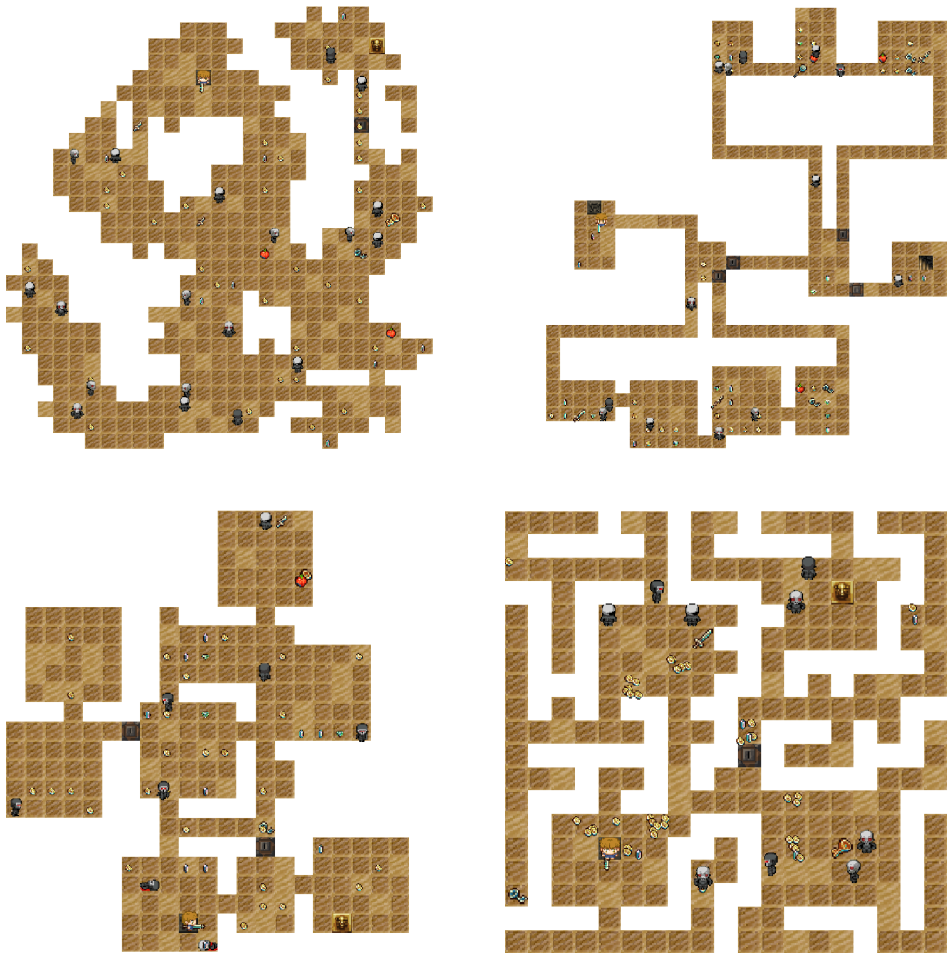


An Exploration of Procedural Content Generation for Top-Down Level Design

Bachelor's thesis in Computer Science and Engineering

Johan Blomberg
Rasmus Jemth
August Lennar
Robin Lilius-Lundmark
Marcus Pettersson Johnsson
Tove Svensson

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

An Exploration of Procedural Content Generation for Top-Down Level Design

Johan Blomberg
Rasmus Jemth
August Lennar
Robin Lilius-Lundmark
Marcus Pettersson Johnsson
Tove Svensson

Supervisor: Staffan Björk
Examiner: Olof Torgersson

Bachelor's thesis DATX02-18-17
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 58 Göteborg
Sweden
Telephone +46 (0)31-772 10 00

Cover:
Four levels generated by four different PCG algorithms, in the game *Fluky*. From top left to bottom right: cellular automata, cyclic, TK and agent-based.

Written in LaTeX
Gothenburg, Sweden, 2018

Abstract

Procedural Content Generation (PCG) is widely used in the game development industry to randomly generate various types of content. However, ensuring quality PCG is very hard, because of its inherent subjectivity; defining exactly what makes for enjoyable gameplay is difficult and depends on a wide range of different factors. We have therefore explored the concept of PCG, specifically procedural level generation, to gather observations and insights, and we present these in the report. The exploration has been performed by creating a simple game called *Fluky* and developing six different level generation algorithms for it. Among these observations are the importance of identifying levels of abstraction for an algorithm, and several aspects that may influence how enjoyable the generated content can be. Our resulting conclusions are very subjective, but not without merit, due to the extensive exploration and evaluation that has taken place.

Acknowledgements

We would like to thank Staffan Björk for his extensive help and support throughout the project.

Contents

1 Introduction

The approach of procedurally generating levels for games is fairly common, and has been used in a lot of popular games during recent years. *Procedural Content Generation* (PCG) means that the content, e.g. game levels, is not handmade by a designer, but instead generated dynamically by a computer. This is achieved through the use of various algorithms[1, p.1]. The use of PCG is not limited to game levels – it can also be used to generate other kinds of content, such as music and stories[1, p.1].

The idea behind PCG is to employ one or several algorithms to generate content. There are many different kinds of PCG algorithms, which fit different kinds of games. These algorithms take some kind of input, which could be a random number, or a set of parameters (e.g. difficulty) to generate a level. The generation itself is done through a set of rules and functions which defines how a level will be created, and what a desired level looks like.

PCG is used in games for several reasons. It can for example be used to increase the replayability of a game, and it can be a way of not having to spend long precious hours on level design when working with a tight budget[1, p.14]. PCG can also be used for purely creative purposes, as it opens up new options for game mechanics or level design.

One of the main challenges with PCG is not how much content one can generate, but the quality of it. A clear example of this is the recent game *No Man's Sky* (2016)[2], which advertised a procedurally generated universe so large that one person would not in their lifetime have enough time to visit even a fraction of all the available planets. Nevertheless, when the game was released, it was met with fierce criticism on account of all the virtually infinite planets looking almost exactly the same[3]. In other words, the amount of content created was huge, but the quality – in this case sufficient variation – was deemed to be very poor.

Other examples of what could be considered to be within the concept of quality in PCG also include: whether a generated level can be completed as intended, if it is challenging enough while not being impossibly difficult, and if it is simply fun enough to play. Additionally, it would seem that the quality of procedurally generated content is so dependent on the game it is generated for, that it is impossible to evaluate separately.

While PCG has been used in the video game industry for about 40 years[4, 5], and has become increasingly popular among game developers, this problem – the insurance of good quality – is far from being resolved. And due to the very subjective nature of the concept of quality content, perhaps it never will be.

1.1 Purpose

In this project, we explore the process of creating PCG systems for generating game levels by implementing several known algorithms. This includes investigating aspects of level generation that affect the quality of the final levels by comparing the levels generated by the different algorithms. Our intention with this project is to present the knowledge acquired during this exploration in a way that can help others who are about to develop a game using PCG for level generation to decide on what algorithm to use.

1.2 Method

To be able to explore PCG, we decided to create a basic game that would use PCG to generate its levels. When deciding the elements and features of the game, we tried to keep it as simple as possible to enable us to direct our focus on the PCG. This was done because the purpose was to explore the process of creating PCG systems. Nonetheless, the features of the game were still important because without a variety of objects and features to choose from, a PCG algorithm would not be able to create something interesting.

We decided on creating a game called *Fluky*, a two-dimensional dungeon crawler seen from above, where the player descends through procedurally generated levels of varying sizes. These levels contain different kinds of labyrinths, enemies, items and puzzles.

To be able to explore PCG as thoroughly as possible, we decided to develop multiple different PCG algorithms, all of which would have the same objectives - to be able to generate a level for *Fluky*, and to utilize all the different game objects and features. This would allow us to analyze and compare different kinds of algorithms.

We decided to use an *Agile* software development approach in this project (Section 3.3). This came naturally due to the fact that the project demanded working with the PCG and the game back and forth. A design document was created where we specified as much of the game's features as we could, which later could be divided into small tasks. In addition, we planned to distribute the development of the PCG algorithms so that each person in the group would be responsible for one of the algorithms.

To summarize our results in order to make our gathered knowledge available, we decided to discuss our experiences from the development of the PCG algorithms. We agreed on a set of relevant attributes that would be interesting to investigate in each PCG algorithm, and evaluated how well the different approaches worked.

1.3 Scope

Since our time was limited, we did not want to spend too much time on the game itself but rather focus on the development of the PCG. We therefore decided to make the game in 2D since that is simpler compared to working with 3D. Similarly, our primary focus with the game was the core mechanics. As such, we did not spend much time on the esthetics (i.e. the graphics and sounds). We did not use any kind of sound in the game, and we did not create any graphics of our own. We also realized that we would not be able to spend time on the final testing, tweaking and bug fixing stage which is usually both necessary and time-consuming when developing any kind of game intended to stand on its own as an enjoyable product.

Another decision we made was to delve deep into one area of PCG, instead of spending a little time on several areas. Therefore we focused on procedurally generated level design, as we feel that is the most interesting part. In level design it is easy to see the results, compared to things like story or music. As a constraint for our level-design, the game was limited to only have finite levels, which are not generated in real time.

1.4 Ethical and societal aspects

In essence, we were aware of two questions relevant to our project: is the player of the game affected negatively, and does gaming have any impact on anyone other than the player – and if so, does the quality of the game affect that?

Regarding the first question: if the procedurally generated elements in a game make the game more immersive or fun, this could give rise to the issue of people getting addicted to playing games to a higher degree. This will however depend on a myriad of other different factors, leaving the PCG as only a small part of a bigger potential issue.

To answer the second question, one could state that there might be an issue of causing unemployment for game developers – level designers especially – as the process is automated, although this will in turn open up jobs for developers implementing the PCG. Nevertheless, the question of potentially higher unemployment for designers is connected to the greater issue of increasing automation throughout society.

On a positive note, further development of PCG tools is likely to empower game developers – especially indie game developers, who often do not have the resources to manually design all aspects of a game. It stands to reason that this could lower the barrier to entry into the game development industry.

2 Background

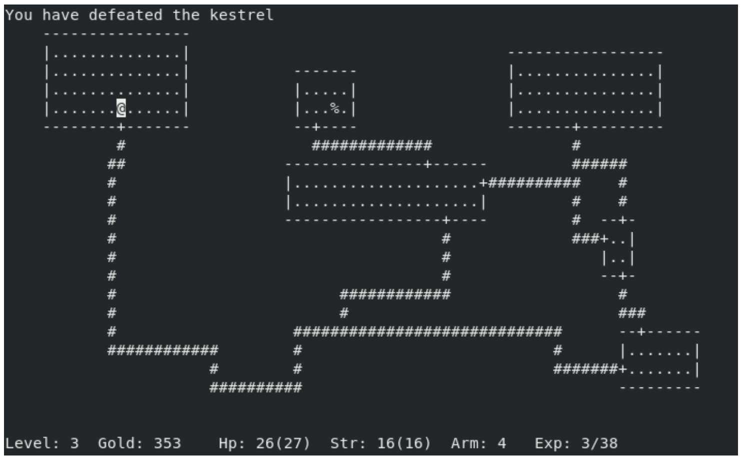
PCG can be used in various fields, for example in games, films or music. This section will focus on the background of PCG in games and previous research on that topic.

2.1 Games

PCG has been used since the early days of the gaming industry. Examples of such early games include *Rogue* (1980)[4] and *Elite* (1984)[6]. In the case of *Elite*, PCG was in fact necessary in order to circumvent the strict memory limitations of early systems[1, p.4]. In *Elite*, the player can explore seemingly endless galaxies including over 2000 stars, while only taking up just over 20 kilobytes of disk space[7, 8].

Rogue and the genre roguelike

Rogue is commonly known as the first game to use a procedural generation system (though this might not technically be true[5]). In *Rogue*, the player is an adventurer exploring a dungeon, searching for a rare artifact. Each level of the dungeon is procedurally generated as the player advances through the game. This gives the player a different set of levels on each playthrough. See Figure 1 for an example of a procedurally generated level in *Rogue*.



```
You have defeated the kestrel
    ----------------
    |..............|                                  -----------------
    |..............|           -------                |...............|
    |..............|           |.....|                |...............|
    |.......@......|           |...%.|                |...............|
    --------+-------           --+----                -------+---------
            #                    #############               #
           ##                 ---------------+------         ######
           #                  |....................+##########    #
           #                  |....................|         #    #
           #                  -----------------+----         #  --+-
           #                                   #             ###+..|
           #                                   #                |..|
           #                                   #                --+-
           #                        ############                  #
           #                        #                             ###
           #                   #############################      --+------
           ############        #                           #      |.......|
                      #        #                           #######+.......|
                      ##########                                  ---------


Level: 3  Gold: 353    Hp: 26(27)  Str: 16(16)  Arm: 4   Exp: 3/38
```

Figure 1: A procedurally generated dungeon in *Rogue*.

Rogue inspired the game genre *roguelike*, which is defined by a list of features present in the original game, including the use of PCG to generate levels[9].

Unexplored

Unexplored[10] (2017) is a roguelike game where the concept of dungeon exploration is kept, but the levels are generated in a circular fashion (Figure 2). This means that instead of the player only having one path to the goal, the level is generated as a cycle, so the player can take at least two different distinct paths to the goal. These paths are usually differentiated in some way. For instance, one path might be longer while the other is more dangerous or difficult[11].

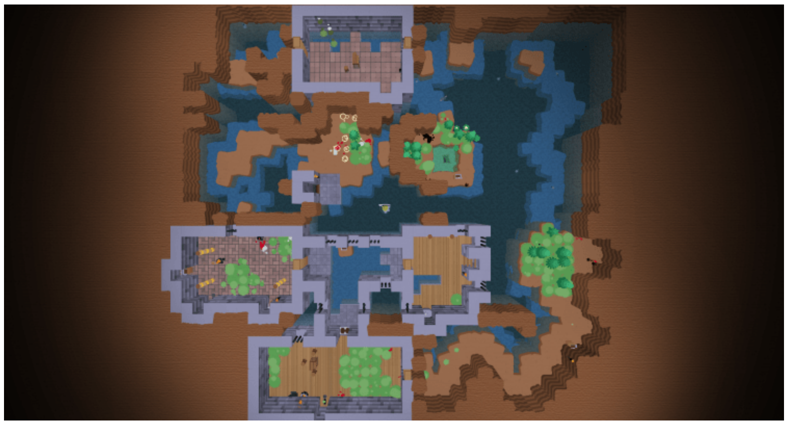


Figure 2: A procedurally generated level in *Unexplored*[11].

Dwarf Fortress

Dwarf Fortress (2006)[12] uses PCG very extensively to generate complex worlds. The player can have some control over certain aspects of the process, for example how large the world will be. Then a world is generated from multiple randomly generated parameters, for example, what temperature an area has or the average precipitation, and these parameters determine if an area will be a forest, desert, swamp and so forth. The world is then populated and its history simulated. This process creates unique and complex maps with mountains that have rivers flowing from them into oceans, the forming of lakes, civilizations that rise, build cities, wage wars and so forth until they perish and leave only the ruins behind. The entire generation is done before the player enters the world and the game starts.

Minecraft

Minecraft (2011)[13] is perhaps the most successful of modern games that make extensive use of PCG (Figure 3). *Minecraft* uses *Perlin Noise*[14][1, p.61] to generate a 3D environment with hills, plains, rivers and mountains, that correspond to different environment *biomes*, such as: desert, tundra, jungle or forest. The player can then explore this generated virtual world that is larger than the surface of the Earth[15]. A world of this size would be nearly impossible to create without PCG.

Figure 3: *Minecraft*, a seemingly endless procedurally generated 3D world.

2.2 Earlier work and research

Most of the work on this subject have been on developing actual games, which is brought up in the previous section. However, there have been numerous studies and papers on this subject which have a similar approach to PCG as this report.

In some of the papers, the authors developed a game alongside the paper. An example of this is the paper “PCG-Based Game Design: Enabling New Play Experiences through Procedural Content Generation” [16]. In this paper, the authors investigate how different PCG approaches impact games and the player’s experience, and develop the game *Rathenn* which highlights PCG.

A paper that focuses more on just the PCG is the paper “Compositional procedural content generation” [17], which describes the general strengths and weaknesses of various PCG methods, and how you can combine different methods to avoid the weaknesses while maintaining the strengths of the methods.

The paper “Understanding Procedural Content Generation: A Design-Centric Analysis of the Role of PCG in Games” [18] analyzes the role and purpose of PCG in games. The authors present a framework for understanding PCG in games, e.g. through giving the reader a proper vocabulary of PCG terms and a broad perspective on PCG. The purpose of this is for the reader to be able to discuss and analyze the role of the PCG in a game, with the help of the framework.

Furthermore, there has been a book written about the fundamentals of PCG in games:“Procedural Content Generation in Games”[1], which is the first textbook written about the use of PCG in games, according to the authors. The book serves as an overview of this subject. It describes multiple different methods and algorithms used in developing PCG for games. We used this book as the basis for our research of PCG, since it gave a good overview of the subject.

3 Theory

This section contains the relevant theory needed to understand the following chapters. Since this project is primarily about the exploration of PCG, all the relevant PCG algorithms and approaches are explained. This exploration is achieved by developing a small piece of software, which is why common practices and tools in software development also are explained briefly.

3.1 PCG algorithms

Procedural content generation (PCG) is the process of a computer generating content from different mathematical functions with random inputs[1, p.1]. Such content can be game levels, music or even narratives. In procedural level design in games specifically, there are several different algorithms that can be used to generate different styles of levels.

3.1.1 Agent-based dungeon generation

Agent-based dungeon generation uses one or more agents to perform tasks that will result in some kind of level. A classical approach is to let an agent move in random directions, always creating a path behind it, with a probability in each step to either change direction or place a room, see Figure 4. This approach can generate chaotic dungeons with very random paths and possibly overlapping rooms. If one takes care of these problems, though, the levels will tend to feel organic, or even planned[1, p.38]. However, the purpose and behavior of an agent can be very different from the above.

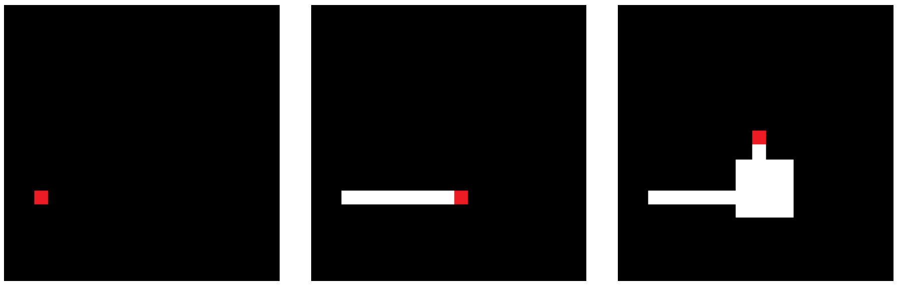


Figure 4: An agent (red square) is placed in a level filled with walls, seen on the left image. The agent is digging through the level (middle image) and places rooms (right image).

3.1.2 Cellular automata

Cellular automata-based content generation uses a grid which represents the level[1, p.42]. At the start of the generation process, the grid can be either randomly filled with different kinds of tiles, such as floors and walls, or have a more ordered layout. A set of transition rules are established, describing what happens to specific kinds of tiles in the grid at the next time step based on their *neighborhood*, an area surrounding the tile that can be defined in a manner of different ways. Time is then advanced to the next step, which changes the grid according to the transition rules. This is repeated a few more times until the grid has started to coalesce into something resembling a level. For an example of how it could look, see Figure 5. It is worth noting that the result will lack the coherency of the other approaches, making it harder to place quest items or control the pace through the level.

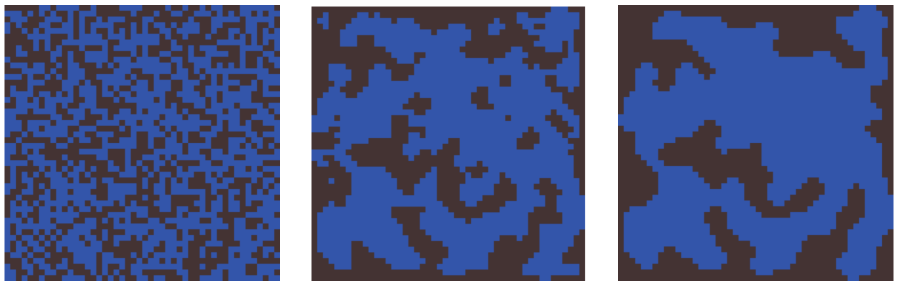


Figure 5: Map generation with cellular automata[19].

3.1.3 Cyclic PCG

Cyclic PCG is supposed to limit the need for players to *backtrack* through a level. Backtracking means that the player, usually after reaching a dead end when exploring an area, retraces their steps to be able to move in other directions. Instead of representing the main path through a level with a tree in a graph, the cyclic approach uses cycles which never generate dead ends (Figure 6. If correctly implemented, this makes it feel like there is some human planning behind it all and the level becomes more connected[11]. One example of a game that uses cyclic PCG is *Unexplored*.

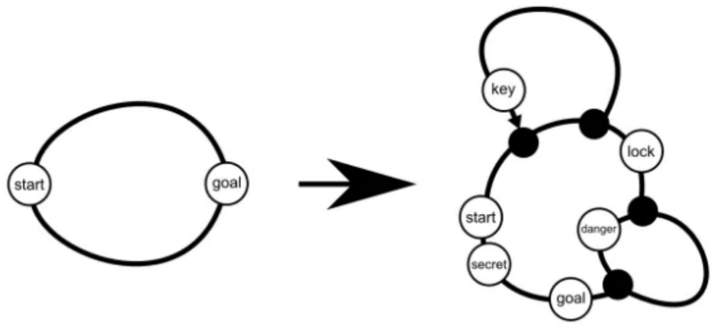


Figure 6: A cyclic level[11].

3.1.4 Grammar

Grammar-based content generation uses the concept of *formal grammars*, a system of rules that determine how structures composed of discrete units (traditionally strings of characters) can be formed and expanded[1, p. 74]. Variations include *graph grammars* that form graphs instead of strings[1, Chapter 5.5.1] and grammars that operate on two-dimensional arrays[20]. The abstract nature of grammars enables a lot of different approaches for level generation. For instance, they can be used to model high-level concepts such as a sequence of tasks the player has to perform[21], but also to directly form the structure of a level[20].

3.1.5 Search-based PCG

Search-based PCG uses an evolutionary algorithm, which is similar to Darwin's theory of evolution, where the computer evaluates the quality of content potentially generated by some other algorithm and removes the worst parts of it. The removed content is then replaced by randomly modified copies of the surviving content. This process repeats until the quality of the content is sufficiently high or the maximum number of iterations has been reached (Figure 7). To evaluate the quality, one can use AI that plays through the content and scores it, or some kind of fitness function[1, p.18][22].

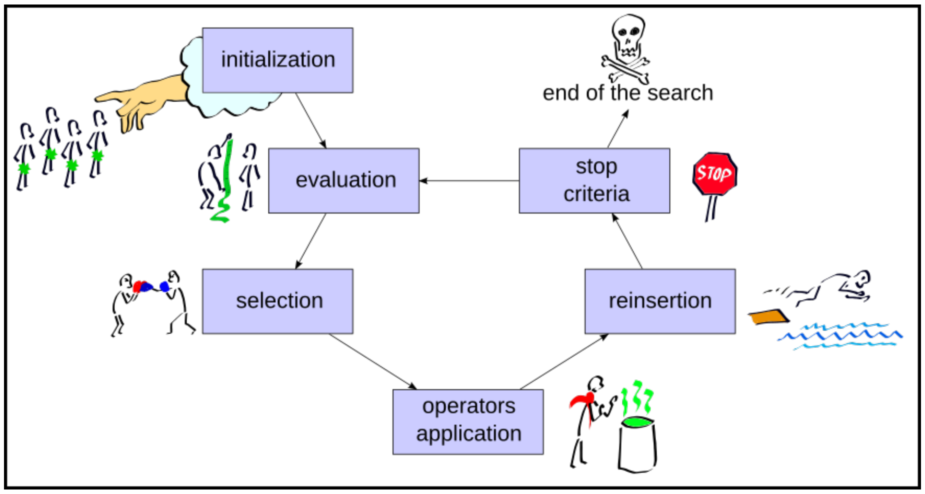


Figure 7: General schema of an evolutionary algorithm[23], image licensed under Creative Commons (CC BY 3.0).

3.1.6 Space partitioning

Space partitioning generates a basic layout of rooms by recursively dividing a given area into smaller disjoint areas along a random axis until they are small enough to be rooms, and then connecting these using doors or corridors. The most common method of space partitioning, called *Binary Space Partitioning*, recursively divides an area into two new areas[1, p.33]. This method generates a highly structured level with connected rooms. For an example of this type of dungeon generation, see Figure 8.

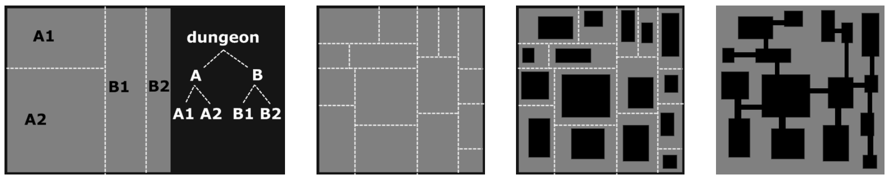


Figure 8: Dungeon generation with space partitioning[24].

3.2 Variations of PCG

As mentioned above, there are multiple techniques and algorithms for generating content. But there are also several approaches to how these are applied, for example: *online* or *offline*, *deterministic* or *stochastic*, *generic* or *adaptive*, and *mixed initiative* or *automatic*. All of our algorithm implementations are deterministic, offline, and automatic.

Online generation is when the world is generated continuously so that the player never reaches an end. On the other hand, offline generation generates a world of a specific size and will not make it larger if the player gets close to its border[1, p.7].

The difference between deterministic and stochastic generation is that deterministic generation generates exactly the same level if you use the same *seed*[1, p.9], while stochastic generation does not guarantee any consistency.

Adaptive and generic generation refers to whether or not the player's actions are taken into account in the generation. Adaptive generation will create different content depending on the choices the player makes while generic generation is done independently of the player's actions[1, p.7].

The concept of mixed initiative means that a user repeatedly is asked for input or has a say in how the PCG algorithm may perform. This is often done during level generation, where for example the algorithm and a developer take turns in generating new generations of a level, until it satisfies the developer. Automatic generation, on the other hand, is when an algorithm has full control and runs from start to end without interruption[1, Chapter 11].

3.3 Methodology

This project is centered around the development of software. Therefore the tools and working methods used have been carefully selected. This chapter gives a brief overview of some different software development processes as well as available tools.

3.3.1 Software development processes

There are multiple software development processes to choose from when deciding how to work during a project. Some of the most common are *Agile* and *Waterfall*. The *Agile* process is iterative and incremental, which makes it very flexible[25]. It is also an umbrella term for methods that share the same characteristics. Examples of these methods are *Scrum*, *Kanban* and *Extreme Programming(XP)*.

The *Scrum* method is based on collecting tasks: user stories, features and bugs, into a product backlog[25]. The backlog is represented by a *task board* or *Scrum board*, which can be either physical (a whiteboard or bulletin board with e.g. post-it notes) or virtual (for example *Trello*[26]). The board contains multiple sets of tasks, each set representing what state the task is in, with labels such as "To do", "In Progress" or "Done"[27]. The task board is used to structure the work into smaller chunks, which are developed during a limited time frame called *sprints*, usually between a couple of days to a month.

The *Kanban* method is similar to *Scrum.* It is based on three main principles: visualize the work, limit the work in progress and enhance flow. The *XP* method on the other hand, involves having continuous testing and planning, rapid feedback loops, close teamwork and dividing the work into short intervals[25].

The *Waterfall* process means that all phases of the development are planned linearly, so that the entire development team is working on the same phase. A team that uses the *Waterfall* process usually structures the project into these six phases: *Requirements*, *Analysis*, *Design*, *Coding*, *Testing* and *Operations.* The advantage of using the *Waterfall* process is that it makes the project structured and it enables relatively easy addition of new development teams[28].

3.3.2 Tools

When developing any kind of software, it is necessary to choose what programming language to use. The choice of language depends on what kind of software you want to develop, and what kind of tools and frameworks you will need. Common languages for game development include *C++*, *Java* and *C#*[29].

There are many different tools, game engines and frameworks for game development to choose between. There is also the alternative to make it from scratch, without any framework or game engine, although this can be very time consuming. When choosing tools, it is important to use something that is suitable for the game being developed. Things to take into consideration when choosing a framework or engine are for example the programming language, which platforms are supported and if the game is in 2D or 3D.

A few examples of frameworks used in Java game development are *LibGDX*[30] and *LWJGL*[31]. *LibGDX* supports both 2D and 3D games, and enables games to support both phones and computers[32]. *LWJGL*, on the other hand, is a library on a lower level, which gives the user more freedom, but also means that a lot of functionality needs to be implemented by the developers themselves[31].

A *version control system* is often used for a software development team to be able to manage the source code and track changes. One of the most popular version control systems is *Git*[33], which is free and open-source[34]. Git is most commonly used through *GitHub*[35], although *BitBucket*[36] also is very popular.

To document meetings and decisions as well as writing reports, there are various different methods available. Examples of these are using pen and paper, normal office software suites such as *Microsoft Office*, or document preparation systems such as *LaTeX*, which allows authors to separate the content of a document from its layout[37].

In regards to the creation of documents, another aspect which is important for group work is the matter of collaboration, namely to be able to work with the same document from different places, simultaneously. This could for example by solved by using *Google Docs*[38] or hosting group documents on *Google Drive*[39]. For collaborating in LaTeX, there is the web-based service *ShareLaTeX*[40].

To plan and structure the work a *Gantt*-chart[41] can be used. A Gantt chart visually shows all tasks in a project, and when they start and end. The chart is a grid with tasks on the Y-axis and time on the X-axis.

3.3.3 System architecture

When building any software of significant size, the architecture and structure are important [42]. Thus, there are a large number of different design patterns and models to use. The following is a small pick of patterns relevant to our project.

Model-View-Controller (MVC) is a common architectural pattern[43, chapter 8.4]. The code is divided into three parts; the Model describes the objects and how they interact, the View contains everything related to presenting the Model to the user, and the Controller lets the user make changes to the state of the Model. This separation of functionality lets developers work in parallel on these three parts of the project without causing compatibility conflicts.

The *Model-View-Presenter* pattern, or *Presenter-View* for short, is derived from the MVC pattern[44]. This version of MVC focuses more on the structure of GUI design than its predecessor does, and is mostly used for just that.

The *Client-Server* model is based on dividing a system structure into two parts: the servers that provide specific services, and the clients which request those services[45, chapter 7]. The internet is perhaps the largest example of this model being used, but one can implement this in simple software without network connections as well.

4 Process

This section summarizes how we have worked throughout the project, by explaining important choices that have been made and describing how we have taken on different challenges. This includes the planning of the project, the implementation of the game and the exploration and evaluation of different PCG algorithms.

4.1 Design document

To be able to properly plan all parts of our implementation, and to have an agile approach, we divided the game into small tasks in a *Design Document.* These were in turn prioritized into three categories:

Priority 1: Needed for completion of the game. Without all of these features, the game would be so simple that it would be difficult to sufficiently explore PCG.

Priority 2: Desirable for added complexity. While none of these features were essential for the game, the higher level of complexity they would provide might help us with more thoroughly exploring our PCG.

Priority 3: Not necessary but interesting. We either felt like they would not add much in terms of depth, or that they might be too time-consuming to implement properly.

The benefits of such a document were manifold: it ensured that we all had a common vision to work towards, it helped us agree on the game's limitations, and listing all of the features made it easy to set up a rough order of priorities (as seen above). Using those priorities, we could agree on a minimum viable game containing only the features we had deemed crucial so we could concentrate on these.

4.2 Implementation milestones

We divided our work on the game into several milestones that could be easily evaluated, so that we would have clear goals to work towards during the implementation. They were as follows:

1. Software structure and architecture, as well as assigning roles and responsibilities.
2. Minimal playable version: to be able to start the game and interact with some makeshift level.
3. Minimal basic PCG: some kind of clearly random element in the level design.

4. Finish with all priority 1 items described in the design document.
5. Finish with all priority 2 items described in the design document.
6. Make sure that the game is presentable, including debugging and testing.

When nearing upon the completion of milestone 4, we were forced to re-evaluate our plan: In the beginning of the project, we had hoped that we would be able to work with PCG in parallel to the game development. This did not work as expected, however, and the exploration of PCG was being given less time and effort than we estimated it would need for us to reach the project goals.

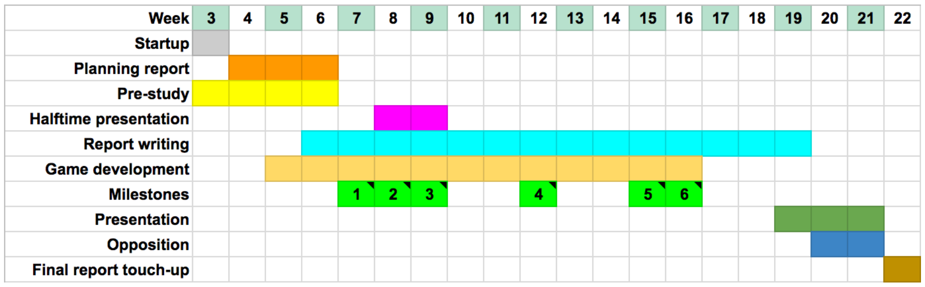
The solution to this was to revise our milestones. Instead of continuing to work on new functionality in the game, we instead decided to focus wholeheartedly on the development of different PCG algorithms, to be able to compare and evaluate their differences and similarities afterwards. As we were just about to start working on the fifth milestone, we decided to change this to reflect our new focus:

New milestone 5: Finish with PCG algorithm implementation

4.3 Time plan

There were two important implementation deadlines to keep track of. The first was the halftime presentation in week 9. We kept this in mind when planning the first milestones because we wanted to have something interesting to display when we got there. The second and final implementation deadline was set three weeks before our final report was due, to allow us to focus on the report.

An overview of our plans for implementing the game, the exploration of PCG, the final report and several other important parts of the project, can be seen in the Gantt chart in Figure 9. Note that the rows "Game development" and "Milestones" include both implementation of all the needed game mechanics and work on the different PCG algorithms.



Week	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Startup	■																			
Planning report		■	■	■																
Pre-study	■	■	■	■																
Halftime presentation						■	■													
Report writing				■	■	■	■	■	■	■	■	■	■	■	■	■	■			
Game development			■	■	■	■	■	■	■	■	■	■	■	■	■					
Milestones					1	2	3			4			5	6						
Presentation																	■	■	■	
Opposition																		■	■	
Final report touch-up																				■

Figure 9: *Gantt chart* of our time plan.

4.4 Development tools

The following is a summary of all the tools we have used during the course of this project, for reference.

Java We decided to use the programming language Java, since all team members had some degree of experience with it. Thus, we would not need to spend any extra time learning a new language.

LibGDX In addition, we decided to use the Java game development library LibGDX, in order to save time compared to making a game from scratch or using a more low-level game library such as LWJGL. A few group members had prior experience with LibGDX and could vouch for its usefulness.

Git and GitHub For version control of our code, we used Git. The repository was hosted on GitHub. Much like with Java, this decision came naturally as all group members had experience with both Git and GitHub, and little or no knowledge of any alternatives.

Google Drive To collaborate online, many of our working documents (including, but not limited to: meeting protocols, project logbook, and design documents) were hosted on Google Drive.

LaTeX The reports were written in LaTeX, using ShareLaTeX. As with Google Drive, this allowed us to access them from anywhere, and simplified cooperation. The reason we chose LaTeX over Google Drive for the reports, is the superior tools for layout and text formatting that LaTeX provides.

Trello We kept an agile taskboard in Trello, to keep track all the subtasks that needed to be completed in order to reach our milestones.

Slack For easy day-to-day communication, we used the messaging agent Slack. It worked well to have the ability to keep several conversations going on in different topics at the same time, as well as being able to upload files, inside a single project structure.

4.5 System architecture

The architecture of any computer program of significant size is of great importance. Software architecture include matters such as the structure and functionality of its classes, which design patterns to use as well as which ones to avoid, and high-level program behavior. It is not possible to fully plan out every aspect of the system structure before writing any code, but that is all the more reason as for why one should try to build an architecture that handles scaling and extensibility in a relatively painless manner. The importance of architecture in software engineering is the reason it was given a separate milestone in our planning.

One of the first things we discussed was if we should use any specific architectural pattern for the entire application. Many in our group had worked with Model-View-Controller before (Section 3.3.3), so this was the first pattern to be suggested. We also considered a couple of other ones, including Presenter-View and Client-Server, but we quickly agreed on using MVC. The main reasons for this were that it provides a strong structure to the program and was a familiar pattern to the majority of our group.

After a general architecture is decided upon, the details of this structure must be worked out. This meant that we had to list all the important components of the game, to determine where in the structure of MVC each component belonged. This is an iterative process, where adding additional components can mean that one is forced to rethink previous decisions. We therefore had to rework the structure of the game several times to make everything fit together in a sensible way.

In the end, we decided to move away from the architectural pattern MVC in a few specific areas of the architecture. We did this because we deemed it unnecessary to force certain design choices onto the system, when a much simpler solution existed outside of the decided pattern. A typical example of this is MVC's separation of Model, View and Controller: rules which we decided to break when modeling all game objects that were supposed to move around on screen. At the same time, we have tried to keep to these guidelines in most other aspects of the program.

When the architecture had been laid out as well defined as it was possible to do before actually writing any code, we created a *skeleton project* and uploaded it to our git repository. This means that empty Java classes were created to fit the architecture model we agreed on, to make it easy for each team member to start writing code, avoiding conflicts with the work of other team members.

The resulting architecture can be viewed in the *class diagram* in Figure 10.

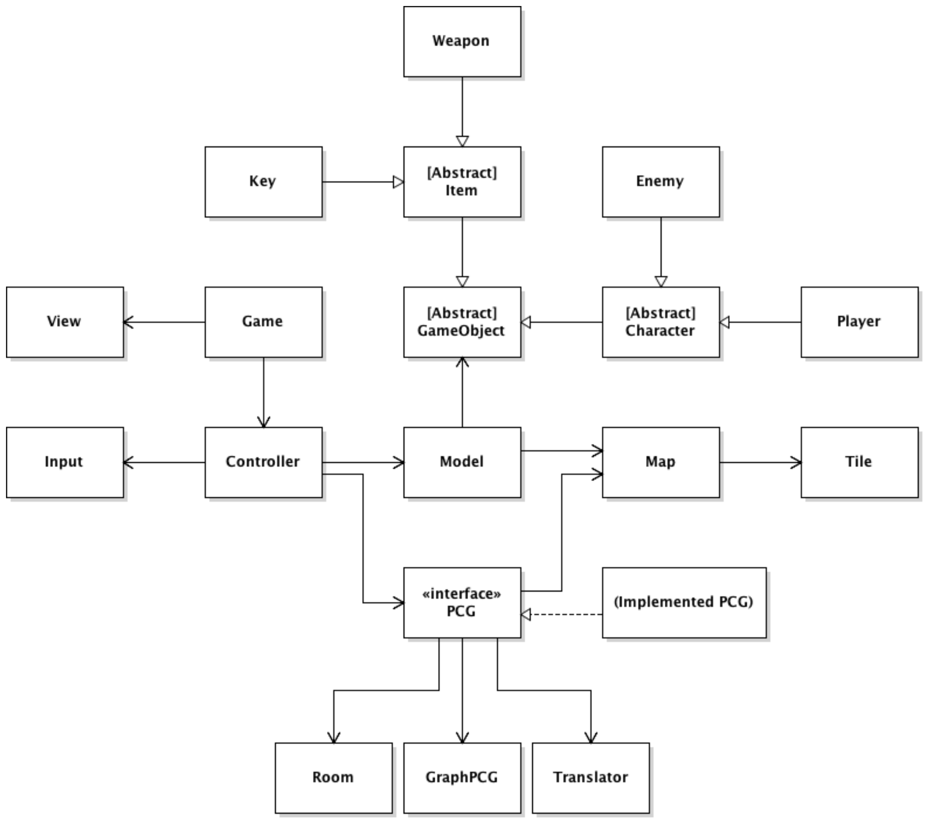


Figure 10: A class diagram showing the architecture of the game.

4.6 Working in an agile manner

Earlier in the project we talked about what kind of features that should be included in our game, and prioritized which of these to create first, in our design document (as described in Section 4.1). When starting the implementation of the game in earnest, we took every feature with the highest priority and divided these into manageable tasks, which we then put in the todo-list of our *Trello* taskboard (Section 3.3.1). The life cycle of a task in our taskboard looked more or less like the following:

First, a team member chose a task and tagged it with their name. The task was then moved to another list labeled "In progress", as the team member worked on it. The amount of work and time it took for a task to be finished was up to the team member, and depended on the size of the task or the grade of difficulty for the team member to overcome. We had regular meetings several times per week, and during these each team member got to explain how their work was proceeding and if any help was needed. In this way, we made sure that we kept making progress on the active tasks.

When a team member felt that a task was finished, the task was moved to another list labeled “Review/Verification”. During our previously mentioned meetings, we also looked through this list. If everybody could agree that it was finished, we moved the task to a weekly “Done” list. If anybody had something to remark upon, the task stayed in “Review” until any issues were resolved.

Most of the work on different tasks went on in parallel, but sometimes (especially in the beginning) several team members had to cooperate on, or at least discuss their respective tasks, to be able to solve conflicts or mutual problems. Several times we realized that some tasks overlapped, and therefore had to reevaluate exactly who should do what.

From time to time, we had to talk about which tasks to prioritize (among the ones with highest priority from our design document), so that important mechanics would be finished in time for other tasks to be able to be implemented.

4.7 PCG algorithms

In parallel with creating the game, we implemented a basic PCG algorithm to generate the content for it. The idea was to create a structure for having multiple PCG algorithms, build an abstract model of the map with some useful functionality and develop a method of translating it to a game level.

The basic algorithm created a number of randomly sized and positioned rooms. Then all the rooms were connected by corridors, regardless of whether they were already connected. We did not focus on trying to make the solution elegant, but rather on quickly creating a connected map. The start and goal were simply put on the far left and right borders of the map, respectively. As the game acquired new features we added simple methods to generate them in the basic PCG. The levels created by this algorithm are solvable, meaning that it is always possible to reach the goal, but they are not very interesting as they often feature plenty of overlapping rooms and unnecessarily long corridors (Figure 11). Although we ended up not using this algorithm in the finished game, the work on it resulted in a base structure for the PCG algorithms we would develop once the game features were all implemented, as well as a fully functioning translator for them.

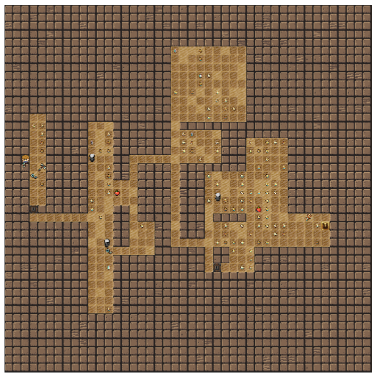


Figure 11: A level created by our first rudimentary PCG algorithm.

As we are six members in the project group, we decided that each person would take on a specific PCG algorithm approach. We tried to diversify as much as we could when choosing what algorithms to explore, as it was impossible to cover all available approaches in just six versions. We specifically chose not to include the search-based algorithm, mainly because it is very complex and therefore would demand more effort to implement than any of the other chosen algorithms. In the end, we decided upon these PCG algorithm approaches:

- Agent based
- Cellular automata
- Cyclic
- Grammar based
- Space partitioning
- An adaption of the algorithm used in the game *TinyKeep*[46]

We chose the agent-based, cellular automata, grammar-based and space partitioning algorithms because they are all well established methods for creating this type of PCG, with a substantial amount of documentation available. The cyclic approach was chosen because it actively tries to avoid some of the issues we are investigating with PCG. We picked the TK-algorithm partly because it is different from all the others, but also to have a less established method to compare to the others.

The work done up until this point (namely, all the priority 1 elements from our design document) was set as a baseline, to ensure that we all had the same starting conditions. This meant that each of us, while working with different algorithms, had exactly the same game elements and mechanics to work with when generating content. The thought was that it would make it easier for us to see similarities and differences between the algorithms. This could benefit us further on, when discussing the different algorithms' characteristics, advantages and disadvantages. In addition, it also helped us avoid collisions between our respective algorithms.

After this, our work progressed almost completely individually and in parallel to each other. We created our own branches in the git repository, to be able to work on our algorithms without interfering with the other team members' work. While working individually, we still met up at least once a week to make sure everybody progressed. Now and then, someone would come up with new functionality that could benefit other team members as well, but other than that, our different areas of work rarely came in contact with each other.

The following sections describe interesting challenges or characteristics specific to some of the algorithms. A detailed description of how the resulting algorithms actually work can be found in Section 5.2.

We have focused on algorithms that are primarily used to generate the layout of levels, so it may not be obvious why we chose to not place items and enemies exactly the same way for each algorithm. This was done because the structure of each algorithm varies and therefore the placement of these objects need to be handled at different times during the generation.

4.7.1 Agent-based approach

The classical approach to agent-based PCG is to let one or several agents dig their way through solid material in order to form the level (Section 3.1.1). However, the idea with this algorithm was rather the opposite: to let agents build the walls in an otherwise open level, creating an intricate maze as a result.

The challenge of generating rooms

A great challenge was to generate a random number of rooms of random sizes, inside a given level size, while also making sure that they did not overlap (Figure 12). When generating a small amount of rooms, there is no issue with just checking for overlap with all other rooms, but when the room count rises, the work needed grows exponentially and gets unmanageable very fast.

Fortunately, in line with the nature of this algorithm, only a small ratio of the level area needs to be covered by rooms (because the rest will be covered by a still traversable labyrinth). This sparsity means that an approach that would be impossible to apply in other scenarios actually works pretty well here: randomizing rooms until the target ratio is achieved without overlapping.

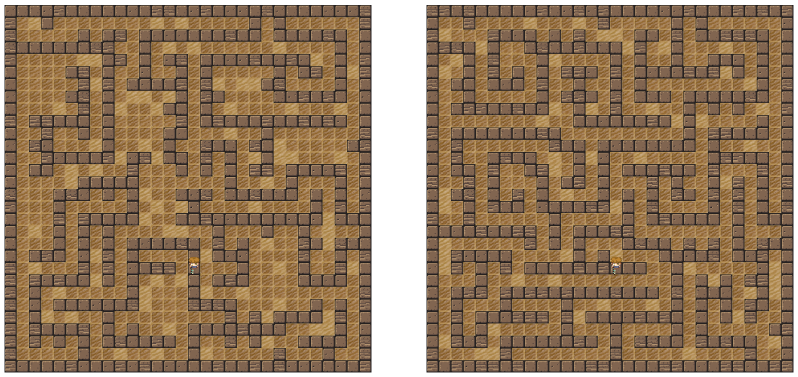


Figure 12: A level generated with rooms (left) and without (right).

Designing interesting agents for wall creation

The initial idea was to let agents originate from each abstract, wall-less room, but the problem with this turned out to be that the outer rim of the level would end up as an obstacle-free corridor. Instead, it was decided that each agent should start at the mentioned outer rim of the level, and instead work their way in towards its middle. Not only did this solve the "outer corridor problem", but incidentally also made sure that there is always exactly one way to reach the goal, making it easier to place locked doors and keys later.

While creating the behavior of these agents, it became obvious that they sooner or later would reach an obstacle and not be able to continue. To remedy this, a backtrack function was added, so that as soon as an agent reaches a dead end, it can keep on going in another direction from an earlier position along its previous path (Figure 13).

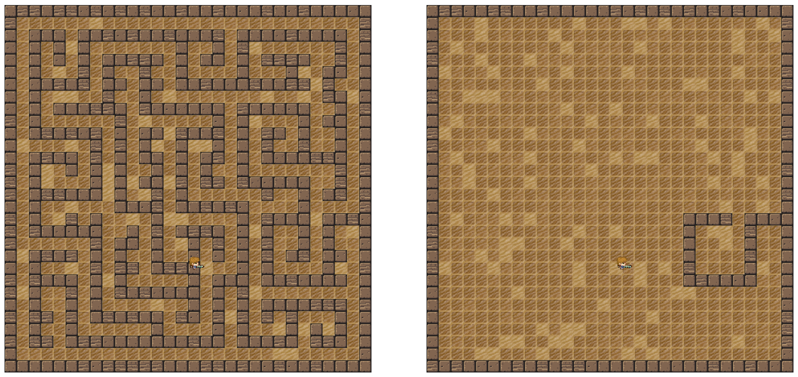


Figure 13: One wall agent traversing the map, with backtracking (left) and without (right).

Using agents for other purposes

For the placement of locked doors and their keys, another agent-based approach was chosen: to let a single door agent make use of a previously created pathfinding algorithm to distribute locked doors and keys (Figure 14). This second agent-based approach was not decided upon from the beginning, yet it proved to be a simple but powerful solution. In hindsight, it was also discovered that this approach ensured that the player never would stand on one side of a locked door, with the key on the other side. In other words, this approach makes sure that the level is always solvable (in regards to the locked doors, at least).

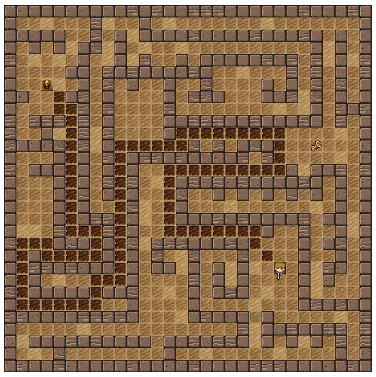


Figure 14: The path an agent walks when placing locked doors and keys.

Solvable levels

When starting out on this algorithm, expectations were that it would need some kind of mechanism that makes sure that the finished level is solvable, i.e. that the player can always make it from the entrance to the goal. Contrary to these expectations, it turns out that such a mechanism was not needed. Instead, the solvability property was instead built into the different agents used in this approach. More on this in Section 5.2.1: Launching wall-creating agents and Launching locked-door-creating agents.

4.7.2 Cellular automata approach

The basic idea behind the cellular automata algorithm is to change individual cells based on their neighboring cells and a set of state change rules (Section 3.1.2 for details). In our implementation, cellular automata was used only for terrain generation and game objects were generated using other methods.

The initial terrain generation was rather straightforward. It took some experimenting to find a good range of settings, but the base algorithm did not require much complexity to generate the sought after result: a map that we perceived as looking unstructured and natural, rather than man-made. This process was very much one of trial and error; it was difficult to make any predictions about the end result based on the settings, and small adjustments sometimes led to considerable differences in the end result.

Lack of structure

Generating the map at the level of individual tiles resulted in a lack of structure. This led to the problem of not being able to ensure the creation of connected rooms. For some settings and some of the time, a map with sufficiently connected rooms was created. However, at other times the map was divided into several completely walled-off rooms. Attempts to avoid the latter problem by changing settings tended to lead to very open maps, which made the path between two points much too straight and simple. For examples of these issues, see Figure 15 below.

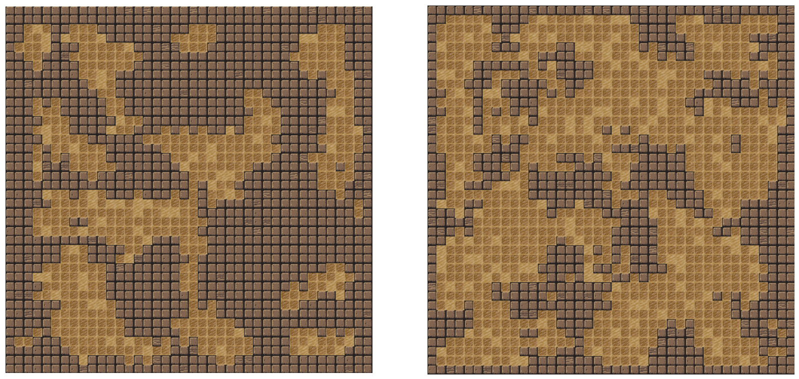


Figure 15: Terrain generated from different settings. In one map, several rooms are walled off (left). In the other, all the rooms are connected but the map is too open (right).

To be able to have levels with different rooms rather than one open space, the decision was made to include corridor generation. This was decided even though it often reduced the natural feel of the map which is one of the main advantages of using cellular automata. This design decision results in maps that often look like natural caves which have been excavated (Figure 16).

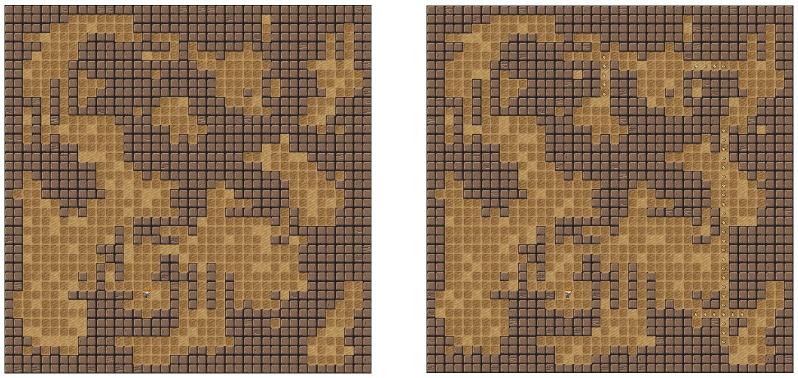


Figure 16: Generated terrain, without corridors (left) and with (right). Coins mark the corridors. Without corridors, several rooms on the right side of the map cannot be reached.

Graph representation

In order to be able to ensure that the rooms were properly connected by digging corridors, as well as ensure the placement of keys and doors in a solvable order, a graph representation of the map was created. This step, along with the corridor generation, took a lot more time than implementing the actual cellular automata step. We felt that this higher-level abstraction was necessary to gain enough control over the level design to be able to control the gameplay to a sufficient degree.

4.7.3 Cyclic approach

The thought behind cyclic PCG is that each important point has at least two paths leading to it with different attributes (Section 3.1.3). We chose to focus on a model where to be able to progress, you need to take a side route to find a key, but once it is found the path back is short.

First, a simple representation of the general layout and flow of the level is generated. This is done by creating *paths* between two points. To make cycles, each node in a path has a chance of creating a new path originating from and ending in that node. The first path is generated between the start and goal nodes.

One could argue that our version of cyclic PCG is not actually strictly cyclic as we do not start with a cycle between the start and goal rooms. Instead we have a linear path with cycles along it. This is because we felt that there was no need to have a short return path from the goal to the start as the player is unable return to previous levels.

Translating the graph

After an abstract graph of the level had been created, the PCG needed to start translating the nodes along the path to rooms. At first, it started by placing the rooms along the main path in the lower part of the map. Any branching paths were placed above and as far to the left as possible.

This proved not to be a very good way of placing rooms though, because if there were two branching paths from the main one, both would place rooms in the same area, making them overlap (Figure 17). This was fixed by generating rooms along the main path in the middle of the map and allow for branched paths to generate rooms both above and below the main path. By only allowing for two branching paths from the main path, placing one above and one below, and only letting branched paths branch again once, solved this problem as it kept the paths separated.

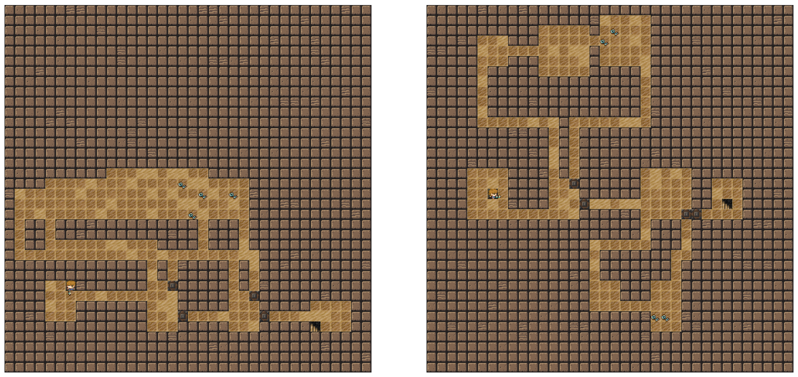


Figure 17: Rooms with the bad placement (left) and rooms with the improved placement (right).

However, this introduced the problem that the maximum number of branches from the main path is bound to two. To fix this, we made sure that two nodes could not branch in the same vertical direction unless there were at least three nodes without branches in the same direction between them. This gave the branches enough space to not interfere with each other, which enabled the algorithm to generate more than two branches if the map is big enough.

Adding corridors

Generating corridors to rooms in another path was problematic, as they often are located diagonally above or below the start room. If we were to generate a normal corridor between two such rooms, this would in many cases cause the corridor to cross other rooms, enabling the player to take shortcuts and ruining the flow of the game.

The solution was to make a special corridor between nodes in different paths that first generates half the vertical part, then the horizontal part and then the rest of the vertical. The horizontal part is therefore generated between the paths and not crossing any rooms in either of them (Figure 18).

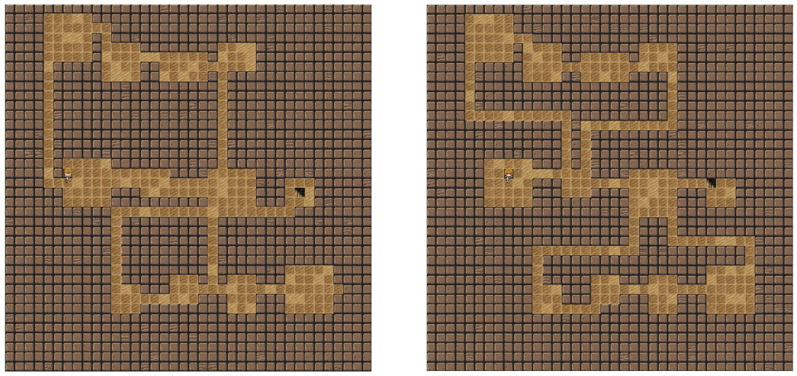


Figure 18: The difference between building normal corridors to branched paths (left) and building special corridors (right).

4.7.4 Grammar-based approach

For grammar based level generation to work, two components needed to be implemented: a system that would use grammar rules to generate an abstract model which would represent the level, and a system that could translate said model into an actual level. As such, the first step was to decide on what kind of model to use.

Choosing a model

Grammars are typically used with one-dimensional strings of characters, but could the level – a two-dimensional arrangement of rooms – be modeled properly with a such a string, or would it require a more complex model of representation? Since it would be preferable if the algorithm could generate levels with branching paths, a graph model seemed more appropriate for representation, meaning that the algorithm would be based on the concept of graph grammars (Section 3.1.4). However, it turned out that even with one-dimensional character strings, it would be possible to model branching paths through the use of special characters which would signify the start and end of so-called *side paths* – sequences of rooms which did not lead towards the level's exit (Figure 19). Thus it was decided that the grammar based approach would forego a graph grammar system in favor of a less complex one-dimensional string grammar.

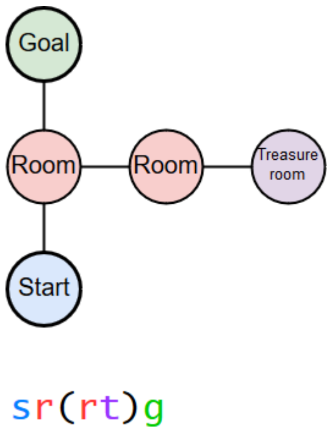


Figure 19: The same level structure modeled as a graph (top) and a string of characters (bottom).

Start of implementation

The algorithm was designed so that it went through the finished string character by character, building a new room for each and making sure it was placed as close to the previous room as possible (Figure 20).

While this initially appeared to work fine, an oversight soon revealed itself. Since corridors could be placed anywhere, it was possible for them to intersect earlier rooms. This could potentially break the intended order of progression (Figure 21). To rectify this problem, corridors had to undergo the same checks as rooms did before being placed.

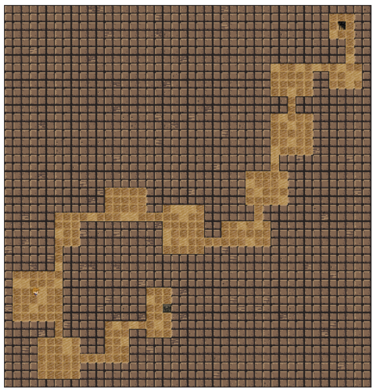


Figure 20: A long sequence of rooms connected together. The irregular shape of the level is a result of how every room is placed only with regards to its predecessor.

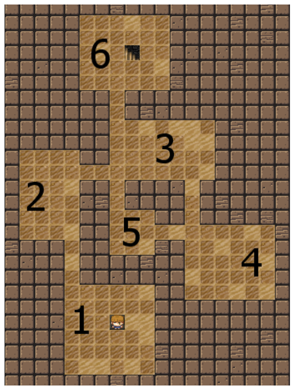


Figure 21: A result of placing corridors without checking for intersection. The correct room order has been indicated with numbers.

Error handling

Ensuring that corridors followed the same rules as rooms led to the discovery of a crucial flaw with the algorithm: since corridors could only be placed where they did not overlap any rooms, the algorithm could now get stuck if a room could not be placed and connected to the previous one. As an example, consider Figure 21. Instead of placing room 6 far away from room 5 and bridging the gap with an invalid corridor, the algorithm would now simply be unable to progress after room 5. To make matters worse, the causes behind this problem were deeply rooted in how the algorithm was constructed. For one, the string used to represent the level contained no information about how the rooms were arranged, only their sequence. This meant that there was no way to ensure a properly constructed level layout before starting to build the rooms. Secondly, the algorithm could not "look ahead". Each room was only placed relative to its predecessor, with no knowledge of whether it was heading into a dead end.

At this point, it was clear that unless the algorithm were to be fundamentally rewritten, there needed to be some kind of error handling system in place. Since the rooms were built in a sequence, it seemed sensible to take a step backwards in the sequence when something went wrong. This prompted the creation of a backtracking system that is initialized when reaching a dead end. It then undoes the placement of each room until one can be placed in a direction that leads away from the dead end, at which point the algorithm resumes as normal.

See Figure 22 for an example of a finished level.

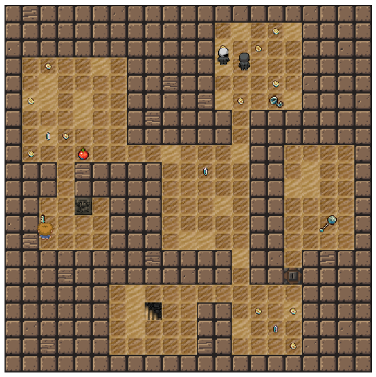


Figure 22: A complete level generated with the grammar-based algorithm.

4.7.5 Space partitioning approach

As briefly explained in Section 3.1, space partitioning is based on dividing an area into a number of smaller areas that are disjoint to each other. This process is repeated until the desired properties of the areas are met. We chose to use binary space partitioning (Section 3.1.6) as the specific space partitioning method (Figure 23).

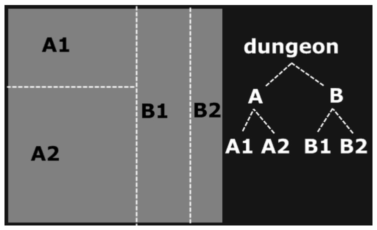


Figure 23: Binary space partitioning with a binary search tree[24].

Generating a space-partitioning tree

The decision of the desired properties, i.e. the desired sizes of the areas, affects the size of the final rooms (since rooms are generated within those areas). To determine the appropriate settings for desired width and height of an area, a series of space-partitioning trees were generated. The trees were generated with different settings, and were then translated into tilemaps to investigate which room sizes worked well with regards to gameplay.

Another aspect of implementing and generating a BSP tree is whether the recursive dividing should stop as soon as an area has reached a desirable width and height. That would create rooms that are generally closer to the upper size limit than to the lower size limit. The alternative to this would be to continue the recursive dividing somehow, for example continue dividing at a certain probability, even though an area has a desired size. This was investigated with a series of different implementations to conclude what was the best option. The resulting partitioning can be seen in Figure 24.

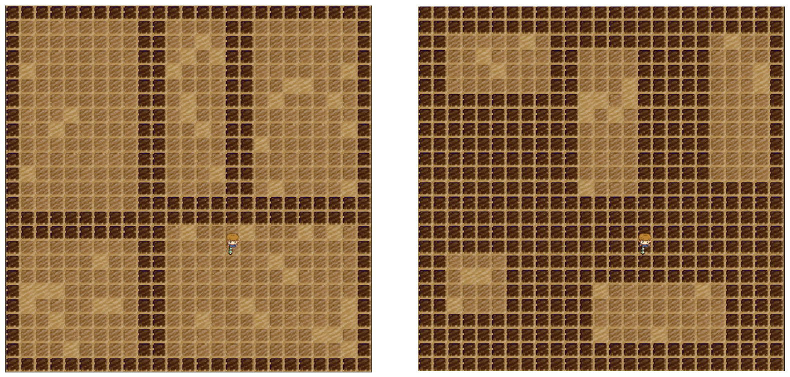


Figure 24: Disjoint areas generated using BSP (left) and rooms generated within those areas (right).

Connecting rooms

When connecting rooms with corridors, there are several choices to consider, e.g. how the corridors should look like, where to place the corridors, between which rooms etc. All these questions could be answered with the objective that the corridors between rooms should be as short as possible. This was decided because the corridors' only purpose is for the player to go from one room to another. If the corridors were too long, it would probably introduce a lot of uneventful moments, which could be seen as boring.

Generating game objects

While space partitioning does not provide any rules on how to generate game objects, there is one common method that space partitioning can provide which can affect how the game objects are generated. That is to introduce various themes, which are assigned to the areas. Themes are assigned when the BST is generated. When a node is assigned a theme, all its children will inherit the theme (Figure 25). The themes can determine where and at what probability different game objects can be generated. This can thereby introduce different environments, such as biomes in *Minecraft*[13], see Section 2.1. Due to the small number of game objects and tiles in the game, it would be difficult to introduce biomes. However, themes that change game objects' probability of being generated were possible, and thereby used in the implementation of the space partitioning algorithm.

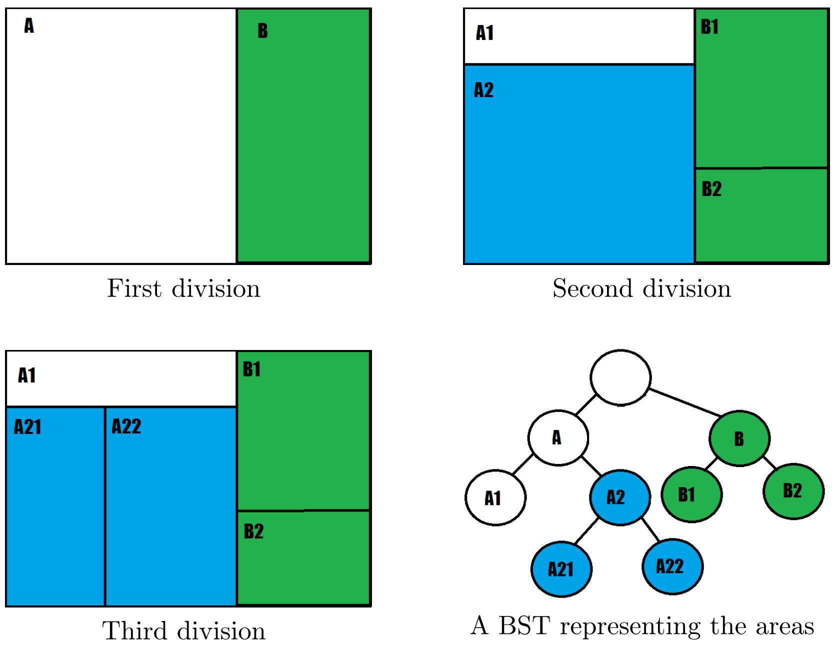


Figure 25: An area dividing into smaller areas, with random theme assignment (represented as a color, excluding white). A2 and B are assigned a theme, and their children inherit the same theme.

4.7.6 TK algorithm

The TK algorithm is named as such because it is an adaption of the algorithm used in the game *TinyKeep*[46, 47]. The basic idea was the same but the final result is quite different as we adapted it to fit our game and the structures we already had in place.

Creating the rooms

The first thing we did was to create a lot of rooms of varying sizes and adding them all as nodes in a graph. This was easy. The tricky part was trying to create them in a way that they did not overlap. Initially we tried spacing them out one at a time, moving them out from the center of the graph until they had no neighbors. However, this was very inefficient, and after trying a few different approaches we in the end came up with another method.

Every room in the graph is placed at random positions, until the entire level is filled. The decision to place the rooms randomly in the map rather than for example systematically filling the map from one corner to another was made to create more diversity in the look of the map. However, sometimes this would lead to areas not covered by rooms. To fix this, we simply create way more rooms than needed to completely cover the map. Finally, we made sure that all the rooms had some space between them, as entirely adjacent ones resulted in too open maps (Figure 26).

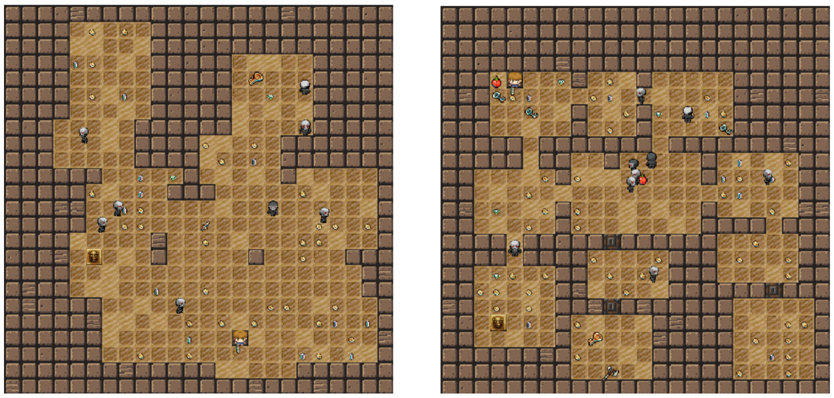


Figure 26: The level generated with the random rooms directly adjacent to each other (left) and with a small separation (right).

Connecting the rooms

The idea of how to connect the main rooms is to add in small 1×1 rooms between main rooms to form corridors. This turned out to be a little bit tricky as well. The original algorithm does this by drawing edges in the graph between all the nodes in such a way that none of edges cross each other. Then a minimal spanning tree[48] is created from those edges, representing the necessary corridors, and the rest removed.

Since the graph structure we already had in place was unweighted and not well suited for this approach, we took a different route: all the rooms are in a random order connected to each other, until every room is reachable from the start. A detailed description of how this works can be found in Section 5.2.6 and the result can be seen in Figure 27.

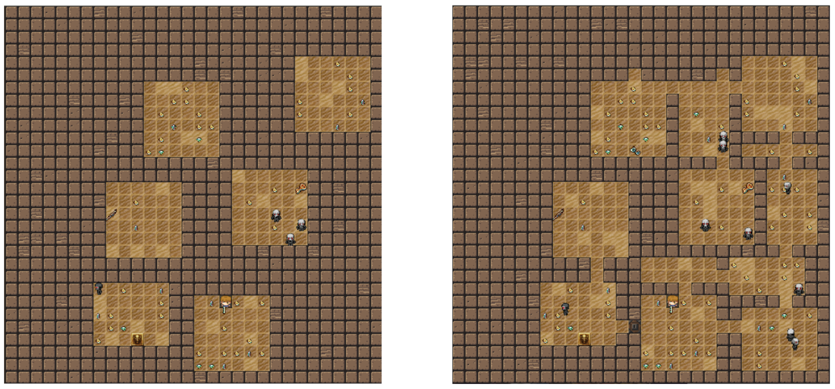


Figure 27: The level before (left) and after (right) corridors are added.

4.8 Evaluating the exploration of PCG

The goal of this project was to learn as much as we can, by getting hands-on experience with the development of PCG in a game. A large and important task has therefore been to evaluate what exactly we have learned throughout the project.

As each team member has explored different algorithms individually, the amount of resulting knowledge and revelations was extensive, but not shared between us. Because of this, a first important step of the evaluation was to sit down and talk about the different algorithms. We agreed on a list of different properties and attributes that we thought could be relevant in such an evaluation. Mainly, we focused on attributes that stood out among the six algorithms, and properties that we all could agree on would influence the experience of a game, while still being relevant to the development of PCG. These properties are listed here in short, with no internal order:

- How hard is the algorithm to implement?
- How many levels of abstraction does it use?
- How easy is it to change or adjust it into something specific?
- How difficult or easy is to use? For whom?
- How easy is it to mix with other algorithms?
- How handmade does it feel? I.e, is it easy to detect that a human did not build a specific level?
- How much variation does it allow? Can one easily see patterns?
- How well does it scale up or down?

- How difficult or easy is the resulting level?
- How enjoyable is the result?

We made sure to have several discussions on this subject. At first, we concentrated on talking about each of the above aspects, to give a better feel for how each of the algorithms worked. After that, we moved on to focus more on the different significant connections between the explored PCG approaches, and making sure that we did not miss anything. In the end, we distilled the original list, by combining similar qualities or removing those that did not tell us anything new. The result was the following list of key properties:

- What we learned from working with the algorithms
- Combination possibilities between them
- The ability to adjust the algorithms
- Quality of the generated content
- Variations in the generated content
- Difficulties of the resulting levels
- Enjoyment of the resulting levels

The resulting observations and insights about these topics are compiled in the Result Section 5.3. This also includes a summary of our thoughts about each of the six explored PCG algorithms specifically, for reference.

5 Result

This section presents the final game *Fluky* along with its features, as well as the PCG algorithms and how they work. It also includes the analysis and comparison of the PCG algorithms, as well as some thoughts on working with them and how work on them could progress in the future.

5.1 The game: Fluky

The game is a tile-based dungeon crawler called *Fluky*, where the player has to navigate a level of rooms and corridors to reach the goal. To make it harder, there are enemies which hurt the player upon contact and doors that require the player to find a key to progress. There are five types of tiles: wall and floor (which both can have different colors), door, start and goal. Areas far away from the player or on the other side of walls are darker, which gives the game more atmosphere (Figure 28).

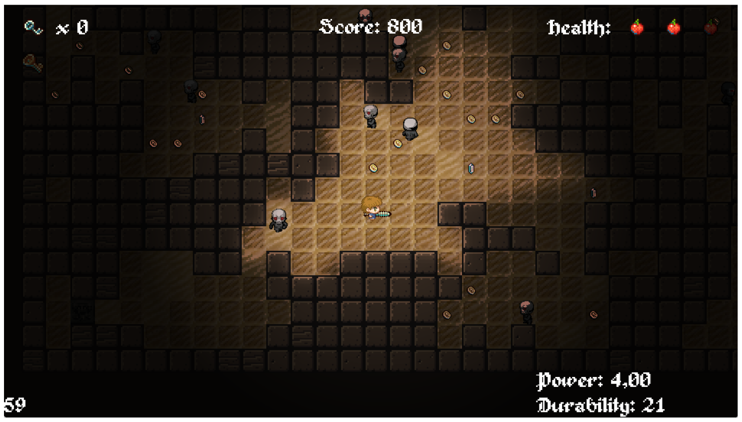


Figure 28: A screenshot from the game.

5.1.1 The player

The player (Figure 29) can walk in eight directions: up, down, left, right and diagonally. They can also attack in four directions independently of the direction they move in, so they can for example walk right while attacking left. The player has an inventory which can hold any amount of keys and up to two weapons at the same time. To pick up keys, the player simply walks over them, while weapons have to be manually picked up. If the player already has two weapons while trying to pick up another, the currently equipped weapon will be tossed to the ground and replaced with the new. If the player comes in contact with an enemy, they will be pushed back, lose one health point and become invincible for a short period of time so that they have time to run away or counterattack.

Figure 29: The sprite used for the player[49], image licensed under Creative Commons (CC BY-SA 3.0).

5.1.2 Enemies and items

Fluky has three types of enemies (Figure 30). The most basic is the patrolling enemy who walks back and forth along a straight path. Then there is the randomly walking enemy who walks in a random direction for a certain time, then chooses another random direction to walk in, and so on. The most difficult type of enemy is the following enemy. If the player is close enough, the following enemy will start to chase them.

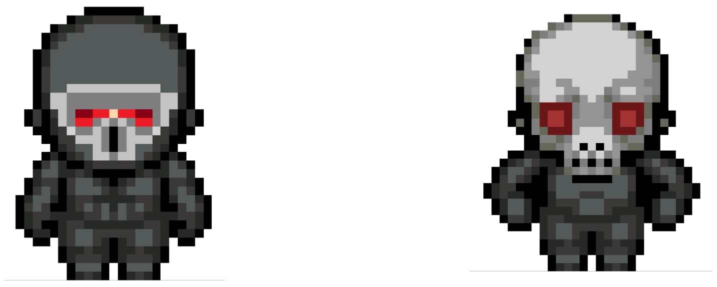


Figure 30: The sprite used for following enemies (left) and the sprite used for both patrolling and randomly walking enemies (right)[50].

Collectibles are items that are non-essential to the progression of a level and are used to give the player an incentive to explore. Whenever the player walks over a collectible it is picked up and the player's score is increased. There are three different tiers of collectibles, worth 10, 50 and 100 points respectively (Figure 31). Also, if the player has taken damage, they can pick up apples which restore one health point each until they are at full health.

Figure 31: The graphics for the collectibles[51]. The left one is worth 10 points, the middle one is worth 50 points and the right one is worth 100 points. The red apple represent a health item.

Fluky also has weapons (Figure 32), which are divided into five weapon classes: daggers, spears, swords, axes and maces, each favoring different stats. The stats are: damage, attack duration, range, attack angle and durability. Daggers tend to have fast attack duration at the cost of damage, spears have long range but low attack angle, swords are a jack of all trades, axes tend to have high damage but can have low durability and maces have high durability at the cost of damage.

Figure 32: The graphics used for the weapons[51]. Weapon types, from the left: Axes, Daggers, Maces, Spears and Swords.

5.1.3 Menus and user interface

When launching *Fluky* the player is first faced with the main menu which has four options: Quit, Credits, Choose PCG and Play (Figure 33). If the player selects Play, a text field appears where they can enter a seed which will be used to generate the level, and the game will randomize which PCG algorithm will be used to create each map. If the player chooses to not enter a seed, the seed will be randomized. If the player selects Choose PCG, they will instead be able to select which PCG algorithm will be used to generate the levels.

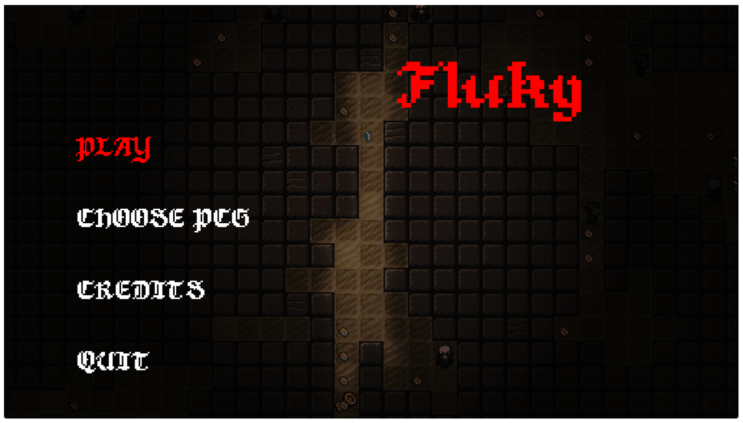


Figure 33: The main menu.

When the player is in the game, there is a *heads-up display* (HUD) that shows the player important information about the state of the game (Figure 28). It shows how many keys the player has collected, how much health is left, the score and the currently equipped weapon's damage and durability. The HUD uses icons instead of text where suitable to make it easier for players to quickly get the information. Health, for example, is represented by icons of an apple.

5.2 PCG algorithms

The game's levels can be generated with one of six different PCG algorithms.

5.2.1 Agent-based approach

As mentioned in Section 4.7.1, this version of the agent-based approach differs from the classical version by using agents mainly to build walls, rather than tunnels. This results in a level design that is very similar to that of a two-dimensional maze. Because of these choices, the way this algorithm works may differ from how the agent-based approach is described in Section 3.1.1.

The following is a short summary of how this algorithm works.

1. Generate random rooms that do not overlap.
2. Assign entrance and goal rooms, as far away from each other as possible.
3. If necessary, assign a third room to hold the key to the goal, as far away from both the entrance and goal as possible.
4. Translate the abstract model to an actual map.
5. Create and launch a number of wall-generating agents onto this empty map.

6. Create and launch a number of locked-doors-generating agents into the maze created in (5).
7. Randomly populate rooms first with valuable items, and then generate enemies guarding these items.
8. Randomly populate maze corridors sparsely with treasure and enemies.

Generating rooms

The rooms in question are represented as two two-dimensional coordinates that mark the lower left and upper right corner of a rectangle. To generate these, the area size of the level to generate is calculated, and then randomly sized rooms are generated (within certain parameters) until a set ratio of the calculated level area is covered by the total area of all rooms together.

Each time a new room is generated, it must be verified with all other accepted rooms that they do not overlap. If they do, the new room is thrown away, and otherwise it is added to the total as another accepted one. This verification process is very inefficient, since it has to do a lot of iterations over all created rooms. However, it still runs fast enough to not impede the game. This is because this specific algorithm only needs a relatively small ratio of the total level area to be covered, resulting in few rooms and few passes before the sought after ratio is achieved.

Assigning entrance, goal and goal-key rooms

Again, a fairly inefficient method is used to find the two rooms farthest away from each other. The distance between every pair of rooms is calculated and then each pair and the distance between them are put in a list and sorted by the distance. The inefficiency of this method does not cause problems in this case either, for the same reasons as in the previous section: the amount of rooms is relatively small.

After assigning entrance and goal rooms, a similar method is used to find a room for the key to the goal. The aim here is to find a third room that is as far from both the entrance and goal as possible. However, as the entrance and goal rooms have already been found, this can be done fast.

Translating the abstract model to a map

There are few concrete attributes to translate at this point. Since the subsequent steps in the algorithm require an empty map, only the specific tiles for the entrance, goal and goal key are placed in an otherwise empty map. The only other thing that is added to the level at this point is an outer wall, to ensure that the level is contained. See Figure 34 for an example.

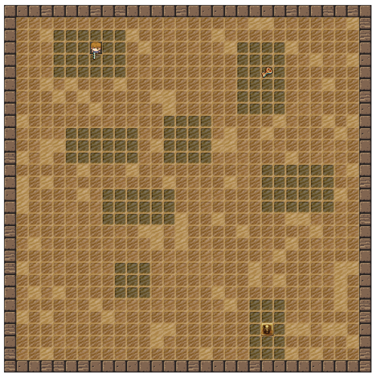


Figure 34: The virtual rooms in the agent-based approach (marked with darker floor tiles) translated to a map, including entrance, goal and key.

Launching wall-creating agents

This is the very core of the agent-based approach. A number of agents are created, based on the size of the map. These agents are then launched from the outer wall of the otherwise empty level, moving across the level in tandem.

A number of rules guide their movement, in combination with a random generator, to make sure each generated map is different. The rules of each agent's movement make sure that no agent moves in a direction that would connect it to any wall or room. Each agent has a direction and in every step it randomly changes direction. In addition, a backtracking mechanism is in place so that if an agent reaches a dead end it can move backwards along its path, and branch out in other directions as soon as there is room. For snapshots from different iterations of the agents at work, see Figure 35.

Wall agents will never connect with each other, so the only connections between them are via the outer wall. And because no agent starts in the middle of the level, but at the outer rim, the resulting corridors between them will all be connected. This property of the algorithm happens to be very important, as it ensures beyond a doubt that every level can be solved. In other words, it does not need to be verified after construction, because the verification is built into the mechanism of the level design.

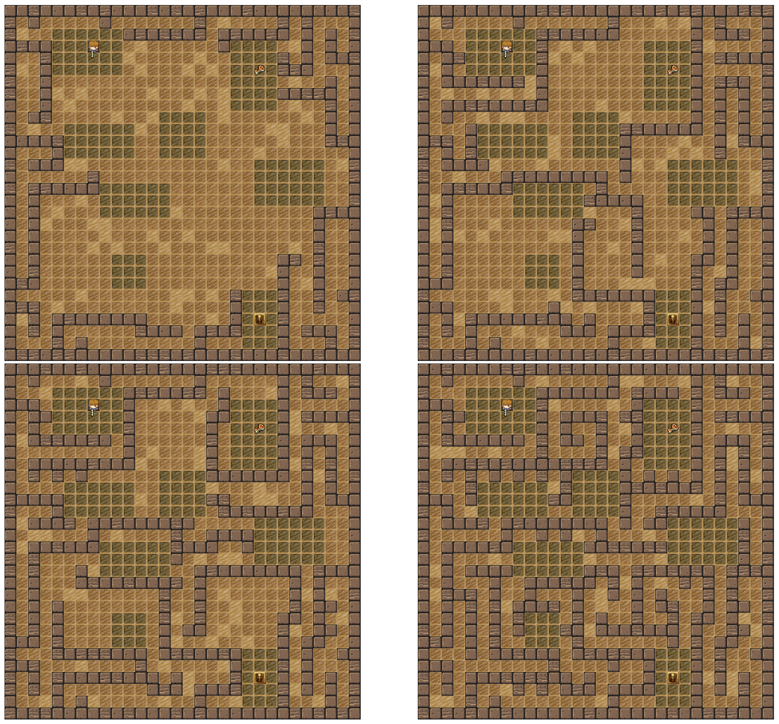


Figure 35: Snapshots from different iterations of the wall-creating agent's process.

Launching locked-door-creating agents

Even though they are both based on the same idea, these agents are very different from the wall-creating ones. A single agent is launched from the goal, using pathfinding algorithms to find its way back to the entrance of the level. As soon as it finds a suitable place (two walls on either side, but none behind or in front of it), it has a chance to place a locked door. When this happens, a smaller utility agent holding a key is launched from that spot, on the entrance side of the door. This agent in turn tries to run as far away as possible from the locked door (up to an upper limit based on level size, to not complicate things too much), and then places the key before terminating. The result can be seen in Figure 36.

The level is in this manner populated with locked doors and keys somewhere on the player side of the door. This means that for the wall-creating agents, there does not exist a scenario where the player cannot reach a key to unlock a door.

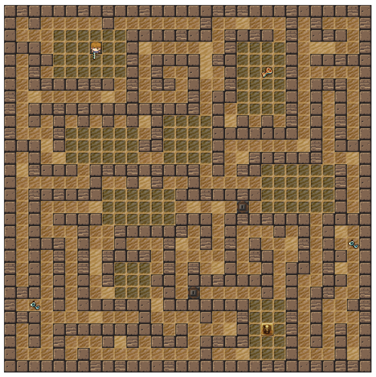


Figure 36: An example of doors generated by the door-creating agents.

Populating rooms and corridors

When populating rooms with content, it is done in two passes. First, valuable things like treasure, health or weapons are generated to make a room interesting. Secondly, the value of these items are evaluated, and according to a specific scale, a fitting number of enemies are generated to guard the treasure. So the more treasure there is, the more dangerous the room is.

When populating corridors, on the other hand, the process is less complicated. Valuable items and enemies are simply thrown out into the corridors at random, but sparsely enough to make the actual rooms interesting in comparison.

After this step, the level is ready to be played. The result of one generation can be seen in Figure 37.

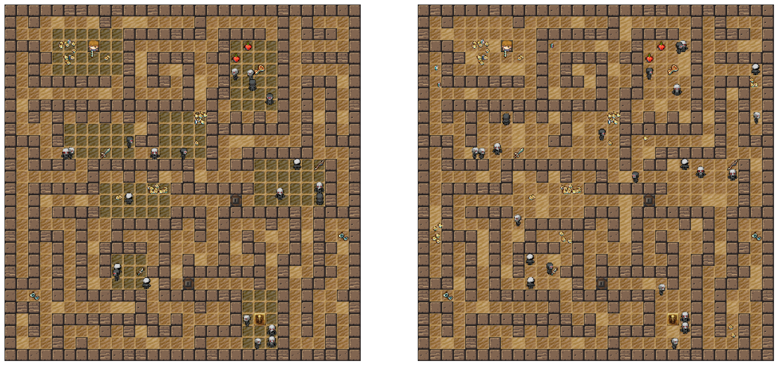


Figure 37: Treasure and enemies have been added to the rooms (left) as well as the corridors (right). In the final version of the agent-based approach, the rooms are not marked by different floor tiles.

5.2.2 Cellular automata approach

Our cellular automata implementation first uses a simple terrain generation algorithm, which makes up the actual cellular automata component. The result of this algorithm is then parsed into a graph representation, which is used to dig corridors and place doors and keys. Finally, items and enemies are randomly placed using probability values. The main steps of the algorithm are listed below:

1. A map is randomly filled with wall and floor tiles, depending on the specified chance for each tile to be a wall tile.

2. The cellular automata step is performed for each tile, for a number of iterations. In every iteration, the algorithm looks at the tile's neighborhood, defined in our implementation as the tile itself and its eight immediate neighbors. Based on the neighborhood and the tile-change settings, the status of the tile in the next generation is then decided. Results of these tile changes are saved in a new map, so that each iteration is based only on the previous map. This step concludes the cellular automata step of our implementation.

3. A graph representation of the tile map is constructed using a *flood fill* algorithm[52]. This means that we choose a random floor tile, and recursively try to move in each direction, adding connected floor tiles to a list of tiles for the current room. A room is for this purpose considered to be any set of tiles which are connected horizontally or vertically. We use a variable to determine the minimum number of tiles which are required for a room to be added to the graph. This determines whether a room will later be connected to the other rooms with a corridor. Each room is represented in the graph by a node. In addition to the room, the node keeps track of its immediate neighbors.

4. Starting with the smallest room in the graph, we try to dig corridors so that there is a path between any two rooms. This is done by finding a set of possible “dig sites” at the tiles at the edges of the room, each with a dig direction of north, east, south or west. Increasingly longer corridors are then tried for each dig site in the appropriate direction until a valid corridor is found. At this point the corridor is created in the map, and a new node representing the corridor is added to the graph, in between the nodes of the rooms it just connected. This is done by updating the neighbor list of each involved node.

5. The graph representation is used to place entrance and exit tiles as well as keys and doors at positions which are sufficiently far from each other and such that the level can be solved. We also consider physical distance, which is useful for cases with very few total nodes.

6. Weapons, collectibles, health items and enemies are each placed throughout the level based on a specified probability per tile. The items and

enemies each have individual probability variables, and there is a master difficulty variable which affects all the probability variables. This placement considers each tile and probability individually, i.e. there is no high-level control of the total number of e.g. enemies per level.

Through some experimentation, we arrived at the following settings:

Wall probability: 38% - 43%
Iterations: 1 - 3
Minimum room size: 10
Tile step: if the sum of wall-tiles in the neighborhood including the middle tile is less than 4: return floor. Else: return wall.

These settings give some variance in the character of the levels, as shown in Figure 38 below; some levels have lots of rooms, thin walls and short corridors while others have few rooms, thick walls and long corridors.

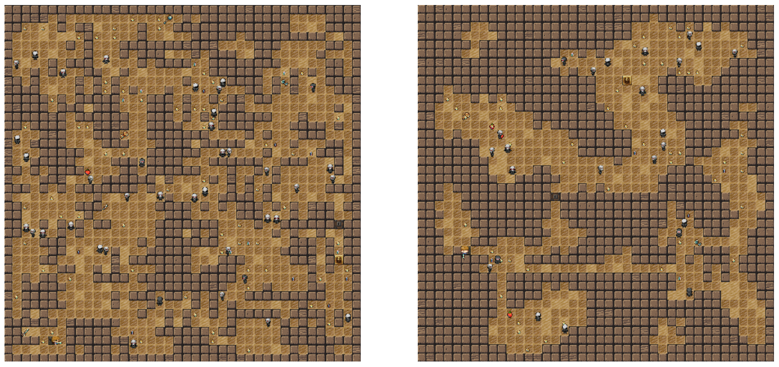


Figure 38: Different settings for wall probability and number of iterations give very different results.

5.2.3 Cyclic approach

The thought behind cyclic PCG is to generate dungeons where the player never encounters a dead end and has to backtrack. This is done by always generating two ways to reach important points. To achieve this, our implementation of the cyclic PCG algorithm generates *paths*. A path is an abstract representation of a sequence of rooms. It has a list of nodes which represent rooms. The nodes are sorted in the list after the order in which they are supposed to be visited. The algorithm starts by generating a main path consisting of 3-6 nodes. It then traverses the list of nodes and for every node there is a chance of creating a new path 2-4 nodes long. This new path is then traversed in the same manner. When the algorithm reaches the last room of the main path the abstract representation of the level is complete (Figure 39).

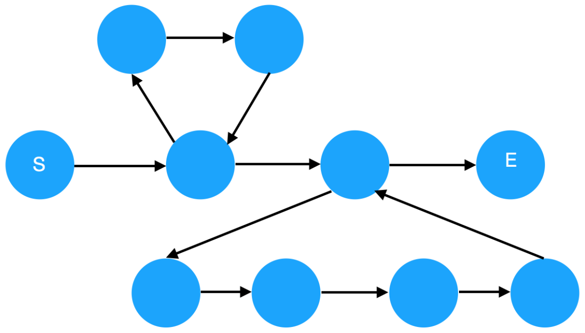


Figure 39: Visual representation of paths generated for a level. Blue circles represent nodes. Start and End nodes are marked with S and E respectively.

After generating all the paths, the algorithm starts generating rooms. It follows the main path and creates a room for each node at slightly random x and y distances to the right of the previous node's room. If there is another path originating from the node it will instead start creating rooms right above or below the room it originated from (Figure 40).

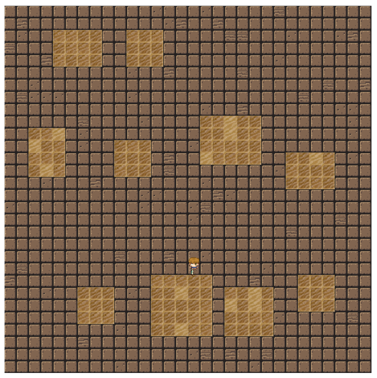


Figure 40: Room layout produced by Cyclic PCG.

After the rooms have been created, the algorithm generates corridors between rooms in the paths, originating from somewhere along the right side of the starting room and ending somewhere along the left side of the end room. A starting point for the corridor is selected randomly along the right side of the starting room and the end point of the corridor will always be in the lower left corner of the end room. First it generates a straight corridor horizontally between the starting point's x coordinate and the end point's x coordinate. Then it generates a vertical corridor at the end point's x coordinate, from the end point's y coordinate to the starting point's y coordinate. When a branching path is discovered, a special corridor is generated to it which first creates half of the vertical part, then the horizontal part and after that the rest of the vertical path. This makes a “zig-zag” corridor. Without this special corridor, there is a high probability that the normal corridors would have intersected other rooms, creating shortcuts and ruining the flow of the level. At this point the basic layout of the map is complete (Figure 41).

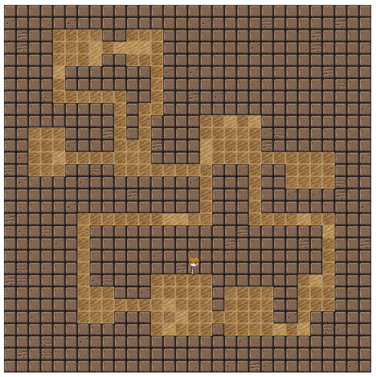


Figure 41: Corridors now connect all the rooms.

Then the algorithm sets a starting point and a goal in the first and last room of the main path so that the player needs to traverse the whole level to finish it. It also generates two doors whenever it encounters a node with a branching path: one door that blocks further progression along the main path and one door in the corridor leading to the last room in the branched path so that player can’t go directly there. Two keys are placed in the last room of the branched path so that the player can open both doors (Figure 42).

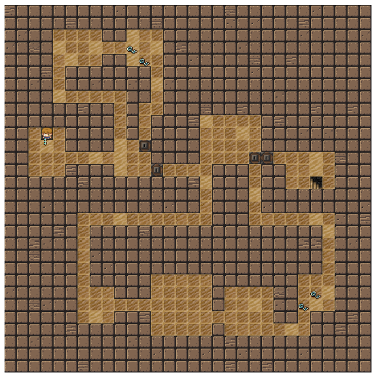


Figure 42: Doors, keys, start and goal have now been generated.

Lastly the algorithm generates all of the game objects (Figure 43). Each room has an *importance value*, which is used to determine the likelihood of items and enemies being spawned there. The importance value is decided by how far away from the main path the room has branched, so the main path has importance 0, any path branching from the main path has importance 1 and any path branching from that one has importance 2 and so on. Rooms of higher importance have a higher chance of spawning high tier loot but also a higher chance of spawning more enemies.

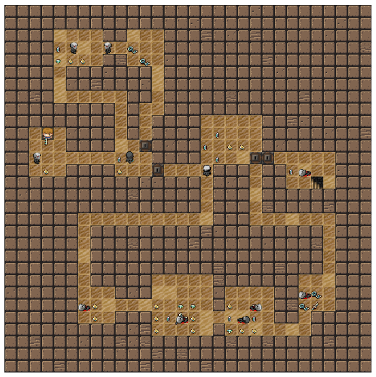


Figure 43: Rooms have now been filled with objects.

A brief summary of how the algorithm works:

1. Generate a main path between the start and goal nodes.
2. Let each node in the main path have a chance to generate a new cycle.
3. Translate the nodes in the paths to rooms on a map.
4. Connect the rooms with corridors, using special corridors when connecting nodes in different paths.
5. Add two doors whenever a branching path is found and place the keys in the last room of the branching path.
6. Fill the map with enemies and items.

5.2.4 Grammar-based approach

The grammar-based approach is named as such because it is based on the concept of formal grammars (Section 3.1.4). The grammar consists of a collection of *terminal* characters and *variable* characters as well as a set of rules, at least one per variable, that determine how that variable can be replaced by other characters. The algorithm starts from some initial string and goes through it character by character. When it encounters a variable character, it replaces it by randomly applying one of that variable's rules. Eventually, the string only consists of terminals. Every terminal character symbolizes some aspect of a level's structure (Table 1), and as such the finished, terminal-only string represents a level. At this point, the string goes through "cleanup" to improve the flow of the level: empty rooms (`[]`) are removed, single keys behind a locked door at the end of a side path (e.g. `(d[k])`) are replaced with an unlocked treasure room and so on.

Terminal	Description
`s`	Level entrance room
`g`	Level goal room
`l`	Locked goal room
`[`	Start of normal room
`]`	End of normal room
`(`	Start of side path
`)`	End of side path
`b`	Goal key
`k`	Normal key
`d`	Locked door
`h`	Item that refills player health
`w`	Weapon
`1`	Patrolling enemy
`2`	Randomly walking enemy
`3`	Enemy that follows player
`^`	Low-value treasure
`+`	Medium-value treasure
`*`	High-value treasure

Table 1: The full list of terminals that are used to represent a level.

Once this step is done, the finished terminal string gets translated into a graph-like structure of nodes representing rooms. These nodes contain references to the preceding node (the *parent*) and any succeeding nodes (the *children*). The algorithm then goes through this structure, and builds each node's room. Starting from the entrance room, the process works as follows:

1. Build the current node's room close to the parent node's room (if the current node is the entrance, it does not matter where it is built).
2. If this node's room is a normal room, fill it with the contents indicated by the terminal characters between `[` and `]`.
3. Build a corridor (in essence just a thin room) between this new room and the parent node's room.
4. Set the current node to the node's child. If there are multiple children, prioritize those that are on side paths. Continue from 1.
   In case the current node does not have any children and the goal room has not been built, the current node is the end of a side path. Then, set

the current node to the node that this side path forked from, and continue from 1.

Thus, the rooms are placed in the sequence specified by the terminal string, which ensures that keys can always be found before their doors etc.

See Figures 44 and 45 for a few examples of levels generated using this approach, along with the strings that specify their layout and contents.

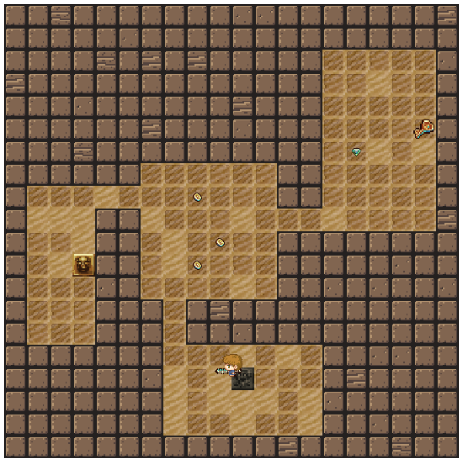


Figure 44: `s[^^^]([b*])l`

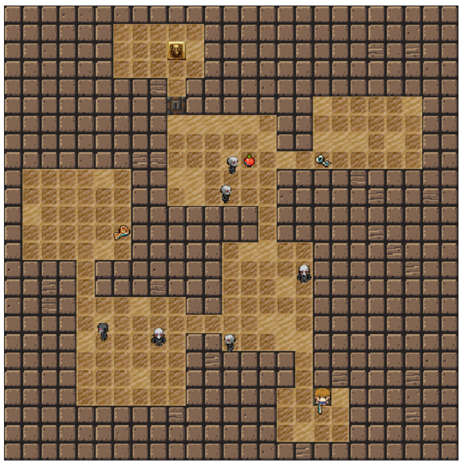


Figure 45: `s[12]([13][b])[22h]([k])dl`

5.2.5 Space partitioning approach

The implementation of the space partitioning algorithm was separated into multiple parts:

1. Generate a space-partitioning tree.
2. Generate rooms.
3. Connecting the rooms.
4. Decide start and goal room.
5. Generate game objects.

Generating a space-partitioning tree

This implementation of the space partitioning algorithm is based on the type of space partitioning known as binary space partitioning (BSP). BSP generates a binary search tree (BST), whose leaf nodes are used to generate rooms, see

Section 4.7.5.

As described in Section 4.7.5, the partitioning depends on desirable properties of the areas. In the case of this approach, the desirable property is simply that the area is of a certain size. The minimum size is constant for all levels, while the maximum size depends on the level's size. This ensures that rooms don't feel too small in larger levels.

Whenever an area's size is within the desirable range, the recursive dividing can continue with a certain probability, as long as the area can be divided into two areas that are larger than the minimum size. See Section 4.7.5 for an explanation.

As an example of how this implementation of space partitioning looks like (Figure 46). The leaf nodes were translated into rooms with adjacent walls, and then combined into a whole level.

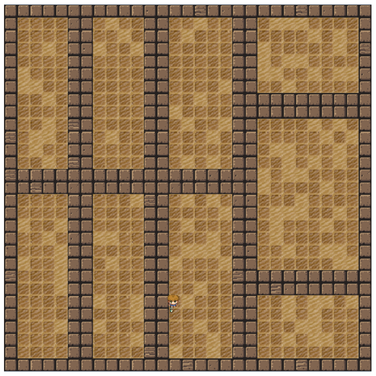


Figure 46: Generating rooms from the leaf nodes in an SP-tree. The adjacent wall tiles are included in the leaf node's area.

Generating rooms

When a BST has been generated from the space partitioning algorithm, the leaves of the tree will be used for generating rooms since the leaves represent disjoint areas. The rooms generated from the leaf nodes will be assigned a width and a height so that the room's size is smaller than its leaf node counterpart, while not being smaller than the minimum room size (compare Figures 46 and 47). The room's width and height have to be smaller than the node's width and height because otherwise there would be no walls for the room, which could result in room collision. In addition, the room's position will be random, while not breaking the node's boundary.

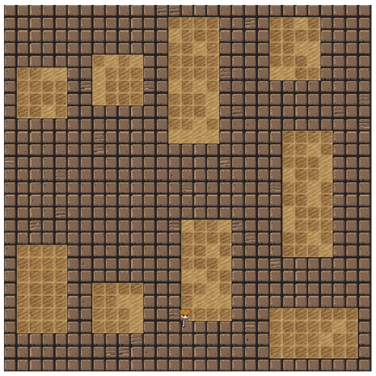


Figure 47: Generated rooms within leaf nodes in an SP-tree.

Connecting the rooms

The rooms are connected through corridors. Corridors are just regular rooms, but with a width or height of one tile. A corridor is created between each pair of children in the BST (Figure 48). The corridors are created as short as possible, either through a straight corridor, or two connected corridors (one vertical and one horizontal). This will result in a level where each room is accessible (Figure 49).

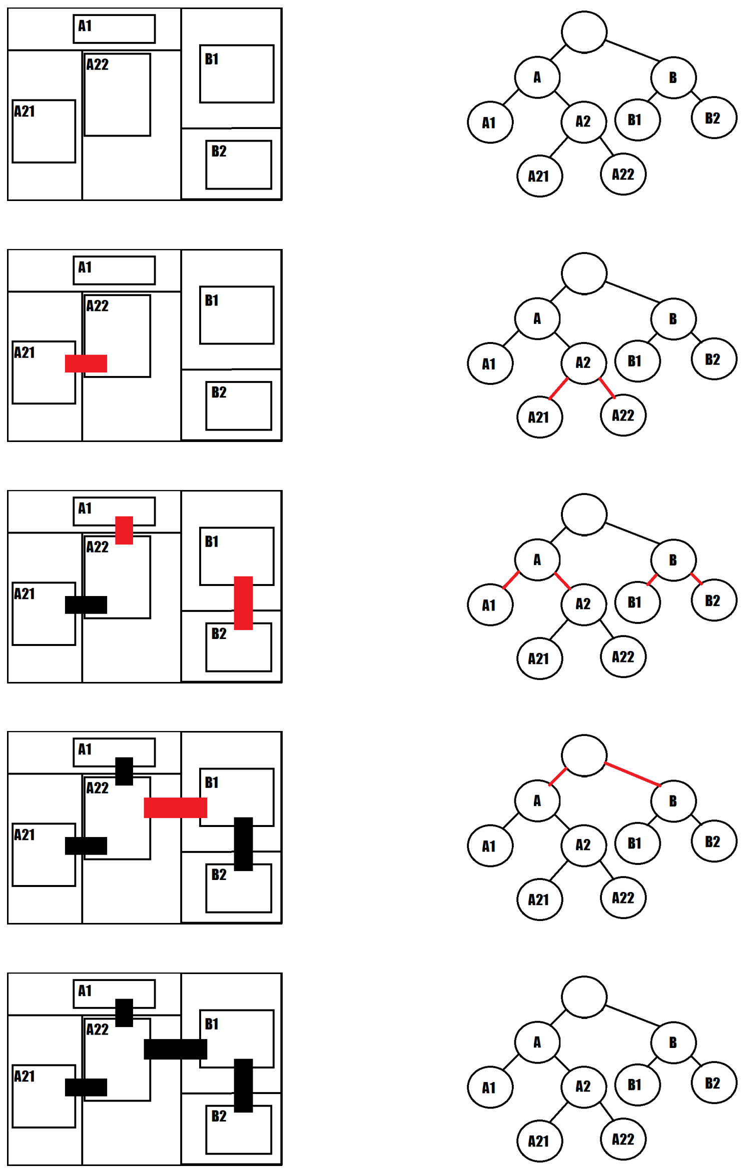


Figure 48: Connecting rooms by connecting in the BST.

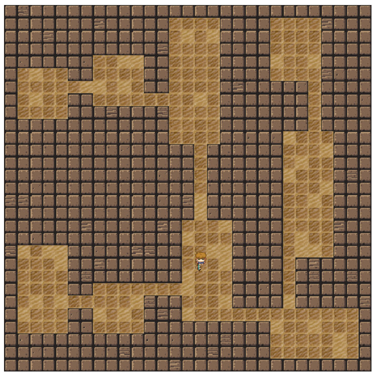


Figure 49: Connecting the rooms in a level.

Deciding start and goal rooms

The start and goal rooms are the pair of rooms with the highest number of rooms between each other. This ensures that most of the level will be explored by the player before they can find the exit. These rooms are selected by counting the number of rooms between every pair of rooms, and picking the pair with the highest number of rooms between them (Figure 50).

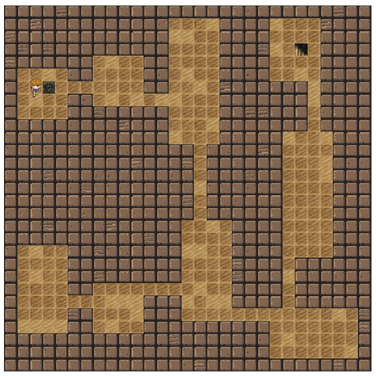


Figure 50: A level with start and goal.

Generating game objects

Generating items, enemies and locked doors is essentially arbitrary when using space partitioning to create levels since the space partitioning algorithm only creates the structure of a level. However, since the goal of each algorithm was to be able to utilize all the different objects in the game, a method of generating game objects in levels generated from space partitioning was created.

A set of rules govern the generation of certain game objects.

Locked goal and goal key: A locked door is generated on top of the regular goal with a set probability. When that happens, a goal key is generated in a random room, with priority given to blind alleys (rooms which do not lead the player anywhere).

Locked door and key: Locked doors are added inside corridors with a set probability. When that is done, a key is added to a room that is accessible without having to pass that corridor, so that the key is not locked behind the door. For a visual explanation, see Figure 51.

Enemies: There is a certain chance that enemies are allowed to be generated in a room. If so, the number of enemies generated will be somewhere between 0 and a maximum value. This maximum depends on the room; it will be higher for rooms with a bigger area, and further increased if there are any items in the room. The generation of an enemy itself is done by randomly deciding the kind of enemy, and possibly adding a random item to it as loot.

Apples: Apples, which restore the player's health, are generated with a certain chance in every room. A room can at most have one apple.

Collectibles: The probability and the amount of collectibles added to a room is dependent on its area. The value of each collectible is completely random.

Weapons: As with apples, only one weapon can be generated per room. The probability of generating a weapon in a room depends on what kind of room it is. The different kinds of rooms are: blind alleys, secondary rooms (rooms that aren't on the shortest path between the start and goal rooms) and rooms on the shortest path. The kind of weapon that is generated and its properties are completely random.

Other: If a blind alley doesn't have any items (excluding collectibles) when all the game objects have been generated, a weapon, an apple or several collectibles are added to the blind alley. This is done because the player does not have an incentive to visit a room that neither leads anywhere nor contains anything of value.

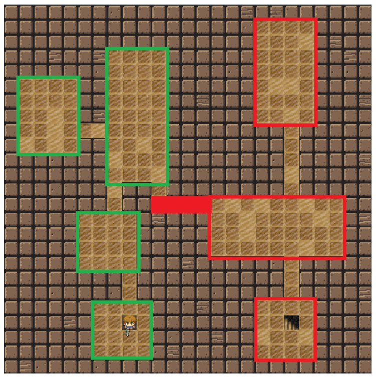


Figure 51: The red filled rectangle represents the corridor where the locked door is placed. The key can be generated in the green rooms, but not in the red rooms.

The probability of an item being generated in a specific room can be increased if the room has a specific theme. Themes are assigned during the generation of the space partitioning tree, see Section 4.7.5 for more. In this implementation, the themes are assigned to each node at a set probability. There are five different themes: Apple, Collectible, Enemy, Weapon and None. The Apple and Weapon themes increase the chance of generating the specified item (apple or weapon). The Collectible and Enemy themes increase the potential number of collectibles/enemies that can be generated in the room. The None theme does nothing.

The result of all these steps is a fully generated level, such as the one seen in Figure 52.

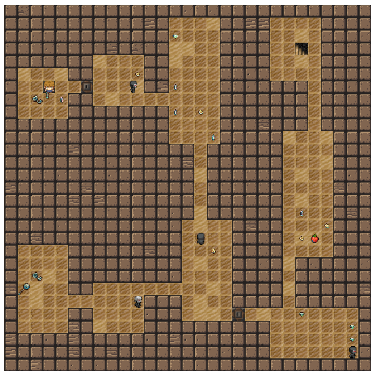


Figure 52: A complete level generated by space partitioning.

5.2.6 TK algorithm

In this approach, the map is modeled as an undirected, unweighted graph where each node is a rectangular room and its neighbors are any other rooms in the graph which it is either directly adjacent to or overlapping with. The algorithm works as follows:

1. A lot of rooms of different sizes are generated. Then the rooms are positioned all over the map in such a way that they do not connect, as shown in Figure 53.
2. Collectibles are generated in all the rooms. Each room can have at most enough collectibles to cover half the room but the amount is random.
3. Enemies are generated in the rooms. Every room can have at most one enemy per row or column of tiles, whichever is smaller. For example: in a 2×6 room there could be as many as two enemies. Each of these potential enemies have a 1/2 chance of actually being generated. The type of enemy is randomly chosen to be either patrolling, following or randomly walking around.
4. To fill in the gaps between the rooms, the rest of the map is filled with rooms of size 1×1.
5. Up to six rooms with an area greater than a fifth of the total area of the map are randomly selected. These rooms are the main rooms of the map and from them two different rooms will contain the start and goal, respectively. The remainder of the rooms are removed from the graph and stored separately. This is shown in Figure 54.

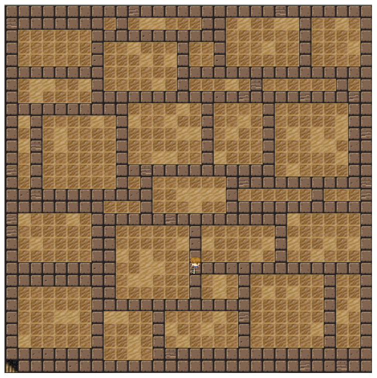


Figure 53: Randomly sized rooms spread out over the map.

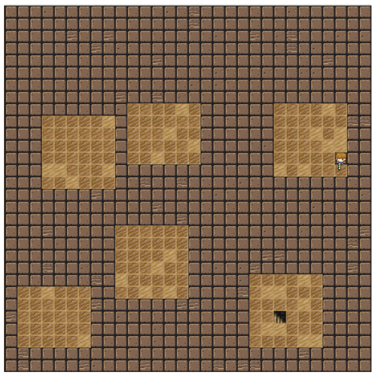


Figure 54: Start and goal are placed in two of the main rooms.

6. One random weapon and up to three apples are placed in random locations in the main rooms.
7. The main rooms are connected to each other by putting back the small rooms between them. This is done by taking one room not containing the starting tile and putting back in small rooms forming a straight line between it and the starting room. Then another room is selected and a path is made between it and the room closest to it from the ones connected to the starting room and so forth until all the rooms are connected as shown in Figure 55.
8. When all the rooms are connected an additional path is added between the starting room and one other random room, to create some diversity in the map as well as make sure that there are always two paths to take when entering the level.
9. The map is finished and the next task at hand is to add in locked doors. The map is searched and every spot considered suitable for a door is saved. Suitable door spots are any spaces with walls on two opposite sides and floors on the remaining opposite sides. Doors are placed in up to three randomly chosen spots from the suitable positions.
10. After the doors are added, keys to unlock them are placed in reachable places. To spread them out in an interesting way, the first key is placed in an area which is reachable from the start of the level, after which one door is marked as unlocked in the model of the map, allowing the next key to be placed behind a locked door. Lastly the goal is locked and a golden key to unlock it is added to the map. The finished map can be seen in Figure 56.

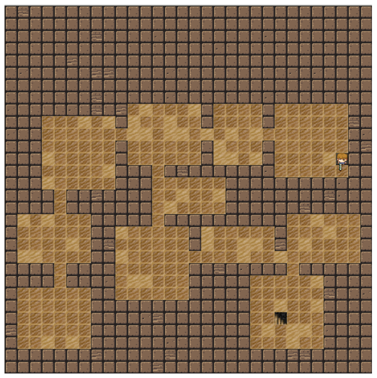


Figure 55: Corridors created from the previously removed rooms are added to connect the main rooms.

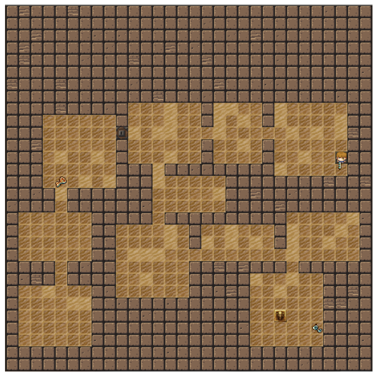


Figure 56: The goal is locked and locked doors, keys for the doors and a key for the goal are added.

5.3 Observations and comparisons

The aim of this project was to explore the process of creating PCG systems for generating game levels, and more importantly, to present our findings so that they may be of use to others. The following section is a compilation of what we have learned by working with, and later comparing, different PCG algorithms. The observations are divided into two main categories: notable features exhibited by the different algorithms when working with them, and how the resulting generated content may be experienced. At the end of this section, we will also summarize what distinguishing characteristics each of the explored algorithms appear to have.

Due to the nature of our evaluation, it is important to note that this section contains subjective views, even though we have strived to anchor these views in experiences, observations and previous studies.

5.3.1 Working with the algorithms

In this section we talk about what observations we have made about the process of working with the algorithms during development. This covers what one can expect when implementing different PCG systems, what can be challenging and what to avoid.

Different algorithms cover very different areas in their generation. For example, the grammar-based approach happens entirely in a high abstraction level, and can later be translated to a map in many different ways that have nothing to do with the actual generation. On the other hand, cellular automata needs to use a map directly to be able to work. This leads us to conclude that when working with PCG, it is important to be mindful of the adaptability or rigidity of each considered algorithm, as this could heavily impact the upcoming work.

Algorithms that are tightly connected to low levels of the software structure can be powerful, but hard to adapt to unforeseen turns in the development. While the cellular automata algorithm was very easy to implement and gave great results, it would probably have needed to be rebuilt if for example we decided to add rivers and bridges into *Fluky*. Algorithms at very high abstraction levels, however, may need considerable work to fit inside a specific game. Our experience of both the grammar and cyclic approach showed us that most of the work with these went into the translation from the abstract model to the concrete game levels.

While there exist many well-documented theories and techniques when working with PCG, it is fully possible to develop completely new PCG strategies or algorithms. However, it is important to consider the extra work that might be required due to the lack of well known and documented practices. As we worked simultaneously on the different algorithms, a recurring issue with the custom made TK-algorithm was that when problems occurred, there were no easily accessible resources to use. This resulted in the work on that specific algorithm progressing slower and being more tedious compared to the other algorithms.

Combination possibilities

Differences in both complexity and level of abstraction affect how easily various PCG algorithms can be combined with each other. Algorithms that operate on a high abstraction level are hard to use inside or even next to each other, but instead lend themselves well to enclosing other, more low-level algorithms. Conversely, low-level PCG algorithms are much easier to integrate into larger or more complex algorithms, than the other way around.

For example, the agents in the agent-based approach that we implemented could easily be used inside a single sufficiently large room in any of the space partitioning, grammar-based, cyclic or TK algorithms (Figure 57). This is because the latter all work with rooms as their smallest building blocks, while the agent-based approach instead works with single map tiles. It would be difficult, however, to use our cellular automata algorithm inside the agent-based approach, since both of these algorithms work with individual map tiles.

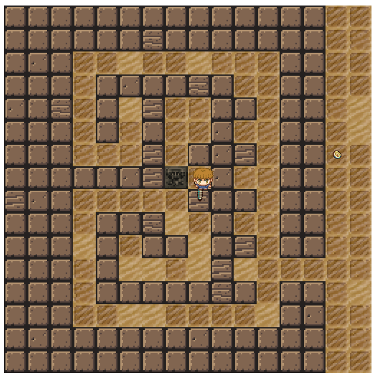


Figure 57: The agent-based PCG used inside a room from the grammar-based approach.

Even though it is not one of our implemented algorithms, the search-based approach operates on even higher abstraction levels than those we have explored in this project. This makes it possible in theory to use each of our algorithms within the confines of a search-based one, further demonstrating the above conclusions (as described in [17]).

One should be careful to consider the above observations about combining algorithms an absolute rule, however. Exceptions exist and can have tremendous potential when exploited. As an example, with some adjustment we were able to incorporate the space partitioning approach into the agent-based approach, using it to set up some parts of the agent-based map. To use several large and complex algorithms simultaneously can produce great results, provided that it is possible to combine them. The combination of different PCG algorithms may in truth even be instrumental when striving to create content of good quality[17].

Adjusting and adapting algorithms

When changing the base conditions of a game, such as adding new game mechanics, PCG algorithms will most likely need to be adapted along with these changes. It is possible to avoid but only with great difficulty, according to Kerssemakers[53]. It would therefore perhaps be more interesting to evaluate how much PCG algorithms can be adjusted within their current implementations.

The easiest way of adjustment is to make constant values variable. When developing any algorithm, constant values will be decided to get a desired behavior. If one instead could determine a range of working values to randomly choose between, the variation of the algorithm could be hugely extended, and hopefully for the better. If variation is not the goal but different behavior is, other values could simply be chosen in the determined span, instead of randomizing them. Such variations may introduce problems or bugs that need to be taken care of, but if done carefully, this has the potential to enhance the result of PCG algorithms.

The number of variable values will differ between algorithms. The cellular automata algorithm we explored showed great potential for adjustment, as changes in its parameters gave very different results. The grammar-based approach also has great inherent possibilities, since the grammar it uses can be changed fairly easy with considerable impact. The space partitioning algorithm, however, can be adjusted but with little visible impact, as the levels always will have a similar look.

A much more complicated but potentially powerful way of adjustment is to extend the algorithm to enlarge the span of variations to choose from. These changes will be much more specific to each algorithm, making it harder to create significant ones.

An example of the above is to create an entirely new agent in the agent based approach, that builds walls with very different rules, thus changing the behavior of the algorithm completely (Figure 58). In our experience of the agent-based approach, it did not have much potential for adjustment within the given mechanics, but does on the other hand offer great possibilities in agent creation. When looking for potential ways to adjust algorithms we therefore suggest that it is important to try many different approaches on many different levels. What works for one algorithm, may unfortunately be useless for another.

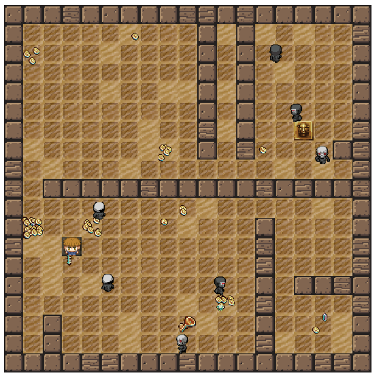


Figure 58: By simply removing the wall-creating agents' ability to turn, the agent-based levels will look very different.

5.3.2 Quality of the generated content

The concept of quality in PCG is subjective and hard to measure. In this section, we present our observations about the resulting generated content. We do this to figure out what to aim for when developing PCG, what to keep an eye out for, and maybe offer a couple of valuable insights about the process.

Quality in PCG can mean many things; there are both technical and emotional parameters to take into account. An example of a technical aspect is how well an algorithm scales when the size of the wanted content grows to large proportions. All of the explored algorithms behaved in undesired manners in these cases, although this might be due to flaws in our implementations. Some of the unwanted behaviors were very prominent, while others did not show as much. However, the fact that every algorithm showed them tells us that it is important to take this into account when developing PCG: make sure it works fine for all intended sizes.

A very abstract aspect of PCG quality, on the other hand, is how "handmade" a procedurally generated dungeon may feel. This relates a lot to the concept of variety (discussed later), but has just as much to do with the architecture and layout of a level. The handmade feel usually comes from the feeling that whoever built the level had a plan, giving it a thought-out sequence of events to be experienced.

Naturally, this handmade quality is very hard to accomplish. An attempt to do this was made with the cyclic approach in the game *Unexplored*, creating a large number of well thought-out possible cycles to extend the map with. The result was considered a success. On the other hand, when we tried the cyclic approach ourselves, the variation of the levels was so low that it became very obvious that they were not handmade. This leads us to believe that a specific algorithm is not the answer in itself, but rather that large amounts of energy and thought need to be spent to be able to evoke the illusion of handmade PCG levels. Of course, the cyclic PCG approach may very well be much more suited to this than other algorithms, but we conclude that there is no such thing as shortcuts to a handmade feel.

Variety

It is important that the levels our PCG algorithms generate feel varied enough, so that they are able to keep a player's interest for any meaningful amount of time. While *Fluky* achieves noticeable level-to-level variety by randomly selecting an algorithm every time a new level is created, the variety between a single algorithm's levels is usually not as high.

Among our algorithms, it appears that the ones with a very clearly defined layout characteristic – the cyclic and agent-based levels – are the ones with the most similarity between levels, by virtue of this very characteristic. While the mazes generated by the agents are technically different in every level, they still end up providing a near-identical experience. The room cycles that make up the core of the cyclic algorithm also feel very similar from level to level, despite being built in different locations and with different frequency every time (Figure 59).

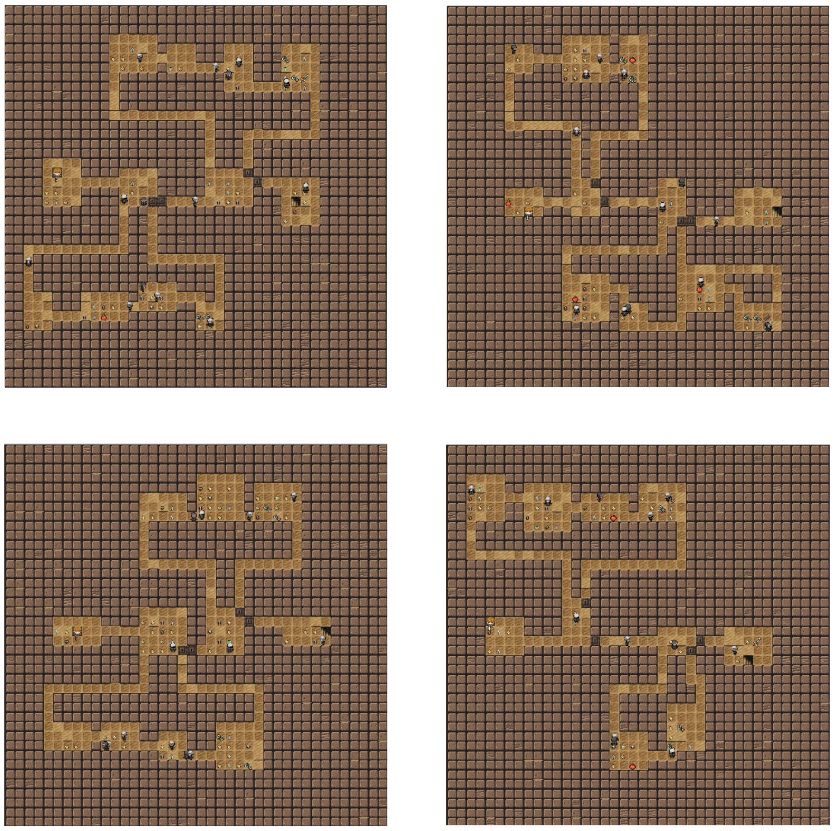


Figure 59: Four different levels of the same size generated by the cyclic approach have very similar results.

Conversely, the other four algorithms, whose unique characteristics had more to do with how the level was built rather than what the player would experience, were perceived as having a higher degree of variety. Especially noteworthy here is the cellular automata-based algorithm, which did not use the standard room-and-corridor layout of the other algorithms, instead focusing on individual tiles. This meant that its levels had very few limitations placed upon them, and as a result could take on nearly any structure.

While not as varied as the cellular automata-based levels, the other three each manage to achieve a certain degree of variety despite their more traditional room-and-corridor layout. The grammar-based algorithm uses its “vocabulary” of small-scale room structures to gradually build a full level independent of its layout, and the TK and space partitioning algorithms are able to create varied level layouts by simply building the rooms first and adding in the corridors afterward.

Difficulty

In this context, the “difficulty” of a level refers to how challenging that level is to complete for a player. Ideally, a PCG algorithm should be able to generate levels that manage to challenge the player without ever feeling too hard. Unfortunately, difficulty was not a primary concern for any of us while we implemented our algorithms for *Fluky*. As such, difficulty is not very consistent between the different algorithms’ levels. After testing the finished algorithms and discussing our findings, we are of the opinion that in *Fluky*, the difficulty of a level is primarily determined by two factors: *exploration* and *combat*.

Exploration-related difficulty

Since the goal of every level is to arrive at the exit, the overall difficulty of a level is naturally affected by how challenging this exit is to find. Our algorithms differ considerably in this regard. It is important to note that although exploration is an especially important factor in *Fluky*’s difficulty, that does not mean that it is irrelevant for games with other win conditions than finding an exit. For example, a game where the objective of every level is to defeat all enemies will be considered hard if finding the enemies is a challenge in itself.

The cyclic algorithm is the only one of the explored ones that generates truly linear levels, due to its “no backtracking” design philosophy which opens up new paths to the player one at a time (Section 3.1.3). This means that even for larger levels, navigation is often a trivial matter for any player. On the opposite end of the spectrum we find the open-ended cellular automata levels and the maze-like agent-based levels. In the former, the irregular and unstructured level layout can make it hard for the player to orient themself. In the latter, the maze sections make it difficult to determine which of the many paths will lead to the goal, even if one knows where it is. While this is the intended purpose of a maze, those built in larger scale tend to become so complex that they are often nigh-impossible to navigate successfully (Figure 60).

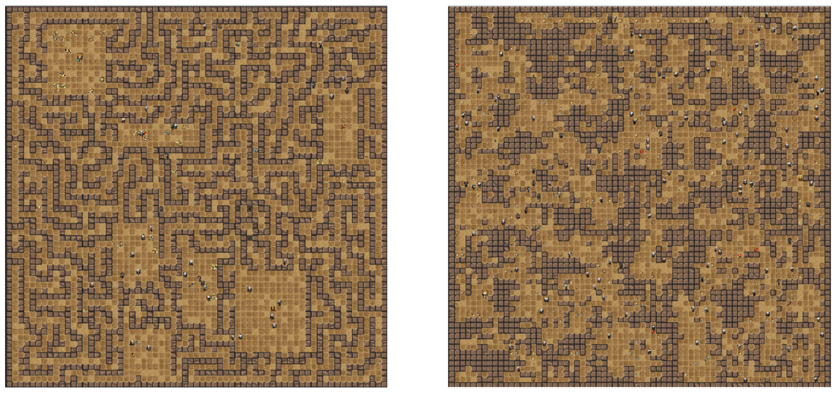


Figure 60: Very large maps generated by the agent- and cellular automata-based approaches.

Between the two extremes of trivial and challenging exploration are the other three algorithms we implemented. The levels generated by the TK, grammar-based and space partitioning algorithms all have a similar structure of rooms connected by corridors, with occasional branching paths. The random nature of the TK algorithm's corridor placement means that it is possible to generate levels with more than one path to the goal, which could serve to reduce the amount of backtracking a player has to do. Furthermore, the TK algorithm inherently packs rooms as tightly as possible, which means that the player never has to cross great distances to get from one end of the level to the other.

In contrast, the space-partitioned levels are laid out in such a fashion that the entrance and exit are always as many rooms apart as possible, which never makes it trivially easy to find the exit. Such a property also means that factors such as level size and room amount can have considerable influence on exploration-related difficulty.

Compared to the previous two, the grammar-based levels are significantly less consistent in terms of exploration difficulty. These levels range from ones that simply consist of only an entrance and an exit room, to a sprawling arrangement of branching paths upon branching paths where it is easy to get lost (Figure 61). This makes them a poor choice for situations where one might desire a sequence of levels which follow some manner of difficulty progression.

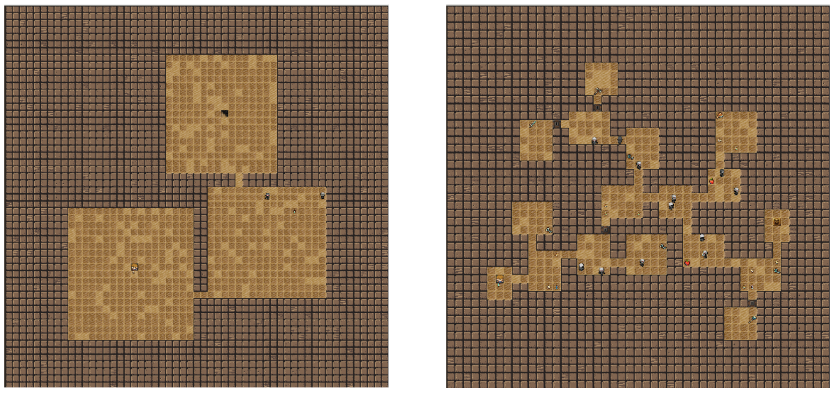


Figure 61: Two levels generated by the grammar-based approach with very different results.

Combat-related difficulty

Since enemies are the only thing that can put a permanent stop to the player's progress, their placement has a considerable impact on a level's difficulty. All algorithms except for the grammar-based one generate enemies in a somewhat similar manner by placing them after the level's layout is finished. This approach lets them have the number of enemies depend on specific elements of the level. Of particular note here are the cyclic and agent-based algorithms, which both try to balance enemy frequency against the value of the treasures that are found in rooms, leading to a correlation between risk and reward.

Since the cellular automata-based levels do not contain rooms in the same way that the other level types do, the algorithm cannot easily take the level structure into account when placing enemies. This can lead to narrow passages filled with a large number of enemies, which can be tough to navigate for the player. The grammar-based levels cannot employ the strategies used by the other levels since rooms are designated to contain enemies before they are built. While this can lead to oddities such as rooms having too few enemies for their size, it also opens up possibilities for letting enemy rooms stand in the way of important rooms further down the path, which can give levels a more "hand-crafted" feel.

Adding to the complexity of combat-related difficulty is the concept of weapons. In *Fluky*, the player character needs to carry a weapon to be able to attack enemies, and these weapons wear down and break with repeated use. This means that if the player is unable to find new weapons on a regular basis, even enemy encounters that are few and far between will eventually prove lethal. Most of our algorithms simply generate weapons randomly, with no guarantee that one shows up in a level at all. The only algorithm that handles weapons in a different manner is TK, where every level is guaranteed to have exactly one weapon.

Enjoyment of the generated content

To analyze and determine the enjoyment of content is basically like measuring "fun", which is difficult, intangible and highly subjective. Consequently, during the analysis of the generated content, only the factors that were deemed sufficiently concrete were included.

The enjoyment is connected to the other aspects already analyzed: variation and difficulty. Generating levels with low variety decreases the replayability of the game and cause the player to get bored faster. A level which is too difficult makes the player frustrated, but if the level is instead too easy, the player will not find the game challenging enough and therefore considers it to be boring. As those factors have already been analyzed in the previous sections, they will not be further discussed in this section.

One important factor in determining if a game or level is enjoyable is if it is engaging. If a game is not, the player could feel bored and unmotivated. It is difficult to procedurally generate engaging content consistently, since it is a complex task to do even manually. There are multiple things that can make a game engaging. One aspect discovered was how close the rooms are to each other. For example, the grammar and TK algorithms generate rooms with little space between them, which increases the pace of the gameplay and thereby keeps the player engaged. The space partitioning PCG, on the other hand, generates levels with a greater distance between the rooms, making the pace generally slower. This causes some uneventful and boring moments. Another thing we found to increase the engagement was the presence of enemies in all parts of the level, like the levels generated by the cyclic PCG has.

Backtracking – when the player is forced to retrace their steps – is an aspect relevant to most of the implemented PCGs because *Fluky* contained locked doors and keys. This can introduce the element of having to return to earlier parts of the level to try to find a key, to in turn be able to unlock a door and continue. This is in some cases a good thing, and even important in puzzles, but can also feel boring and like a waste of time if it is overused. The agent-based PCG generates levels in a way that makes backtracking a vital part of the level, and often necessary to be able complete a level. This is fun on smaller levels as the backtracking does not take so much time, but on larger levels it can take a very long time. The levels generated by the cyclic PCG are completely different, never forcing the player to backtrack. While this eliminates the potentially long and boring backtracking, it makes the levels linear and removes the element of free exploration. The other four PCGs are somewhat similar to each other in this regard, which is that backtracking is sometimes necessary. However, the space partitioning and the cellular automata PCGs generally generate larger levels, increasing the risk of a level having tedious backtracking.

Another important aspect is to have levels that are interesting and fun to explore. This is somewhat coupled with variety since if the variety of the content is bad, there is not much to explore. Another important aspect of exploration is to encounter and find new interesting things, which none of the explored PCGs can provide due to the small number of game objects and tiles in *Fluky*. However, the levels created by the cellular automata algorithm has a unique structure that makes it interesting to explore. However, exploring these levels can sometimes be time-consuming, because it can be difficult to navigate through the level due to their unpredictability and large size. The levels generated from the agent based algorithm have a similar structure to cellular automata, but are smaller and more narrow, thus limiting the space needed for proper exploration.

Among the algorithms we explored, the four not mentioned above generate levels with quite uninteresting structures, since all of them are based on generating rectangular rooms in some way. However, out of those four, the space partitioning PCG generates levels that are somewhat explorable. This is due to the fact that it creates large levels with a lot of rooms, and that it always generates items in the blind alleys, which encourages the player to explore more. Also, the TK algorithm generates levels where the player is often able to pick several paths. This makes it easier and more engaging to explore since the player is faced with multiple alternatives. But the exploration is still limited by the small sizes of the levels.

5.3.3 Summary of each algorithm

We have learned a lot from the exploration of different PCG algorithms, as described in previous sections. Our insights about each algorithm are therefore summarized in the following subsections, to make it easily accessible.

Agent-based

The agent-based approach is seldom a complete solution. It may produce all of the content, or only small parts of it, entirely depending on its agents' implementation. We have learned that it is a very flexible approach, as the customization of the agents is simple yet has much potential. It can also be used as part of other larger, more complex algorithms: either as a means to that algorithm's end, or to fill in blind spots that the other algorithm cannot solve. See Section 5.3.1 for examples.

This approach is very dependent on its agents, and is therefore difficult to evaluate as a whole. But if one is careless, the resulting agents have a tendency to be very rigid and hard to tweak, or use in other situations. Our implementation showed a lack of variation and had scalability issues, so these problems should be kept in mind when using the agent-based approach, since they could impact the quality of the result negatively.

An interesting example of a possible area of use is to have agents simulate player behavior, which might be a powerful tool when verifying playability. However, these verified levels would still need to be checked for gameplay value, as the levels can be solvable while still being tedious or even boring.

Nevertheless, agent-based PCG is a powerful approach with great potential, that can allow for many different agents to work together towards a shared goal, or even in concert to iterate towards complex solutions that would be hard to attain in other ways.

Cellular Automata

Cellular automata can be useful in situations where an unstructured and natural-looking element is desired. However, its low-level nature results in a lack of structure, which can require a lot of work to parse into some kind of high-level representation in order to gain more control over the level design, if this is necessary.

Areas where this lack of control is not a problem could be more esthetic elements, like grass or any other element where the result should look unstructured and natural, but where there is not a hard requirement for validation since the result will not affect the playability of the level.

This algorithm requires a lot of trial and error to find appropriate settings – the results of these settings can be hard to predict, and a difference in the number of iterations can make the end results very different.

Cyclic

Cyclic PCG creates levels with high tempo as the player never has to backtrack, always encountering new enemies and puzzles. It is an algorithm that needs a lot of variation to avoid feeling repetitive and predictable, like our implementation does. This algorithm is therefore best implemented in a rather substantial game with lots of features that can be used to increase the variety of cycles. However, adding lots of different cycles would naturally increase the time it takes to develop the algorithm.

A good thing about cyclic PCG is that it can easily represent a level as a graph before all the heavy work of translating it to a map has begun. This way it is easy to check the graph for major flaws or bad level design, and correct it before too much time has been spent. The developers also need to build a good system for placing rooms along the cycles that will not overlap with each other, as well as smart corridor algorithms, as it can be hard to place rooms in a well-structured manner when there can be cycles upon cycles of rooms in the map.

The algorithm seems to handle scaling pretty well. In larger maps, you can simply allow the algorithm to generate larger cycles and add more cycles that originate from these, but then again you need to watch out so that rooms and corridors do not start to overlap.

Grammar-based

Grammar-based level generation is well-suited for situations where one wants to build a level in terms of smaller-scale structures, with little or no regard to the level as a whole. One of the primary challenges that needs to be overcome for such an algorithm to work is the matter of constructing a suitable model that is both able to properly represent a level and be used with a grammar system. Provided a good model is used, it might even be possible to decouple the level's layout from the intended purposes of its components, as seen with our implementation. When taken to its extreme, such a decoupling could enable a grammar that is only expressed in terms of the player's actions, which enables a lot of interesting design possibilities.

The downsides of the grammar-based approach are the aforementioned lack of control over the level as a whole, and the unpredictable manner in which it applies its rules. These factors mean that ultimately, the algorithm's creator has very little control over how a finished level will look, as seen in our approach. These could be diminished by allowing different sets of rules to operate in different phases of the level generation process, or by letting certain rules have an increased probability of being applied.

Space partitioning

The space partitioning approach is easy to grasp and generates consistent content. The algorithm itself is straightforward and does not have a lot of different approaches and parameters that can be adjusted. This makes it easy to recreate the space partitioning algorithm and get similar content. However, the content generated by the algorithm is rather repetitive and predictable, since it consistently generates rooms that look like each other.

Another problem is that space partitioning only provides a level with connected disjoint rooms, which in itself is not enough for an enjoyable game. It would probably be able to generate better and more diverse content if it were to be combined with another PCG algorithm. The other algorithm might in that case take the level generated by the space partitioning algorithm and make it more interesting, unique and complete. For example some agent from the agent based PCG could traverse the level and generate items. Another example idea is to apply cellular automata to make the structure and shape of the rooms more unique and natural.

TK algorithm

This approach creates a closely connected map, with many rooms and short corridors. This in combination with the fact that there are often many ways to reach different parts of the map keeps the tempo in the level high. In the way our implementation works, it does not scale very well, as the rooms get too large and the variety of content is too small. It is however possible to tweak it in a number of ways to remedy this. The algorithm could in fact easily be modified to create quite different maps. For example changing the size of the rooms, the distance between them or the number of main rooms, could give the map a very different look. This modular design allows it to easily be combined with other algorithms at different stages. You could for example take a map of unconnected rooms and use this algorithm to generate corridors between them, creating a more interesting map than regular corridors perhaps would.

The algorithm could work well for a variety of games. However, the settings have to be tailored to the game and the size of the level desired. One of the major drawbacks with this approach, as it is quite unique and not widely used, is the lack of resources available to the developer.

6 Discussion

In this section we discuss our results and what they entail, as well as the validity of them and how they might have varied had we done things differently. Furthermore, we bring up some of the issues we had along the way and how they affected the project. We also consider the effects of our work and what can be done to further explore this topic in the future.

6.1 Result

As we had limited time to develop the game and the different algorithms we tested, the comparisons are based upon rather basic implementations of both the game and the algorithms. This may have affected the results slightly as some of the algorithms would have benefited from more development time and a more content rich game to really show their full potential and to get rid of some of their problems. The cyclic PCG is an example of an algorithm which with more development time could have implemented more kinds of cycles, solving the problem of it being very predictable. With that said, we still feel like we managed to develop all parts to a satisfactory level where we were able to get a general feel for how the different algorithms differ and what their general strengths and weaknesses are.

There is also the fact that our results from the comparison of the different algorithms are based on subjective thoughts which others may disagree with, since it is difficult to evaluate the quality of content in an objective way. Based upon the fact that we are six people and all of us agreed on the results we got when comparing the algorithms, we feel confident that the results are valuable to others.

6.2 Method

Before we started development of *Fluky*, we considered another approach for exploring PCG: developing one or several PCG algorithms for an existing game. This would have allowed us to focus solely on the exploration of PCG, and as such may have given us a deeper understanding of the field. However, such an approach also carried with it certain drawbacks. For one, we would need to spend time finding an appropriate game and familiarizing ourselves with its codebase. It would be very difficult to accurately predict how much time we would need for this step when making our time plan. Secondly, our PCG algorithms would have to take the game's features and mechanics into account when generating levels, which could have made them significantly more complex than we had time or need for. These drawbacks were what made us settle for building a game of our own design, where we had full control over all systems and mechanics.

Our decision to begin implementing six different PCG algorithms halfway into the implementation phase had a significant impact on the way our work progressed, and was the only notable departure from our initial plan. Ultimately, we feel that this decision helped us gain a broader understanding of PCG. However, there is a non-negligible possibility that our original plan – to work on the game and a PCG algorithm in parallel – could have given us some other insight that now eluded us, since it would have hopefully let us create a more complex game.

Throughout the implementation phase, we stayed true to our initial decision to work in an agile manner. This felt like a very natural way to work, considering that we implemented new features one by one on a priority basis. Even when we started working on the PCG algorithms, the agile workflow of implementing and testing parts of the algorithms piece by piece felt like the natural way to structure our work.

Another aspect of our work that remained true throughout the implementation phase is that we usually programmed independently. This way of working granted us a lot of flexibility and made it possible to implement many different components and algorithms in parallel. However, it is possible that working on tasks in groups or pairs could have let us solve certain problems faster.

Even though we programmed independently, our work was still very much of a collaborative nature. We held meetings regularly and frequently, and communicated via *Slack* and various other collaboration tools. We are certain that the ever-present focus on communication was crucial for our success.

6.3 Validity and generalization

The result of the development of our game *Fluky* and the included six PCG algorithms is quite easy to measure objectively: we have clearly created a working game and six different PCG systems for that game. We are also confident that we have learned a lot from our exploration of PCG. The resulting knowledge, on the other hand, is harder to verify. As mentioned in Section 6.1, most of our evaluation results are very subjective. We have strived to base our conclusions on sound argumentation, references and experience. Yet the results are still far from facts due to the subjectivity.

Our project has been an exploration of PCG with the goal to benefit game developers, students, and researchers that want to explore similar areas. Whether our work is beneficial to these groups or not is therefore the most important question when considering the validity of our results.

We deem our resulting comparison to be very useful, primarily to those that are interested in dungeon generation, or want to use some of the approaches we have explored. To others that are looking for general insights about PCG, there might still be useful things to be found. However, the level of generalization in this case is very high, and our work should therefore not remain the only, or even the primary source of knowledge, but rather more of an additional reference.

6.4 Ethical and societal aspects

Early on in the project, we considered the effects that our work, as well as PCG in general, could have on society. We came up with a few possible effects. Firstly, that PCG could potentially put level designers out or work, and secondly, that it could allow indie game developers to create more content with less resources. Furthermore we considered if there could be a connection between PCG and video game addiction. However, we concluded that this is a far larger issue where PCG is only one small factor. For this reason, we did not investigate this issue any further.

It can be argued that PCG could lessen the need for content designers, causing unemployment in the field. As content is generated automatically, far less time is needed to create vasts amounts of it. However, it allows smaller companies with fewer resources to easier get into the game industry. By this reasoning, PCG would rather create opportunities.

Considering our project in particular, the risk of somebody losing their job on account of our work is low. This is because our work has been exploratory in its nature. We have not tried to create any new tools or improve PCG to take it to a level where it could replace a designer. What we have done is try to map some of the properties of different types of PCG. As we discovered during this project, trying to figure out what algorithm will work well or provide a specific result is no small task. We therefore believe that someone in the early stages of developing a game could benefit from this report when implementing their own PCG.

6.5 Future work

There are several aspects of our project that could easily be extended. There are also many possible projects that could continue what we have started in this project.

First off, the game itself is very rudimentary and it could therefore be interesting to expand it with at least most of the features outlined in the Design Document, if not even more. This would result in many more mechanisms to experiment on with the different PCG algorithms, allowing a deeper exploration of the PCG concepts.

Secondly, we have only scratched the surface of the vast subject that is PCG. We think that to explore even more algorithms, as well as explore them more thoroughly, would really benefit our research. More algorithms could add greater insights and multifaceted views of how the concept of PCG works. Longer work on each algorithm might also allow us to really figure out exactly what works and what does not, try different approaches to the same algorithm, compare additional aspects between algorithms, and other interesting venues.

Finally, our evaluation and comparison of the algorithms have followed a straightforward arrangement of discussing our experiences and then summarizing the result. Additionally, one could perform user tests on the different algorithms and gather feedback from the users. Another approach could be to do a much more detailed study of each algorithm, that possibly could give us different or additional realizations.

Taking a step away from this project and looking at the research area in general, a similar study to ours but on a three-dimensional game instead would be really interesting to see. Perhaps even with the exact same PCG algorithms, to allow for useful comparisons between the projects. A completely different approach could also be to start a project around trying to develop a completely new PCG algorithm, that is sufficiently generic to benefit others.

7 Conclusion

Procedural content generation is a powerful tool used frequently in game development to randomly generate content that otherwise would take too long to create or take up too much storage space. A significant problem with PCG, however, is that it is very hard to ensure quality, or even to clearly define high-quality content. This is because both the PCG system and the definition of its quality depend largely on the game for which it is developed.

This project has therefore been about the exploration of PCG, through the gathering of knowledge about different kinds of PCG algorithms and approaches. A significant part of this exploration has been the creation of the small game *Fluky* and the PCG that runs within it. The game is presented in two dimensions, viewed from above and involves steering a boy with a sword through an endless number of dungeons filled with monsters and treasure. The implemented PCG is therefore focused on the generation of game levels.

We have explored PCG mainly by implementing six different algorithms, listed here:

Agent-based
: Lets agents move around and perform certain missions, such as building corridors or placing items, until the result can be used as a level.

Cellular automata
: Iterates through several generations in a grid of random tiles, where every tile has a life cycle depending on its neighborhood. In this way, coherent levels can emerge.

Cyclic
: Creates a graph with cycles, and gradually expands it by adding new cycles, which generates levels with uninterrupted flow.

Grammar-based
: Uses the concept of formal grammars. A simple string of characters is evolved by randomly exchanging different parts of it with other versions, and then translating the abstract concept to concrete levels.

Space partitioning
: Generates a dungeon by dividing a map a number of times, assigning each part a room of random size and then connecting them.

TK algorithm
: A large number of rooms are created and placed adjacent to each other. Certain rooms are then chosen and connected, creating compact levels.

Our goal has been to accumulate all possible knowledge and insights about the development of PCG, to help future game developers and others interested in PCG to make well-informed choices. To meet this goal, we have compiled our observations and discussed our thoughts. The resulting summary of our knowledge has then been presented in the following manner:

Working with the algorithms

Realizations about challenges during development of PCG.

Combination possibilities
How combining different PCG approaches with each other can benefit a game.

Adjusting and adapting algorithms
Different ways to adjust and adapt algorithms, and what to keep in mind when doing so.

Quality of the generated content

Our conclusions about what influences the quality of the generated result of PCG algorithms.

Variety
How variety in procedurally generated content can be both good and bad in different ways.

Difficulty
Observations about how difficulty can affect a player's experience, and different aspects that in turn affect difficulty.

Enjoyment of the generated content
The aspects of PCG which influence how enjoyable the content is.

Summary of the observations of each algorithm

Brief accounts of what to expect when using one of the six PCG approaches that we have explored during this project.

As an example, we realized how much the level of abstraction in which an algorithm primarily performed could influence how well it worked in combination with other algorithms on different abstraction levels, as well as how easy or hard it was to adapt the algorithm to new conditions. This means that it is an important aspect to take into consideration when choosing PCG algorithms.

We also realized that when it comes to the enjoyment of levels generated by PCG algorithms, many different aspects affect the result. The variety of the level plays a big role, for example, as too little makes it feel very mechanical, but too much can feel chaotic. You would rather want just the right amount of variety in combination with an illusion of planning, to make the level feel handmade.

The implementation of *Fluky* is rather simple, so the algorithms used in the game did not have as much to interact with as we would have wished. Our observations are of course also subjective and are therefore not entirely reliable in all situations. However, we believe that those with an interest in developing or researching PCG algorithms may find this report and our observations beneficial.

The area of general practices in procedural content generation could definitely need more research, as we found few earlier papers covering it. We ourselves have only scratched the surface; exploring additional algorithms, investigating them more thoroughly and adding extensive user tests are all remaining avenues to look into. One could also do a similar exploration of PCG but in three dimensions, or attempt to develop new and original PCG algorithms, to add to the field.

References

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*. Springer, 2016.

[2] (2016) No Man's Sky. Hello Games. [Online]. Available: https://www.nomanssky.com/

[3] (2016) Metacritic: No Man's Sky reviews. [Online]. Available: http://www.metacritic.com/game/playstation-4/no-mans-sky/user-reviews

[4] (1980) Rogue. Epyx.

[5] (1978) Beneath Apple Manor. The Software Factory.

[6] David Braben, Ian Bell. (1984) Elite. Acornsoft. [Online]. Available: http://acornsoft.co.uk/

[7] (2009) The History of Elite: Space, the Endless Frontier. Gamasutra. [Online]. Available: https://www.gamasutra.com/view/feature/132375/the_history_of_elite_space_the_.php

[8] (2014) Elite: the game that changed the world. The Telegraph. [Online]. Available: http://www.telegraph.co.uk/technology/video-games/11051122/Elite-the-game-that-changed-the-world.html

[9] (2013) Berlin Interpretation. [Online]. Available: http://www.roguebasin.com/index.php?title=Berlin_Interpretation

[10] (2017) Unexplored. Ludomotion. [Online]. Available: http://store.steampowered.com/app/506870/Unexplored/

[11] A Handcrafted Feel: 'Unexplored' Explores Cyclic Dungeon Generation. [Online]. Available: http://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/

[12] (2006) Dwarf Fortress. Bay 12 Games. [Online]. Available: http://www.bay12games.com/dwarves/

[13] (2011) Minecraft. Mojang. [Online]. Available: https://minecraft.net

[14] (2015) Here's how 'Minecraft' creates its gigantic worlds. Engadget. [Online]. Available: https://www.engadget.com/2015/03/04/how-minecraft-worlds-are-made/

[15] Minecraft Wiki: World boundary. Mojang. [Online]. Available: https://minecraft.gamepedia.com/World_boundary

[16] G. Smith, E. Gan, A. Othenin-Girard, and J. Whitehead, "PCG-based Game Design: Enabling New Play Experiences Through Procedural Content Generation," in *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, ser. PCGames '11. New York, NY, USA: ACM, 2011, pp. 7:1–7:4. [Online]. Available: http://doi.acm.org/10.1145/2000919.2000926

[17] J.Togelius, T.Justinussen, and A.Hartzen, "Compositional Procedural Content Generation," 2012.

[18] G. Smith, "Understanding Procedural Content Generation: A Design-Centric Analysis of the Role of PCG in Games," 2014.

[19] (2013) Generate Random Cave Levels Using Cellular Automata. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664

[20] D. Maung and R. Crawfis, "Applying Formal Picture Languages to Procedural Content Generation," in *2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES)*, July 2015, pp. 58–64.

[21] R. Linden, R. Lopes, and R. Bidarra, "Designing Procedurally Generated Levels," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013. [Online]. Available: https://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/view/7450

[22] V. Valtchanov and J. A. Brown, "Evolving Dungeon Crawler Levels with Relative Placement," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, ser. C3S2E '12. New York, NY, USA: ACM, 2012, pp. 27–35. [Online]. Available: http://doi.acm.org/10.1145/2347583.2347587

[23] J. Maturana. (2009) General Schema of an Evolutionary Algorithm (EA). [Online]. Available: https://commons.wikimedia.org/wiki/File:Evolutionary_Algorithm.svg

[24] (2017) Basic BSP Dungeon generation. [Online]. Available: http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation

[25] M. McLaughlin, "What Is Agile Methodology?" 2018. [Online]. Available: https://www.versionone.com/agile-101/agile-methodologies/

[26] Trello homepage. [Online]. Available: https://www.trello.com/

[27] "What is Scrum?" 2018. [Online]. Available: https://www.scrum.org/resources/what-is-scrum

[28] A. Powell-Morse, "Waterfall Model: What Is It and When Should You Use It?" 2016. [Online]. Available: https://airbrake.io/blog/sdlc/waterfall-model

[29] C. Addis. (2018) What Are The Best Programming Languages For Game Design? [Online]. Available: https://www.gamasutra.com/blogs/ConnorAddis/20180411/316420/What_Are_The_Best_Programming_Languages_For_Game_Design.php

[30] LibGDX homepage. [Online]. Available: https://libgdx.badlogicgames.com/

[31] LWJGL homepage. [Online]. Available: https://www.lwjgl.org/

[32] LibGDX: Goals and Features. [Online]. Available: https://libgdx.badlogicgames.com/features.html

[33] "What is version control," 2018. [Online]. Available: https://www.atlassian.com/git/tutorials/what-is-version-control

[34] Git homepage. [Online]. Available: https://git-scm.com/

[35] GitHub homepage. GitHub, Inc. [Online]. Available: https://github.com/

[36] BitBucket homepage. Atlassian. [Online]. Available: https://bitbucket.org/product

[37] The LaTeX Project homepage. [Online]. Available: https://www.latex-project.org/

[38] (2018) Google Docs homepage. Google. [Online]. Available: https://www.google.com/intl/en/docs/about/

[39] Google Drive homepage. [Online]. Available: https://www.google.com/drive/

[40] ShareLaTeX homepage. [Online]. Available: https://www.sharelatex.com/

[41] Gantt.com. [Online]. Available: http://www.gantt.com/

[42] D. Garlan, "Software Architecture: A Roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 91–101. [Online]. Available: http://doi.acm.org/10.1145/336512.336537

[43] D. Skrien, *Object-Oriented Design Using Java.* McGraw-Hill, 2009.

[44] GWT Project. [Online]. Available: http://www.gwtproject.org/articles/testing_methodologies_using_gwt.html

[45] G. Reese, *Database Programming with JDBC and Java, Second Edition.* O'Reilly & Associates, 2000.

[46] (2014) Tiny keep. Digital Tribe. [Online]. Available: https://store.steampowered.com/app/278620/TinyKeep/

[47] (2013) Procedural Dungeon Generation Algorithm Explained. [Online]. Available: https://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/

[48] E. W. Weisstein. (2018) Minimum spanning tree. MathWorld–A Wolfram Web Resource. [Online]. Available: http://mathworld.wolfram.com/MinimumSpanningTree.html

[49] sheep @ opengameart.org. [Online]. Available: https://opengameart.org/content/alternate-lpc-character-sprites-george

[50] mechaelite @ DeviantArt. [Online]. Available: https://mechaelite.deviantart.com/art/RPG-Maker-Warhammer-40k-Assassins-Sprites-463389091

[51] Orteil. [Online]. Available: http://pixeljoint.com/pixelart/90714.htm

[52] V. Jaimini. (2017) Flood-fill algorithm. HackerEarth. [Online]. Available: https://www.hackerearth.com/practice/algorithms/graphs/flood-fill-algorithm/tutorial/

[53] M. Kerssemakers, *Procedural Adventure Generation: The Quest of Meeting Shifting Design Goals with Flexible Algorithms*. Cham: Springer International Publishing, 2017, pp. 151–173. [Online]. Available: https://doi.org/10.1007/978-3-319-53088-8_9

A Design Document

Design Document (first draft)

Prio 1: Needs to be in
Prio 2: Really want
Prio 3: Not necessary
Other: General decisions

- (Story)
  - *See PCG*

- Characters
  - Main character
  - Enemies
    - Hazardous critters (damage on contact)
    - Melee
    - Ranged
    - Magical

- Level/environment design
  - Top-down
  - Finite levels
  - Rectangular levels (grids of some sort)
  - Increasing level size (to make it harder / more interesting)
  - Some kind of goal (maybe a door to stairs)
  - Some kind of labyrinth - must be solvable!!
  - collectibles (coins, health, weapons, experience, etc.)
    - weapons
    - experience
    - health
    - coins
  - *May or may not be larger than the screen*
  - Cannot revisit levels
  - Traps / dangerous parts of the environment (spikes, etc)
  - Holes/non-traversable sections that one can see across (and ranged attacks reach past)
    - Bridges over these sections (activated by puzzles: switches, keys, put a box in a hole, etc)

- Gameplay
  - Sometimes environmental puzzles that (maybe) needs to be solved to reach the goal - must be solvable!!
    - Find a key + locked door (Variation: buttons that activate bridges)
    - Pushing boxes to specific locations

        - Multiple activation spots before progression (destroy stuff, push buttons, etc) Variation: activation IN SEQUENCE (maybe with clues?)
        - Push boxes into holes to make bridges
    - Some kind of player progression (level ups, skills, weapons, etc.)
    - Non-static environment (parts of the map is replaced by other random sections)
        - "Stick of randomness" → Randomize sections (or rules)
    - Weapons:
        - melee (swords, knuckles?)
        - projectile
        - magic? (ranged, area damage, etc.)
        - *+ default starting? No starting weapon?*
        - Durability / charges of weapons (to balance the power of weapons)
        - Throwing weapons?
    - Only have one weapon at a time - and dropped weapons stay on the ground
    - Permadeath?
    - Reset key-ring between levels (and all other puzzle related items)
    - Line of sight => do not see through walls
        - Fog of war => see where one have been before

- PCG
    - Per level
    - Seed-based (deterministic)
    - Puzzles and labyrinth need to be solvable (multiple solutions?)
    - Difficulty as a parameter (explore the concept etc.)
    - Gameplay rules? => For the whole game? For each level?
    - Enemy stats
    - Player behaviour => PCG decisions
    - Generate chapter names for each level (that takes some detail of the level into account)

- Art
    - Keep it simple!
    - Wall, floor, hole, door, stairs, main character, enemies, weapons, puzzle stuff, keys, collectibles

- Sound of Music
    - Keep it even more simple!
    - Sounds > Music
    - VERY low priority

- User Interface
    - Health
    - How many / what kind of keys
    - Which weapon and its stats (power, durability, range?)
    - Counter for collectibles / puzzle items?

    - a minimap for larger levels (maybe as a (non-essential) collectible that points to the goal etc.)
    - Pause? + seed indicator
    - Menus?

- Game Controls
    - 4 Direction movement + attack + interaction = 6 buttons
    - cycle button (for items) => + 1 button
    - 4 attack directions => + 3 buttons