, not that interesting, such as empty solar systems or barren rocky planets. If the goal is to generate a varied and interesting world, strictly following reality will actively hinder instead of help. However, realism to some degree can be useful, in the sense that it can make the player more immersed in the game. The game will on occasion try to mimic realistic concepts, but genuinely simulating real situations can take much time. There are simply not enough resources to provide full realism. The focus will instead be on using PCG in a way that can produce content that would seem reasonable to a player.

1.5 Games Using Procedural Content Generation

Considering the long history of PCG in games, it is impossible to provide a comprehensive list of all games which have used PCG, and how they have used it. However, it is important to present a brief overview of which games have been especially im-

portant in the history of PCG in games, which this section aims to provide.

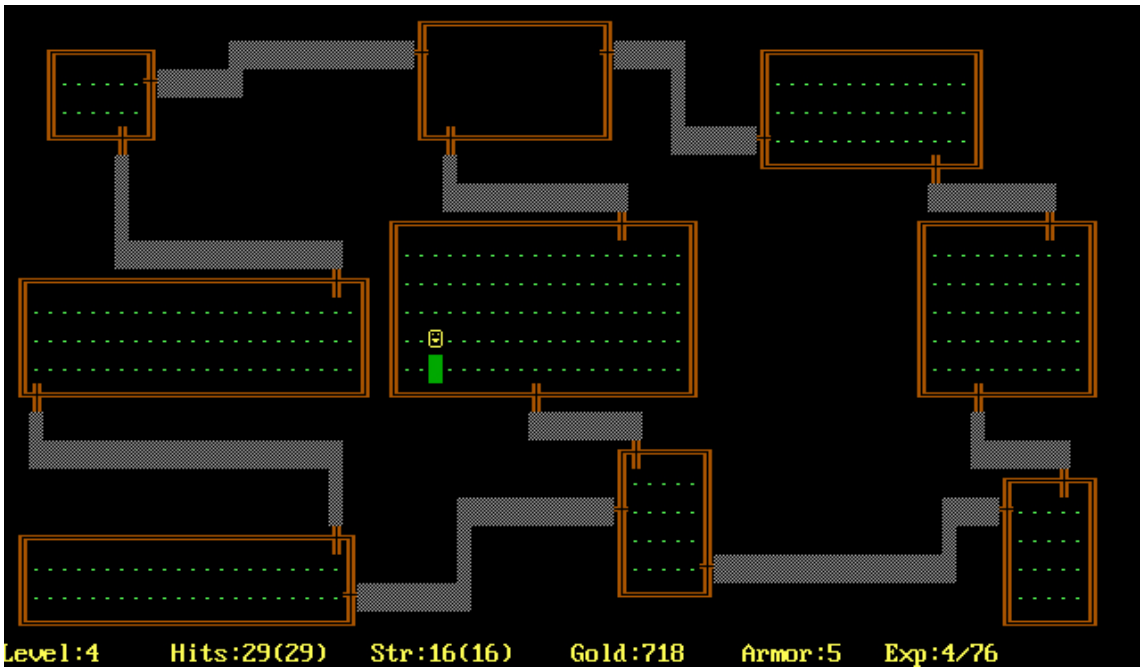


Figure 1.1: A screenshot from Rogue, showing the generated dungeons. Image by Artoftransformation under the license of CC BY-SA 3.0, via Wikimedia Commons.

One of the first games that featured PCG heavily was Rogue [6], which was released in 1980. In Rogue, the player explores a randomly generated dungeon, like the one seen in figure 1.1. The fact that the dungeons were randomly generated made it possible to play the game many times, with each playthrough being unique. The popularity of the game spawned an entire sub-genre, called Roguelikes, where the focus is on exploring randomly generated dungeons. The act of generating dungeons was quickly adopted by genres aside from roguelikes, with the Diablo [7] series being perhaps the most famous example.

In 1984, Rescue on Fractalus! [8] was the first title to use fractals to generate the mountains for their first-person space-shooter, a technique that since then has become widely adopted for creating terrain in general.

In later years, games started using PCG in many more ways than before. The Elder Scrolls IV: Oblivion [9] does not simply generate dungeons but instead used PCG to help create vast swaths of land that the designers could use as a starting point for creating their world. The space game EVE Online used PCG to generate their playable universe consisting of over 7 800 star systems. In Borderlands [10], all of the guns which players collect have randomly generated attributes, forcing players to make the best use of what they get. Minecraft [11], on the other hand, generates

every single piece of a 3D world which the player can explore and interact with by destroying parts of the world and building new things.

Figure 1.2: A screenshot from Terraria, a 2D game containing large procedurally generated worlds.

Finally, two games which helped inspire this project, Starbound and Terraria also make heavy use of PCG to generate large 2D worlds which players can explore. The worlds created are filled with different environments to explore, in which the player can fight and collect different resources. Figure 1.2 shows an example of such an environment.

These game examples are but a glimpse of the enormous amount of games which use PCG and the varying amounts of contents which can be generated, but hopefully, it will give some insight into the different possibilities of PCG and in which ways it has been used.

2

Theory

This chapter provides the theory that is relevant to both the process and result of this project. Some fundamental concepts and terms commonly used in procedural content generation are introduced to provide a basic understanding of some of the issues that have to be addressed in PCG. It also covers a few common methods and algorithms for PCG that have been used or are of other importance to this project. Lastly, some general methodologies for software development are presented.

2.1 Offline and Online Generation

There are two general approaches to PCG, known as offline and online generation [12, p. 173]. Offline generation means that all the content is created, evaluated and finalised before the product is released. It is used in a way that helps designers create content which is too difficult or time-consuming to create manually. As an example of this, in an interview, Gavin Carter from Bethesda explains how their, at the time, new procedural content generation for terrain and trees significantly sped up the development of environments for The Elder Scrolls IV: Oblivion [13]. Another common practice for offline generation is to create an initial level structure which serves as a template that human designers can then improve upon and finalise afterwards [12, p. 173].

Online generation, on the other hand, is done during run-time of the program. An advantage of this is that new content, like levels, can be generated almost endlessly. For example, a world can be expanded as the player approaches the end of it, as can be seen in Mojang's Minecraft. Similarly, Diablo generates new dungeons as the player progresses through the game and Borderlands generates different weapons as the player defeats enemies. Online generation also makes it possible for the generated content to be adapted to the player's actions in real time. This could mean generating more difficult levels for more skilled players [14] or generating new content based on the player's preferences [15].

An important aspect that sets the two approaches apart is the fact that online PCG needs to be fast so that it is not detrimental to the game experience. This limits the complexity of the algorithms which can be used to generate content, and

by extension, the complexity of the content which can be generated. Since offline generation does not suffer from that issue, much more complex algorithms can be used.

Furthermore, when generating content, some sort of content validation is necessary to ensure that it is of at least acceptable quality. With offline generation, the developers themselves validate the content and can even make alterations if the generated content has some flaws. However, this is not possible when using online generation, which means that the content must be validated automatically.

2.2 Validation Approaches

When designing a content generator, it is important to have a way to ensure that the generated content is valid. The criteria for what content is valid are defined by the game designers and can vary in complexity. They can be simple things such as a requirement that the end of the level is reachable, or more nuanced like controlling the difficulty of one area. The criteria can also vary in strictness: the terms *necessary content* and *optional content* are used to describe how strict a criterion is. For example, having bonus treasure chests in a region might be optional content while there being a path to the goal is necessary content [1, p.8]. When it comes to making sure that the game follows the necessary criteria, there are a few different approaches.

The first is the constructive approach, where the generation is done only once and is considered done when the content has been generated. This approach requires a generator that can only produce valid results, which means that the possible results are limited [12]. One difficulty of this approach is to limit them in a way that does not exclude valid results.

The second approach is called generate-and-test. There are different ways of doing this, but the common concept is that there are two components to the generation: a generating program and a test function [12]. When a result has been generated, it is inputted to the test functions which determines whether it is valid or not. If the result is not good enough, the generating program is used again, to produce a different result. This process is then repeated until there is a satisfactory result. One variation, called search-based, grades each result on a scale and then uses it to affect the generating program, resulting in an evolutionary algorithm that successively becomes better and better [12].

2.3 Markov Chains

One way of generating a variety of content such as text, game levels [16] and music [17] is through the use of *Markov chains* [18]. A Markov chain is a finite state

machine, meaning that it consists out of a set of states among which it can transition between. The process is stochastic, and has no memory of any previous states, and each transition can be traversed with some fixed probability.

An important aspect of using Markov chains for generating content is the N-gram [19, Chapter 6], which is an n token long sequence which defines how much information is stored in each of the Markov chain's states. For example, if the tokens are letters, then a 2-gram (bigram) is a sequence of two letters, while a 3-gram (trigram) is a sequence of three letters. Any word could then be split up into a set of n-grams. If the word 'gram' were to be split into a set of 3-grams, the two states would be 'gra' and 'ram', and the only transition would be from 'gra' to 'ram'.

N-grams have proven to be one of the greatest strengths of Markov chains, as they make it possible to automatically build the set of states and transitions by analysing a sequential set of tokens and recording the successor for a specific n-gram each time it appears. On the other hand, since it is an entirely stochastic process, it is hard to fully control the output. The output can be affected by changing the analysed token set, but it is hard to know the exact effects of the change.

2.4 Formal Grammars

Formal grammars[20, Chapter 2.2] are another technique which has proven to be very useful for generating a variety of content in games [1, p.97]. Just as how a grammar describes what is considered a valid sentence in a natural language such as English, in the field of formal grammars, all valid strings are defined by a set of production rules.

Production rules describe all the possible ways a string (state) can be transformed into another string (state) and a set of production rules define a grammar. This is perhaps best explained with an example. A simple grammar can be created with two production rules, where the string on the right-hand side describes what the string on the left-hand side is replaced with.

1. A $\rightarrow$ B

2. B $\rightarrow$ AB

The string on the left-hand side can be a subset of the current state string of the system. Assume that the starting point of the string is A, then the first few expansions of this grammar could look as follows.

1. A

2. B (Rule 1 used on letter 1)

3. AB (Rule 2 used on letter 1)
4. AAB (Rule 2 used on letter 2)
5. ABB (Rule 1 used on letter 2)
6. AABB (Rule 2 used on letter 2)

This process can then go on indefinitely.

One of the advantages that formal grammars offer is that developers have a large degree of control over the results. They get to decide what the production rules are and what the probabilities of using them are. Unfortunately, this is also one of the disadvantages as it requires that developers manually design grammars, which can be a highly time-consuming process.

2.4.1 L-systems

A more specific type of grammar which is used readily for PCG is the Lindenmayer system (L-system) [21], which is most commonly used for generating various kinds of plants but has also been used to generate a city [22] and music [23]. What sets L-systems apart from an ordinary grammar is that it makes use of parallel rewriting, that is, all of the replacements for the iteration are done concurrently.

Some common variations on the L-system are *stochastic L-systems*, *context-sensitive L-systems* and *parametric L-systems* [21, Chapter 1] which can be used individually or in conjunction. Stochastic L-systems simply add an element of randomness to the rules, for example, there may be two production rules associated with one string, with one of the production rules being chosen at random. In context-sensitive L-systems, the rules can depend on the context in which they are to be applied. For instance, if 'A' can be replaced by 'B' only if there is a 'B' before the 'A', the L-system would be context sensitive. Parametric L-systems introduce the ability to pass parameters to each symbol which can then be used both by functions operating outside the L-system such as a drawing method, or by the production rules to either decide whether the rule will be applied or the production rule could modify the parameters.

2.5 Binary Space Partitioning

Binary space partitioning (BSP) is a way to split some space into partitions. It works by recursively subdividing the space into two smaller parts until some stop criterion is fulfilled for each subspace [1]. It has many uses and is relevant to this project for its use in dungeon room placement.

Dungeons are a common element in digital games, where they are most often enclosed environments and encourage exploration [1, pp. 31-32]. One common dungeon type is the so-called *rooms and corridors*, which is made up of a set of rooms connected by corridors. BSP can be used to divide a dungeon map into non-overlapping areas, which can then be used to place rooms of varying sizes. In this case, the stop criteria can be a maximum room size where the recursion stops when the size of the space goes below the threshold.

2.6 Cellular Automata

Cellular automata are a widely studied computational model [1, p. 42]. They can be described as a large number of cells updating their states by using simple and identical rules as a function of the states of their neighbouring cells. The concept of cellular automata was first introduced by John von Neumann and Stanislaw Ulam in the 1950s and gained much popularity in 1970 when John Conway introduced his *Game of Life* at a conference [24]. Conway's game of life is one example of a cellular automaton and it featured a matrix where each cell could have one out of two different states represented by zero and one. In this game cells with the state zero were considered dead while cells with value one were considered alive. Every time step the state of each cell is updated by following two simple rules: a cell dies if it has too many living neighbours (overpopulation) or if it has too few living neighbours (under population) and a cell becomes alive if it has precisely three alive neighbours. If none of the rules are fulfilled, the state of the cell remains the same.

What makes this game, and cellular automata in general, interesting is the varied type of interactions that can arise from simple rules. It can appear as if the cells have some central coordination, even though they are just updated according to their local interactions with other cells. This self-organising property that arises from local interactions is commonly seen in nature, for example in ant colonies [25], and makes cellular automata well suited for emulating many different kinds of naturally occurring systems. For example how forest fires spread [26] or how rock is eroded to create caves.

Cellular automata can be modified by changing the birth and death limits, or by changing which neighbours are taken into account when updating the state of a cell. The number of possible states can also be changed; making it possible to perform more complex actions than simply switching between two states [1, p. 42]. Cellular automata have been proven to be able to generate relatively good-looking cave systems. This is done by applying the rules for a number of iterations on an initially randomised grid of cells representing either cave walls or open spaces [27].

2.7 Terrain Generation

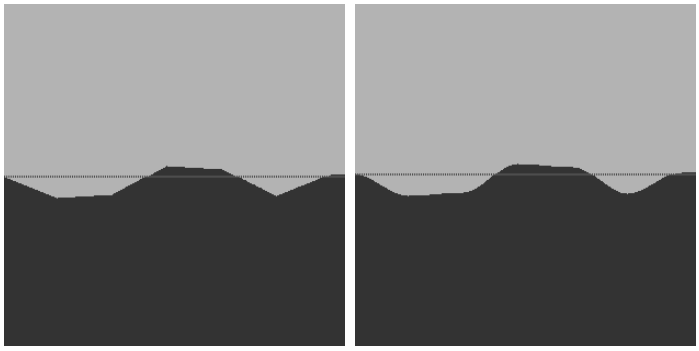
Terrain and landscapes play an important role in many games, as they can provide both scenery and gameplay opportunities [1, p.57]. The use of procedurally generated terrain has been prevalent in games for a long time, with one of the earliest examples being 'Rescue on Fractalus!' , where fractal algorithms were used to create terrain dynamically during runtime [28].

One simple and very common way of representing a terrain is through the use of *heightmaps* (or heightfields), which essentially is a set of data containing height values for the surface [29, p.71]. A way of generating procedural terrain is to introduce noise into such a height map. While simply randomising each height value in most cases will not result in anything that can be used as terrain, there are several different methods using noise for constructing more suitable environments [1, p.59]. Two noise algorithms are explained below, as well as how to make the terrain more natural-looking by utilising fractals.

2.7.1 Value Noise

A simple way of generating a height map, value noise simply randomises the heights of a number of starting points and then interpolates between them. The starting points are set at a constant distance from each other, and that distance can be adapted for the specific situation. The range of the height randomisation can also be set to a suitable value [1, pp.60-61].

The interpolation is done using a slope function s. For any point p in between the starting points, the nearest starting points are given weights according to a chosen function $s(d)$, where d is the distance from p to the starting point. If a point p is 1/10th of the way between lattice point a and lattice point b, the height at p will be $(1 - s(1/10)) * height(a) + s(1/10) * height(b)$ [1, p.60]. The shape of the function s is what determines the shape of the terrain, and thus the choice of function has a significant effect on the result. Common functions to use are linear, cosine, or cubic functions [30]. A linear function will create straight lines and sharp tips, a cosine will have a rounded result, and cubic functions will produce slopes with varying S-shapes depending on the coefficients used. The cubic functions can be said to have the best results, but are also the slowest [30]. As can be seen in figure 2.1, the cubic function produces a smoother curvature compared to the jagged lines of the linear function.



(a) Linear interpolation. **(b)** Cubic interpolation.

Figure 2.1: Value noise using different interpolation function. The linear interpolation clearly results in sharper lines, and can be considered less natural than the more rounded cubic interpolation.

2.7.2 Perlin Noise

Perlin noise works in a similar way to value noise. The difference is that instead of generating random heights, the starting points get assigned random *slopes* [1, pp.61-62], while their heights are all set to the same value. Then the points in between are given a height based on the surrounding starting point slopes. The height at point p is calculated by, for each surrounding starting point, calculating what the height of p would be if it were only affected by that single starting point. That is done through a simple calculation of the slope's value times the distance from that point. If the slopes are two- or more-dimensional, this means using the dot product of the slope vector and the distance vector. Then, the different calculated heights of p are interpolated into a single value in the same way as the heights are interpolated in value noise [1, pp.61-62].

Perlin noise is a more complex algorithm than value noise is, but since the computations can be optimised in various ways, it is not necessarily much more computationally heavy than value noise is [30].

2.7.3 Fractal Noise

One effect of using noise functions is that the outputted terrain is very smooth and fluctuates at very noticeable frequencies because of the lattice spacing. Real terrain, however, has variation at multiple levels, meaning that it has large mountains and valleys, but as one looks closer, the same pattern occurs at a much smaller scale, causing roughness to the surface. A way to achieve this effect is through the use of fractal algorithms [1, pp.62-64]. Due to their simplicity and computational efficiency, fractal algorithms are a common approach for generating realistic looking terrain, especially in game development [31].

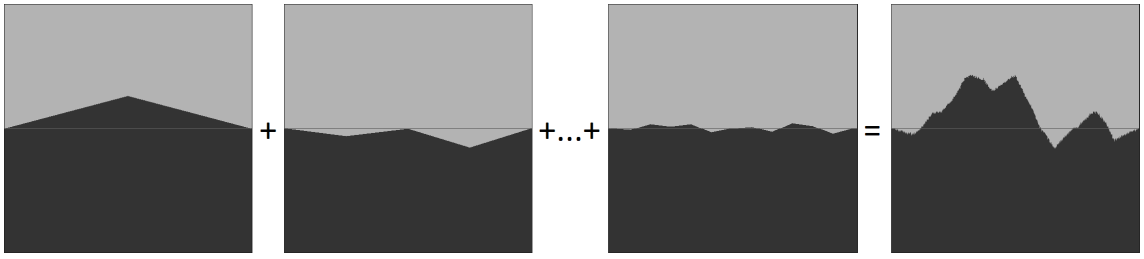


Figure 2.2: Fractal terrain created by adding multiple generated heightmaps together, which are in this instance generated using the midpoint displacement algorithm.

The idea is that multiple heightmaps are created at different scales and are then blended together. Initially, the large-scale properties are created with the high intensity or amplitude of the noise function, and each following step the intensity is decreased while the frequency is increased, as illustrated in figure 2.2. Hence, any noise function can be considered a fractal noise algorithm if used in several passes, but there are also other approaches to creating fractal terrain [1, p.63]. One of the most simple and widely used is called *midpoint displacement* [30].

Midpoint displacement works by initially placing two points and considering a line segment stretching between them. The line is then split in half, and the centre point is displaced randomly with some maximum intensity. This step is then recursively repeated with each newly created line segment with a decreased displacement factor. This process is further repeated until some terminating condition is met, such as if a line segment is considered too short.

2.8 Diffusion Limited Aggregation

Diffusion Limited Aggregation (DLA) [32] is a sort of crystallisation simulation. The algorithm starts with some initial structure, which is usually just a single point (particle). New particles are then added, and do a random walk until they stick to the already existing structure. After a particle gets stuck, it becomes possible for other particles to stick to it, effectively expanding the structure. This tends to create crystal-like, branching structures as illustrated in 2.3.

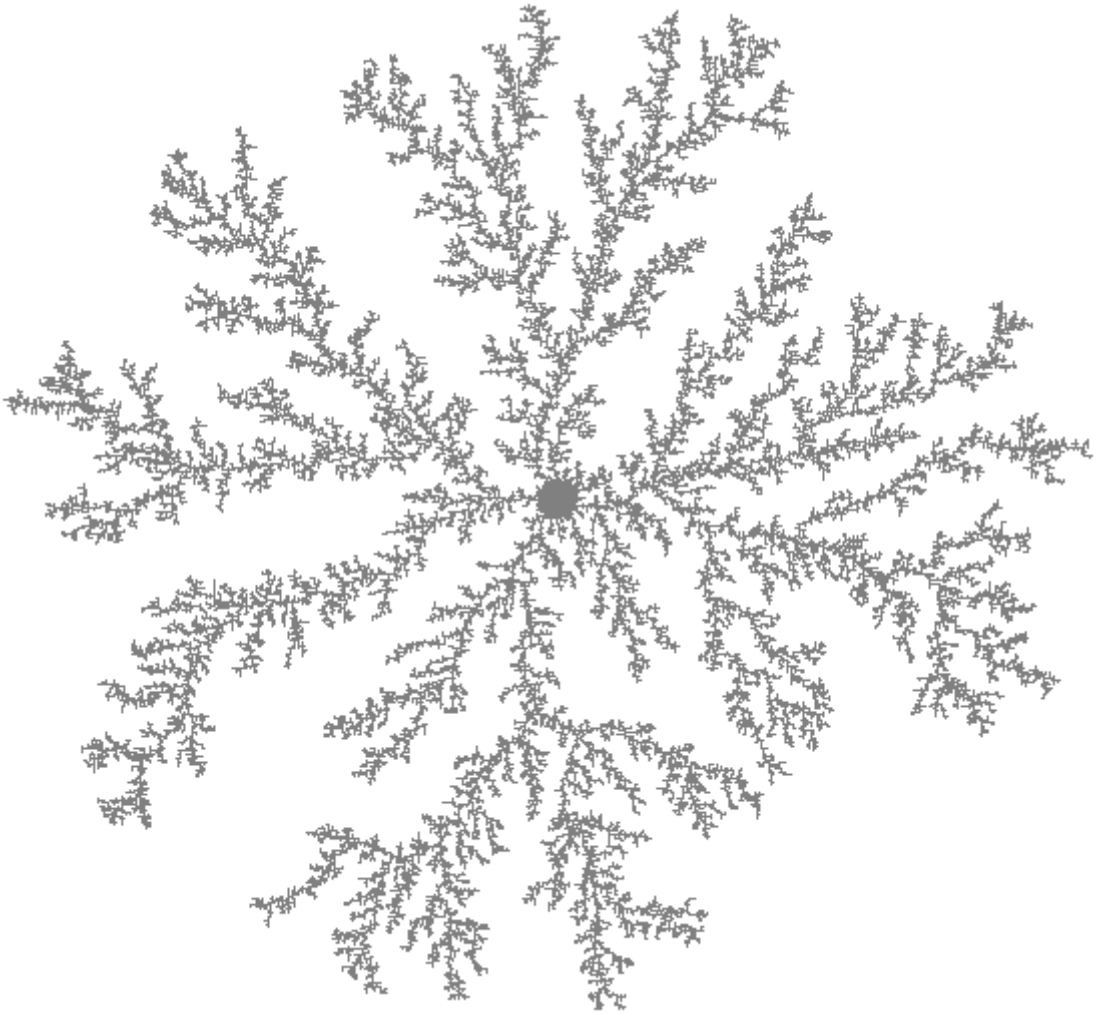


Figure 2.3: Example of a '2D DLA structure' by Alexis Monnerot-Dumaine under the license of CC-BY-SA-3.0

DLA can be used to create the structure of a galaxy as is done in EVE Online [33]. The more random crystal-like structure could be a good fit for irregular galaxies, and with this method, the stars can be ensured to be connected and within some minimum distance of each other.

2.9 Methodology

Beyond common issues in PCG and technical background to specific algorithms, it is also important to reflect upon how one approaches the development phase of a project. There are many different examples of software development processes, where some examples of established methodologies are Scrum [34], Waterfall [35] and extreme programming [36].

The Waterfall model follows more traditional design principles where much time is spent planning at the start of the process. This approach made sense in the production of physical products, where the model originated, since many criteria about the product have to be identified and specified early on in the process, but has later been viewed as an archaic practice not well suited for software development [37]. The reason for this is mainly that requirements for the product change much more often in software development. This problem is something which the agile development process tries to address.

Scrum is a software development framework that is part of the family of agile development processes. It has a strong focus on working efficiently in small teams with short iterations from one to four work weeks called Sprints. At the start of every Sprint, a Sprint meeting is held where the previous Sprint is evaluated and the features to be implemented in the coming Sprint are decided upon and planned.

Extreme Programming is another software development framework whose lead principle is that if there is a development practice that is good, then that practice taken to the extreme is even better. For example, code review, the practice of team members reviewing each others code, is considered to be a good practice and taken to the extreme this means that code review is done continuously throughout the project in the form of pair programming.

3

Process

This chapter aims to describe the general process of this project, explaining all of the decisions that were made, as well as the reasons for why they were made. As this project has gone through a very iterative process, it is difficult to express the entire chain in chronological order while keeping coherency among the different technical parts. Therefore, this chapter has been split into four pseudo-chronological stages.

3.1 Planning Stage

The first stage of the project was to gather an understanding of the field of procedural content generation in games, and to formulate a plan regarding what the project would focus on, both concerning goals and what problems would need to be solved.

3.1.1 Field Review

The project started by researching PCG in general and gaining a basic understanding of the most common PCG algorithms so that it would be easier to discuss any specific ideas which anyone had. Further research was then done to explore how PCG has been implemented previously and how each algorithm is best used. By now, a clear trend had been noticed. Most of the research either focused on a particular type of content, e.g. terrain, quests, puzzles, or was at the other end of the spectrum, focusing on content generation in general. Also, much of the research seemed to be focused on generating levels for platformers, with a lack of research which focused on generating cohesive game worlds or generating different types of content and combining them.

The lack of research regarding those aspects, and a shared interest in a few algorithms, which are described in Chapter 2, led to the idea of a more specific style of game, ultimately leading to the finalised purpose of this project.

3.1.2 Game Plan

Initially, several different game ideas were considered, stemming from the fact that PCG is versatile and can be implemented in some capacity into pretty much any genre. Eventually however, the game idea was steered towards a space exploration game, since that could provide the opportunity to show several different environments. The idea was to create a galaxy composed of stars surrounded by planets. The player would then travel between these different planets and explore their different environments.

Generating a full galaxy to explore seemed like a good fit for the project for several reasons. Firstly, because it could showcase the power of PCG in the sense that creating so many stars, planets and levels by hand would be a herculean effort, but can be done in a few seconds by a computer. The task of distributing the stars and planets alongside the generation of the individual planets would also open up interesting but separate problems to solve. Additionally, for the different planets to feel distinct a lot of variety would be needed, allowing for many different PCG methods to be implemented.

There was some discussion about whether the gameplay on the individual planets should be displayed in a side view or seen from above. Both methods have several potentially interesting problems and opportunities for PCG, but eventually it was decided upon a side view. A side view makes it easier to in an intuitive way show planet surface altitudes and also allows for the player to explore both the surface and the interior of planets in a single screen. These attributes were the main reason for why a side view was chosen, because solving the issue of generating planet surface altitudes and cave structures seemed like interesting problems to tackle.

Another element that was early decided should be implemented was a set of galactic parameters. These parameters were planned to be things such as sun size, planet gravity, wind speed, and similar things. The idea was that these parameters should be generated in different stages and affect the content generation of other systems. Some parameters would be created on their own for the galaxy, which would then influence the parameters created for the solar systems which would then finally influence the planets. The question of exactly what most of these parameters would be and how they would affect generation was left until later, because the team was not yet sure exactly how modifiable the different generating algorithms would be. However, one parameter that was fully defined at this stage was the *alien presence*. This would would represent the spread throughout the galaxy of an old alien civilisation. One solar system in the galaxy would be appointed as their home system, and the closer a solar system was to that home system, the more alien remnants should be found on the planets within that system.

3.1.3 Problem Identification

After the game idea had been established, a discussion was held about what the most important issues concerning the project were. Creating a game as described in 1.3 was separated into making the three stages: galaxy, solar system, and planet, and then connecting these.

Regarding the galaxy, there were two crucial aspects: properly placing the stars in the galaxy and generating the attributes of each star. The placement of stars should be done in a way that feels natural to the player and is suitable for gameplay in a finished game. For example, to avoid making it tedious to navigate from star to star, they should not be placed too far away from each other, but an otherwise seemingly realistic galaxy was deemed appropriate. The stars' attributes would be used to create variation on the planets, and should be chosen based on that. At this stage, there were no decisions made yet about exactly which variables would be included, but examples such as alien presence, size, temperature, and age of the star were brought up. It was also considered important to name the stars so that it would be easy to distinguish between them.

The solar systems similarly needed to place planets and generate attributes for them. Here too the placement should feel natural to the player, creating a need for at least partial realism, and be suitable for gameplay. Attributes that were considered for the planets were temperature, ability to sustain life, size, and humidity. Of these some would depend on the star's attributes, but not all. The way that the star's attributes affected the planets should also be done in ways that could provide interesting variation, for example by having several parameters depend on each other.

The planets are where most of the gameplay would take place as well as where the different variables should show their effects. What these effects should be was not decided yet but some possibilities included surface structure, appearance of caves, amount of buildings and amount of vegetation. It was planned that the decisions on what to include should be made later on.

What was decided at this stage was that as the player should be able to explore both above and under the planet surface, there should be effects of the parameters clearly visible in both areas. This should include effects on the terrain as well as on the inclusions of various points of interest such as spreading fire, vegetation, or decorative objects.

3.1.4 Choice of Methods and Tools

This project dealt with many, largely separate, issues which made it easy to divide the work into smaller parts and allowed team members to work in parallel. When choosing development process the waterfall strategy was quickly dismissed, as it

is not very suitable for software development. The team instead used an agile development process inspired by Scrum, where tasks were assigned on a weekly basis. Each week, the team evaluated the work which had been done the week before and set goals for the following week while keeping upcoming deadlines for the project in mind. All issues were split into as small tasks as possible to facilitate this work flow. In order to keep track of the tasks an online planning board called Trello was used. Google Drive was used to share various documents within the group efficiently, and Slack was used for online communication.

Both version control and code sharing were accomplished by using Git [38] through GitHub [39]. It was also planned that code reviewing would be enforced through Git. Before any changes were to be committed to the code base, someone other than the author was supposed to review the code. It quickly became apparent that this was more of a burden than initially estimated. Because the different members of the team worked on such different parts of the code, other people's work was hard to review. The time required to get a deep enough understanding of the written code to be able to point out issues was often deemed too great to afford.

To help alleviate this problem, the team moved more towards the practice of pair programming. This was inspired by extreme programming. By working together when designing systems and writing code, there is always at least two people there to spot any potential issues. In this way something akin to the effect of continuous code reviewing can be achieved, hopefully reducing the number of revisions needed.

3.2 Base Implementation

During the next part of the project a plan had been formed, and the initial groundwork of the project started up. The goal was to get the different core components of the system up and running as quickly as possible so that a foundation which could be demonstrated and expanded existed.

3.2.1 Code Architecture and System Design

Due to the team's shared experience with the language, Java [40] was chosen as the programming language for the development of the product. Furthermore, it was decided that the game development toolkit LibGDX [41] should be used for its tools regarding window handling, input and especially its graphics framework. This framework was chosen because the team had previous knowledge of it and thanks to its simple to use graphical methods. The functionality of LibGDX was expected to heavily facilitate the development phase of this project; project structure would also have to be built with their integration in mind. Though the project is heavily constrained regarding game mechanics, it was decided that some basic physics would be needed in order to demonstrate it adequately. Thus, it was later decided that LibGDX's physics framework Box2D and lighting engine Box2DLights should also

be included.

Before any actual implementation could start, an overall system architecture had to be decided. A few common architectural design patterns were discussed. *Model-view-controller* (MVC), a pattern which is commonly used to separate logic from view data [42], was among the most discussed patterns out of those, though it was decided that this project should not follow MVC to a full extent. The motivation for this was that within LibGDX, it is customary for game objects to include both model and graphics data. LibGDX offers its own architectural solutions in their framework, however, it was decided that those should not be used either in order to ensure that the architecture could be tailored to this project, to not be limited by their implementation. Both MVC and LibGDX's architecture however inspired the final architecture.

Since the project would have at least three different stages, it was necessary to have an architecture that supported smooth transitions between them. As those stages, in this case, are arranged in a hierarchical order, it would also be necessary for the state in each stage to be preserved between transitions. If a player were to return to the galaxy after visiting a particular solar system, everything should be exactly as the player left it, without any extra processing. Since the transitions between these stages would always happen hierarchically (e.g. planet to solar system or vice versa, never planet to planet), it was decided that the stages should be stored in a stack-like data structure. By doing this, it would be easy to create or remove stages while preserving parent stages' states and never having to store more than one instance of each unique stage at any given time.

For this storage system to work, all generation had to be deterministic. What this means is that given the same initial input parameter to a generator, it should result in identical output. If this requirement was not upheld, then every time a specific planet or solar system was revisited, it could have changed drastically for no reason since each stage is regenerated from scratch every time it is visited. While our generators aren't deterministic, they all base their randomness on Java's random class which is deterministic. Thus we can ensure determinism by initialising each random instance with the same seed each time.

Each level (at the planet stage) was represented by a matrix containing data for all tiles, since that would enable quick access to each tile in the level, making it easy for different generators to modify the level at specific coordinates. Another approach could have been to have tiles as individual objects which keep references to their neighbours, which would make it possible to have a dynamically sized level, but would make it slower to access specific tiles and would use more memory.

3.2.2 Game Mechanics

In order to adequately showcase the generated content, a few basic gameplay mechanics had to be implemented.

The game started with a view of the galaxy, showing a multitude of stars. In order to navigate the galaxy and solar systems, controls were implemented to make it possible to move the camera and select individual stars and planets. A player character was created for exploration of the individual levels. This character can fly around the levels freely but is affected by a constant gravity which pulls it downward.

The LibGDX library Box2D was used for physics handling on the planets, which requires specific collision forms to be specified for all objects that should be solid and interact with each other. Initially, an individual collision box was stored for every tile present on a level, which worked fine with small levels. However, as soon as the level size was increased, memory issues appeared. This issue was solved by instead only storing collision boxes in an area around the player which was updated as needed. The first implementation of this solution simply stored a matrix of collision boxes, which was emptied out and refilled every time the player moved between two tiles. Even though this design did not have an especially noticeable performance impact with the area size used, the system was redesigned to increase scalability and performance.

The optimisation was done so that instead of updating the full matrix every step, only one row or column was updated at a time. When the player moved one step in a direction, the collision boxes for the tiles that the player moved too far away from were replaced with new collision boxes for the tiles that the player was approaching.

3.2.3 Galaxy

The first issue was to decide what aspects are important when generating a galaxy. Due to the potential gameplay and generation being considered more important than realism, the generation need only be semi-realistic. There are several classifications of galaxies but one of the most common and perhaps widely recognised is the spiral galaxy, so this was chosen as the model [43]. Limited research has been found regarding this topic, but there are several other, similar algorithms available for generating a spiral galaxy, for example, Evan's work [44].

There is also research done in creating large-scale game worlds effectively with an implementation of a galaxy generator similar to the above implementation [45]. While creating a full-scale galaxy could be done using Carpenter's work, doing so would probably take too much time and be out of scope for this project. At the same time, the need for a large galaxy is not great since exploring it would take too much time for the player and though maybe not preferable, a new one can always be generated.

The implementation in this project is based on Evan's idea but adapted for two dimensions which could be done without much trouble. A core is generated with 2-6 arms spread out evenly around it with each star rotated around the centre to create a swirl effect. The result is a fairly realistic looking spiral galaxy as can be seen in figure 3.1.

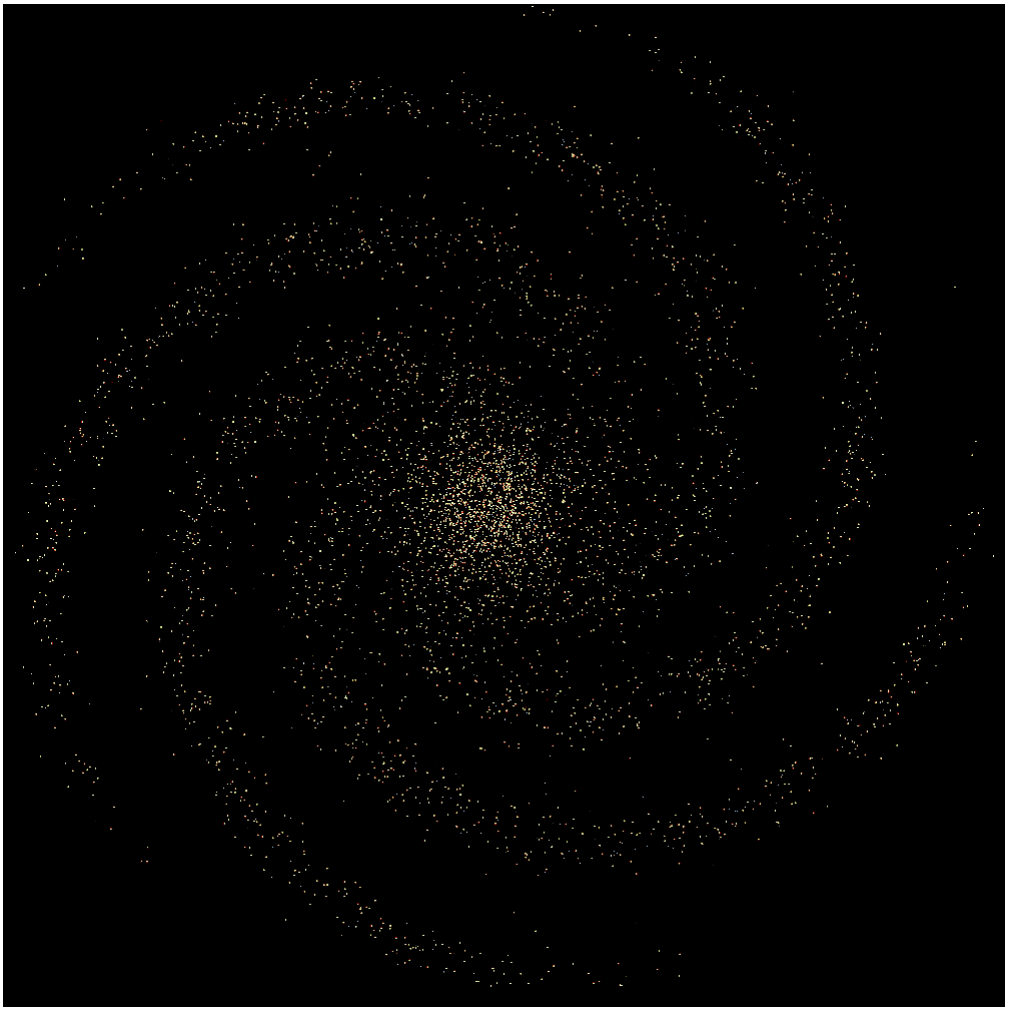


Figure 3.1: A generated spiral galaxy

The next issue was to implement some variation to the generated stars. This was done while keeping in mind both what is suitable for the game as well as some realism. Again, what most players would recognise was chosen and implemented with some degree of realism, e.g. the types of stars and their mass and temperature.

This is also when the aliens, and their presence throughout the galaxy, was implemented. A random solar system was picked as the alien home planet, and then the level of alien presence for each solar system was set based on that. Lower presence

further from their home planet was desired, but at the same time, either extreme (none or max presence) should sometimes occur even if the star is close or far away from the alien origin as this would showcase how varying levels of alien presence affect the generation of planets without forcing the player to travel long distances.

3.2.4 Solar System

The generation of the solar systems, like the generation of the galaxy, did not have to be very realistic. The important part was that it did not actively seem too unrealistic. Due to this, the model for the solar system was kept as simple as possible, with little to no scientific basis behind the design choices. The reasoning behind this decision is that the solar systems and their planets serve as no more than a vehicle for level selection. Furthermore, since the purpose of the project is not to generate scientifically valid solar systems, it was deemed unnecessary to waste valuable time on any physics simulation.

As a result of this choice, there are many phenomena that can be seen in the real universe which were deemed unnecessary. For example, it was decided that neither black holes nor star systems with several stars orbiting each other should be included. Furthermore, some simplifications to the model were merely done because they would not be conducive to the game idea. For example, it was decided that no gas giants should be included as they cannot be conventionally explored.

With all that in mind, it was decided that the solar systems should consist of a single star with at the very least one planet orbiting it since there would be no point to a solar system with no planets to explore. Figure 3.2 shows an example of a generated solar system.

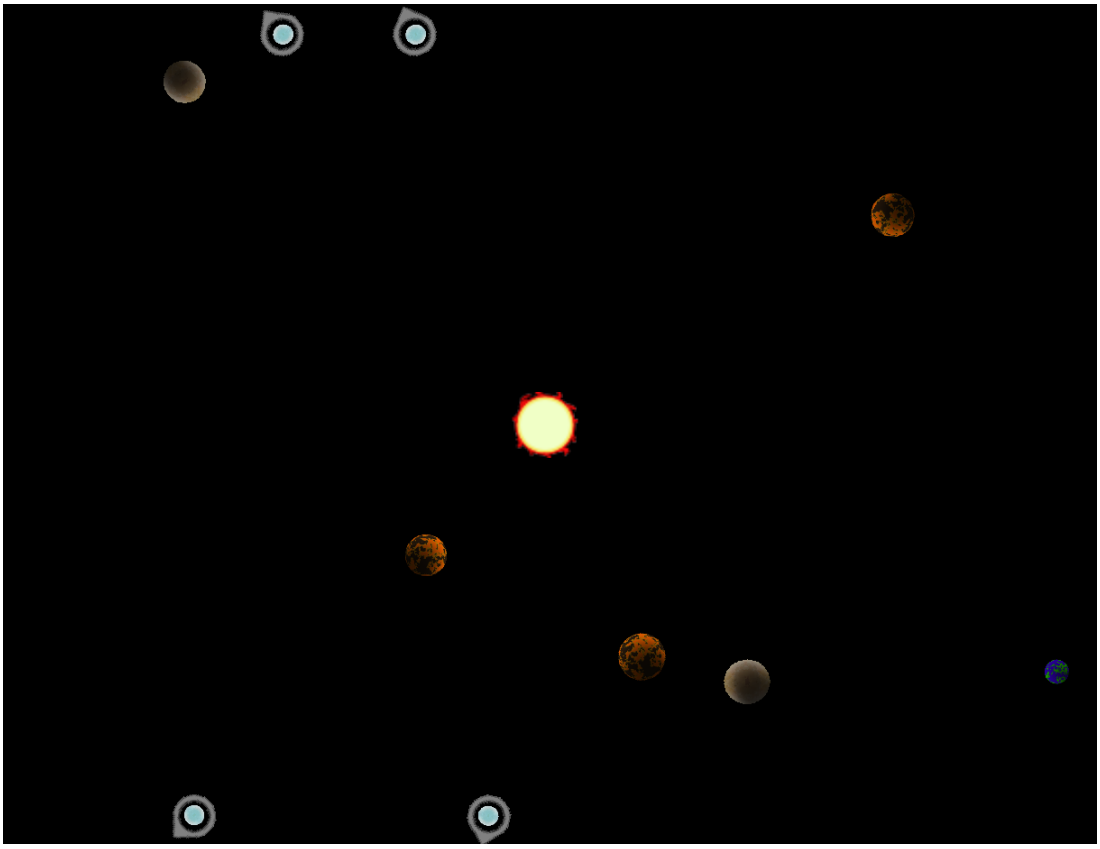


Figure 3.2: An example of how a generated solar system can look. The sun is in the middle, surrounded by planets of different types. Planets outside the view of the screen are shown by markers along the edge of the screen.

3.2.5 Surface

Surface generation was considered a key focus since the surface would be what the player first sees on a planet. Thus, there needed to be a significant amount of variation for the surfaces, to make planets distinguishable and provide opportunities for exploration. The goal during this stage was to get a functioning terrain by beginning to test a few methods. A review of the research field revealed that noise functions, especially fractal noise functions, are a common way of generating terrain. This because they can create somewhat realistic-looking terrain without being too computationally heavy. Many are also quite simple to implement. Three oft-mentioned noise functions were tested: midpoint displacement, value noise, and Perlin noise.

For this first version, value noise and Perlin noise were done in a single pass only. The implementations were not perfect at this point; they could generate a surface, but they were not polished. The Perlin implementation, in particular, was not implemented correctly. Figure 3.3 shows the sharp, unnatural turns it produced at this time.

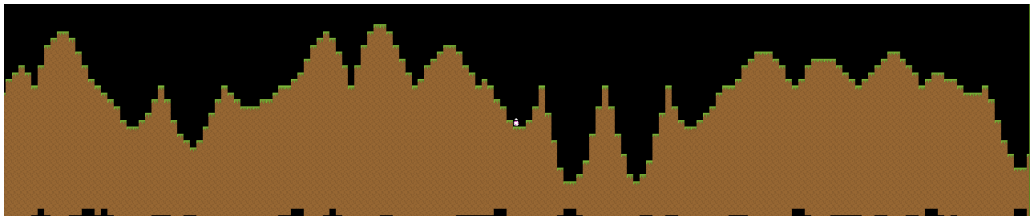


Figure 3.3: An example of the unnatural look of the Perlin noise algorithm, resulting from a faulty implementation.

No decision was made at this stage about how to use these algorithms. Some possibilities that were discussed were to choose one and use only that in the final product or use two or three and change which was used based on what the planet's parameters were. Perhaps one would be more suitable for a windy planet than the others. It was decided that the algorithms should be tried out and evaluated for usefulness later.

The surface generation also needed an interface with the cave generation. They both had access to the same level grid object, so they needed to be compatible. To keep them as separate as possible, it was decided that the surface would take care of everything above a surface level that was set for the Level. In this way, they were not dependent on what the other did and could be executed in any order. However, this total separation led to some problems and needed to be changed later (see section 3.3.4).

3.2.6 Caves

Another very important part of the levels are the cave systems. Cellular automata are a common way to generate caves, and for this project, an online tutorial [46] was used as a starting point for the implementation. The algorithm described is a simpler version of Conway's game of life. In Conway's game of life a living cell can die both from starvation (too few alive neighbours) and from overpopulation (too many alive neighbours), but in the simpler version, the overpopulation criterion is removed. Instead, a cell becomes alive if enough of its neighbours are alive and it only dies if there are too few neighbours that are alive. If there is a difference between the birth limit and death limit, then the state of the cell does not change if its number of live neighbours is between the two limits. This led to fairly good results, as shown in figures 3.4 and 3.5.

Figure 3.4: The earliest implementation of a cave.

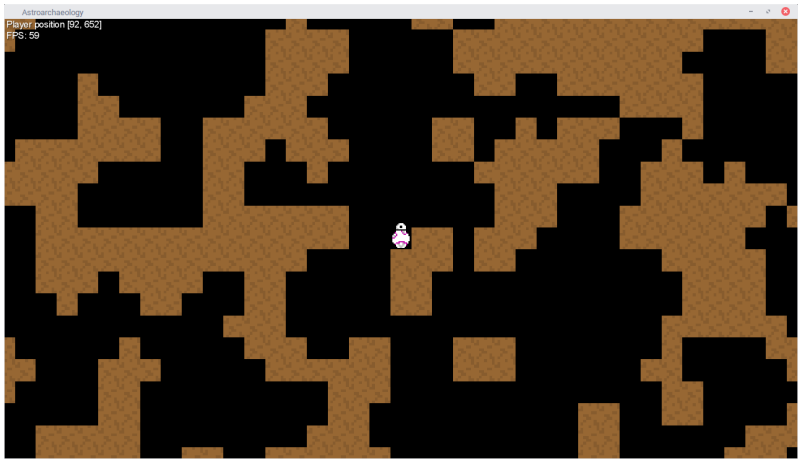


Figure 3.5: A slightly improved cave with a player character, although still producing narrow passages.

3.2.7 Name Generation

A decision was made to generate names for stars and planets, in order to make them easily distinguishable from each other. This also provided the opportunity to investigate another type of generation.

As mentioned in Chapter 2, both Markov chains and formal grammars can be used to generate text, and by extension, they can also be used to generate names. Therefore, the first order of business was to decide on which method to use.

The primary requirement for the name generation was that it must be able to

generate a considerable number of names due to the immense number of stars and planets that are generated. Fortunately, this requirement was all that was necessary for a decision. While the fact that formal grammars need to be designed by hand does ensure that all the generated names are of high quality, it does make it much more difficult to generate a large set of names. Markov chains, however, only need to analyse a training string to be able to generate an extensive set of names which makes it a much more appropriate approach to this problem.

Since the theory behind Markov chains is relatively simple, an n-gram Markov chain could be implemented quickly and without any issues. More specifically, n was set to two as this provided a good balance of quantity and quality.

However, some issues still needed to be solved. One of the most important issues was the difficulty of finding good input data to be analysed. In the end, this proved to be the most challenging issue as experimentation was necessary to find a set of data that resulted in output that felt fitting for astronomical objects. A set consisting of the names of Roman deities, Roman place names and constellations resulted in output which was, for the most part, satisfactory.

3.3 Feature Completion

During this phase of the project, most of the work was put into combining the different existing systems into a more cohesive whole and refining them from their initial implementations into something better.

3.3.1 Parameters

At this stage, work began on the parameters that should influence the generation of planets. This work consisted mostly of deciding which parameters should be included and how they would depend on each other.

The parameters were discussed in two layers, for the stars and for the planets. The stars were given a few parameters: alien presence, temperature, mass, and state, with state meaning whether the star is a white dwarf, a regular star, a giant, or a supergiant. These types of stars were chosen as kinds of stars that players would likely recognise.

The planets were given a larger number of parameters: size, temperature, gravity, humidity, wind speed, and alien presence. These were chosen based on what seemed likely to be possible to integrate with the level generation and give a noticeable result. The planets were also given planet types, which included ice planet, verdant planet, desert planet, and lava planet. A planet's type was calculated based on its temperature and humidity, and different tilesets were made to distinguish between the types. The aim was also that later, the level generation would take the

parameters into account in a way that would further differentiate the planet types.

3.3.2 Caves

The caves up to now had been quite narrow and difficult to navigate. To solve this, different values for the automata birth and death limits were tried, as well as the initial distribution of wall and space tiles, but this made little difference. What did help was including more neighbours in the calculation of each cell, making a cell depend on all cells within two tiles in each direction. This is commonly done in cellular automata, and it helped make it easier to fine-tune the death and birth limits since the granularity became greater.

Earlier, the caves had had a very strict area in which they were generated. This led to a sharp line where the surface generation began, and the mountains in that part of the map had no caves, as can be seen in figure 3.6. It was decided that the caves should be made to go up all the way to the surface and to do that, the interface between the surface generation and cave generation was changed. Instead of having the algorithms operate on separate parts of the map, the surface generation was now done first and simply marked tiles close to the surface. The caves were then generated on all tiles below the marked ones.

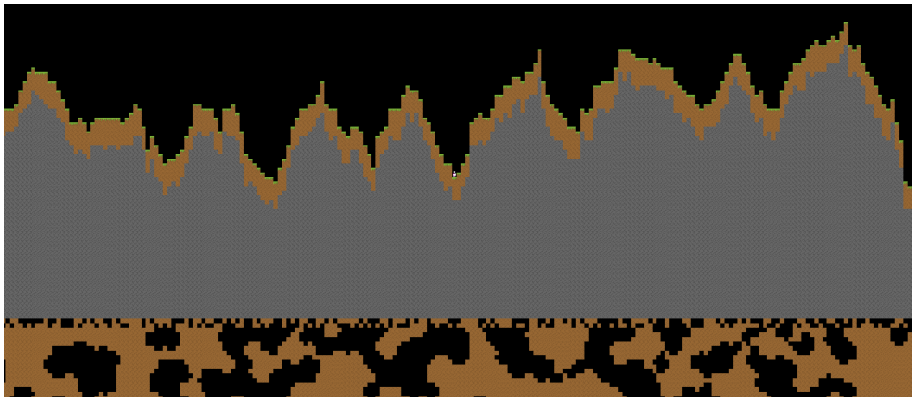


Figure 3.6: A clear distinction can be seen between the caves and the mountainous surface.

Cave openings were also implemented during this stage. At first, the idea was to create the surface first and then mark spots that should be made into openings to the caves on it. Then the cave generation should be done in such a way that it would always generate reasonably large connected systems from the opening spots.

At first Diffusion Limited Aggregation (DLA) was considered for this specific cave generation, which would guarantee paths leading away from the opening. DLA's ten-

dency to create branches seemed promising for creating paths in the caves. However, this approach proved to be quite a time costly procedure due to the random walk element of the algorithm. It would frequently freeze the game for several seconds during the level generation, so another approach was needed.

The new approach was to generate the caves before the openings, and then make openings in suitable places. This was done by finding a random point on the surface, then checking whether there was a large enough cave beneath it. Given the random element, repeating this until the preferred number of openings had been made could theoretically result in long computation time, so the number of tries was limited to 200.

The question of how many openings there should be on a planet was left until later.

3.3.3 Parallax Background

A simple solution to further create a sense of immersion to the planet's surface was to implement basic parallax scrolling backgrounds, a technique which many games have employed since the early 80's. Introduced in 1982 with Irem's Moon Patrol [47], parallax scrolling is a technique where different speeds are assigned to different elements of the background and the foreground [48]. Basically, elements which are further away from the player move more slowly, which creates an illusion of depth and more realistic movement in an otherwise flat 2D scene.

The idea, in this case, was to generate mountains in the distance, which in itself was a fairly straightforward task as multiple algorithms for terrain generation were already implemented. To keep it simple, it was decided that the number of background layers should be limited to two in this implementation. To preserve performance, it was decided that each parallax should not generate infinite terrain, but instead only produce repeatable chunks as this was believed not to make a noticeable difference to the player. Thus each layer is made out of two images at the size of the game's resolution that are always drawn beside each other. This is shown in figure 3.7. As one image distances off on one side of the screen, it is moved to the other side preventing any gaps in the background.

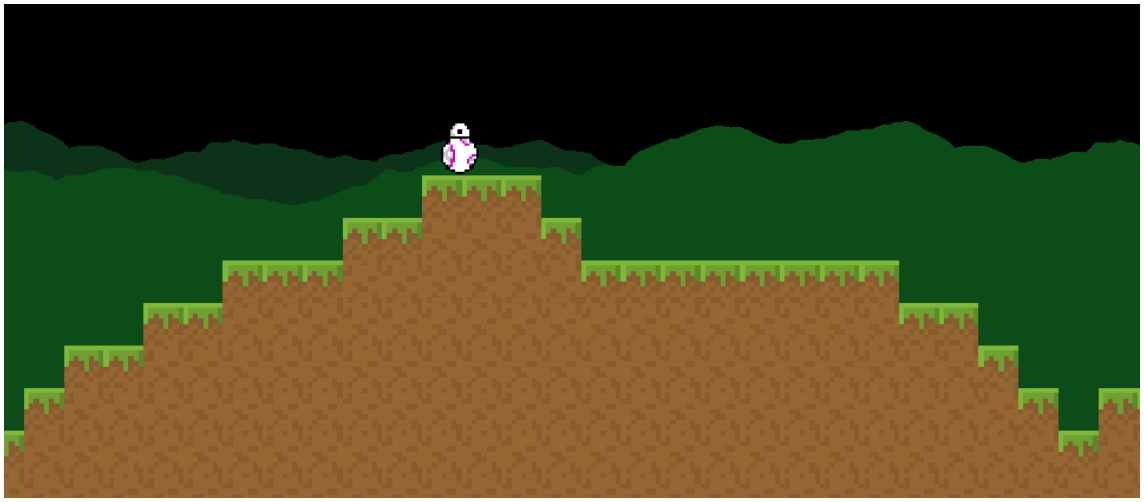


Figure 3.7: An early version of parallax backgrounds, where both images are set at the same average height.

Initially, the layers only tracked the camera's horizontal movement. This did not look very pleasing, so a slight vertical tracking was also included. At this point, some artefacts started to appear. For example, it was noticed that the terrain was generated at a different altitude depending on the algorithm, causing the parallax to look misplaced at times. Furthermore, there was no requirement at this time that the terrain had the same altitude at the start and end position, and thus sharp edges could occur at the point where the image was repeated. This led to the realisation that the terrain generation needed to be refactored. After this was finished, the angles of the slopes near the start and the end of the images were similar enough to look natural.

3.3.4 Surface

At this stage, the work done on the surface revolved around fixing bugs and adapting it to other systems, most notably the caves and the parallax. The bugs in Perlin were fixed so that it also produced acceptable terrain.

The parallax system needed to use the surface algorithms several times and needed them to be able to mesh together without any noticeable seam. Value noise and Perlin noise did not necessarily end and begin at the same height, see figure 3.8. To fix this, the terrain was simply set to always start and end at height 0. Midpoint displacement already did this from the design of the algorithm, and value noise was well suited for this simple fix since it works by interpolating between random heights either way. Perlin was a bit more complicated since it works by slopes, merely setting the height to 0 could still result in a very bad match-up of slopes as two heightmaps were joined, see figure. This was resolved by setting the slope at the first and last point to zero as well.

Figure 3.8: The ends of the surface segments used for the parallax did not match up.

The noise functions used for the surface all generate heightmaps, which means that more irregular terrain formations, such as raised plateaus or cave overhangs are improbable or impossible. Finding another way to include features such as these was discussed, however, the only relatively simple method found was to create chunks of terrain with these features and then placing them during the generation. This solution was judged as being too focused on handcrafting for this project and was therefore not included.

Another extension to the terrain generation could be to simulate erosion, which usually works by dissolving material from slopes in the terrain and transport it downhill. This often smooths out the surface, enhances steep slopes and creates a flatter base, which may result in a more realistic looking terrain [49]. It can however also be used to do the opposite to create more elevated plateaus [30]. It is not clear how well erosion would have fitted into this project, as the smallest unit in the terrain is a tile which is quite large. As it was uncertain how the result would look, and it was expected that integrating erosion could be very time consuming, it was set aside for more highly prioritised features.

The final work done on the surface during this stage was that the planet types were made to affect which surface algorithm would be used. No other use of the parameters was implemented yet.

3.3.5 Rooms Underground

At this point, a rudimentary cave generator was implemented, but it did not generate anything particularly interesting yet. Many games in the roguelike genre have dungeons that consist of only rooms connected by corridors [6], and that is how the idea of combining natural looking caves with artificial rooms inside of them first came up.

At first rooms of random sizes were simply placed at random in the cave, which meant that rooms could overlap, something that often did not look very good. Instead of facing the problem of classifying if an overlap is inappropriate or not head on, the problem was circumvented by not allowing any overlap at all which was achieved by using Binary Space Partitioning to divide the map into smaller, non-overlapping, parts where it is certain that rooms can be placed without creating any overlaps. This partitioning also allowed the rooms to be grouped in small clumps instead of being more evenly spread out. This choice was made because it seemed more realistic and perhaps more interesting.

The next big challenge with creating rooms was the creation of reasonable doors. Since the rooms are placed in an already existing cave, the doors should be opened so that the room is reachable from the rest of the cave and so that the player can get deeper into the cave system. Preferably, there should be doors in the room that lead to all different openings in the surrounding cave, but there should not be any doors which lead straight into a cave wall. The difficulty lay in identifying unique cave rooms so that only one door is opened to each of them. This is because it does not generally make much sense to have two doors right next to each other that lead to the same place.

At first simply making a fixed number of doors for each room was tried. It worked by simply choosing a wall block at random and if it had a large enough cave outside it, it was removed together with one of its wall neighbours. This approach led to a few problems however. First of all it was possible for doors to end up far too close to each other and lead to the exact same cave, which does not make much sense and looks quite bad.

3.3.6 Surface Buildings

To make the surface more interesting and to showcase the presence of aliens on the planet more, it was decided that buildings should be added to the surface. A seemingly simple solution was to find flat areas and create rectangular rooms on top of these, with doors on each side to allow the player to access them. Building several rooms on top of each other was done to create several floors for each building and running the algorithm more than once enabled smaller rooms to be built on top of already existing rooms to create tower-like structures. All of this was scaled depending on the level of alien presence on the planet. Low alien presence generated

some basic structures to resemble a small colony while higher levels generated both taller and more numerous buildings to resemble a city of some sort.

Due to the nature of the surface generating algorithms, most structures were generated on top of hills or at the bottom of valleys, as most other spots were not flat enough. The severity of this problem is dependent on which surface algorithm is used, which in turn depends on what type of planet it is. Another problem was that the rooms only have one shape which means there is little variation in the result. While the generation still had some problems it was deemed good enough to focus working on other parts instead.

3.4 Expanding and Polishing the Game

During this stage, most of the key features that had been planned from the start had been brought into working order. What was left was to properly use the parameters from the galaxy and solar system to adjust how the planets looked. Other than that, some extra features such as fire and decorative items were brought into the game. This stage also held a fair amount of bug fixing and refactoring.

3.4.1 Parameters

At this stage, the properties of planets on the solar stage were changed to depend both on the parameters from the galaxy stage and on each other. This change was made to increase the cohesion in the game world by attempting to give more reason behind the values.

Afterwards, the work with parameters mostly consisted of actually integrating them with the other systems on the planet, which unfortunately proved to be difficult. The problem was using the parameters in a way which could give noticeable variation in the generated content beyond choosing different planet types or surface algorithm etc.

3.4.2 Game Mechanics

As the project progressed, a few additions and improvements were made to the basic game mechanics. The upwards force applied to the player character was no longer constant as the player pushed the up key. Instead, the force gradually increased as the up key was kept pressed, up to a maximum force. This change made the controls feel a bit more natural, similar to an engine working its way up to maximum power, and made it easier to fine-tune the player's altitude. In addition to this, the gravity that the player is affected by was changed from a constant value to one based on the gravity generated for the current planet. Furthermore, to facilitate debugging, a debug mode was added. While not very extensive, it enables zooming in and out,

so that it is easier to see the whole level or minor details, and the ability to break blocks, to be able to traverse the level in a faster manner.

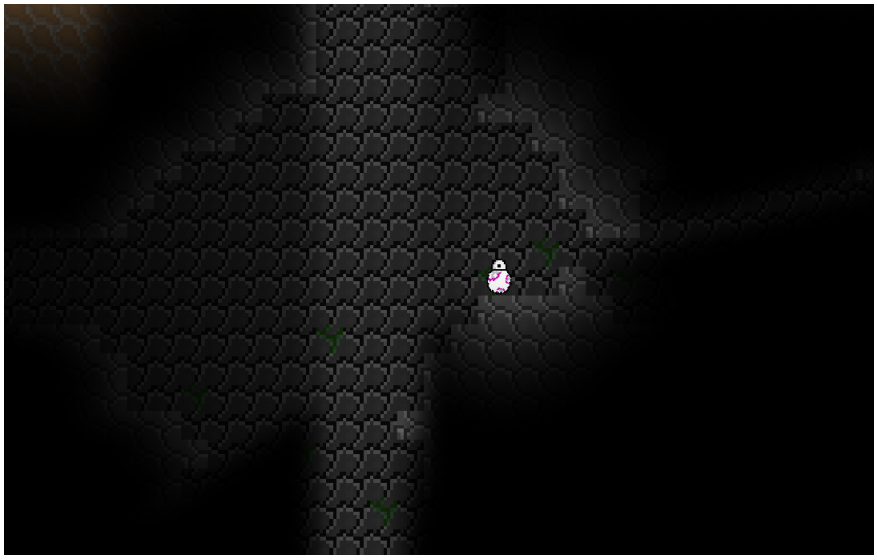


Figure 3.9: The player character in a cavern close to the surface, with sunlight spilling down into the cavern.

In order to make the game a bit more visually interesting, the lighting engine Box2DLights was implemented into the game, as seen in figure 3.9. It had the added benefit of enhancing the exploratory part of the game. By only being able to see that which was lit up by the sun or the player character's light, it made it more tricky to find a way through the dark cave systems.

Figure 3.10: The main menu of the game, with an input field for a seed.

Up until this point the galaxy was always generated from the same initial randomisation value, also called a seed. To more quickly be able to try out different seeds and to give a less abrupt introduction to the game after start-up, a menu screen was added, see figure 3.10. In this menu screen, the player can enter any arbitrary string, which is then parsed into a seed. The menu made it possible to generate galaxies based on different seeds while still including the ability to reproduce the same galaxy again thanks to the deterministic generation of the system in an intuitive way.

3.4.3 Fire

The procedural content generation discussed so far has been focused on static environments and play areas, i.e. things that do not change over time. However, it might be interesting to explore systems which are dynamic and change over time, for example, having some fire that spreads while the player is exploring the level. Simulating how fire spreads by using cellular automata is an established method [26] and was therefore used for this purpose.

The basic idea of how it would work was that fire would be able to start somewhere in the cave during runtime randomly and would then spread by using cellular automata. Unlike in Conway's Game of Life, there was no reason for cells to die due to overpopulation as in this case that would represent fire being extinguished because of too much fire nearby, which did not seem very reasonable. In the game of life cells also die because they have too few neighbours; this could certainly be implemented to some extent to represent fires dying out because they are not hot enough. However, this did not seem like a good rule either, since fires often start from a small spark and then spread to become very big. Instead, it might be more reasonable to let a cell die after burning a certain amount of time, representing the fire running out of fuel in that cell. Finally, a rule that allows the fire to spread was required. In Conway's Game of Life, that rule is simply that any cell with precisely three alive neighbours also becomes alive. That did not seem suitable for fires since, as mentioned, fires can spread from a single starting point.

Other options considered was to have the limit at one or two alive neighbours. Regardless of which of these options are chosen, it was deemed a good idea to introduce some uncertainty and give the fire a certain chance to spread if able. This was to make sure the fire does not spread too fast, which is also a good idea from a gameplay perspective since it would be quite boring if the fire always spread across the entire map if it starts somewhere. The advantages of spreading with just one alive neighbour could be that it is realistic in the way that there does not need to be a large fire for it to spread, but the downside is that the fire can spread in a very irregular way as can be seen in figure 3.11.

Figure 3.11: In this figure the fire has a chance to spread to a cell if that cell has at least one neighbour that is on fire. The fire is moving upward and in the middle there is a rather large line of fire that by chance got ahead of the rest.

If the fire only spreads to a cell if that cell has two or more neighbours that are burning, then it looks more like in figure 3.12. The fire stays more grouped and spreads in more of a wall shape which was considered more realistic and is why this option was implemented.

Figure 3.12: In this figure cells need two neighbours on fire to start burning. Here the movement of the fire is more cohesive.

3.4.4 Trees

At this point, the above-ground portion of the levels was mostly empty, aside from the occasional building. One of the ideas which were proposed to remedy this was to generate some vegetation for the levels since that could be appropriately done with PCG [1, p. 73].

The development of the tree generation went through many changes. At first, simple trees were generated with a recursive method as this was very simple to implement. However, this approach quickly proved not to be very flexible, and it was difficult to generate different kinds of trees with it aside from the very simple tree seen in figure 3.13

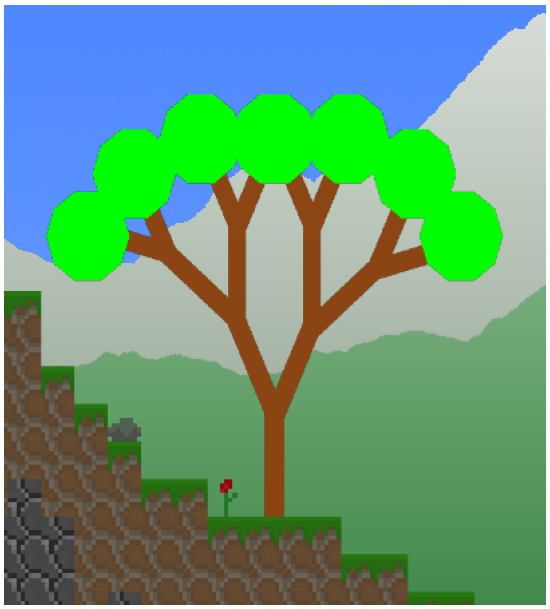


Figure 3.13: Recursively generated tree.

As a result, the recursive approach was scrapped in favour of L-systems. Initially, L-systems were ignored because of their perceived complexity relative to time. However, since this appeared to be the industry standard, with plenty of information available, it was deemed worth the effort in the end albeit with some compromises to simplify the development. A majority of the examples illustrated in The Algorithmic Beauty of Plants by Prusinkiewicz and Lindenmayer, [21] are rendered in 3D, with only a few of the examples beinf rendered in 2D. Since implementing the system in 3D seemed much more involved than doing it in 2D, it was decided to limit the implementation to 2D even though it would limit the strengths of the generation. Furthermore, only the simplest kind of L-system was implemented, neither stochastic, nor context-sensitive nor parametric aspects were planned to be included due to their increased complexity.

Since L-systems are a kind of formal grammar, it was first necessary to implement a system which can hold a set of rules and use them to expand a string. After that, the graphical aspects were implemented. With this in place, it was possible to generate trees based on grammars. However, a specific grammar always generates the exact same kind of tree, so variation between individual trees had to be accomplished elsewhere. Variation was achieved by introducing variables which govern the way in which an individual tree grows. Since time was limited, only a few variables were added, for example, initial height and width, height and width contraction and a rotational angle, although it is possible to have much more specific variables. All of these are chosen within an interval which had to be manually configured since most values do not lead to satisfactory results. In fact, a significant amount of time was spent fiddling with both the graphical aspects of the trees as well as the variable intervals since it was difficult to make everything satisfactory. More experience with LibGDX and OpenGL might have made this process simpler.

At this point, the actual tree generation was finished, however, since all trees were rendered directly in the level the trees came with a rather hefty performance loss. By rendering each tree to a texture and then simply rendering the texture instead, it was possible to decrease the computational complexity of the trees drastically. After this, the tree generation was considered satisfactory and thus finished.

3.4.5 Surface

The surface had been almost completed for a while, but during this stage, some more bugs were handled. Perlin was not smooth, in many places one single block would be either one step above or below its neighbours, which was determined to be a problem. It was solved by simply using a smoothing function on the result, setting every point to the average of itself and its two neighbours. Other than that, Perlin and value noise both had troubles with their end sections. Since both methods work by calculating the heights of most points by the two nearest fix points, they both needed to have a fix point at the end of the level. Because of rounding issues, this meant that the final segment could be shorter than the others, sometimes much shorter. In some cases, this led to abrupt height differences (as seen in figure 3.14) since the algorithm would be trying to fit the same height change into a smaller lengthwise distance. This problem was avoided by removing the second to last fix point. The last segment would then be longer than most others, not shorter, resulting in more drawn-out height differences instead which resulted in much less noticeable seams between the endpoints and their neighbouring point.

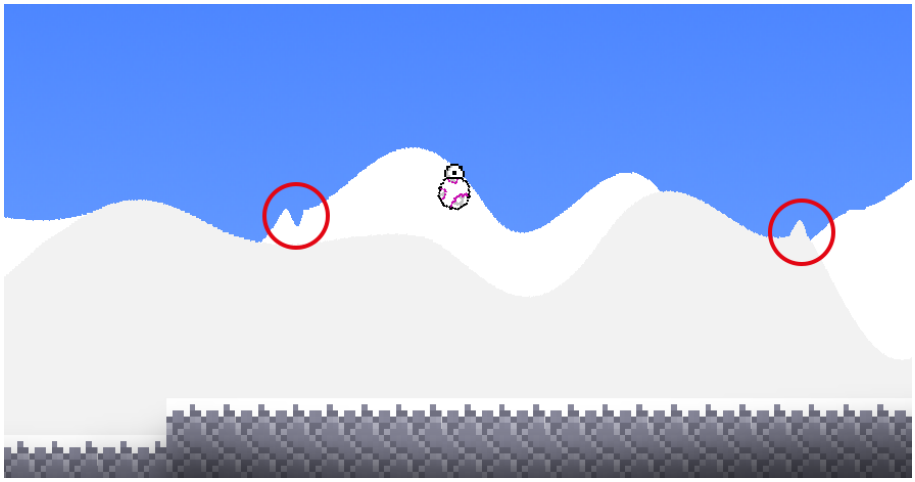


Figure 3.14: Small, steep bumps in the parallax, which were fixed in this stage.

In this stage, the Perlin noise and value noise algorithms were also redone to use several passes, for an approximation of fractal noise. This led to the results looking more natural.

Other than fixing bugs, the final adjustment to the surface was to involve the planet's parameters more in deciding how the surface would look. All three algorithms were made to make gravity affect the height of the peaks. Midpoint displacement was also made to use the wind speed to determine the roughness variable: higher wind speeds meant less rough terrain, emulating the stone being smoothed by the elements.

3.4.6 Liquids

Initially, a water system was to be implemented using cellular automata, with each cell containing their level of water. Research has shown that it is possible to use cellular automata to create both complex models of liquids as well as simpler implementations which are more suitable for use in games [50][51]. During the development of the water system, it became apparent that it could be reused for liquids in general if some parameters were changed. In this project, it could be suitable for both water, lava and perhaps some fluid alien substance.

Using a simpler implementation, a cell would be one game block in size. Each cell would then update its values depending on the previous state of the system and a few rules. This presented a few problems; since each cell updates purely depending on the previous state, more than one cell could flow into the same cell and create an overflow of water. Handling this could be done by returning the extra water to the neighbouring cells on the same height coordinates. Another problem was how the water would flow in discrete steps, i.e. one tick empty, one tick filled, which is very unnatural. Allowing cells to update depending on the current state would

partly solve this, but the problem would still be present if the cells were updated in the wrong order. How the water would be displayed when flowing down sides was another problem.

Without a clear solution at the time, a different approach to the water system was explored. Creating a water cell for each 'pixel' in the game could create a smoother simulation. At the same time, allowing the cells to update on the current state of the system while making the different directions of movement into separate steps further smoothed out the simulation. As a result, all cells first tried to move down, and then if they had not moved, they tried to move sideways. Keeping the direction of a cell when moving sideways also created a more coherent movement, i.e. each cell has a sideways direction stored to stop it from moving back and forth between two cells.

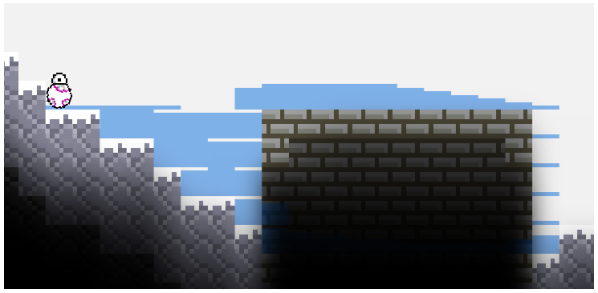


Figure 3.15: Block-based water cells, showing some issues of overflow and visual representation.

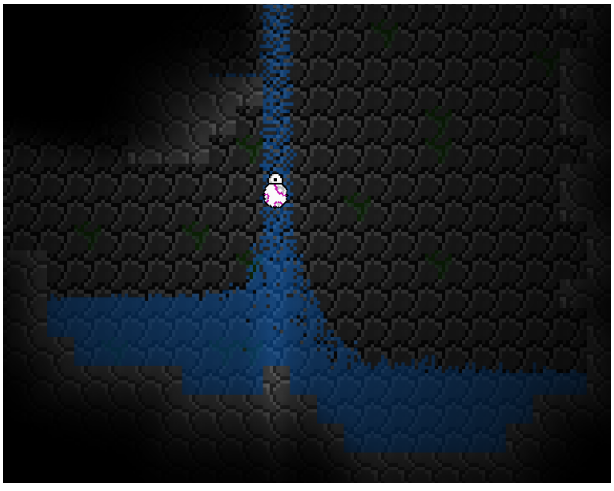


Figure 3.16: 'Pixel-based' water falling and spreading out.

While this system created a more realistic and smooth simulation in most cases (see figure 3.15 compared to 3.16) it still had some glaring problems that could occur if water spawned in certain ways. This system also presented a scaling issue. A game block full of water would be 64 cells, with each of them being frequently updated. Solving this could be done by either not updating cells outside some distance of the player, or by marking stable cells not to be updated unless changes to the terrain occur. With limited development time left and problems with both scaling and the movement of the fluid, the liquid system was not included in the final product.

3.4.7 Tile System

As was explained in section 3.2.1, the data structure of a planet level consists of a large matrix, where each cell of the matrix holds the information for one tile of the game world. Initially, this tile data was just a very simple index system. For example, if a cell contained the number one, that tile would be a cave-rock tile, and if it contained a 3, it would be a dirt tile. All of those tiles were defined arbitrarily, and this system worked well on a small scale. However, it was quickly shown not to be flexible when new tile types were added to the game. In this system, each time a new tile type was added, extensive changes had to be done in the code. For example, even when simply adding another kind of simple, solid tile, it was necessary to update the line of code where collision detection is done, adding the tile's number so that its type is included in the collision detection.

To fix this issue, the system was redesigned. Instead of the matrix holding numbers, it instead holds references to predefined tile types. These tile types each have a number of tags associated with them which other systems can access. These tags can be things such as Solid, Diggable, Background or other such attributes that different systems might care about. This change makes it possible for any system which depends on tile types to simply check if the tile has the relevant tag. As a result, it is much simpler to add new types of tiles, as all that needs to be done is to create new sets of tags describing them. In the same way, if a system is interested in a specific set of tiles, one need only create a new tag describing the tiles and add it to them.

Additionally, even though much more data can be accessed from each cell in the matrix, not much more memory is used. Each entry in the matrix is merely a pointer to an enum definition; there is no class instantiation happening.

3.4.8 Decorative Items

In order to increase the number of things to find on the different planets, a system was designed for adding decorative foreground and background images to the levels. Some examples of such items are shown in figure 3.17. This system was implemented by adding an additional matrix to the level data class. This matrix had a list for

each tile in the world, and that list then contained decorations to be drawn there.

It quickly became apparent that certain decorations needed to be linked to certain tiles, since if they were not, flowers could start floating mysteriously in the air if the ground beneath them was removed. The link would ensure that the decoration would be removed along with any of the linked tiles in case the tile was destroyed (for example by a generator modifying the world after a decoration has been placed). The linking was implemented by letting each decoration store the tile positions they are dependant upon, and then get notified whenever a tile is destroyed. If the destroyed tile matched one of the ones the decoration was linked to, it removes itself from the decoration list containing it.

Considering that many decorations might be added, it was deemed important that the process of creating and adding new decorations is as simple as possible. This was achieved by requiring, at most, width and height of which the decoration is dependent on. This width and height simply state which other tiles around it the decoration is dependent on. An extra pass during the level generation then goes through the level to find suitable spots for them. For example, flowers are placed with a certain percentage chance above every grass tile on temperate planets.

Figure 3.17: A few decorative items such as rocks and flowers on the ground.

4

Results

This chapter presents the results of the project, divided into four different aspects of it. Firstly an overview of the finished game and its mechanics. Following that the underlying code structure and architecture is explained in more detail. The same is done for all of the different PCG systems of the project. The chapter finishes with the thoughts and conclusions about the different PCG algorithms.

4.1 *Astroarchaeology*

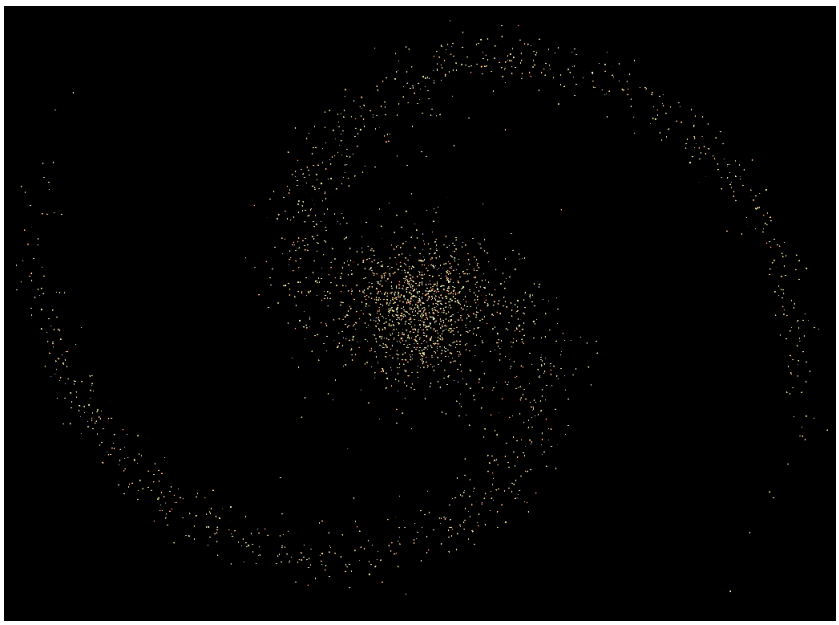


Figure 4.1: One example of a generated galaxy

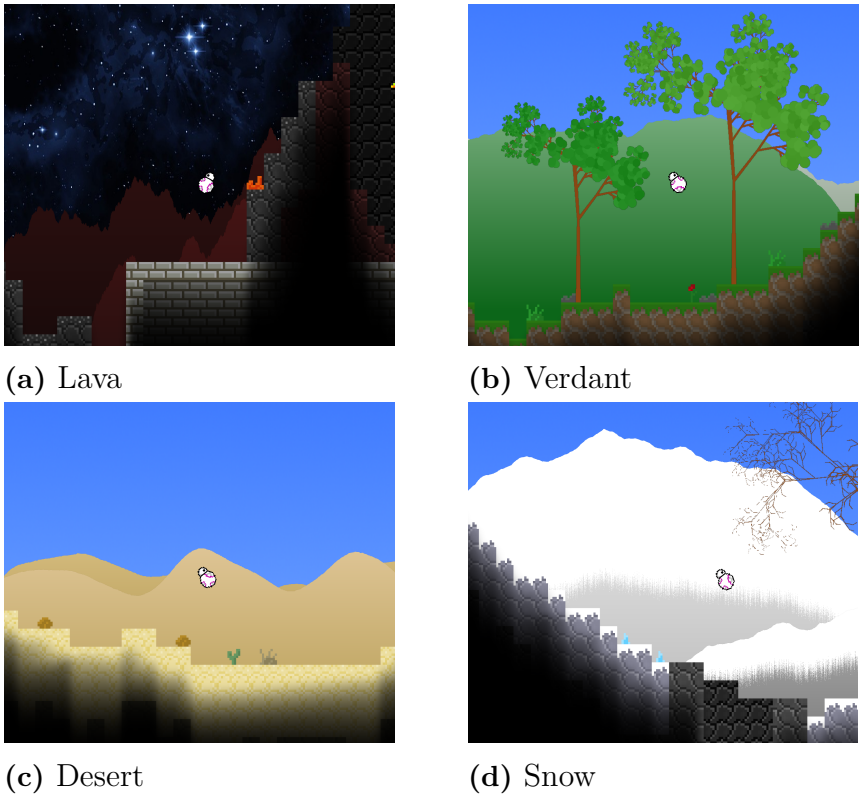
As the game starts, the user is prompted with a text field asking for a name for the galaxy that will soon be generated. The submitted string is hashed into a seed for the generation, meaning that a specific name will always result in the same galaxy being generated without the need for any data storage. As the seed is submitted,

the user is presented with a spiral galaxy containing a few thousand stars, ranging from white dwarfs to fierce supergiants, all with uniquely generated names. Figure 4.1 shows the entirety of one such galaxy. Somewhere is the origin of an ancient alien civilisation, and the remains from their colonisation can be found scattered around the galaxy.

Figure 4.2: One of the many solar systems found in the galaxy

Each star constitutes the centre of its own solar system (see figure 4.2), where a few types of planets have the possibility of existing. Closest to the warm sun, fiery magma planets are usually found, with black ashes covering the rough surface. As the temperature decreases further away from the star, cold and rocky planets with a subarctic setting are more often encountered. In between, where the temperature allows for it, more welcoming planets can be found. Most commonly those contain dry deserts with sand dunes stretching away into the horizon, but if there is an atmosphere with high humidity in place, verdant planets may also be found.

As the user hovers over a star or a planet with the cursor, an information screen appears, displaying all necessary data about the celestial object in question, such as temperature, gravity and mass. To provide visual feedback, the planet hovered over is also highlighted. As the stars are always displayed at high brightness and are often clustered close together, stars are instead slightly enlarged when hovered over.



(a) Lava **(b)** Verdant

(c) Desert **(d)** Snow

Figure 4.3: The different environments the player can encounter

When the user clicks on a planet, the planet is immediately generated from scratch. The player undertakes the duty of controlling a robot named 'Rover' which falls from the sky onto the surface of the visited planet. The world is divided into an invisible grid, where a single type of terrain occupies each square in the grid. The surface varies depending on the climate of the planet, from smooth dunes to vast, rocky mountains. Figure 4.3 shows the possible climates. Parameters such as wind, humidity and gravity may also affect the final, more subtle properties of the surface. As the player starts exploring the surface, he or she may encounter alien structures given that they have once colonised the planet. The structures can vary from small houses to tall skyscrapers with several floors. The player may also find a cave opening, serving as an entrance to the world beneath the surface, where they may explore the connected cave systems.

4.2 System Architecture

In this project, the game development library LibGDX was used as the initial starting point for the system architecture. LibGDX provides some architectural tools, and for this project a simple architecture with inspiration from those tools was crafted. The different game parts are divided into different stages (i.e. Galaxy, So-

lar system and Planet), where each stage is composed by its own theoretical MVC pattern, though it is not separated in different components as traditional MVC suggests. Instead, each stage is composed of three primary methods - one for initialisation, one for logic updates and one for rendering. Data is kept on a per-stage basis (model), decisions as to how the model should be drawn is handled in the rendering call (view), and modifications to the model are made through the update method (controller).

Stages are managed through a stack-like data structure as illustrated in figure 4.4, which stages can be added to and removed from from anywhere in the system. As stages from a game perspective are hierarchical, any data needed for, e.g. a planet is passed through the constructor from its parent solar system. When a stage is appended or pushed on to the stack, it is automatically initialised, and the topmost stage is always considered active.

This approach makes transitions between the different levels very smooth. When the player wants to return from a solar system to the galaxy, the active stage is just popped, and the player is returned to the state where he/she left the galaxy without any extra processing needed. Furthermore, this also means that content does not need to be loaded into memory until the player visits it, and as the content is generated through PCG, it does not have to be stored on disk either. This allows for a theoretically near infinite set of content to be included.

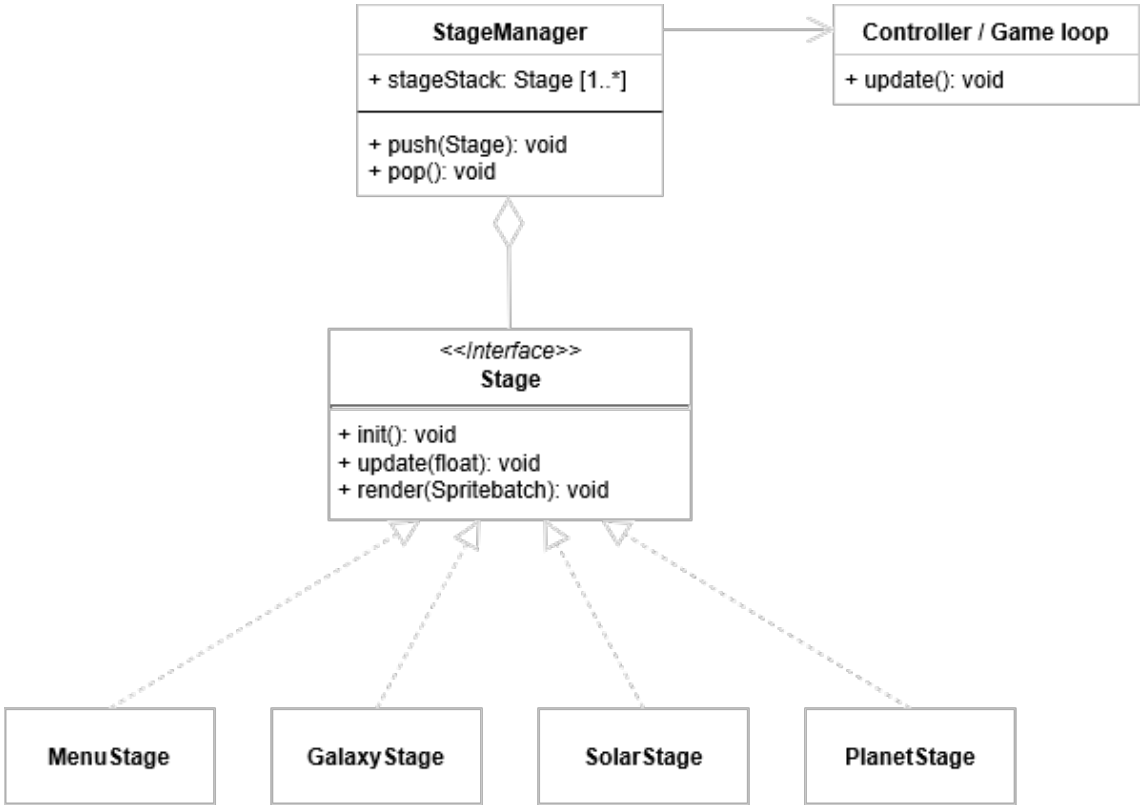


Figure 4.4: A UML diagram showing the architecture behind management of stages

As with most games, a game loop is running at all times. In this project, the game loop is provided by LibGDX, and acts as a controller, triggering logic updates and draw calls sequentially for the active stage. The non-active stages receive no such updates and are such simply kept paused until they become active again.

The tile data for the planet stage is kept within a large matrix, where each cell of the matrix corresponds to a position in the game. Each of these cells holds a reference to one specific tile-type, as well as a set of decorations. For modularity, each tile type can be associated with specific properties through a set of 'tags'. A tile can, for example, be classified as 'solid', meaning that physics will be applied to it, so that the player, light, et cetera will collide with it. Another tag may decide whether or not the tile should be drawn as back- or foreground.

The procedural content generation happens during the initialisation phase of its associated stage. Different generators are run one after another, modifying the data of the newly created stage. In the galaxy and solar stages, only a single pass is done, but the planet stage has a lot more generators implemented. The planet worlds are generated in the order of surface, caves, cave wall decoration, underground rooms, surface buildings, trees, ground- and ceiling-decoration. All of these are implemented as separate generators. It is very easy to expand this system

to include more alterations or additions to the level generation. An additional pass can be added by simply creating a new class that modifies the data in some new way, and letting it have access to the stage data at a good spot in the generation order (if for example windows should be added to buildings, it would be wise to do it after buildings have been added to the level).

4.3 Procedural Content Generation Approaches

In this section, the final implementation of the generators and their results are described more in depth.

4.3.1 Name Generation

The name generation is done with a simple implementation of Markov chains using n-grams. By using a string containing the names of constellations, Roman deities and Roman names of places, the generated names feel like somewhat appropriate names for astronomical objects. Below are some examples of names which can be generated with this dataset.

- Langulum
- Garegius
- Album
- Rumna
- Cernax
- Sces

Because of the immense number of objects which are generated, it was of particular interest to find out how many of the generated names are unique. This aspect was tested by generating a large number of names at once, noting whenever a name is repeated. The results of this can be seen in table 4.1. Note, however, that these numbers are approximate since it is a random process they can vary a bit.

Table 4.1: Amount of unique names and duplicate names generated

Names Generated	Unique Names	Duplicates	Percentage duplicates
100	100	0	0%
1000	950	50	5%
10 000	8078	1922	19.2%
100 000	64 186	35 814	35.8%
1 000 000	461 482	538 518	53.8%

Based on the results, it can be surmised that as the number of generated names increases so does the relative amount of duplicates. This trend can be expected to be stable as there is a finite amount of names which can be generated, so at some point, no more unique names can be generated. However, considering that the amount of unique names generated keeps increasing, and most likely keeps increasing as more names are generated, it seems unlikely that there would be a problem with running out of unique names.

4.3.2 Galaxy and Solar System Generation

The idea of the galaxy generation follows Evans' implementation applied to a two-dimensional environment [44]. A circle of stars is generated in the centre, with two to six arms spread out evenly. The stars are spread out within the core and arms following a Gaussian distribution. Finally, each star is rotated around the centre scaling with the distance to the centre, with stars further out being rotated more. Figure 4.5 shows an example of this.

Each star then has properties generated in a separate data class which can be passed along to the solar system generator when the player decides to visit that solar system. This data includes the position in the galaxy, star classification, temperature, mass, name and level of alien presence. Each property is randomised with the distribution being semi-realistic. The presence of aliens is generated through a Gaussian distribution that is scaled to the distance to their home planet which is a randomly selected star. The result is a sort of linear fade of the level of presence from the home planet while also producing some outliers.

The implementation is fairly lightweight and fast since it mostly used two-dimensional positions that are created and manipulated and all other generation is delayed until needed when the player chooses to explore a solar system. The size of the generated galaxies is around a few thousand stars, which is more than enough to showcase the variation of solar systems and planet levels.

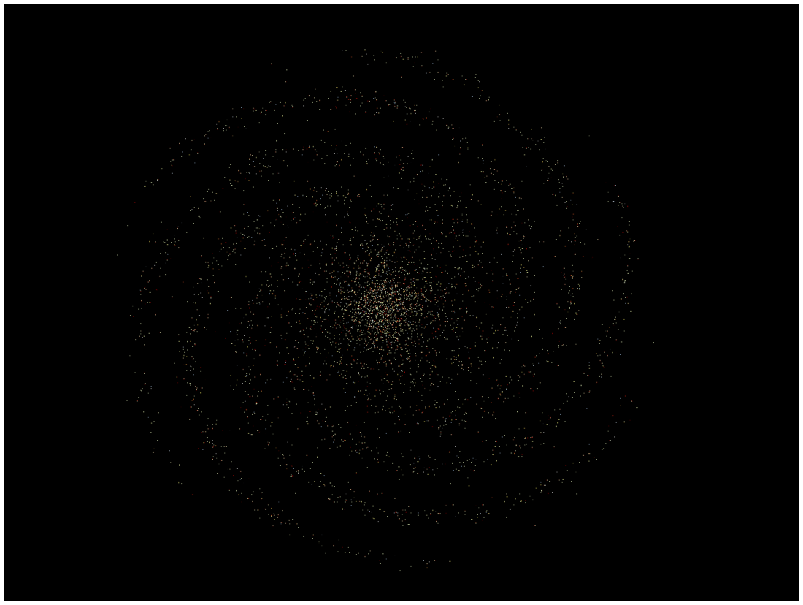


Figure 4.5: A galaxy with several spiral arms.

The solar system generation is also relatively simple. Starting with the selected star in the middle and randomly creating between 1-10 planets orbiting it at varying distances. Like the stars, each planet has a separate data class with its properties. This data includes their position, name, size, gravity, temperature, humidity, wind speed, planet type and alien presence. These properties are influenced both by their parent star's properties but also by each other to some extent. For example planets closer to the star tend to be lava planets while the stars furthest away become ice types instead, as can be seen in figure 4.6.

Figure 4.6: A solar system, with lava planets closer to the sun and ice planets farther away.

4.3.3 Surface Generation

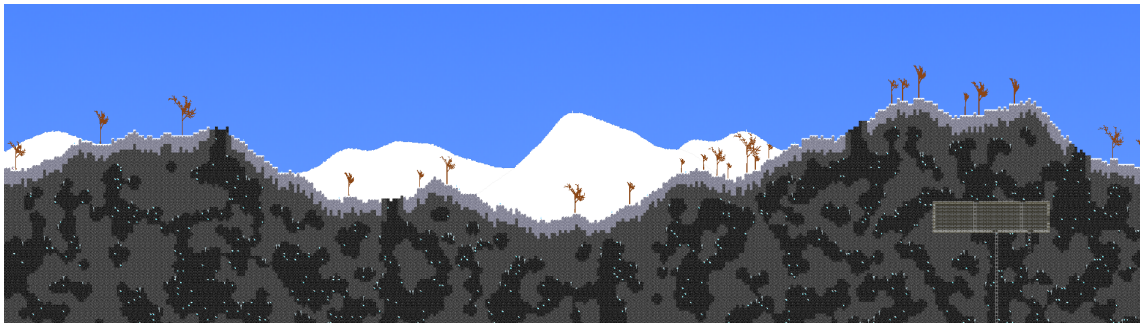
The surface generation of the game is done using three different terrain generation algorithms: midpoint displacement, value noise and Perlin noise. Which algorithm is used depends on the type of planet, and in some cases the wind speed of the planet as well.

Midpoint displacement is used for lava planets, as well as ice and green planets with low wind speeds. The roughness is automatically amplified for lava planets, and humidity causes further adjustments to the roughness in order to emulate erosion to some extent. The initial amplitude which is allowed depends on the gravity, though the actual height of the surface is not particularly deterministic. This happens partially because the obtained amplitude is never specified, but also because the surface may re-scale if the mountains are high enough to exceed the boundaries of the level. This was an easy solution to allow for greater variance to the roughness parameter.

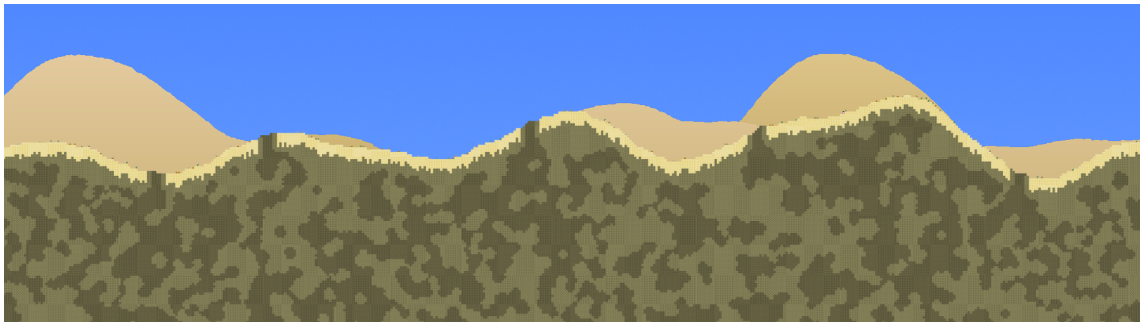
Value noise is used for ice and green planets with high wind speed. The algorithm is done in several passes, with each pass halving the amplitude and doubling the frequency of the noise function. The gravity of the planet sets the range of the fix points' heights (and thus the amplitude of the terrain) for the first pass. The ratio between the planet's gravity and the maximum possible gravity is multiplied with the adjustment range for the amplitude, which means that when a planet has very high gravity the amplitude is increased. Value noise is used for planets that are not supposed to look mountainous, and as such this adjustment range is set to a rather low number. The distance between the fix points of the first pass depends on the planet's width. This gives larger planets wider hills, although all planets will have about the same number of hills.

The Perlin noise is done similarly to the value noise. It is used for desert planets. The distance between the initial points is affected by the planet's size, and the height of the peaks is affected by the gravity of the planet. Since Perlin noise does not directly use height variables, the height was manipulated by changing how steep the slopes could become.

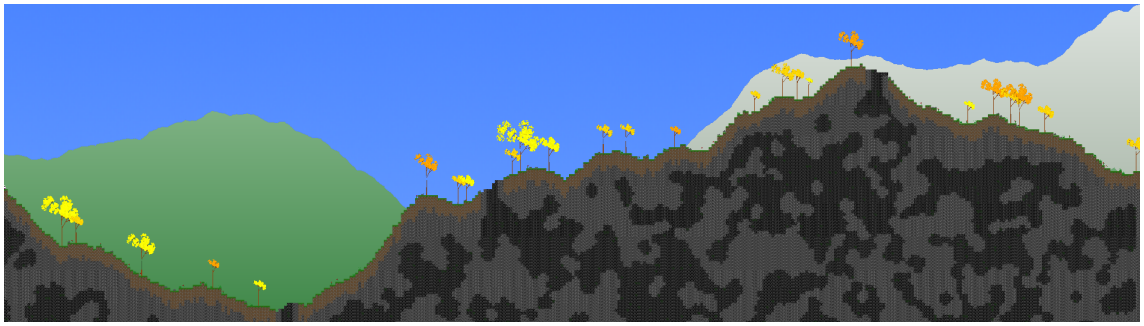
The algorithms have all been adapted for use only for the mentioned planets, for example, Perlin noise can only produce smooth, dune-like hills and midpoint displacement is always prone to mountains. See figure 4.7 for examples of the different terrains produced.



(a) Value noise



(b) Perlin noise



(c) Midpoint displacement

Figure 4.7: The results of the three different surface generation algorithms.

4.3.4 Cave Generation

As mentioned in section 3.2.1 the level is represented by a matrix of tiles, representing earth, cave and air. However, in this section, only whether a block is passable or solid will be considered. To generate the caves a cellular automaton is used. For the purpose of this automaton, passable tiles are considered to be alive and solid tiles to be dead. Some special case elements have also been inserted into the algorithm for this project, such as cave openings and some cleanup. The algorithm works as follows:

1. Each cell in the level matrix that is marked as mountain, is with a 50 % probability set as either passable or solid.

2. For each tile: if the tile has more than $3 \cdot birthlimit$ alive neighbours around it in a 5x5 square or if it has more than $birthlimit$ alive neighbours in a 3x3 square around it then the tile becomes alive. Else, if the tile has less than $3 \cdot deathlimit$ alive neighbours in a 5x5 square or less than $deathlimit$ alive neighbours in a 3x3 square then the tile dies. Otherwise the tile does not change state.

3. Cleanup step where some degenerate behaviour can be detected and handled. In this case that means handling checkers patterns by killing alive cells whose closest vertical and horizontal neighbours were all dead and vice versa.

4. Step 2-3 are repeated 7 times

5. Depending on how large the planet is, between 2 and 8 cave openings are set into the surface by first finding a random surface tile, then checking whether there is any large cave beneath that tile. If there is, a chunk of mountain is removed; otherwise another random surface tile is found.

6. Step 2-3 are repeated 7 more times to smooth the openings into the rest of the caves.

It was found that the algorithm worked well for $birthlimit = 5$ and $deathlimit = 3$.

How many times step 2-3 are repeated has a dramatic effect on the end result. If no repetitions are done at all the cells states will all have the same random values they were assigned in step 1. Figures 4.8, 4.9 and 4.10 show how the caves appear after three, nine and fifteen repetitions respectively.

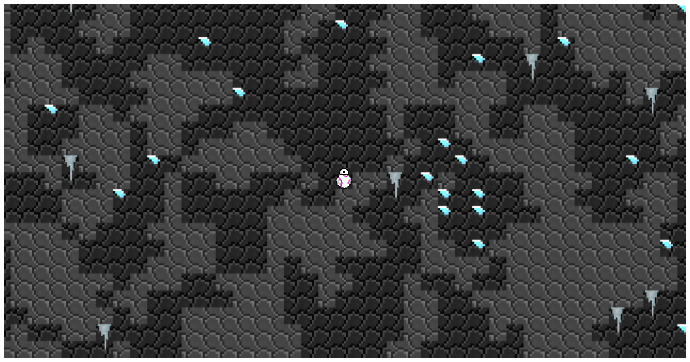


Figure 4.8: Cave after doing just 3 iterations of step 2-3. After three iterations there are a lot blocks connected diagonally and a lot of checker patterns. The cave paths are also narrow and resemble an earlier implementation in figure 3.5

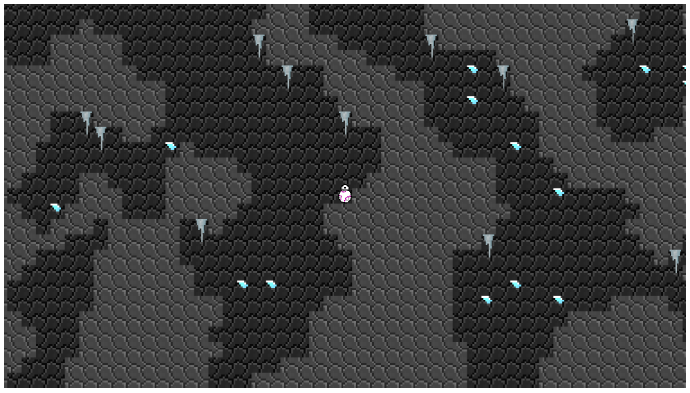


Figure 4.9: Cave after doing nine iterations of step 2-3. After nine iterations the cave paths have widened a lot and there are some larger rooms. The walls are also much smoother.

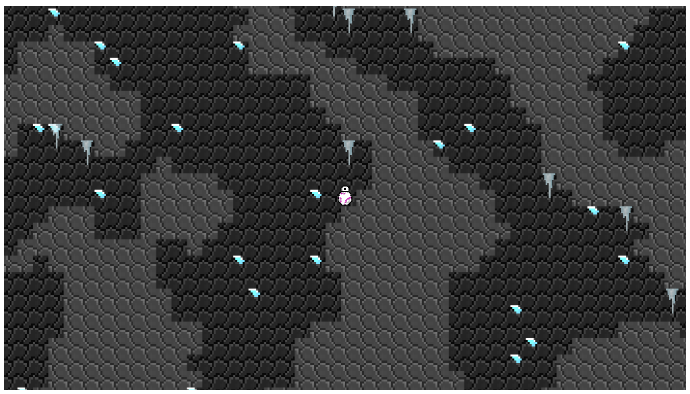


Figure 4.10: Cave after doing fifteen iterations of step 2-3. After fifteen repetitions not much changed after the first nine.

As can be seen in figure 4.8 and 4.9 there is a substantial difference between nine and three iterations. After nine iterations the checker patterns have disappeared, and the walls are generally much smoother, which is, of course, a significant improvement. Between nine and fifteen iterations, however, not much has changed. This shows that the configuration in the figure is somewhat stable after nine iterations and that the number of iterations run after that does not matter very much.

4.3.5 Cave Rooms

Rooms from an alien civilisation can be found within the cave systems, as in figure 4.11. They are always found in small groups that are distributed uniformly throughout the cave. Binary space partitioning is used to divide the world into smaller spaces

where about 2-4 rooms can fit. Some of these spaces are chosen randomly with the probability being proportional to the alien presence of the planet. The chosen spaces are then further split with BSP until they contain sufficiently small subspaces fitting for individual rooms. Rooms are then placed in each subspace, and as a result, a room is always grouped with at least one other room.

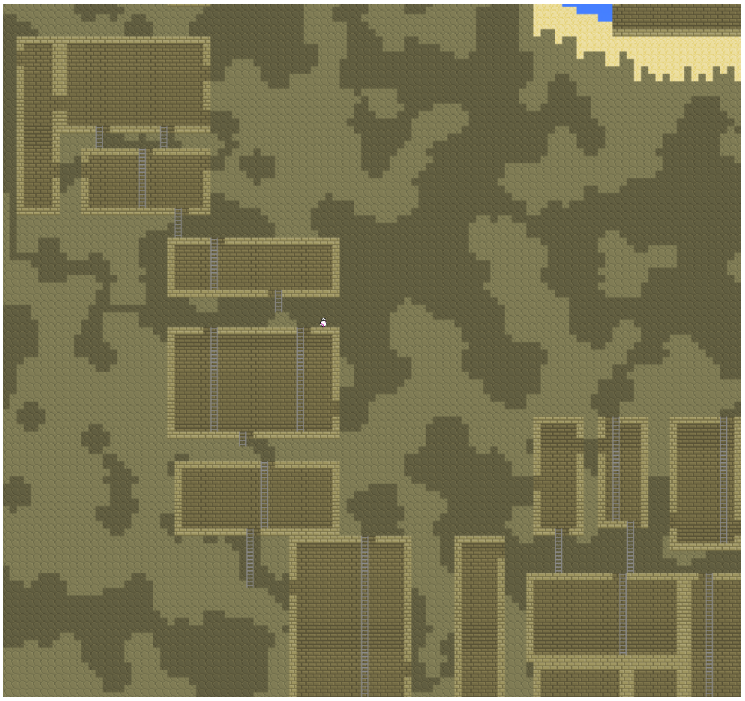


Figure 4.11: An example of rooms created in the caverns, clumped together by their partitions.

The positioning of the doors is chosen by taking a random wall block from the room walls and finding all of the other wall blocks that are 'connected' to it, meaning there is a short path outside the room that connects the blocks to each other. After identifying the entire wall strip which the original wall block is a part of, two neighbouring blocks in the strip are removed to create an opening.

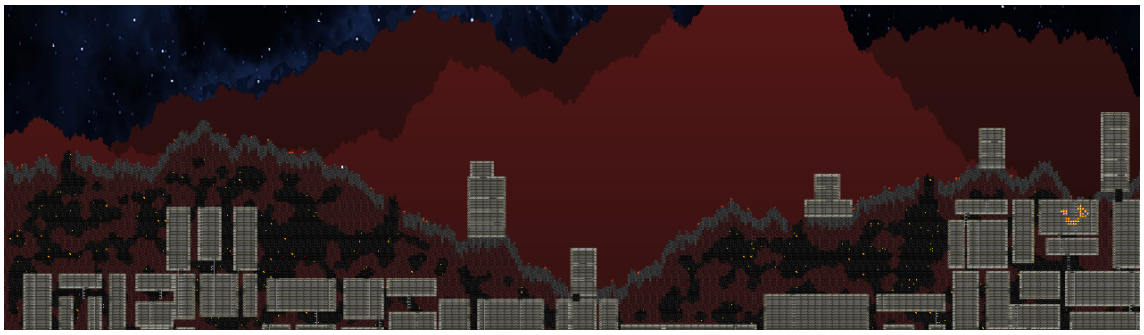
The algorithm for identifying a strip of blocks that are connected works as follows:

- Start with one block and check if its neighbour block outside the wall is free (the player can walk there).
- If it is not free return the list of found blocks so far.
- If it is free then it adds itself to the list of found blocks and recursively does the same for its neighbouring blocks that are part of the wall.

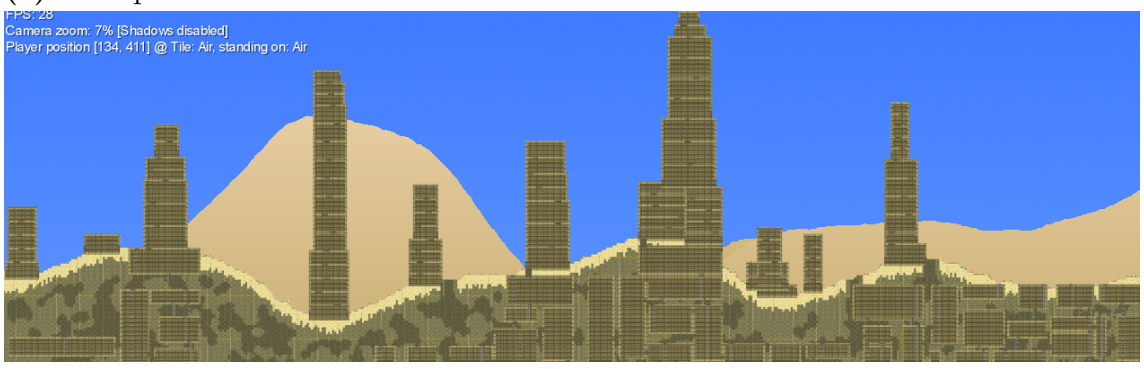
In essence, the algorithm returns the list of blocks that are in a line and that have empty space outside the room the whole way. The upside of this algorithm is that it is fast, but the downside is that if there is a single block in the way it can split a strip into two, even though they lead to the same large room very quickly.

4.3.6 Surface Buildings

Rooms were also built on the surface of the planets. To avoid creating rooms embedded too much in the ground, an algorithm for finding suitable building spots on the surface is used. A suitable area is defined as a minimum width of 14 blocks that are all within three blocks vertically. Rooms are then created in these areas randomly depending on the amount of alien presence on the planet. It is also possible to generate buildings with more than one floor by adding a room on top of a room and connecting them with a door at the bottom of the new room. This is done by doing several passes of the room creator, which can then create tower-like structures.



(a) Lava planet



(b) Desert planet

Figure 4.12: Surface rooms generated on high alien presence planets.

Figure 4.13: Doors to a room being blocked by terrain.

The number of rooms which are generated on the surface is highly dependent on the terrain of the planet which can be seen in the difference between the number of buildings in the erratic lava and smooth desert planets in figure 4.12. As can be seen here, most buildings on large alien presence planets become towers due to the few spots available to build for the algorithm. Lastly, there is still a problem with some rooms being generated with their doors blocked by the surrounding terrain as seen in figure 4.13.

4.3.7 Tree Generation

L-systems are used to generate various kinds of trees. The following six L-systems, either taken directly from The Algorithmic Beauty of Plants[21] or slightly modified versions of grammars therein, were implemented.

Grammar A:

- F → FF
- X → F-[[X]+X]+F[+F[+X]XL]-X

Grammar B:

- F → FF
- X → F-[[X]+X]+F[+FLX]-X

Grammar C:

- F → FF
- X → F[+XL][-XL]FX

Grammar D:

1. F → FF
2. X → F[+XL]F[-X]+X

Grammar E:

- F → F[+F+]F[-F-][F]

Grammar F:

- F → FF-[-F+F+F]+[+F-F-F]

Here, F means that the tree is built on, L means that a leaf is placed, + and - represent a rotation, clockwise and counterclockwise respectively, [means that the current location and orientation is pushed to a stack and] means that a location and orientation is popped from the stack.

What kind of tree each grammar yields can be seen in Fig. 4.14

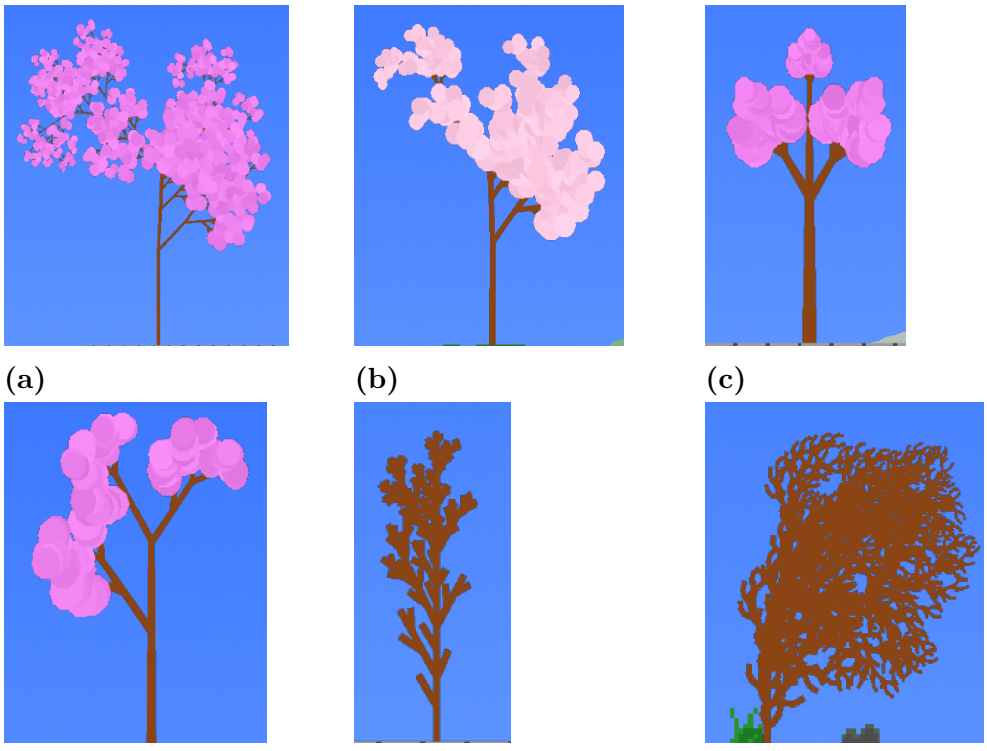


Figure 4.14: The six different trees resulting from the different grammars.

The trees are included on two types of planets, the Earth-like planets and the ice planets. While the same kind of trees can be seen in both types of planets, the trees do not have any leaves on the ice planets.

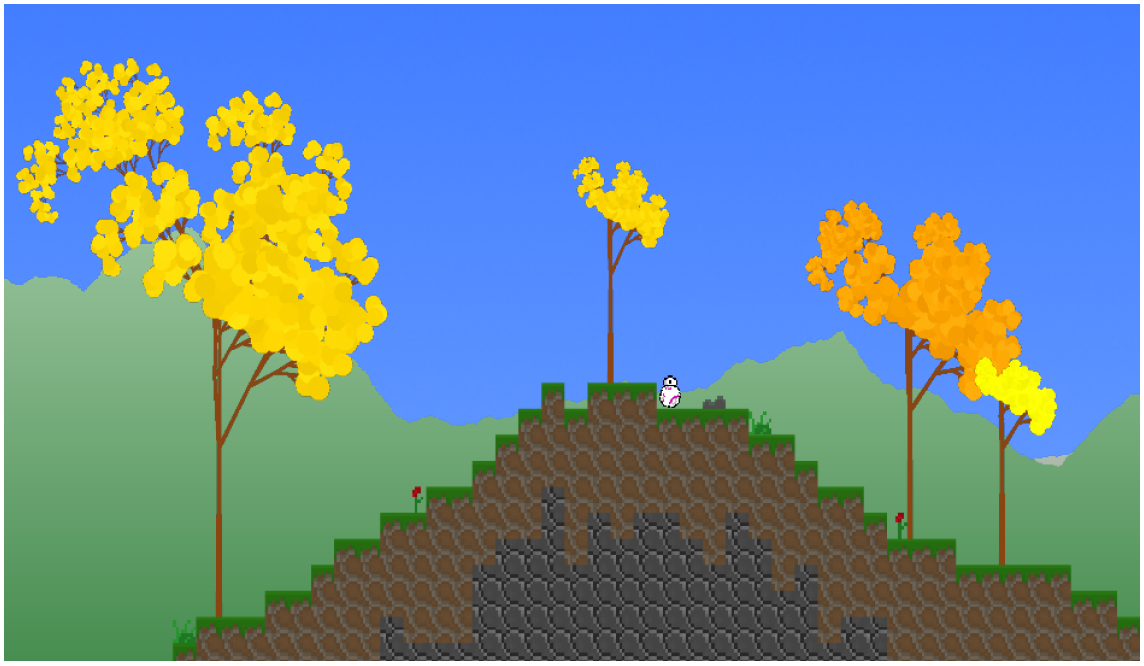


Figure 4.15: One variant of the trees that can be generated

The amount of variation between different individuals of the same type of tree varies from grammar to grammar and is acceptable for the most part. It was enough that none of the trees looked too alike at all times, as that would look too artificial, and this was avoided. An example of variation within one type of tree can be seen in figure 4.15.

4.3.8 Fire system

The final implementation had a matrix that spanned all blocks in the level, where an integer denoting if the cell is currently burning was saved. Each time the fire updated there was a small chance that a random cave cell would catch on fire. Every update the age of all cells that were currently burning were also increased by one, and if they passed a certain threshold, five, in this case, they burnt out. After that each cell with at least two neighbours that were on fire had a certain probability to catch on fire, signifying that the fire is spreading. Because two cells needed to be on fire for the fire to spread, a neighbour was also ignited whenever a fire started at a random place in the level. If a cell had recently burned out, it was impossible for it to start burning again for a while, representing a depletion of fuel. The probability

that a random cell ignites was increased when planets got warmer, and the same was true for the probability of the fire spreading. There was also a small chance that fire could spread to wall tiles where the player cannot walk. When this happened, the wall would be turned into a cave tile, which meant the fires slowly eroded the level.

Due to the dynamic nature of the system, it is difficult to properly show the results. However, in Fig. 4.16, the way in which a fire might spread can be seen in a set of time-lapse photos.

(a) **(b)** **(c)** **(d)**

Figure 4.16: Fire spreading out, from left to right

4.4 Observations and reflections

This section covers different strengths, weaknesses and other things of note regarding the different procedural content generation methods implemented in this project.

4.4.1 Name Generation

The name generator provides satisfactory results for the purpose. It generates a large number of unique names, and it is simple to change what kind of names are generated by changing the data set. This is incredibly useful if there is a particular kind of style which is sought after in the naming sense.

There are, however, some minor issues which need to be taken into account and most of them arise from the fact that the name generation is a random process. For example, it is possible that the generated name is gibberish, a possibility which gets more probable as more names are generated. At least two things can be done to minimise the risk of generating gibberish. First of all, it is advisable to avoid mixing different languages in the input string as this seems to have significantly increased the occurrences of odd names being generated. Furthermore, it is likely good practice to include some form of validation or extra control when generating names, for instance, not allowing too many vowels or consonants in a row.

Another problem which arises from the random process, and lack of control over the output, is the fact that inappropriate words can be generated. This problem occurred on several occasions, but was not considered a problem in this project, as it is being done solely for research purposes, but is likely something which has to be considered in many other projects. The best solution to this problem is likely to keep a list of words which are unacceptable, and cross-referencing with it each time a name is generated.

Furthermore, some interesting aspects were noticed when experimenting with the data set. At one point, Roman emperors were included in the dataset, an addition which resulted in a large number of the generated names to end with -us, as it is a common ending in Roman names. As a result, some transitions became much more likely to happen than others, introducing a clear bias in the random generation. This problem can be avoided by simply manually removing the offending names from the data set, but with an extensive data set, it might be difficult to gauge which the offending words are. If this is the case, the problem can likely be solved by implementing the Katz back-off model [52], which 'backs off' to a lower n-gram under certain conditions, for example, if a specific transition is extremely common, or there is only one possible transition due to a lack of data. The generator could then back-off to a lower n-gram if the transition to -us, or any other transition, is too likely.

4.4.2 Galaxy Generation

For generating spiral galaxies or objects of similar shape, the method used in this project is straightforward, quick and gives good results. If a more non-standard form of a galaxy is wanted, Diffusion Limited Aggregation could be used instead as long as the slower calculation time is not an issue. DLA would create a galaxy with a more crystalline shape, while still having a dense central core with arms that branch out and become thinner as they move away from the centre.

4.4.3 Level Generation

The two most prominent methods used in the generation of the level terrain are noise functions and cellular automata.

Something which can prove to be frustrating when working with these methods is the fact that both noise functions and cellular automata are very difficult to control. For example, the cellular automata that are used for caves are considered to work well in this particular configuration; however, when the variables of the algorithm are tweaked just a little bit, the results can be wildly different. When one rule is changed, it affects all tiles in the whole system, making it hard to predict what results they will give. The same is true for the noise functions used to generate the topography. Height restrictions, frequencies and fractal repetitiveness can often be

modified, and may with high probability result in different levels of steepness and more or less rough terrain. However, due to the unpredictability of noise, it also has to be highly constrained to minimise the risk of generating improper results, which of course also limits the output range.

On their own, there is only so much that can be done with either of these algorithms. Having one set of rules for each tile in the cellular automata or a single pass of the noise function will always produce a somewhat homogeneous result. Thus if more variation is wanted, a suggestion would be to post-process the generated output. For example, start by generating a base terrain and then simulate erosion and possibly carve out cave openings or run additional passes of the functions but with different rules. All in all, those are trade-offs for the simplicity and speed of the algorithms, and both of these approaches can be very useful to generate a good starting foundation for different types of terrain.

4.4.4 Structure Generation

The rooms generated on the planet surface are rather small and often built on top of other rooms. This is because the surface generally does not generate many flat areas, while the tops of any existing rooms are always flat. The tower-like structures can be quite interesting, but they should preferably be generated independently of how the current planet surface happens to look. There is no justification for aliens building more towers on uneven terrain as opposed to flatlands. To keep the surface interesting, the surface generation should be allowed to create as flat or as uneven terrain as needed. Instead, the building algorithm should modify the terrain more if there are no suitable areas while a larger quantity of buildings is desired.

Increasing the allowed height difference when searching could increase both the size and amount of areas fulfilling the minimal width, but this would create buildings even more entrenched in the terrain which might not be preferable. To alleviate this problem, the algorithm could instead find more areas defined by looser constraints and then modify the areas to better suit buildings. This approach would more accurately depict human behaviour in that we often build our foundations instead of simply building where the ground is already optimal.

Another approach could be to randomly pick building spots and form the rooms around the terrain instead of being perfect rectangles. This would force more varied building shapes and be more interesting but could appear less realistic. The algorithm for this might also be harder to implement in a way which produces good enough results. A mix of building around the terrain and modifying it could produce a better result.

To create more variation, several different simple algorithms could be used instead of one more complex. This would create the problem of combining them though, which might in itself be more complicated.

With regards to the underground rooms, binary space partitioning was found to be very useful to, quickly and efficiently, ensure that no overlap between different segments occurs, while still allowing for all parts of the level to be used. Something to keep in mind when using it, however, is that it can look very artificial and obvious where the partitions are if the partitions are filled to their edges. An example of this can be seen in figure 4.11, where rooms are grouped in very obvious boxes. To possibly alleviate this, try to reduce the size of the content within partitions to better where the edges are.

4.4.5 Tree Generation

Due to the difficulties which have been present throughout the development of the tree generation, a few observations have been made. One of them is that L-systems are the standard for a reason, and it is unwise to try and reinvent the wheel. Furthermore, it is recommended to have some knowledge of graphical programming when trying to implement tree generation as it makes the process much more manageable.

Some other general problems were also noticed. First of all, it is difficult to design grammars which result in visually appealing trees. This difficulty is further exacerbated by the fact that most work about modelling trees with L-systems has been done in 3D, resulting in a relative lack of known grammars which look good in 2D. A s a result, a lot of time would be spent on trying to design adequate grammars, or alternatively, designing a tool which aids in the modelling of L-systems such as [53].

Secondly, there are a plethora of difficulties with drawing the trees during run-time which has had a major impact on the visual quality of the trees. One of these issues is that it was difficult to introduce better colouration or texturing. Closely related to this problem is the difficulty of adapting the look of the trees to the graphical style of the game.

Although it would not improve the problem of graphical style, the aesthetics of the generated trees could be significantly improved by rendering them in 3D and then projecting to 2D, both due to more complicated structures being possible to render but also because there is a lot more information about L-systems in 3D. However, depending on the complexity of the L-system, this is not something that can be done during run-time.

All things considered, generating trees at run-time, at least in the way in which it has been done in this project, does not appear to be a worthwhile effort. It is difficult to recommend it, mostly due to the difficulty of making the generated trees fit in graphically unless the rest of the game is built around it. Instead, the creation of a tool like SpeedTree [54] focused on 2D trees could be a good idea.

Another possible approach which was thought of very late into development and

could not be tried would be to combine a fairly large number of pre-created art assets for each type of tree and L-systems. By defining an L-system which decides how different art assets can be combined, it would be possible to generate a sizeable amount of trees while still keeping to a specific graphical style. This approach could perhaps work very well, though it would require a fair amount of manual work to create the art assets.

4.4.6 Dynamical Systems

The implementation of fires within the game worked moderately well. Fires are started randomly and have an acceptable life-length on the map. The most substantial issue with the system is arguably how the fires are started. The fires simply appear out of nowhere, which does not make much sense.

Something else that might have been improved is how the fire spreads. As discussed in 3.4.3 there are pros and cons for having the fire spread with either just one burning neighbour or requiring two burning neighbours. If there had been more time, it might have been interesting to allow the fire to spread with just one burning neighbour, but to at the same time increase the probability of fire spreading to a cell the more burning neighbours that cell has. This could potentially provide the advantage of allowing small fires to grow, while still avoiding that fires spread very erratically. It would also naturally make sure that gaps in the fire are avoided since such a gap would have a lot of burning neighbours.

While the liquid system was scrapped, there were still some lessons to be had. Increasing the amount of particles in the system did create better a simulation but when doing so one has to keep in mind the increased performance cost and evaluate if it's worth it. Depending on what the system is to be used for, this increase in performance cost could be negated by not updating particles outside a certain range.

If the larger cell approach is kept, some solution to there being gaps in the flow is needed. Regarding the graphical problem of water flowing downwards, a solution found online later was to display those cells as if they were full of water but instead scaling the intensity of the colour to the amount of water.

The approaches found certainly work for some cases but in this project the result was not satisfactory.

5

Discussion

In this chapter, both the developed systems and the methodology employed throughout the course of the project are evaluated, potential ethical and social aspects of the project are analysed, and potential future work is discussed.

5.1 Results Evaluation

The purpose of this project is deemed to have been fulfilled. During the course of this project, a way of generating cohesive 2D levels has gradually emerged, generating many key features of the levels of similar games.

With that said, there are still many issues remaining. It can, for example, be concluded that even though each planet is uniquely generated, the perceived variation is still insufficient. It proved very difficult to provide a broad output range of substantial differences to the environment. The most significant variety arguably comes from the different planet types that were defined. These are of course based on several procedurally generated parameters, such as temperature, gravity and humidity. However, the planet types still needed to be explicitly specified and also needed much manual handcrafting. Creating more planet types should preferably be taken care of by the system itself, though it would probably, if at all possible, require substantially more PCG algorithms, and another validation approach.

Furthermore, one of the stated goals of this project was to use the mentioned parameters to create a chain of events that defined the outcome of the different planets. This also proved very difficult and was poorly executed throughout the entire development phase. This was due to multiple reasons. Perhaps the most substantial reason for this is the fact that no consideration for the parameters was made when choosing which algorithms to implement. It was, therefore, challenging to integrate the parameters in the generation during the later phase of the project.

Also, it proved to be a rather complex task to make reasonable values for the parameters. For an extended period of time, due to debugging reason, those parameters were generated more or less completely randomly within unreasonable ranges. At a late stage, some attempts were made to resolve this by changing how the parameters

were generated. Although the maximum and minimum values are now better, some values are still simply normally distributed around some fixed mean with no regard to any other parameters. This is somewhat of a letdown compared to the grand visions at the beginning of the project of having a multitude of parameters working together and logically depending on one another.

5.2 Evaluation of Methodology

While a formal methodology was devised early on, it was not followed ardently throughout the entire project. In particular, code reviewing was quickly down-prioritised as it was considered less important than getting more things implemented. Features were often reviewed by the group and tested for obvious bugs, but more extensive testing was not considered a top priority. This was most likely the right decision, considering that the purpose of the project was to try out different techniques and not to create a polished product.

Assigning tasks on a weekly basis proved advantageous due to several reasons. The weekly evaluation of the progress, more specifically, discussing how the work for the past week had gone was especially constructive. This process made it possible to identify any possible future problems with a task early on, making it easier to know if the task had to be dropped or approached differently. It also made it easy to request for help on a task if it was more difficult than expected.

However, the task assignment was not always very formal, and several times there were no clearly defined goals for the week. At the same time, the scope of tasks and estimation of how much time they would take were kept simple. For example, many times work on a larger task would begin by letting one member work on it for a while so they could evaluate the task better next week.

This was at times beneficial since it meant that the team knew what needed to be done and could spend time working rather than planning it. However, at other times, the lack of structure made it difficult for members to know what exactly they could work on, or how they should try to solve a task. If an assignment is too small, it is possible to have nothing to do at the end of the week, and without a clear goal, it could be more difficult to find motivation.

In general, a more structured approach might have been helpful in some situations, such as when a team member did not know what to do. However, the weekly evaluative meeting and the fact that the team was eager to work together meant that any issues which arose were handled relatively quickly. Generally, the best thing about the team's methodology was the close communication about the work itself.

5.3 Validity and Generalisation

All of the results presented in this thesis are in no way an objective truth. There were no user tests done to judge whether content was suitable or good. Many of the conclusions and reflections which are given here are simply subjective thoughts from the development team about the different systems and generation techniques. This is not to say that the results are meaningless, there are reasons given for the different conclusions, but one should keep in mind that no rigorous scientific validation has been conducted.

Little research has been found prior to the writing of this study on how to generate this type of environments with all parts woven together. Though the approach that this project has taken in doing this is probably far from the best solution, this study may provide useful insight into what to think about when attempting a similar project. It also discusses several specific algorithms that can be considered when generating different types of content, both for games but possibly also other types of applications.

5.4 Ethical and Social Aspects

While this project is of a pretty small scope from a big picture perspective, if PCG, in general, develops enough it could cause some issues.

One concern about PCG is if it could eventually replace traditional content creators entirely and result in unemployment. However, as Shaker et al. argue in their book, 'content generation, especially embedded in intelligent design tools, can augment the creativity of individual human creators' [1, p. 3]. They argue that PCG will overall create a greater good by allowing smaller teams or individuals to create content-rich games without worrying about the busy-work. This thesis should have also made it clear that PCG is not simply a matter of pushing a button which generates perfect worlds, there is still much work needed to be done in designing the content generation and tuning it to specific purposes. This means that game development can continue to evolve in new ways, without necessarily creating a net loss of work opportunities.

While it was not a major concern in this project, since it will not be released to the public, it is worth noting that the random nature of most PCG techniques makes it possible for offensive content to be generated. As discussed in 4.4.1, there is a risk that inappropriate names will be generated for the planets and stars. Similarly, when generating caves, there is a risk that offensive symbols or imagery will appear. Disallowing certain words can be done easily, however, disallowing certain patterns in the cave generation is much more difficult.

Already with today's quality of games, some people become addicted to them with no regard to the outside world. In this way, it could be feared that if PCG comes to

surpass handmade content and games started offering an endless amount of quality experiences, perhaps this would worsen the issue with game addiction? However, stifling the quality of media or prohibiting the development of PCG does not seem to be the right way to tackle the issue. Instead perhaps if such a correlation is found, more development could be put into developing systems to identify and restrict excessive gaming.

5.5 Future Work

While a fair amount of progress has been achieved during the course of this project, there is still a lot more which could, and should, be done in the future.

During the project, using cellular automata to emulate the behaviour of fire was explored and an example of such a system was implemented. However, the model implemented was quite simple and it would certainly be interesting to explore this to a larger extent. For example a more complex stochastic approach that varies the probability that a cell catches on fire with how many burning neighbours that cell has.

Some topographical features of surface terrain cannot be expressed with a single heightmap. For example, cliffs with overhangs could add an interesting touch to the environment. This could perhaps also be used to create more smooth transitions into the underground, as opposed to the vertical fall-pits encountered in this implementation. How this can be done without the use of pre-defined assets would be interesting to see, and could possibly be studied in a future project.

In a more general sense, one large issue that this project did not manage to solve was how to create meaningfully varied content. This is an important aspect of PCG, and more research should be conducted on how to achieve noticeable variation in generated content.

5.5.1 Improving the Game

First of all, an extensive amount of work should be focused on trying to increase not only the amount of variation in the generated levels but also in finding ways in which to make the generated levels more interesting. This might entail generating more interesting terrain such as cliffs and crevasses, more varied caves or better integrating the underground rooms with the caves and connecting them better to other rooms.

Furthermore, it would be worth finishing the development of the liquid system and then explore how it could be used to generate more interesting and varied levels. For example, it would be possible to include lakes, seas and rain which could affect the levels. Some other examples would be possibly including an oasis in desert planets

or liquid lava deep in caves or on the surfaces of lava planets.

Additionally, there is plenty of potential to also generate other kinds of content which are not present at the moment, such as various non-player characters (NPCs) or narrative content, to name just a few examples. Procedurally generating narrative especially has great potential, particularly for exploration based games where it could be a central feature. While the most obvious application of procedurally generated narrative is dialogue for NPCs, many other things can be done. Perhaps the most beneficial type of narrative which could be generated is environmental narrative of some sort, where the game world itself tells a story.

6

Conclusion

The purpose of this study was to explore how complete environments could be generated for explorative adventure games. This was done through the implementation of a tech demo representing a game idea somewhat similar to the aforementioned games. A diverse set of PCG algorithms for many different purposes were studied, and a selection of these approaches combined resulted in the space exploration game *Astroarchaeology* which this thesis covers. First, a spiral galaxy is generated by placing stars in a starfish shape and rotating the arms around the centre of the galaxy. Based on the properties of each star, a solar system is generated around it, containing different types of planets that the player can explore. Each planet has a noise generated surface terrain, and through the use of cellular automata, natural caves are generated beneath it. The remnants of an alien civilisation are then displayed through cave rooms and buildings generated through binary space partitioning, and decorations such as trees generated by L-systems are scattered around the planet.

It was concluded that though each planet is different from the others, much variation is still not perceived during gameplay. Additional passes of different algorithms could alleviate the issue, but it proved difficult to generate variety to the overall setting of a planet with the methods and passes used within this project. Reflecting the effects of planetary parameters such as gravity, wind speed and temperature on a planet's terrain is not a simple task. Also, even though the portion of each planet that may be visited is heavily limited, it still feels like it is too uniform to encourage exploring it thoroughly, at least with such limited gameplay. The results obtained here can hopefully help other developers who are trying to develop something similar in the future. The hope is that it will facilitate understanding of how procedural generation can be used to create cohesive game worlds and also what aspects of each algorithm make them suitable, or unsuitable, for a specific task.

In conclusion, while the purpose of the project has been fulfilled, showing how several algorithms used in conjunction can generate a cohesive game world, it proved to be more difficult than expected to introduce significant variation to the generated levels. Therefore, there still exists a great need for further research into how to best generate the type of game worlds which have been explored here.

Bibliography

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016.

[2] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley, "Procedural Content Generation: Goals, Challenges and Actionable Steps," in *Artificial and Computational Intelligence in Games*, ser. Dagstuhl Follow-Ups, S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, vol. 6, pp. 61–75. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/4336

[3] R. E. Cardona-Rivera, "Cognitively-grounded procedural content generation," in *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*. Association for the Advancement of Artificial Intelligence, 2017.

[4] Re-logic Games, "Terraria." [Online]. Available: https://terraria.org/

[5] Chucklefish, "Starbound." [Online]. Available: https://playstarbound.com/

[6] M. Toy, G. Wichman, and K. Arnold, "Rogue," 1980.

[7] Blizzard North, "Diablo."

[8] Lucasfilm Games, "Rescue on fractalus!" 1984.

[9] Bethesda Softworks, "The Elder Scrolls IV: Oblivion," 2006.

[10] Gearbox Software, "Borderlands." [Online]. Available: https://borderlandsthegame.com

[11] Mojang, "Minecraft." [Online]. Available: https://minecraft.net

[12] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186,

Sept 2011.

[13] RPGamer, "Rpgamer feature - the elder scrolls iv: Oblivion interview with gavin carter," [Accessed 2-April-2018]. [Online]. Available: http://www.rpgamer.com/games/elderscrolls/elder4/elder4interview.html

[14] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin, "Polymorph: Dynamic difficulty adjustment through level generation," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ser. PCGames '10. New York, NY, USA: ACM, 2010, pp. 11:1–11:4. [Online]. Available: http://doi.acm.org/10.1145/1814256.1814267

[15] E. Hastings, R. K. Guha, and K. Stanley, "Automatic content generation in the galactic arms race video game." vol. 1, pp. 245–263, 01 2009.

[16] S. Snodgrass and S. Ontañón, "Controllable procedural content generation via constrained multi-dimensional markov chain sampling," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, pp. 780–786. [Online]. Available: http://dl.acm.org/citation.cfm?id=3060621.3060730

[17] K. Verbeurgt, M. Dinolfo, and M. Fayer, "Extracting patterns in music for composition via markov chains," in *Innovations in Applied Artificial Intelligence*, B. Orchard, C. Yang, and M. Ali, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1123–1132.

[18] J. R. Norris, *Markov Chains*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997.

[19] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[20] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*. Upper Saddle River, NJ, USA: Ellis Horwood, 1990.

[21] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. Berlin, Heidelberg: Springer-Verlag, 1990.

[22] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," in *in Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. Press, 2001, pp. 301–308.

[23] P. Worth and S. Stepney, "Growing music: Musical interpretations of l-systems," in *Applications of Evolutionary Computing*, F. Rothlauf, J. Branke, S. Cagnoni, D. W. Corne, R. Drechsler, Y. Jin, P. Machado, E. Marchiori, J. Romero, G. D. Smith, and G. Squillero, Eds. Berlin, Heidelberg: Springer

Berlin Heidelberg, 2005, pp. 545–550.

[24] J. L. Schiff, *Cellular Automata: A Discrete View of the World*. Wiley-Interscience, 2008.

[25] J. D. Davidson and D. M. Gordon, "Spatial organization and interactions of harvester ants during foraging activity," *Journal of The Royal Society Interface*, vol. 14, no. 135, 2017. [Online]. Available: http://rsif.royalsocietypublishing.org/content/14/135/20170413

[26] L. H. Encinas, S. H. White, A. M. del Rey, and G. R. Sánchez, "Modelling forest fire spread using hexagonal cellular automata," *Applied Mathematical Modelling*, vol. 31, no. 6, pp. 1213 – 1227, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0307904X06000916

[27] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular automata for real-time generation of infinite cave levels," in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, ser. PCGames '10. New York, NY, USA: ACM, 2010, pp. 10:1–10:4. [Online]. Available: http://doi.acm.org/10.1145/1814256.1814266

[28] Procedural generation. [Online]. Available: https://en.wikipedia.org/wiki/Procedural_generation

[29] T. Betts, *"Procedural Content Generation" in Handbook of Digital Games*, H. A. Marios C. Angelides, Ed. Wiley, 2014.

[30] T. Archer, "Procedurally generating terrain," 2011.

[31] F. F. d. V. Miguel Frade and C. Cotta, "Evolution of artificial terrains for video games based on accessibility," *Applications of Evolutionary Computation*, vol. 6024, 2010, http://www.lcc.uma.es/~ccottap/papers/frade10accessibility.pdf.

[32] T. A. Witten and L. M. Sander, "Diffusion-limited aggregation," *Phys. Rev. B*, vol. 27, pp. 5686–5697, May 1983. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.27.5686

[33] (2010) Eve online. [Online]. Available: http://pcg.wikidot.com/pcg-games:eve-online

[34] N. S. J. Linda Rising. (2000) The scrum software development process for small teams. [Online]. Available: http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/scrum/s4026.pdf

[35] H. D. Benington, "Production of large computer programs," *Annals of the His-*

tory of Computing, vol. 5, no. 4, pp. 350–361, Oct 1983.

[36] G. Melnik and F. Maurer, "Introducing agile methods: three years of experience," in *Proceedings. 30th Euromicro Conference, 2004.*, Aug 2004, pp. 334–341.

[37] P. A. Laplante and C. J. Neill, "The demise of the waterfall model is imminent," *Queue*, vol. 1, no. 10, pp. 10–15, Feb. 2004. [Online]. Available: http://doi.acm.org/10.1145/971564.971573

[38] Linus Torvalds, "Git." [Online]. Available: https://git-scm.com/

[39] GitHub, Inc., "Github." [Online]. Available: https://github.com/

[40] Oracle Corporation, "Java." [Online]. Available: oracle.com/java/

[41] Bad Logic Games, "Libgdx." [Online]. Available: https://libgdx.badlogicgames.com/

[42] N. Broberg, "Model–view–controller," 2016. [Online]. Available: http://www.cse.chalmers.se/edu/year/2017/course/DIT952/slides/6-1a%20-%20Model-View-Controller.pdf

[43] Types of galaxies. [Online]. Available: https://space-facts.com/galaxy-types/

[44] M. Evans. (2016) Procedural generation for dummies: Galaxy generation. [Online]. Available: http://martindevans.me/game-development/2016/01/14/Procedural-Generation-For-Dummies-Galaxies/

[45] E. Carpenter and M. Haahr, "Procedural generation of large scale gameworlds," 2011.

[46] (2013) Generate random cave levels using cellular automata. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664

[47] Irem. (1982) Moon patrol.

[48] T. Stahl. (2013) Chronology of the history of video games. [Online]. Available: http://www.thocp.net/software/games/golden_age.htm

[49] J. Olsen, "Realtime procedural terrain generation," 01 2004.

[50] S. Wolfram, "Cellular automaton fluids 1: Basic theory," *Journal of Statistical Physics*, vol. 45, no. 3, pp. 471–526, Nov 1986. [Online]. Available: https://doi.org/10.1007/BF01021083

[51] (2009) Simple fluid simulation with cellular automata. [Online]. Available: https://w-shadow.com/blog/2009/09/01/simple-fluid-simulation/

[52] S. M. Katz, "Estimation of probabilities from sparse data for the language model component of a speech recognizer." *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 35, no. 3, pp. 400–401, 1987.

[53] R. Sun, J. Jia, and M. Jaeger, "Intelligent tree modeling based on l-system," in *2009 IEEE 10th International Conference on Computer-Aided Industrial Design Conceptual Design*, Nov 2009, pp. 1096–1100.

[54] I. Interactive Data Visualization. [Online]. Available: https://store.speedtree.com/