Constructive Generation Methods for Dungeons

Seminar-Thesis in Procedural Content Generation for Games

Author: Marco Niemann
Heekweg 12
48161 Münster
+49 (176) 9320 6605
marco.niemann@uni-muenster.de

Supervisor: Dr. Mike Preuß

Date of Submission: June 30, 2015
Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this Seminar Thesis titled ‘Constructive Generation Methods for Dungeons’ is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citations and references.

Münster, June 30, 2015
# Contents

List of Figures II

List of Tables III

List of Listings IV

Abbreviations V

Symbols VI

1 Introduction 1

2 Definitions 2

2.1 Procedural Content Generation (for Games) 2

2.1.1 Content 3

2.1.2 Games 4

2.2 Dungeon 4

2.3 Motivation 5

3 Constructive Generation Methods for Dungeons 7

3.1 Space Partitioning 7

3.1.1 Areas of Application, Pros and Cons 7

3.1.2 Background Information 8

3.1.3 Dungeon Creation Algorithm 9

3.2 Cellular Automata 11

3.2.1 Areas of Application, Pros and Cons 11

3.2.2 Background Information 12

3.2.3 Dungeon Creation Algorithm 14

3.3 Evolving Cellular Automata 17

3.3.1 Areas of Application, Pros and Cons 17

3.3.2 Overview over Genetic Algorithms 17

3.3.3 Dungeon Creation Algorithm 18

3.4 Generative Grammars 22

3.4.1 Areas of Application, Pros and Cons 22

3.4.2 Overview over Grammar Types 22

3.4.3 Dungeon Creation Algorithm 24

4 Limitations and Conclusion 26

References 27

A Statistics Regarding Percentage of Gamers in Germany 32
## List of Figures

1. Dungeon-Screenshots from different games ................................................. 6
2. Different Tree Types for different dimensions ............................................ 8
3. BSP Dungeon Generation Algorithm .......................................................... 10
4. 3D-Dungeon Generation Issue ................................................................. 12
5. Grid and Neighbourhoods of Cellular Automata ........................................... 13
6. State change in CA with a Moore neighbourhood of level 1 ............................ 13
7. CA Dungeon .................................................................................................. 16
8. CA Dungeon Creation Process Schema ......................................................... 16
9. GA Offspring Creation .................................................................................. 18
10. GA Algorithm Schema .................................................................................. 18
11. Illustration to clarify ECA Attributes ........................................................... 19
12. Representations/Rule Tables for the CA in ECA ........................................... 19
13. Different Grammar Types Overview ............................................................. 24
List of Tables

1  Five desirable PCG properties ................................................. 3
2  Six classes of game content ................................................... 3
3  Key elements of games ......................................................... 4
4  Advantages and Disadvantages of Space Partitioning .................. 7
5  Advantages and Disadvantages of Cellular Automatons ............... 12
6  Cellular Automata Steering Variables .................................... 14
7  Advantages and Disadvantages of Evolving Cellular Automata ....... 17
8  Attributes for the Cellular Automata of the ECA ....................... 18
9  Attributes for ECA ............................................................ 19
10 Comparison of the different representation types for ECA .......... 19
11 Advantages and Disadvantages of Generative Grammars ............ 22
List of Listings

1. BSP Algorithm for Dungeon Generation - Source: (Shaker et al. 2015) ........................................... 9
2. CA Algorithm for Dungeon Generation - Source: (Johnson et al. 2010) ........................................... 16
3. CA Flood Fill Cavern Identification - Source: (Pedersen 2014b) ....................................................... 16
4. Evolving CA Algorithm for Dungeon Generation - Source: (Pech et al. 2015) .................................. 21
5. Generative Grammar Algorithm for Dungeon Generation
   Source: (Dormans 2010; Dormans and Bakkes 2011; van der Linden 2013) .................................... 24
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM</td>
<td>Attribute Similarity Measure</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur-Form</td>
</tr>
<tr>
<td>BSP</td>
<td>Binary Space Partitioning</td>
</tr>
<tr>
<td>CA</td>
<td>Cellular Automaton/Automata</td>
</tr>
<tr>
<td>CS</td>
<td>Computer Sciences</td>
</tr>
<tr>
<td>ECA</td>
<td>Evolving Cellular Automata</td>
</tr>
<tr>
<td>env.</td>
<td>environment</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
<tr>
<td>FV</td>
<td>Fitness Value</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm(s)</td>
</tr>
<tr>
<td>ms</td>
<td>millisecond(s)</td>
</tr>
<tr>
<td>NPC</td>
<td>Non-Player Character</td>
</tr>
<tr>
<td>PCG</td>
<td>Procedural Content Generation</td>
</tr>
<tr>
<td>PCG-G</td>
<td>Procedural Content Generation for Games</td>
</tr>
<tr>
<td>prev.</td>
<td>Previous</td>
</tr>
<tr>
<td>RPG</td>
<td>Role-Play Game</td>
</tr>
<tr>
<td>RTS</td>
<td>Real-Time Strategy</td>
</tr>
</tbody>
</table>
Symbols

# number of/count
1 Introduction

Chances are high, that most people who will read this will have played some kind of a video game; be it the retro games back in the 1980s and 90s when computer games became popular or one of the more up-to-date mobile games. However chances are probably equally high, that terms like Procedural Content Generation (PCG) or Constructive Generation Methods are unknown to the very same reader. And the justified question at this point could be:

*Why should anyone actually care about it?*

→ Because PCG could very well be the next Big Thing in the gaming sector!

And this holds true for both - those who are ‘just’ playing them, but also for those who are more interested in creating them. For those who belong to the group not yet convinced by this answer, the Definitions section - and there especially the part concerning motivation - will be a good start to get a better insight. Just to foreshadow some bits: While the idea of ‘Time being money’ will be proven again, the idea that ‘Everything has an end’ will possibly be one to question after reading the thesis.

*But why dungeons?*

→ As they are one of the PCG origins and still a popular element of many RPGs and FPS.

Those who not already fought through a virtual dungeon could have a look at the part concerning Dungeons to get a better idea of what this is all about.

Another part of the answer of course is, that this seminar thesis only has limited space whereas the research area of PCG is already pretty large. And as some preliminary research in the ‘PCG Book’ by Togelius et al. (2015b) revealed some interesting algorithmic solutions in the area of dungeon generation, the decision was made to write this thesis in exactly this area.

Finally three algorithms (Space Partitioning, Cellular Automata and Generative Grammars) and one extension (Evolving Cellular Automata) were chosen to be discussed in more detail in the main part of this paper.

So for everyone who is now eager to get deeper into the topic, the answer to the final question:

*And where are the algorithms?*

→ Right here: Constructive Generation Methods for Dungeons.

Having read the complete thesis a reader should have a basic understanding of the discussed algorithms, their strengths and downsides. Furthermore it should become clear, that there is no single truth in creating dungeons and that most algorithms can be improved, e.g. by combining them with others.
2 Definitions

After already having talked about some topic-specific terms/ideas in the introductory section, it might be helpful to discuss and define some of the terms. This shall help to establish a common understanding for this thesis, but also enable readers with no or no strong background in this thematic setup to get straight up into the topic.

2.1 Procedural Content Generation (for Games)

As this thesis is settled in the research area of Procedural Content Generation (PCG), it makes sense to get a definition for this subject first.

Being no particularly young research field, dating back to 1980 when Rogue, the first game employing PCG mechanisms (Khaled et al. 2013) was published, it seems that PCG is only receiving larger scientific interest for the last 10 years, when own journals and conferences where established (ACM 2015; IEEE 2015; Bidarra et al. 2010).

This may be one reason, that - instead of one consolidated one - there still is a range from rather abstract definitions like the Gamasutra (2012) one

*The platonic Procedural Content Generation algorithm allows you to create entire universes by pressing a button.*

...to more concrete and precise ones like the one Togelius et al. (2015a) use in their yet unpublished PCG-Book:

*The definition we will use is that PCG as the algorithmical creation of game content with limited or indirect user input.*

...Here it is interesting to note, that some authors (e.g. Hendrikx et al. (2013)) use the term Procedural Content Generation for Games (PCG-G) rather than just PCG - probably to clearly separate it from approaches using PCG for art creation (Wikidot 2009) or movie creation (Massive 2011). Yet most papers and sources tend to simply use PCG synonymously to PCG-G, so this thesis will follow this pattern.

What can be derived from both above cited (and also other) definitions is, that the overall aim of PCG is to go away from manually generating game content, towards an automated/procedural approach based on algorithms - or to make it short: a mostly\(^1\) computer-based instead-of human based content generation. A real-world example according to Togelius et al. (2015a) could be a tool, that generates dungeons for a game like The Legend of Zelda without human interaction, whereas map editors where a player can create RTS game maps would not be considered a PCG tool.

---

\(^1\)mostly, as for example Togelius et al. (2015b) and others propose so called mixed-initiative approaches where computers and humans co-create content
Constructive Generation Methods for Dungeons

To make the topic a little more quantifiable/tangible Togelius et al. (2015a) propose the following five desirable properties when discussing PCG solutions (as we are going to do in the course of this thesis):

<table>
<thead>
<tr>
<th>Speed</th>
<th>Reliability</th>
<th>Controllability</th>
<th>Expressivity and diversity</th>
<th>Creativity and believability</th>
</tr>
</thead>
<tbody>
<tr>
<td>having content in time</td>
<td>having content in a desired quality</td>
<td>human can specify certain aspects</td>
<td>diverse set of content</td>
<td>content should not look 'generated'</td>
</tr>
</tbody>
</table>

Table 1: Five desirable PCG properties

2.1.1 Content

Since it is the outcome of PCG efforts, as a next step it is helpful to get a better idea of what is subsumed under the term content.

In their paper on 'Search-Based Procedural Content Generation' Togelius et al. (2011) (also taken up in the upcoming PCG Book (Togelius et al. 2015a)) defined (game) content as

\[ \text{content [that] refers to all aspects of the game that affect gameplay other than nonplayer character (NPC) behaviour and the game engine itself} \]

where they consider these aspects to be things like terrain, maps, levels, stories etc.

A somewhat different yet more extensive approach is taken by (Hendrikx et al. 2013) in their survey on PCG for games. They provide a six-class taxonomy of procedurally generatable game content which allows further insights into what content is, but also indicates that content is a rather multi-then single-dimensional idea:

Table 2: Six classes of game content (Hendrikx et al. 2013)

<table>
<thead>
<tr>
<th>Game Bits</th>
<th>Game Space</th>
<th>Game Systems</th>
<th>Game Scenarios</th>
<th>Game Design</th>
<th>Derived Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>textures</td>
<td>indoor maps</td>
<td>ecosystems</td>
<td>puzzles</td>
<td>System Design</td>
<td>News and Broadcasts</td>
</tr>
<tr>
<td>sound</td>
<td>outdoor maps</td>
<td>road networks</td>
<td>storyboards</td>
<td>World Design</td>
<td>Leaderboards</td>
</tr>
<tr>
<td>vegetation</td>
<td>bodies of water</td>
<td>urban env.</td>
<td>story</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buildings</td>
<td>entity behaviour</td>
<td>levels</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>behaviour</td>
<td>fire, water, stone, clouds</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this thesis multiple classes will be - at least partly - covered: this includes Game Space (outdoor and indoor maps), Game Scenarios (Levels), Game Design (System Design) and implicitly Game Bits. Why these types of content matter for the constructive generation of dungeons will hopefully become clear in the following sections.

\(^{2}\)the difference is, that Hendrikx et al. (2013) regard NPCs as part of PCG whereas e.g. Togelius et al. (2015a) argue it is not - but as this is no concern of this thesis there will be no further elaboration on this fact.
2.1.2 Games

Talking about games in the PCG context does - in contrast to what one might think regarding the previous sections and the literature often coming from a CS background (e.g. see the references of this paper) - not only include computer games, but can as well mean board games, card games and puzzles (Togelius et al. 2015a). For this thesis however, the range will be limited to computer games, as the setting for the dungeons to be created.

When it comes to formally defining the term Games there is some conflict, as for example Salen and Zimmerman (2003) point out the difficulties to define the term, but yet provide a definition (that for example Browne and Maire (2010) take up), whereas e.g. Togelius et al. (2015a) reject the idea to define the word completely. Yet all of them propose some key elements of games:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>design</td>
<td>rules/means</td>
<td>system</td>
</tr>
<tr>
<td>affordances</td>
<td>play</td>
<td>players</td>
</tr>
<tr>
<td>constraints</td>
<td>outcome/ends</td>
<td>artificial</td>
</tr>
<tr>
<td>playable</td>
<td></td>
<td>conflict</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>quantifiable outcomes</td>
</tr>
</tbody>
</table>

Table 3: Key elements of games

So it can be stated that a Game somehow involves playing under certain conditions/rules and must lead to a certain end with a certain outcome.

2.2 Dungeon

While the original definition of a dungeon, as e.g. given by the Oxford Dictionary (2015), refers to an underground prison cell, the definition for dungeons in games is less restrictive characterizing

adventure and RPG dungeon levels as labyrinthic environments ([rooms connected by hallways]), consisting mostly of inter-related challenges, rewards and puzzles, tightly paced in time and space to offer highly structured gameplay progressions

that can include additional elements like characters (NPCs), decorations etc. (Shaker et al. 2015).

As a distinguishing characteristic of dungeon levels van der Linden et al. (2013) identified the 'close control over gameplay pacing' respectively the 'tighter bond between designing gameplay and game space'. To illustrate this idea one can build on the proposed comparison to open world games or platform games (van der Linden et al. 2013): So for example in race track game (platform game) there will usually a predefined track without much variation (as in it’s real world pendant), whereas in an open-world games like GTA V the player can freely explore the world. In a dungeon game however, the player may encounter something like a main track/hallway, that leads to an end boss.

---

3 even though there are also board games based on dungeons available (Dungeon!; Wizards of the Coast 2015) that according to Shaker et al. (2015) are predecessors of the modern computer versions

4 the definition can be found in Salen and Zimmerman (2003, p.11) - the underlying definitions can be found in Salen and Zimmerman (2003, p.4-9)
or a treasure, which has multiple rooms and/or hallways diverging from it - so there is control as
the player will have to follow the main path, but also exploration as there is no prescribed way to
explore the other areas of the dungeon.
Known examples for games employing dungeons are Diablo, Rogue, Doom, Half Life, Sacred etc.
(Adams 2002; van der Linden et al. 2013). Screenshots (Figures 1a-1d) from some of those games
can be found on the next page and maybe help to get a better visual impression of what a 'real
virtual' dungeon looks like.

2.3 Motivation

So before finally starting with the main part, the last remaining question may be, why the whole
topic is relevant.
One reason is the growing complexity of games\(^5\) (Hendrikx et al. 2013) in combination with the ris-
ing number of people playing games (Hughes 2012; BITKOM 2014) (statistics regarding Germany
see: Appendix A), both creating a rising demand for more and more content. The resulting growth
in numbers of content designers from single persons per game to hundreds (Hendrikx et al. 2013)
resulted in rapidly growing expenditures for content generation\(^6\). This and also the lack of scala-
ibility (Iosup 2011) and timeliness (Kelly and McCabe 2007) made PCG a candidate to either reduce
artists or at least increase their efficiency (Togelius et al. 2015a).
Furthermore real-time PCG instead of manual content generation can help to automatically adjust
difficult levels (Togelius et al. 2015a), so that gamers would always be challenged at their current
level. Also games like Super Tux would not have to end after a fixed amount of levels, as theo-
retically at the end of each level a new one could be generated (Togelius et al. 2015a; Super-
tux.lethargik.org 2014).
The last advantage Togelius et al. (2015a) see, is that a PCG approach, e.g. for dungeon or level
generation, could potentially create dungeons and levels that a human designer might not even
think of and thus offer completely new experiences.
That is plenty of reasons to now have a closer look at algorithms and methods to understand what
is the current state of the art, and where there is still limits.

---

\(^5\) Hendrikx et al. (2013) found evidence that already in 2008 the game World of Warcraft contained about 30,000 items,
5300 interactable creatures etc.

\(^6\) If the evidence Hendrikx et al. (2013) found is correct, and about 40% of game cost is for content, e.g. for modern
blockbuster productions like GTA 5 the content cost would be about 106 million dollar (McLaughlin 2013)
Figure 1: Dungeon-Screenshots from different games
Source: (Venator_Noctis 2011; Valencia 2006; Willbr 2006; Artoftransformation 2008)
3 Constructive Generation Methods for Dungeons

In the main section of this thesis, it will be the aim to discuss a selected set out of the variety of methods to procedurally generated dungeons. While Shaker et al. (2015) and van der Linden et al. (2013) typically distinguish three steps to make up a dungeon generation method, most of the following methods will focus on the representational model as well as the method to construct that representational model. The translation into the actual dungeon will not be a central aspect.

Another limit that should be mentioned beforehand is the large focus on 2D games - so what van der Linden et al. (2013) state regarding genetic algorithms (‘may allow some form of 3D mapping, the current work only focuses on 2D’) holds true for most of the other presented approaches.

3.1 Space Partitioning

To allow a more or less easy entry into dungeon map/level PCG, space partitioning was chosen as a first method.

3.1.1 Areas of Application, Pros and Cons

This is based on the fact, that Shaker and Liapis (2013) and Pedersen (2014a) categorize this method as an easy to implement, relatively simple approach. It would be the recommended way to go, whenever the aim is the creation of a rather structured, neatly aligned dungeon, without overlapping rooms. So recalling the desirable PCG properties (see Table 1) one can say that at least in terms of diversity and believability the structuredness might be a problem - simply because natural cave like dungeons could not be generated by Space Partitioning, but rather only 'man-made' dungeon structures (like 'real' dungeons). And even there too much symmetry might create a feeling of generated content, so those two points make clear, why the structuredness is one of the downsides of the space partitioning approach.

Another downside has been found by Williams (2014) regarding the lack of control e.g. over the number of rooms (reasons see algorithmic description in section 3.1.3), which would violate the controllability property.

To allow a better overview over Pros and Cons they are summed up in Table 4.

<table>
<thead>
<tr>
<th>Pro</th>
<th>Contra</th>
</tr>
</thead>
<tbody>
<tr>
<td>easy implementation/simple</td>
<td>very neat/organized</td>
</tr>
<tr>
<td>no overlaps</td>
<td>looks 'generated'</td>
</tr>
<tr>
<td>easy creation of groups of rooms</td>
<td>very limited control</td>
</tr>
</tbody>
</table>

Table 4: Advantages and Disadvantages of Space Partitioning

Source: (Shaker et al. 2015; Shaker and Liapis 2013; Williams 2014)

---

7to have an arbitrary example of such a model, one could just imagine treating a dungeon as an array of 0 and 1 to distinguish between wall and stone

8a quote illustrating the notion would be ‘There are no straight lines or sharp corners in nature. Therefore, buildings must have no straight lines or sharp corners.’ (Antoni Gaudi)
3.1.2 Background Information

This paragraph will give some details about the theoretical background of this method before its application for dungeon generation will be illustrated. The basic idea behind the space partitioning algorithm is taking a given 2D/3D-space and dividing it into disjoint subsets - typically hierarchically with a recursive algorithm (Shaker et al. 2015). As a data-structure usually a tree is used to represent the subsets and their hierarchy - also called space-partitioning tree (Shaker et al. 2015).

For the dungeon-creation the special form of binary space partitioning (BSP) - a technique originally developed to efficiently display computer graphics around 1980 (Toth 2005; Fuchs et al. 1980) - can be used (Shaker et al. 2015). The BSP method will be used to recursively split a (2D) space in halves until a certain level of granularity is reached - here it has to be noted, that the partitions do not need to be of equal size, but can be defined by custom rules (Shaker et al. 2015). This can be visualized by the so called BSP tree, a binary tree like in Figure 2a just with the difference, that it operates on a 2D space like the Quadtree in 2b. So the overall idea of Figure 2 is to show, that BSP has multiple variants to treat different dimensions and that the Quadtree could be used for 3D space (Shaker et al. 2015).

---

9 This all-encompassing view of the dungeon right from the start is the reason, why this approach is also called a macro approach (Shaker et al. 2015).

10 subsets being disjoint is the reason why no overlaps can exist in BSP - because by the definition of disjoint no piece of space can be stored twice within the tree and can thus not be used twice.

11 one game where BSP for graphics was used is DOOM - here David Fetter developed a project that displays this system (see: https://www.youtube.com/watch?v=e0W65Sc2mQu).

12 the level of granularity may vary depending on the context - but e.g. for a dungeon rooms with the size of one pixel would make no sense - thus breaking a given space down to that level is not necessary.
3.1.3 Dungeon Creation Algorithm

The following algorithm finally describes the generation of a dungeon level and is based on the algorithmic descriptions of Shaker et al. (2015) and Williams (2014) and can - with small deviations - be seen in implemented form in Simon (2009) and Hely (2013).

**Code-Lines 1-6 in Listing 1 and Images 3a to 3d**

As one might assume at this position, the previously (see section 3.1.2) introduced BSP method will be used to divide a given dungeon space into smaller subsets, that can then be filled with rooms. The decision in which direction a section will be split can e.g. be random (Shaker et al. 2015) or be biased by given cell-sizes (Simon 2009) - the only limiting factor in that regard is that the resulting subsets must still have a certain size\(^{13}\) (e.g. not less than either a quarter of the original width or height).

**Code-Lines 7-8 in Listing 1 and Image 3e**

Once subdividing of the dungeon area is finished, the leaf nodes representing the dungeon structure have to be filled with rooms. Shaker et al. (2015), Simon (2009), and Hely (2013) propose drawing rectangles within the partition, underlining the importance of having rooms smaller than the partition itself and being 'in' the partition to prevent connected rooms\(^{14}\). To add more variation, Williams (2014) proposes to only randomly place rooms in partitions or to use handmade tiles or procedurally generated shapes as rooms instead of simple rectangles.

**Code-Lines 9-10 in Listing 1 and Images 3f to 3h**

To satisfy the game requirement of being playable (see Games) in a final step the rooms have to be connected. Here random methods to connect different rooms are possible (Williams 2014) but the BSP tree can be used again: By connecting rooms sharing the same parent node (siblings) from the leaves to the top, all rooms will be connected and on top of that, intersections of corridors with other corridors and rooms will be prevented (Shaker et al. 2015; Williams 2014).

```plaintext
1 start with the entire dungeon area  // root node of the BSP tree
2 divide the area along a horizontal or vertical line
3 select one of the two new partition cells
4 if this cell is bigger than the minimal acceptable size:
   5   go to step 2  // using this cell as the area to be divided
6 select the other partition cell, and go to step 4
7 for every partition cell:
   8   create a room within the cell by randomly choosing two points ("top left and bottom right") within its boundaries
9 starting from the lowest layers, draw corridors to connect rooms in the nodes of the BSP tree with children of the same parent
10 repeat 9 until the children of the root node are connected
```

**Listing 1: BSP Algorithm for Dungeon Generation - Source: (Shaker et al. 2015)**

\(^{13}\)This is the reason for the lack of controllability - because due to the random splits it is not clear how many of them happen before the constraint is reached (and as splits create the spaces for rooms the room number depends on the number of splits)

\(^{14}\)the connection between rooms would come to pass, if in two neighboring cells the rooms would be placed exactly on the border of those two cells
Constructive Generation Methods for Dungeons

**Add on**

Even if not represented in Listing 1, one could continue the dungeon creation by adding decorations to the dungeon (enriching the *Game Space with Game Bits*) to increase both *diversity* and *believability* of the dungeon. Again the BSP tree can be used, e.g. to define thematic areas\(^5\) (Shaker et al. 2015) or to ensure 'playable' placement of keys\(^6\) or other elements (Williams 2014).

![BSP Dungeon Generation Algorithm](image)

*Figure 3: BSP Dungeon Generation Algorithm*

---

\(^5\)Shaker et al. (2015) points out due to the BSP tree hierarchy, there will typically only be one entrance to the rooms represented by the child nodes of one non-leaf node in the tree - so for each non-leaf node a multi-room thematic area can be created.

\(^6\)Williams (2014) proposes e.g. placing a key the child nodes of the node with the corresponding door.
3.2 Cellular Automata

After discussing the Space Partitioning method used to generate very structured and symmetric dungeons, the next step will be to have a look at an approach capable to generate more natural looking dungeons - the Cellular Automaton.

3.2.1 Areas of Application, Pros and Cons

This already forestalled a potential area of application as well as the first advantage of this method. While, as has been discussed in the previous parts, the Space Partitioning method could be used to generate dungeons that look human made, the Cellular Automaton can be useful to generate more 'natural'/organic looking caverns (Shaker et al. 2015; Anonymous 2014; van der Linden et al. 2013). In terms of diversity and believability this can be seen as an advantage.

Shaker et al. (2015) also underline the versatility of CA (potential areas of application see: Gibson et al. (2013, p. 12)) as well as their ability to generate infinite dungeon levels\(^{17}\) and this even online/while the game is being played. To prove the efficiency, Johnson et al. (2010) - the CA for PCG algorithm developers - tested it and reported the low average generation time of \(4.1 \cdot 10^{-1}\) milliseconds (\(\approx 0.06\) FPS), even with a from today’s point of view slow computer with a 1.73 GHz single-core CPU\(^{18}\).

Another implicitly mentioned pro-argument is the theoretically higher control with four (respectively five) variables (see section 3.2.3) - yet this is also one of the contra arguments, as van der Linden et al. (2013) and Shaker et al. (2015) argue, that the parameters exist, but due to interaction effects the influence of a single one is hard to predict. This makes it difficult to adjust the game to any specific technical or gameplay requirements other than by trial and error (Shaker et al. 2015).

Moreover van der Linden et al. (2013) and Johnson et al. (2010) are concerned that the CA method as proposed by Johnson et al. (2010) is only viable for 2D but not 3D both for reasons of control as well as playability. To illustrate it one can imagine a simple path in a 2D space - if it is free it is free - yet in 3D the same 2D path could exist and yet special procedures/rules would be needed to ensure that the path also has a 'walkable' height to be passable (see Figure 4).

As a last issue Anonymous (2014) mentions problems with larger maps not looking very well, but only provide indirect proof by mentioning issues with non-connected caverns in the dungeon, respectively larger free areas.

For a better overview, the arguments for and against the Cellular Automata are compiled into a small table (Table 5) like in the Space Partitioning section.

\(^{17}\)meaning that a given player could move in every direction in a dungeon, without ever reaching an end

\(^{18}\)yet in another section of his article he reports a generation time of 349 ms for another CA map - which today would most likely not be seen as real time as for example gamers complain about a game being unplayable if their latency goes over 300 ms (as a comparison), cf. I_2_i et al. (2010)
### Constructive Generation Methods for Dungeons

<table>
<thead>
<tr>
<th>Pro</th>
<th>Contra</th>
</tr>
</thead>
<tbody>
<tr>
<td>versatile</td>
<td>impact of parameters often unclear</td>
</tr>
<tr>
<td>not looking generated</td>
<td>no requirement specific adjustments</td>
</tr>
<tr>
<td>fast/low computational cost</td>
<td>connection to gameplay is trial &amp; error</td>
</tr>
<tr>
<td>control (4 Parameters)</td>
<td>more control issues when in 3D</td>
</tr>
<tr>
<td>infinite dungeons possible</td>
<td>difficulties for larger maps</td>
</tr>
</tbody>
</table>

**Table 5:** Advantages and Disadvantages of Cellular Automatons  
Source: (Shaker et al. 2015; Johnson et al. 2010; van der Linden et al. 2013; Anonymous 2014)

---

#### 3.2.2 Background Information

The idea behind the CA is not new, as the first ideas and papers in that direction where published around 1950 by John von Neumann and Stanislaw Ulam (Wolfram 2002). Since then CA have been used in many science areas (*biology, physics, ...*) (Gibson et al. 2013) leading to a large amount of research regarding CA. So here only a short overview over the most important aspects will be provided.

The very basic element the CA will operate on, is a 2D grid of cells (Figure 5a) (other dimensionalities possible but uncommon) with each of the cells being in a finite amount of states (something like \{0, 1\}, \{wall, path\}, \{gray, red\}) (Shaker et al. 2015; van der Linden et al. 2013).

Another characteristic of this method is the concept of so called *neighbourhoods*, which refers to the cells surrounding a chosen cell (van der Linden et al. 2013). Two common patterns are the Moore and the von Neumann neighbourhood (more exist, e.g. for reference Tyler (1970)\(^{19}\):

- the *Moore* neighbourhood (Figure 5b) spans all cells (marked in dark red) surrounding a selected cell (marked in gray) - it can even have multiple levels (second level indicated in light red) (Shaker et al. 2015)

- the *von Neumann* neighbourhood (Figure 5c) only includes the cells in the north, south, west and east of the selected cell (Shaker et al. 2015)

\(^{19}\) 1970 is not the publication data of that web-page - unfortunately this date had to be used as a placeholder as the used citation tool does not support empty areas for the year and with no year \LaTeX{} would use the website title in brackets which looks rather bad
As CA are time-discrete systems (Shaker et al. 2015), they change their state in time-steps and not continuously (so a sequence like $t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$ exists).

The type of the state change is determined by a fixed set of rules and the given neighbourhood and usually all cells change their state after a time step (Shaker et al. 2015; Pedersen 2014b). So there could be the simple rule in a dungeon level, that whenever there are three cells of wall in the neighbourhood of a cell, it will be transformed to a wall as well (or to a path if there is less than three wall cells and the cell is a wall). The cell's state in time $t_n$ would then be based on the sum of wall-cells in its neighbourhood in $t_{n-1}$ and its own state in $t_{n-1}$ (Shaker et al. 2015).

The statechange process is visualized in Figure 6, where gray cells represent wall, the dark red cell the currently selected cell and the lightred cells are supposed to indicate a Moore neighbourhood.

---

**Figure 5:** Grid and Neighbourhoods of Cellular Automata

---

**Figure 6:** Statechange in CA with a Moore neighbourhood of level 1
Constructive Generation Methods for Dungeons

3.2.3 Dungeon Creation Algorithm

The preceding section already introduced a lot concepts that make up the Cellular Automaton Dungeon Creation Algorithm, as it is proposed by Johnson et al. (2010). So this section will focus on the topics not yet discussed, that are needed to successfully generate a dungeon.

At first that will concern the parameter- and rule-set that is used to steer the CA. Table 6 introduces the shortcuts, a short description and proposed values:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>initial percentage of rock cells</td>
<td>0.5 (50%)</td>
</tr>
<tr>
<td>$n$</td>
<td>CA iterations</td>
<td>2</td>
</tr>
<tr>
<td>$T$</td>
<td>neighbourhood value threshold that defines a rock</td>
<td>5</td>
</tr>
<tr>
<td>$M$</td>
<td>Moore neighbourhood size</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6: Cellular Automata Steering Variables - Source: Johnson et al. (2010)

Given the knowledge of the previous section a fifth variable - the type of the neighbourhood - could be discussed, but is neglected for the current approach.

The only rule Johnson et al. (2010) use is close to the one used e.g. in Figure 6:

- $T = 5$ with the meaning, that if in the neighbourhood of a cell there is five rock cells it is converted to rock - otherwise it will be converted to a floor cell

Codelines 1-8 in Listing 2

In the first half of the algorithm should not be too many surprises, as it in large parts resembles the general work pattern of a CA discussed in the previous part - it starts with the initialization of the grid in a certain size and the conversion of $r$ (or here 50%) of the cells to rock with uniformly distributed random numbers. Afterwards the automaton runs $n = 2$ times, calculating the neighbourhood value and converting cells depending on the value being bigger or smaller $T = 5$.

Codelines 9-10 in Listing 2

To ensure that a player can continuously play it is important, that once he reaches the end of a grid he can directly advance to the next. This is achieved by not only generating the so called base grid (codelines 1-8), but also generating the adjacent grids of it in the same step (Johnson et al. 2010). Again it can be discussed how the neighbourhood is defined, as Johnson et al. (2010) for adjacent grids proposes a von Neumann neighbourhood, Shaker et al. (2015) a Moore neighbourhood.

So basically the code steps 1-8 are then applied to the neighbouring 4 respectively 8 grids to set up the dungeon structure there as well (Johnson et al. 2010). In a somewhat reduced\(^\text{21}\) version this procedure will be run repeatedly throughout the game, as every time the player enters a new grid, its neighbours have to be generated (Johnson et al. 2010).

---

\(^{21}\)the neighbourhood value is the number of rock cells in a given neighbourhood, so e.g. in a Moore Level 1 neighbourhood this value will lie between 0 and 9

\(^{21}\)reduced as depending on the neighbourhood not each neighbour will have to be generated as some already exist
Constructive Generation Methods for Dungeons

**Codelines 11-12 in Listing 2**
After this step is completed, there is five respectively nine dungeon-grids, yet it is not sure that those are connected (Johnson et al. 2010). If no connection can be found, Johnson et al. (2010) propose to establish it by picking the two floor cells closest to the border of two grids and digging a tunnel between them. This ensures the overall playability of a level (see Games).

**Codelines 13-14 in Listing 2**
Upon completion of the previous steps theoretically a playable dungeon would be available to the player. However, as the each grid was generated by a separate CA run and tunnels were dug in a straight line, the believability may have suffered, as the dungeon may be rather inconsistent in design (Johnson et al. 2010). The proposed solution are another $n$ CA runs on the whole dungeon (all five/nine grids) to remove the given flaws (Johnson et al. 2010).

**Codeline 15 in Listing 2**
This last step in the algorithm is necessary to ensure that a player can actually return to a location in a dungeon and will encounter the same 'setting', as has been there before (Johnson et al. 2010). To achieve this, an implementation of the algorithm will have to store the seed for the base grid including pointers to neighbouring grids (Johnson et al. 2010) (which then will be in need to be updated on expansion of the dungeon due to player exploration).

**Add on in Listing 3**
Having used a CA to create a mobile app, Pedersen (2014b) observed a problem that Johnson et al. (2010) do not address in their paper explicitly: within a grid the CA might generate lots of different caverns, which are not necessarily connected.

This is why he proposes to use the illustrated flood fill algorithm to identify the different caverns (Pedersen 2014b). The first lines of the code (1-5) are used to traverse the whole grid, meaning that each cell that is a path cell and not already 'filled' with a fillNumber, will be the starting point of a new cavern with a new fillNumber.

For this new cavern a floodFill(fillNumber)-operation is called, that will set a cell to a given fillNumber whenever it is no rock and recursively invoke itself for all it’s neighbouring cells (here a von Neumann neighbourhood was chosen). At a certain point the recursion will stop, as it will only hit cells that are either filled or rock cells - then control goes back to the steering function described in codelines 1-5. To not overwrite the grid, the flood fill will be done with a copy of the original grid structure (Pedersen 2014b).

Having identified the caverns Pedersen (2014b) considers two ideas worth implementing:

- filling all but the largest cavern to keep the natural dungeon look, but with the issue of loosing playable area
- connecting the caverns with the downside of having straight tunnels between caverns

---

22An animation illustrating the process can be found at [http://cdn3.raywenderlich.com/wp-content/uploads/2014/02/Flood-Fill.gif](http://cdn3.raywenderlich.com/wp-content/uploads/2014/02/Flood-Fill.gif)

23Anonymous (2014) chose a comparable flood fill approach and to solve the problem of loosing area the dungeon was recreated when the remaining path/floor area was below a certain threshold like 45% of the dungeon

24Whereas when integrated into Johnson’s algorithm depending on its placement the new tunnels would also be subject to a rerun of the CA like the tunnels between grids, which would solve this problem
Constructive Generation Methods for Dungeons

Listing 2: CA Algorithm for Dungeon Generation - Source: (Johnson et al. 2010)

initialize empty grid with a x b cells  // e.g. 50 x 50
initialize floor cells
for n iterations
    go through each cell
    calculate neighbourhood values
    if past threshold T
        convert
create adjacent grids  // for square cells that will be 4 or 8
    for each repeat steps 1-8
if two adjacent grids are not connected
create a connection  // between two accessible areas
run n additional CA iterations  // removing inconsistencies
create wall cells  // special rock cells
store base grid and pointer to adjacent grids  // allows restoring the grid

Listing 3: CA Flood Fill Cavern Identification - Source: (Pedersen 2014b)

// Function to iterate through all cells
for each cell
    if not already filled and a path cell
        floodFill(fillNumber)
        increase fillNumber

// floodFill-Function to fill a cavern with a number to identify it
if cell is not of type floor/path
    return
set cell to fillNumber
  go recursively through neighbour cells
    floodFill(fillNumber) for northern cell
    floodFill(fillNumber) for eastern cell
    floodFill(fillNumber) for southern cell
    floodFill(fillNumber) for western cell

Figure 7: CA Dungeon
Source: (Johnson et al. 2010)

Figure 8: CA Dungeon Creation Process Schema
Source: (Anonymous 2014)
3.3 Evolving Cellular Automata

The Evolving Cellular Automata as recently proposed by Pech et al. (2015) uses Genetic Algorithms (GA) to improve CA for dungeon/maze generation.

3.3.1 Areas of Application, Pros and Cons

The area of application and most of the Pros and Cons are basically equal to those discussed for the Cellular Automata. This is the case, as this approach again uses a CA to generate dungeons (Pech et al. 2015), which naturally will enjoy the same benefits like other CA, including the organic and 'realistic' look, as well as being fast and easy to compute at the same time.

As already indicated above, the major difference in terms of advantages is, that the Evolving CA (ECA) does not suffer from the difficulties of manual rule creation, but uses a GA to evolve the rules automatically (Pech et al. 2015). For completeness it should be mentioned, that Genetic Algorithms themselves are an option to generate dungeons and are combined with CA, as they generate proper levels but lack real-time capability, taking up to 20 minutes per dungeon (Pech et al. 2015). Yet even this algorithm has downsides when used: As it will use pre-generated mazes/dungeons and versions that have been modified by a CA (see Dungeon Creation Algorithm) to evaluate the CA (ruleset), it depends on a reasonable selection of attributes to judge whether two mazes/dungeons have a comparable style (Pech et al. 2015).

<table>
<thead>
<tr>
<th>Pro</th>
<th>Contra</th>
</tr>
</thead>
<tbody>
<tr>
<td>advantages of a normal CA</td>
<td>difficult to specify attributes capturing visual style</td>
</tr>
<tr>
<td>no difficult manual rule creation</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Advantages and Disadvantages of Evolving Cellular Automata
Source: (Pech et al. 2015)

3.3.2 Overview over Genetic Algorithms

As GA are not covered in more detail in this thesis, this part shall provide a brief overview (more detailed introductions covering more variations e.g. in de Weck and Wilcox (2010) and Weise (2009)). The GA as a subclass of the evolutionary algorithms (Weise 2009) try to computationally simulate the natural/biological genetic evolution. To do this, the basic unit is the so called chromosome or gene, which typically is a fixed-/variable-length tuple (Weise 2009) of integers, characters or even colors (de Weck and Wilcox 2010) (examples, e.g. Fig. 9a). A number of these chromosomes, the so called population, is then evaluated by a fitness function, which is important to the GA (de Weck and Wilcox 2010) as it evaluates the quality of a gene for a certain aim.

---

25 one example for this is described by Ashlock et al. (2011)
26 Pech et al. (2015) created the algorithm for maze generation - yet visually the generated mazes equal dungeons and the authors themselves mention dungeon environments in their introduction
27 as a complement to chromosomes so called phenotypes exist, which are what the chromosome represents - so as a rough example a 2D grid can be the chromosome of a dungeon
28 in a dungeon could be number of rooms, percentage of accessible area, ...
The resulting numeric value can then be used to select a number of most fit genes/individuals to be reused/move on to a new generation (de Weck and Wilcox 2010). Furthermore the selected or additionally selected candidates\(^{29}\) can be used to replace less fit individuals in two steps:

- **crossover**: two selected genes are split and exchanged to form two new genes (Fig. 9a)
- **mutation**: a certain number of bits/elements in a gene are randomly changed (Fig. 9b)

The above mentioned steps are repeated until the number of selected and/or the newly generated individuals is sufficient to replace the prev. population (de Weck and Wilcox 2010; Pech et al. 2015). Then again, the new population will be evaluated and it will be checked if the resulting Fitness values are sufficient to terminate the algorithm (or if other criteria like a certain number of iterations are met) (Pech et al. 2015). Otherwise another iteration will be carried out.

![Figure 9: GA Offspring Creation](source)

![Figure 10: GA Algorithm Schema](source)

### 3.3.3 Dungeon Creation Algorithm

While the regular CA typically starts from a 'blank page' with randomly distributed rock cells, the Evolving CA use pre-generated maps as an input (Pech et al. 2015). To capture the visual characteristics nine attributes (see Table 9) are used, also illustrated in Figure 11 to clarify some terms\(^{30}\) (potentially some attributes would need to be changed to apply the method for dungeons). Moreover the CA is steered slightly different for the ECA, as it does not use the four variables \(r, n, T, M\). It instead uses the following variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S)</td>
<td>number of cell states</td>
</tr>
<tr>
<td>(N)</td>
<td>size of the neighbourhood used for the CA</td>
</tr>
</tbody>
</table>

Table 8: Attributes for the Cellular Automata of the ECA - Source: (Pech et al. 2015)

\(^{29}\) e.g. Tournament selection, where two or more members of a population are randomly chosen, compared and only the stronger one will move on to the crossover/mutation stage (de Weck and Wilcox 2010; Miller and Goldberg 1995)

\(^{30}\) here it should be noted, that traversable areas are not marked in the picture - yet one traversable area is displayed by all accessible tiles (another traversable area would look similar but would not be connected)
As with the lack of the former threshold $T$ the Cellular Automaton would lack a transformation rule, a new ruleset is introduced for the ECA. This ruleset does not have a fixed number like $T$ to determine how a neighbourhood value is handled, but uses a rule table, that based on a given neighbourhood value is used to decide, how a cell should be transformed (Pech et al. 2015) (these rule tables also make up the genes or chromosomes that will be manipulated by the GA).

Pech et al. (2015) distinguish two types of rule tables or, as they call it - representations - differing in the following aspects (more explanations: Pech (2013) / representations: Ashlock et al. (2011)):

<table>
<thead>
<tr>
<th>Application Case</th>
<th>Direct Representation</th>
<th>Indirect Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output States</td>
<td>simple Cellular Automata, max. two cell states</td>
<td>more complex CA</td>
</tr>
<tr>
<td></td>
<td>for each possible CA neighbourhood configuration</td>
<td>for every possible sum of neighbourhood values</td>
</tr>
<tr>
<td>Output State Representation</td>
<td>list of integers in lexicographic order</td>
<td>list of integers</td>
</tr>
<tr>
<td># Neighbourhood Configurations</td>
<td>$S^{N^3}$</td>
<td>$(S - 1) \cdot N + 1$</td>
</tr>
</tbody>
</table>

Table 10: Comparison of the different representation types for ECA - Source: (Pech et al. 2015)

How the cell transformation is handled can be best shown graphically in a simple 1D CA:

Figure 12: Representations/Rule Tables for the CA in ECA - Source: (Pech et al. 2015)

$S^{N^3}$ neighbourhood size 1, two cell states: $N = 3$ and $S^{N} = 2^3 = 8$ (1D CA); $S^{N} = 2^{(3 \times 3)}$ (2D CA) (Pech 2013)
Constructive Generation Methods for Dungeons

Having discussed these basic elements of the ECA, the algorithm itself can now be discussed:

**Codeline 1 in Listing 4**
The whole procedure starts off with the initialization of the rule tables respectively chromosomes, each cell being set to a random value in \([0, S - 1]\) (for the generation of dungeons two states - path and rock - can already be enough) (Pech et al. 2015).

**Codeline 2-5 in Listing 4**
After the rule tables have been initialized, the evaluation starts by applying the associated CA to a collection (e.g. 100) of mazes/dungeons that have been generated before as a needed input for the CA (Pech et al. 2015). In a next step Pech et al. (2015) use a series of image process techniques to obtain the nine Attributes for ECA without specifying the exact method. Using the extracted attributes, for each generated layout an attribute similarity measure (ASM) is calculated and finally for each individual/ruleset a fitness value \(FV\) is calculated (average of ASMs for each individual):

\[
ASM = \sum_{i=1}^{9} \left(1.0 - \frac{|da_i - aa_i|}{ma_i}\right)^3 \times aw_i \quad \text{and} \quad FV = \frac{1}{\text{#layouts}} \sum_{j=1}^{\text{#layouts}} ASM_j
\]

**Codelines 6-9 in Listing 4**
In the next part of the ECA algorithm a new population has to be generated, where the overall procedure mostly follows the one described in the Overview over Genetic Algorithms.

As a first step, the five individuals with the highest \(FV\) are transferred to the next generation via elitist selection; the remaining 45 candidates for the new generation are found via tournament selection between five individuals and follow-up crossover and mutation (Pech et al. 2015). In difference to what is described for the general GA, crossover is only applied with a certain likelihood (e.g. 60%), which means that for every chromosome a random number generator draws a number in \([0, 1]\) and if it is smaller than the likelihood, crossover takes place (after an additional chromosome is added as two are needed for crossover) (Pech 2013; Pech et al. 2015).

The final step is the mutation of the chromosome (or after crossover: the two chromosomes), where each cell is changed with the probability of \(1 / \text{chromosome length}\) (using uniform mutation) (Pech 2013).

**Codelines 10-12 in Listing 4**
Having created the new population, the algorithm must check, whether one of the following termination criteria is met or whether the CA evaluation and the population generation have to be repeated to further improve the solution (Pech et al. 2015):

- **Convergence**: last 100 consecutive runs did not increase \(FV\) by more than 0.0001
- **max. \(FV\)**: \(FV\) has reached its maximum of 1.0 and cannot be further improved
- **# Generations**: the other conditions are not met, but 5000 runs have been executed

---

32 remember the standard CA that also needs a non-empty initial map-configuration, see section 3.2.2
33 a rather precise overview over the techniques like erosion or region growing can be found in Pech (2013)
34 \(da = \) desired attribute value // \(aa = \) attribute extracted value // \(ma = \max. \) possible value // \(aw = \) attribute weighting factor → more information on each variable in Pech (2013)
35 Pech (2013) and Pech et al. (2015) argue that they took most of the parameters for the GA from De Jong (1975) except the mutation parameter which they adapted to better fit the potentially varying chromosome lengths
Constructive Generation Methods for Dungeons

Listing 4: Evolving CA Algorithm for Dungeon Generation - Source: (Pech et al. 2015)

```
1 initialize population // random value between [0,S-1]; S = #cell states
2 evaluate CA ruleset
3 run CA with collection of perfect mazes/dungeons
4 extract attributes from generated layouts using image process techniques
5 evaluate extracted attributes based on set of goal attributes
6 repeat until new population is created
7 select // elitist top 5 (only first step) + tournament selection
8 crossover // single-point variant
9 mutate // using uniform mutation
10 check if termination conditions are met
11 yes - solution is found
12 no - go to line 2 and continue search
```
3.4 Generative Grammars

While the previous approaches mostly focussed on generating dungeon maps with certain qualities like room count, or a natural look, the Generative Grammar has the intention to allow map-generation based on the gameplay respectively story (idea introduced by Dormans (2010) and taken up by van der Linden (2013)).

3.4.1 Areas of Application, Pros and Cons

In addition to the above mentioned general idea of Generative Grammars, this approach is also meant to improve the ease of use for designers (controllability, see Procedural Content Generation (for Games)) by allowing them to express the dungeon layout in a more high-level design-oriented way (Dormans 2010; van der Linden 2013), instead of more or less ‘cryptic’ variables as for Cellular Automata or Space Partitioning. Similar to what Pech et al. (2015) proposed for the CA, Dormans and Bakkes (2011) point out, that generative grammars can also be modified by evolutionary algorithms. Another PCG property that is satisfied by this approach, is Speed, as Dormans (2010) and van der Linden (2013) both characterize it as performant. Dormans and Bakkes (2011) even propose to use player modelling in combination with generative grammars to adjust the dungeon at play time (space, mission, difficulty,...).

A last advantage, the syntactic correctness, is a side effect of using grammars and according to Dormans and Bakkes (2011) makes correctness tests and selection processes unnecessary. One downside of using grammars is, that (at least at the moment)36 grammar are still game-dependent, which means that two games require two different grammars (Dormans 2010). Dormans and Bakkes (2011) also restrict the idea that grammars make dungeon-generation easier for designers, as they figured out, that the knowledge of generative grammars is often not common among designers. This and the difficulties to foresee the final outcome (Dormans and Bakkes 2011) may lead to what Dormans (2010) calls maps having a ‘random feel’ and ‘lack[ing] overall structure’.

<table>
<thead>
<tr>
<th>Pro</th>
<th>Contra</th>
</tr>
</thead>
<tbody>
<tr>
<td>map-structure and mission related</td>
<td>game-dependent</td>
</tr>
<tr>
<td>easier to use (high-level ‘language’)</td>
<td>designers lack knowledge</td>
</tr>
<tr>
<td>modifiable with evolutionary algorithms</td>
<td>difficult to estimate outcome</td>
</tr>
<tr>
<td>fast and adjustable to player</td>
<td>random/unstructured feel of maps</td>
</tr>
<tr>
<td>syntactical correctness</td>
<td></td>
</tr>
</tbody>
</table>

Table 11: Advantages and Disadvantages of Generative Grammars
Source: (Dormans 2010; Dormans and Bakkes 2011; van der Linden 2013)

3.4.2 Overview over Grammar Types

Like for the previous algorithms, it makes sense to get an overview over the underlying ideas - in this case three different types of grammars, that in combination are used to create a dungeon map.

36Van der Linden et al. (2013) have worked towards a more generic approach, where only the translator from grammar to real content is still game-dependent)
Constructive Generation Methods for Dungeons

**Generative Grammar (Fig. 13a)**
This type can be seen as the 'basic' grammar, which is the basis for the other two types and is derived from linguistics, where among others Chomsky laid the foundation for generative grammars as constructs to produce all correct phrases of a language, based on an alphabet (symbols) and a set of rules (transformation of symbols to other symbols) (Dormans 2010; Hamp 2015; Pavle 2015).

From there it was carried over to Computer Science as the basis for code parsers (Dormans 2010) and is formally defined as a 4-tuple \( \langle V, \Sigma, S, P \rangle \) (also called Backus-Naur-Form\(^{37}\)) (Adams 2002):

- \( V \): is a finite alphabet, called variables/nonterminal symbols
- \( \Sigma \): finite alphabet, called terminal symbols with \( V \cap \Sigma = \emptyset \)
- \( S \in V \) as the start symbol
- \( P \) as production rules of ordered pairs \( \langle \alpha, \beta \rangle \) with \( \alpha, \beta \in (V \cup \Sigma)^\ast \) and \( \alpha \) containing min. one symbol from \( V \)

To work with this, one starts to replace the start symbol \( S \) with one rule from the BNF, e.g. \( S \rightarrow e \ C \ G \ \text{bm} \ g \) and then continues, till all non-terminal symbols \( V \) are replaced with terminal symbols \( \Sigma \), e.g. \( e \ t \ km \ km \ km \ km \ lm \ bl \ g \)\(^{38}\). Combining the above mentioned ideas, it becomes clear that if words and rules are correctly given in some type of BNF, a computer cannot only analyse the language, but also use the given BNF to generate language, respectively missions/stories in case of dungeon generation (Dormans 2010).

**Graph Grammars (Fig. 13b)**
Graph Grammars build on the principles of Generative Grammars and thus are very similar to them, with the difference, that no strings are handled, but graphs with nodes and edges (Dormans 2010). Graph Grammars are used to build dungeons, as they provide more flexibility in rewriting things (see replacement for G in Fig. 13a and Fig. 13b) and as a result are considered to be more suitable to create desirable outcomes (Adams 2002).

**Shape Grammars (Fig. 13c)**
Shape Grammars, originating from work of Stiny and Gips (1972), build on the concepts of Generative and Graph Grammars, but instead of strings and nodes they use 2D shapes (Dormans 2010). The used shapes can represent any type of 2D layout element, like the walls and spaces in Fig. 13c or the test, key and lock rooms on page 6 of Adventures in Level Design by Dormans (2010). Like Graph and Generative Grammars, Shape Grammars work based on rewriting, so e.g. in Fig. 13c every connector could be replaced with one of the given rules to create a larger spatial structure.

\[
\begin{array}{|c|c|}
\hline
\Sigma & e \ [\text{entrance}], \ bl \ [\text{level-boss}], \ km \ [\text{multi-part Key}], \ lm \ [\text{lock for multi-part key}], \ t \ [\text{test}], \ g \ [\text{goal}] \\
\hline
V & S \ [\text{start}], \ C \ [\text{chain}], \ G \ [\text{gate}], \ CL \ [\text{chain linear}] \\
\hline
S & S \ [\text{chain}] \\
\hline
P & S \ ::= \ e \ C \ G \ bl \ g \\
C & ::= \ CL \ t \\
G & ::= \ km \ km \ km \ lm \\
\hline
\end{array}
\]

\(37\) not called that way by Adams (2002), but the given notation resembles the BNF described e.g. in Vahrenhold (2013)

\(38\) intermediate steps: \( e \ C \ G \ bl \ g \rightarrow e \ CL \ t \ G \ bl \ g \rightarrow e \ t \ t \ G \ bl \ g \rightarrow e \ t \ t \ km \ km \ lm \ bl \ g \)
3.4.3 Dungeon Creation Algorithm

The algorithm discussed in this subsection is primarily derived from Dormans (2010) and Dormans and Bakkes (2011), but also incorporates some elements of the approach of van der Linden (2013) and Adams (2002). Basis for this decision was the fact, that only Dormans (2010) and Dormans and Bakkes (2011) do not work towards any kind of implementation and thus focus on the description of the algorithm and necessary background information.

```plaintext
1 define symbols and vocabulary
2 define relationships and constraints
3 transform to graph grammar
4 define shape grammar
5 create min. one shape for each terminal graph node
6 build mission with graph grammar
7 build space/layout with shape grammar
8 for each mission element
9 find suitable shape element
10 place at random, suitable location
11 save reference to mission element
12 compute player model // optional
13 adapt mission/space/difficulty based on player model // optional
```

**Listing 5**: Generative Grammar Algorithm for Dungeon Generation

Source: (Dormans 2010; Dormans and Bakkes 2011; van der Linden 2013)

**Codelines 1-2 in Listing 5**

Right at the beginning, the dungeon designer must identify the basic elements of the dungeon story (and through this step also for the layout). This, in an easy case, comes down to set up a basic generative grammar as in Figure 13a, specifying the core elements, like in a dungeon case rooms, boss-enemies, keys etc. and on top of that basic relations, like a dungeon having rooms, rooms containing enemies and keys.
Constructive Generation Methods for Dungeons

**Codelines 3-5 in Listing 5**
Based on the generated vocabulary and basic rules, a Graph Grammar is created as the basis for the story generation (Dormans 2010). Again, the reason basically is the greater flexibility compared to Generative Grammars, allowing non-linearity and more randomness (Adams 2002; Dormans 2010). As the Graph Grammar will only be the ‘intermediate’ step to generate the dungeon level, Dormans (2010) creates at least one shape for each terminal node in the Graph Grammar to enable the translation from mission to space. Multiple ones would also be possible, then one element would randomly be selected (Dormans 2010), creating greater diversity.

**Codeline 6 in Listing 5**
Before or also after the creation of the Shape Grammar, the specified Graph Grammar is used to generate the underlying story (Dormans 2010). This, as indicated in Overview over Grammar Types, works by replacing the start node (in Fig. 13b) with what is specified in the ruleset. When all non-terminal nodes are replaced with terminal ones (so based on the example in Fig. 13b only grey circles would remain) a mission is created (Dormans 2010). Here it has to be noted that Fig. 13b only represents a very limited example, as typically multiple replacement rules for a non-terminal symbol will exist, to ensure diversity.

**Codelines 7-11 in Listing 5**
Given the mission in Graph Grammar, the dungeon structure will be created by iterating through the (terminal) nodes of the graph, placing a suitable shape into the map for each node (Dormans 2010). Diversity is ensured by randomly selecting the shape from multiple fitting ones and can be increased by using dynamic parameters to influence choice, e.g. to select more complex/difficult shapes in the final parts of the dungeon (Dormans 2010). Keeping references to the mission graph is a guarantee that e.g. a key containing shape will be placed before a gate shape (Dormans 2010).

**Codelines 12-13 in Listing 5**
Dormans and Bakkes (2011) propose not to stop the algorithm/generation process once a dungeon is generated, but to make use of the speed of the method to continuously modify the map. Based on an analysis of the actions and preferences of players, they propose to alter shape grammars or the mission graph dynamically, or to use terminal graph nodes representing different difficulty levels. To bring some of those elements into a given dungeon, Dormans and Bakkes (2011) introduce special non-terminal mission nodes (e.g. ?) that are not replaced in the initial map generation, but can be filled up during a game in the dungeon.

**Differences in the van der Linden (2013) approach to the one of Dormans (2010)**
vander Linden (2013) takes a slightly different approach by using the software Entika to define objects and their relations already including parameters and semantics39. Instead of the general Generative Grammar he uses so-called Gameplay Grammar, defining action and subactions (e.g. Acquire key ::= Kill Enemy → Loot key). This is later translated to an initial graph with all top-level actions, which are then iteratively replaced with subactions. Based on players' actions, they are grouped together, before applying several optimization steps to the resulting graph to get a representation that an existing map generator for the target game Dwarf’s Quest can work with.

---

39 e.g. a player could have a certain level - based on that finding a key could either include fighting monsters or just searching for different treasure chests (van der Linden 2013); more about semantics and Entika in (Kessing et al. 2012)
4 Limitations and Conclusion

Any reader who made it through the previous sections should now have a good overview over a quite diverse set of dungeon creation algorithms. At the same time, he/she most likely will have noticed the often very brief and abstract style in which the algorithms are presented. While this is inevitable for a seminar thesis due to length restrictions, it still represents a first limitation, as a more complete understanding of each of these algorithms will require further reading. The given references constitute a good starting point to gain deeper insights wherever required or desired. Moreover not only the description of individual algorithms is limited - the same holds true for the selection of algorithms presented in this thesis. Only three categories of dungeon creation algorithms made it into this thesis (plus one variation of a category), while a multitude of other approaches is not discussed. Examples include:

- Agent-based dungeon growing (Shaker et al. 2015)
- Relative Placement (Valtchanov and Brown 2012)
- Genetic Algorithms (e.g. Hartsook et al. (2011))
- Constraint-Based (Rodden and Parberry 2004)

And this again limits the overall view on dungeon generation, as it up to this part only considers the scientific point of view. Considering that PCG research strives for practical applicability, existing practical work can also be a valuable source for information, especially for anyone aiming at implementing a PCG dungeon game himself. Some examples cited in this thesis would include Hely (2013) with a tutorial for BSP trees, Simon (2009) with his 7 Year Roguelike project or Pedersen (2014b) with his tutorial on CA.

This thesis has three major contributions: The first one is supposed to be an easy introduction into the concept of PCG and existing constructive generation methods for dungeons. This includes a clarification of the basics of PCG, content, games and dungeons as well as simple descriptions for four algorithms, supported by pseudo-code snippets and illustrating graphics. The explanation of used techniques, as e.g. BSP trees for space partitioning, before explaining their usage in the creation algorithm, shall provide people with no background in Computer Science or related faculties with the necessary toolbox to understand the more specific dungeon creation algorithms.

A second contribution is the provision of information on algorithm application cases, advantages and disadvantages, allowing the reader to quickly compare the explained algorithms in order to find the right one for a given scenario.

The provision of a vast set of references and additional literature is the last contribution of this thesis, enabling the reader to continue learning about algorithmic dungeon creation approaches. Here the “PCG Book” by Togelius et al. (2015b) would be suitable for newcomers to the PCG domain, whereas the cited conference proceedings and papers are rather recommended for more experienced readers.
Constructive Generation Methods for Dungeons

References


Constructive Generation Methods for Dungeons


Preuß, Mike (2015). Search-Based PCG.


A Statistics Regarding Percentage of Gamers in Germany

Anteil der Computer- und Videospiele in Deutschland in den Jahren 2013 und 2014 nach Geschlecht

<table>
<thead>
<tr>
<th>Geschlecht</th>
<th>2013</th>
<th>2014</th>
</tr>
</thead>
<tbody>
<tr>
<td>Männer</td>
<td>42%</td>
<td>44%</td>
</tr>
<tr>
<td>Frauen</td>
<td>30%</td>
<td>39%</td>
</tr>
</tbody>
</table>

Quelle: BIKOM © Statista 2015
Weitere Informationen: Deutschland; ARIS, BIKOM Research; 2014; 1,205 Befragte, ab 14 Jahre