Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelor Thesis

**Sven Freiberg**

**Procedural Generation of Content in Video Games**

*Fakultät Technik und Informatik*
*Studiendepartment Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

# PROCEDURAL GENERATION OF CONTENT IN VIDEO GAMES

SVEN FREIBERG

Bachelor Thesis handed in as part of the final examination

| | |
|---|---|
| COURSE OF STUDIES | Applied Computer Science |
| DEPARTMENT | Computer Science |
| FACULTY | Engineering and Computer Science |
| | Hamburg University of Applied Science |
| SUPERVISOR | Prof. Dr. Philipp Jenke |
| 2ND REFEREE | Prof. Dr. Axel Schmolitzky |
| HANDED IN ON | March 3rd, 2016 |

Bachelor Thesis eingereicht im Rahmen der Bachelorprüfung

| | |
|---|---|
| STUDIENGANG | Angewandte Informatik |
| DEPARTMENT | Informatik |
| FAKULTÄT | Technik und Informatik |
| | Hochschule für Angewandte Wissenschaften Hamburg |
| BETREUENDER PRÜFER | Prof. Dr. Philipp Jenke |
| ZWEITGUTACHTER | Prof. Dr. Axel Schmolitzky |
| EINGEREICHT AM | 03. März, 2016 |

## ABSTRACT

In the context of video games Procedrual Content Generation (PCG) has shown interesting, useful and impressive capabilities to aid developers and designers bring their vision to life. In this thesis I will take a look at some examples of video games and how they made used of PCG. I also discuss how PCG can be defined and what misconceptions there might be. After this I will introduce a concept for a modular PCG workflow. The concept will be implemented as a Unity plugin called Velvet. This plugin will then be used to create a set of example applications showing what the system is capable of.

*Keywords:*

procedural content generation, software architecture, modular design, game development

## ZUSAMMENFASSUNG

Procedrual Content Generation (PCG) (prozedurale Generierung von Inhalten) im Kontext von Videospielen zeigt interessante und eindrucksvolle Fähigkeiten um Entwicklern und Designern zu helfen ihre Vision zum Leben zu erwecken. In dieser Thesis werde ich einen Blick darauf werfen, wie sich Spiele PCG in der Vergangenheit zu Nutze gemacht haben. Es wird diskutiert, welche Defintion sich für den Begriff der Procedrual Content Generation (PCG) eignent und welchen Missverständnissen man begegnen kann. Danach wird ein Konzept vorgestellt, dass die Arbeit mit PCG in einer modularen Art und Weise strukturiert. Danach folgt die Beschreibung von Velvet, eine prototypische Implementierung des Konzepts, umgesetzt als Plugin für die Unity Engine. Es werden weiterhin Beispielanwendungen präsentiert, welche die Arbeitsweise und Möglichkeiten des Plugins veranschaulichen.

*Stichworte:*

prozedurale generierung, software architektur, modulares design, spieleentwicklung

*If we have learned one thing from the history of invention and discovery,*
*it is that, in the long run, and often in the short one,*
*the most daring prophecies seem laughably conservative.*

— Arthur C. Clarke [8]

# ACKNOWLEDGMENTS

I want to really thank my family, especially my father Jürgen Freiberg, who helped me focus on the bigger picture and enabled me to gain back motivation when it was really needed.
I also want to thank Prof. Dr. Philipp Jenke for being an open, interested and understanding supervisor, helping and giving advise even beyond the topic of this thesis.

# CONTENTS

Part I

# INTRODUCTION

This introductory chapter will give you a brief overview of the contents of this thesis and discusses what Procedrual Content Generation (PCG) is and how it can be defined. This is followed by a short motivation why I think PCG is interesting and what my goals with this thesis are.

# 1 SETUP

## 1.1 OVERVIEW

In this thesis, I take a look at Procedrual Content Generation (PCG), what this term means and how it has been and is being used in the context of video games. I will first argue for the use of PCG from the point of creativity as well as buisiness interests. After that I am going give an overview of the goals I've set to archive in this thesis. The following part will be a brief look into the history of pcg in video games. Succeeding this will be a look into some techniques found in PCG and recent research. The two subsequent chapters will be concerned with the concept for a modular workflow for PCG and a prove of concept implementation in the form of a Unity plugin called VELVET. The last two chapters will be the evaluation of my concept and the prototype as well as the conclusions I have come to while writing this thesis.

## 1.2 TERMINI: PROCEDURAL, CONTENT AND GENERATION

Before I will make some notes on why I think it is well worth your time, incorporating PCG in your workflow, let me first try to define, what the termini procedural, content and generation mean in the context of this thesis. In the publication *Design metaphors for procedural content generation in games* [24], the authors define PCG as:

> "the algorithmical creation of game content with limited or indirect user input"

This definition in turn has been attributed to the article *What is procedural content generation?: Mario on the borderline [51]*, in which Julian Togelius et al. looked at what might and might not be considered PCG. The difficulties on how to come to terms with a definition, is in my opinion well summarized in the introduction:

> "PCG has been attempted by too many people with too many different perspectives for this to happen. A graphics researcher, a game designer in the industry and an academic working on artificial intelligence techniques would be unlikely to agree even on what "content" is, and much less which generation techniques to consider interesting." *[51]*

### 1.2.1 *Procedural*

The Merriam-Webster dictionary defines *procedural* as *"of or relating to procedure"* [31] and in turn *procedure* as *"a series of actions that are done in a certain way or order" [32]*. I think it is reasonable to argue that the procedural part of PCG communicates the intent to describe the way something is created, as a sequence of steps, which when fed the same set of parameters, will yield the same result. To get such a set if instructions or steps, any designer or developer concerned with the creation of some part of the game, has to do some introspection on how exactly she proceeds when working on a certain task. The alternative would be to have a more exploratory workflow, where the designer or developer comes up with a basic set of intrustions and works her way towards the vision.

#### 1.2.1.1 *Randomness*

Procedural does in no way imply that it has anything to do with randomness. Although randomly generated values used to initialize the procedural part of PCG are vital for giving it a certain dynamic and unpredictability [1] as well as exploratory feel, randomness is not a necessity for PCG to function. No Man's Sky is a good example of the interplay between deterministic behavior and randomness [2]. It aims to have its universe completely procedurally generated on the fly and not "stored on the disk [...] not stored in the cloud [...] its just generated there and then on the fly. When you fly away we just throw that data away" [15]. In an interview with GameSpot, Sean Murray founder of Hello Games said:

> "The way the universe is create in No Man's Sky, the way everything is created is this term that we use; procedural [...] We're [...] really picky about that, we say it's not random. [...] 'Cause random to me is this [...] chaotic kind of mess, potentially. But procedural is [...] mathematical formulas". [15]

He then goes on to explain how deterministic behavior is used to create the universe in No Man's Sky:

1 In the human sense, not in a machine sense. The argument could be made, that with enough information about the state of the game and the entropy of the Pseudo-Random Number Generator (PRNG), every outcome of a procedure seeded with a value generated by a PRNG could be predicted.

2 At the time of writing this thesis, No Man's Sky has not yet been released, so the conclusion are soly based on press releases and interviews.

> "When you put in the same inputs, which is just: I am here, this point in the world, you will always get the same output. Which is the mountain in that exact same position or the rock that's on top of that mountain, down to the [...] blade of grass. It will always be in the exact same place. If another player comes along and they [...] input the same inputs, they will get the same outputs." [15]

But this behavior is not unique to NO MAN'S SKY. Lots of other games use a similar system. See ELITE in section 2.2.2 or .KKRIEGER in section 2.2.4 for further example applications of Procedrual Content Generation (PCG) with focus on predictability and recreatability.

### 1.2.2 *Content*

The first types of content one might think are of particular interest when it comes to PCG might be traditionally, levels or dungeons, animations or textures. I would go so far as to include all pieces of information [3] to be found in a specific game. This might be partially in context of the mechanics and the game feel. So lets imagine an example where a game would generate characters participating in its story with a different set of traits, which in turn would alter the interaction of said characters, therefor creating a unique story. I would argue that all parts involved in this process, the traits making up the characters and the story unfolding depending on the characters interaction are eligible to be considered content. So in consequence, there are in my opinion no real boundaries to be set on the topic of what is to be considered content or not. Limiting the meaning of the term, could also limit what people consider worth creating and therefor might lose us interesting content to be encountered in research, games and commercial tools to come.

### 1.2.3 *Generation*

The generative part of PCG is where the knowledge on how to create something (the procedural part), gets executed to produce the desired type of content. That means you have to put into place a system which is capable of being fed a certain set of instructions, preferably parametrizable so that it will produce a certain result. This is also the part which describes how the process of creating something is started and where the result is stored.

[3] Even generated code.

#### 1.2.3.1 *Parameter*

An emphasis should be made on the part of being parametrizable. The versatility of Procedrual Content Generation (PCG) comes from its ability to produce a slight or not so slight alteration on the set of instructions it is based on. If you do not allow the procedural part to be parametrizable, this will still in my opinion be PCG, but may be applicable for tackling a subset of interesting problems. One of such sub-problems might be compressing data, in order to save disk or memory space, with the cost of increased computational effort. See the section 2.2.4 about .KKRIEGER for an example of texture compression with the help of PCG.

#### 1.2.3.2 *Starting the process*

To procedurally generate some content, this process has to start somewhere. Be it an actual human or a machine, pressing a button or calling a function. If the processes should be controlled by a developer or designer, she would need a user interface containing at least a start button. An interesting concept is to allow for some sort of feedback loop, where previously generated content could evokes some reaction by actors [4] in a game which in turn yields data capable of being fed into the Procedrual Content Generation (PCG) itself. Therefor if the game would hypothetically be completely simulated, a specific set of start parameters, or a seed, could be used to control the whole dynamic of the game. This can be in my opinion be very well visualized through a game proposed by John H. Conway in 1970 [17] called LIFE (see appendix A.1).

### 1.2.4 *Summary*

So in summary, my definition of Procedrual Content Generation (PCG) in the context of video games is:

> *Using a parametrizable sequence of instructions, yielding the same result when given the same input, to create content which is considered to be an integral part of a specific video game.*

## 1.3 MOTIVATION

A huge part of the time and effort spent developing a game is the conception and production of assets used to give the game a certain audio-visual style. Integrating those carefully crafted assets be it whole levels, single models, music clips, sound effects or textures is crucial and sometimes complex undertaking.

4 Player, AI and environment alike.

This gets quite apparent, when reading different literature mentioning the creative and technical effort needed to get a vision from an idea to integral part of a game. In the book *The Game Asset Pipeline* the author states:

> "Just managing the creation of content and getting that content into the right place at the right time is an enormous challenge. Hundreds of development hours are lost dealing with asset and pipeline issues, so the need for a working system is immense." [6]

Eric Lengyel description of a game asset pipeline from his book:

> "The asset pipeline takes care of obtaining the data from the tools used for their creation, and then optimizes, splits or merges, converts, and outputs the final data that can be used by the game engine." [26]

### 1.3.1 *Creativity*

Seeing how the life-cycle of a game asset is rather long, it is imaginable that feedback processes are also long. I think Procedrual Content Generation (PCG) could proof a valuable piece in the endeavor of crafting game assets. On the one hand providing a tool for the developers and designers to automate small, repetitive tasks. On the other hand, it may open up a more exploratory way of approaching design through working together with an algorithm to create content.

### 1.3.2 *Business interest*

In the previous paragraph, I talked about the effect PCG could have on the workflow for designers and developers. In the article *Guest Editorial: Procedural Content Generation in Games* the authors state:

> "In many contemporary game productions, creating all this game content requires a significantly larger effort and expense than the actual programming of the game." [52]

If you look at it from a financial point of view, it might be interesting to explore the use of Procedrual Content Generation (PCG) for the goal of cutting time. This might free up human resources, to be used in other areas or simply save money and shorten development cycles.

## 1.4 GOALS

With this thesis, I want to take a look at how PCG has been, and is being used in the context of video games. Furthermore I would like to propose a concept and prototypical implementation for a modular procedural generation workflow. This will be realized partly as portable dynamically linked library implemented in C# as well as a prototypical implementation of a Unity plugin called Velvet. While developing this plugin, I will take a look at how to build a basic user interface to control the generative part of PCG as well as how to create a parametrizable system for procedural part. The focus of Velvet should be:

### 1.4.1 *Modularity*

One of the first contacts I had with the concept of modularity was the promotion of composition over inheritance in the *"Gang of Four" [16]* book about design patterns. While the focus of the authors was mainly the design at the level of code, I would like to also take a look at a conceptual or workflow level. Especially the concept of breaking down the procedural part of Procedrual Content Generation (PCG) modules. One early publication on how to decompose a system into modules in the context of software development can be found in the paper *"On the Criteria To Be Used in Decomposing Systems into Modules"* [37]. There the authors state two advantages of modular programming:

(1) allow one module to be written with little knowledge of the code in another module, and

(2) allow modules to be reassembled and replaced without reassembly of the whole system.

These two principles are almost exactly the vision I had in mind for the concept. So with this thesis, I want to discuss a design for a system which adheres to those two principles. Although I would like to strive for the change from *little* to *no* knowledge of code in another module. It should be possible for someone to develop a module and not having to worry that another module having side effects. The two points mention earlier, are in my opinion a very good guideline when it comes to modular design.

But I think a very important fact is revealed in the conclusion of the aforementioned paper *"On the Criteria To Be Used in Decomposing Systems into Modules"*:

> "[...] it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution [...]" [37]

My interpretation of what this conclusion tries to communicate, is the importance of modules being part of the overall design; concerned with solving a specific sub-problem. A module has to be just independent enough to not be a static part of a flow chart, bearing the risk of introducing side effects, or sharing of information between modules, leading to inter-dependencies and therefor undermining point 2 in the list above.

### 1.4.2 *Chainability*

Modules should take an input and provide an output of the same data type. This allows for any type of module processing the same data type to be assembled in a sequential order.

### 1.4.3 *Nestability*

It should be possible to nest one or more modules in order to archive a tree like data structure, holding the information about the procedural part of PCG.

### 1.4.4 *Integration with Unity*

The concept should be integrated into the eco-system of the Unity engine and its tools.

### 1.4.5 *User interface*

When working in Unity it should be possible for a developer or designer to interact with a user interface specifically tailored to what ever generation she might want to do.

### 1.4.6 *Parametrization*

When creating a chained and nested structure of modules, it should be possible for a developer or designer to pick certain parameters of any module in any arbitrary depth within the tree structure and bind it on the top level. This would greatly increase the ability to control the generation part of PCG.

## 1.5 CHALLENGES

When designing the architecture and developing Velvet, I expect to come across a set of challenges, which would be amongst others:

- Abstracting the work done to process the modules so that the user can focus on writing the code needed to create her vision.
- Creating a system allowing me to cater to designers (visual interface) and developers (code interface) alike.
- Which value does the system bring to its user?

# 2 A BRIEF HISTORY OF PCG

## 2.1 PROLOGUE

In the past games often used Procedrual Content Generation (PCG) to generate parts of the level. This might have been motivated by the memory limitations encountered while developing games in the 1970 and 1980. But it certainly was not the only reason as we will see in the following descriptions. In this part I want to discuss some of the games which in my opinion are relevant in the context of PCG and show a broad application of it. This is not an exhaustive list and as of 2016 there are certainly a lot of games making use of PCG in one way or another. But I will limit this discussion to eight of them.

## 2.2 LOOKING AT SOME EXAMPLES

### 2.2.1 *Rogue*

At the start of this list is ROGUE (see figure 2.1). Arguably one of the most influential games when it comes to not only procedural dungeon generation[1], but also a whole slew of game mechanics [2] today referred to as *rouge-like* (see 2.2.1.1). First developed by Michael Toy and Glenn R. Wichman around 1980 [59], ROGUE is what today might be considered a dungeon crawling game. The story was not really the focus, but it had a to offer a considerable amount of depth when it comes to the use of items such as armor or weapons to combat the 26 different enemy types [53]. To beat the game the player has to explore the dungeon all the way down to the last level. Each layer or level is procedurally generated, initialized by a random set of parameters for every new character. Each level in turn is composed of rooms, connected by corridors. This unique combination of game mechanics and procedural level generation led to the growing of a rich fan base, which is still active today [3].

*One enemy for each character in the English alphabet.*

1 Some suggest [42] that the game called Beneath Apple Manor (BAM) preceded ROGUES effort of procedurally generating dungeons by 2 years. Curiously BAM never got the some level of attention ROGUE received.

2 Including but not limited to turn based movement and actions, and permanent death of a character.

3 http://web.archive.org/web/20160120145301/http://www.roguebasin.com/index.php?title=Main_Page

Glenn R. Wichman, the co-author of the original version states in his essay *A Brief History of "Rogue"*:

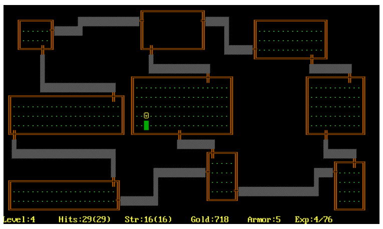


Figure 2.1: Creative artistic rendering of the appearance of Rogue

"Version 4.2 of BSD UNIX included Rogue – suddenly, the game was available on university computers all over the world. At the time, there was no other game like it. Over the next 3 years, Rogue became the undisputed most popular game on college campuses." [59]

He attributes the initial success to the rather unusual approach of not having the game developer decide what the level would be like, but instead "[...], the program itself should "build the dungeon", giving you a new adventure every time you played, and making it possible for even the creators to be surprised by the game" *[59].*

### 2.2.1.1 *Rogue-like*

This has originally been a description for a game true to the original style and rule-set envisioned by Rogue. Namely the procedural dungeon generation, permanent death of a character, as well as the turn based movement and combat. In recent years games developer and media alike adapted the term for games featuring one or more of its core mechanics. An example of a game described as rogue-like, despite having very little in common with the original, would be the game FTL[4]. It plays in a sci-fy setting and features pausable ship to ship combat and crew management as well as permanent death.

4 It is has been tagged as *Rogue-like* on a major digital marketplace called Steam [55].

### 2.2.2 *Elite*

Another great milestone not only for its use PCG, but also for its revolutionary visual style is ELITE [5]. It used wire-frame 3D graphics [6] with hidden line removal [7] to present the player with a vast virtual universe to explore. The game-play is based on a sophisticated flying system, which includes docking to space stations, scooping fuel from stars or traveling between solar system via hyper-drive. The player is able to earn credits via hauling goods and trading, head hunting pirates or flying missions for different factions. I previously (see 2.1) talked about games making an effort towards Procedrual Content Generation (PCG) due to memory limitations. ELITE certainly was one of them. In the book *Backroom Boys: The Secret Return of the British Boffin* the author describes the ambitious struggle of the developer to build a vast and compelling universe for the player.

> "Their first idea had been to furnish the machine with the details of (say) 10 solar systems they'd lovingly handcrafted in advance: elegant stars, advantageously distributed, orbited by nice planets in salubrious locations, inhabited by contrasting aliens with varied governments and interesting commodities to trade. But it quickly became clear that the wodge of data involved was going to make an impossible demand on memory." [47]

How impossible this demand really was, has been stated by David Braben in an interview [3] with GameSpot:

> "We wanted a huge world, but we had 22K of memory–which is probably even less than a single Frontier icon today."

#### 2.2.2.1 *Generating the universe*

With that amount of information the developers of ELITE wanted to use in any given solar system, they needed a technique, allowing them to get a predictable result when generating a system, base on as little input information as possible. To archive this David Braben and Ian Bell developed a system based on the Fibonacci series. Starting with the first two digits in the series, all subsequent digits can be predicted and are consistently recreatable. To conquer the habit of the numbers Fibonacci series two grow in size very fast, they simple dropped all digits in the generated numbers, except the last one.

5 https://web.archive.org/web/20100127094607/http://frontier.co.uk/games/elite
6 Only the edges of a 3D model are rendered.
7 The edges in a wire-frame model, covered by surfaces facing the viewer are not rendered.

So if we would start with $[1,2]$, generating 6 additional numbers, the resulting sequence would be $[1,2,3,5,8,3,1,4]$. I provided an simple implementation in then appendix A.1. This simple yet beautiful system, allowed Braben and Bell to reduce the size of input information they needed to encode any given solar system, to to only two integers [47].

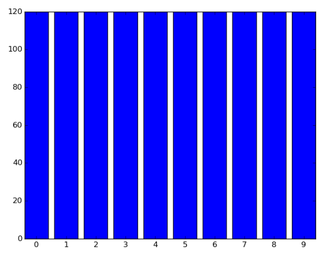


Figure 2.2: Distrubtion of each number from 0 - 9

Analyzing the behavior of this function (see listing A.2) by creating 100 lists each with 12 numbers (two initial and 10 subsequent) on the basis of the permutation of $a$ and $b$, where $0 \leq a \leq 9$ and $0 \leq b \leq 9$, this number generating function appears to have a uniform distribution of occurrence for each number $i \in [0,9]$ (see figure 2.2). This uniformity lead to a very noisy generation, which is why the 256 system featured in Elite were - although initially procedurally generated - hand picked and reviewed, to make sure they adhered to the standards [8] set by the publisher.

8 "One of the first galaxies we tried had a system called Arse. We couldn't use the whole galaxy. We just threw it away!" [47]

### 2.2.3 *Diablo*

Aside being well acclaimed for its tense atmosphere and sound design, Diablo [9] might be one of the most successful *rogue-like* (see 2.2.1.1) games of contemporary history. Diablo got initially released in 1996 and lead the player to Tristram, a plagued town haunted by the devil himself. The goal of the game is to descent through all dungeon levels; ultimately entering hell, to fight Diablo in person. The PCG nature of Diablo has since then been part of its franchise.

To get a slightly different play-through for each character, the developers wanted diverse dungeons. The map generation system, implemented in Diablo turned out to be a mix of static and generated content. Prefabricated rooms or parts of rooms [10] and procedurally generated ones, as well as an entry and an exit to a dungeon level. Controlled by a seed value, these parts then got assembled to build a coherent dungeon for the player to explore. On his way down towards hell, one could encounter a variaty of items, ranging from weapons to scrolls and armor. These items where also procedurally generated. Picking for example a rarity, type of weapon and its stats, the algorithm then figured out a name describing the assembled parts by picking words associated with each characteristic.

### 2.2.4 *.kkrieger*

In the introduction (see 1.2.3.1) I briefly mentioned, that some games used PCG to decrease their memory footprint. An interesting example of this was shown by the award winning German group of demo-makers called Farbrausch [11]. In 2003, they released a game called .kkrieger [50] under their subdivision named .theprodukkt. Through the clever use of procedural texture generation at runtime based on a custom format, they were able to reduce the size of the application to only 96kB [12].

9 http://web.archive.org/web/20160224123017/http://us.blizzard.com/en-us/games/legacy/
10 Important for placing quest items.
11 http://web.archive.org/web/20160113194346/http://www.farbrausch.de/
12 Quote from the read-me of .kkrieger *"A kilobyte is, historically, defined to be 1024 (2^10) bytes, not 1000. Thus .kkrieger is a game in 96k even though it's actually 98304 bytes."*

### 2.2.5 *Dwarf Fortress*

A procedurally generated and incredibly detailed simulation of different fantastical civilizations, among which the player takes the role of the overseer for a dwarfen society, can be found in DWARF FORTRESS [13]. Still under development, the game first appeared in 2006 in the form of an alpha version. The PCG in DWARF FORTRESS is structured into layers, splitting responsibilities and promoting emergent game-play. Emergence [14] can lead to fascinating results in complex systems, such as software and has also been subject to research, trying to providing a way to identify which parts of a system are more likely to cause emergent properties [49]. At the beginning of a session the player can specify a few parameters and let the game generate a tile based map. Each tile has details on e.g, its elevation, temperature, vegetation, savagery [15] or its alignment to either good a evil.

The game then simulates the evolution of the map. Rivers will spring, mountains erode, volcanoes may spew lava and civilizations may be brought to life. Each will be tested against the parameters specified and a set of internal rules, to make sure the map is sufficiently playable. Should the test fail, the progress is rejected [60] and the algorithm will try another approach. After the map generation, the game will then create a fictional history, based on the generated map its native civilizations. Over a previously specified range of in-game years civilizations will grow, fight each other, or starve to death leaving behind abandoned towns and fortresses.

### 2.2.6 *Left 4 Dead*

In the online multiplayer game LEFT 4 DEAD [16] released in 2008, you take on the role of one of four survivors in a post-apocalyptic world overrun by zombies. The goal of each level is for the survivors to reach the exit. On the way to the exit, the players will have to fend of different kinds of zombies. One of the key features of the game is what is called the Director AI. It is responsible for procedurally generate unique narrative structure for each play-through of a level. By monitoring the states of the survivors, the AI then sets spawn points for zombies and places items, to modulate the flow of the game.

13 `http://www.bay12games.com/dwarves/` (visited Feb. 2, 2016)

14 The concept of finding properties in a system which are not associated with any component of said system.

15 Inidicating how 'wild' a tile is.

16 `http://web.archive.org/web/20160223063915/http://www.l4d.com/blog/`

### 2.2.7 *Spore*

Also released in 2008, SPORE [17] is a game where you manage and evolve a species from the beginnings, a single cell organism, through to a space faring civilization. Over the time you gain evolution points which can be spent to evolve certain traits of your species, giving it two more legs, a set of horns or changing its eating habits from carnivore to herbivore. When designing your creature the game procedurally generates a set of animations, like walking, fighting or dancing and generates a set of texture to give the creature a unique look [11].

In later stages it is also possible for the player to create buildings and vehicles, which will also be animated and textured by the system. Instead of using a flat landscape, the player is placed on a procedurally generated spherical planet [9]. Those planets are then spread in a generated galaxy by using a specially engineered technique based on pseudo-random sequence [61]. Not only the behavior and animation of the creature gets procedurally generated, the background music is also depending on the parts your creature is build of. In an interview the executive producer Lucy Bradshaw describes the system:

> "One of our original visions . . . was to do procedural music, [which we achieved with help from electronic musician Brian Eno]. So, as you create your creature in the editor, if you're putting on a more aggressive part, the music starts to turn a little more ominous. If you're putting on a more socializing part, it turns a little more perky and happy. And that happens throughout the game, in fact." [33]

17 https://web.archive.org/web/20160227170036/http://www.spore.com/getSpore

### 2.2.8 *Minecraft*

Certainly one of most well known games featuring PCG is MINECRAFT [18]. First released as purchasable alpha build in 2009 and having its full release in 2011, MINECRAFT is a open-world sandbox game featuring a voxel [19] based environment. Starting the game, the player is asked to either provide a seed or let the game choose one randomly. This seed is then used to create the blocks used to form the game world. The landscape comes in different bioms, each with their own unique set of blocks. A block could be for example a piece of dirt, sand, stone or a flower. It not only is very popular with gamers, but also makes frequent appearance in scientific publications as well. From analyzing the player behavior in the game and correlating motives in real life [5], using it as a basis for researching Massively Multi-user Virtual Environments (MMVE) [12] to enabling *"children and adults to co-design within an environment regardless of their location"* with KidCraft [57].

18 http://web.archive.org/web/20160215173801/https://minecraft.net/

19 A Voxel represents a value on a regular grid in three-dimensional space; contrasting a Pixel as value on a two-dimensional regular grid.

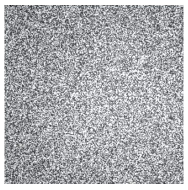
# 3 COMMON TECHNIQUES

## 3.1 PROLOGUE

In this chapter we look at some techniques frequently used when working with PCG.

## 3.2 MAKING NOISE

In computer graphics, noise is commonly associated with image processing techniques or the procedural generation of textures. The term noise basically describes a function, often based on statistical methods. The simplest noise is generated by using a PRNG with uniform probability for all its possible values independent of their position. When interpreting the values as heights and coloring them from black when maximal to white when minimal and projecting it to a texture, it generates the look typical for the so called White Noise (see figure 3.1a). Every value or pixel in this texture is independent from its neighbor. Some of these functions are design to represent a noise space, which in the case of textures would be two-dimensional. So the function is capable of producing values depending on the two-dimensional coordinate provided to it. When trying to create a landscape for example, it would be beneficial to have a more smooth transition from pixel to pixel.


(a) White Noise

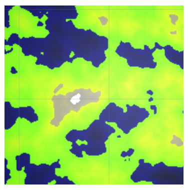
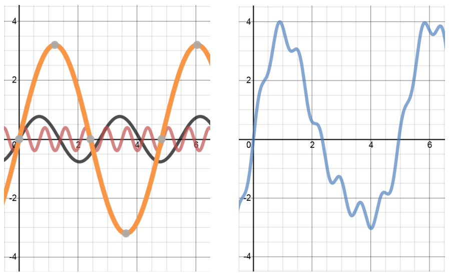

(b) Colored Perlin Noise

Figure 3.1: Noise Textures

COHERENT NOISE One such type of noise would be *Perlin Noise* named after Ken Perlin, who proposed this algorithm in his 1985 paper *An Image Synthesizer* [38]. Values in Perlin Noise are coherent; every pixel has a relation to its neighbor. This allows for parts of the noise space to form recognizable structures. The range of the values $v$ produced by the noise function is commonly $0 \leq v \leq 1$.

SUPERPOSITION To get more control over the structures produced in Perlin Noise, we might use superposition. When superimposing noise, or signals, the amplitudes of the different sources get added together. For example we could use a set of three octaves (see figure 3.2a). An octave is the same function but with increased frequency and decreased amplitude. Taking the first octave for generating a rough outline of the terrain. Then adding the second and third octave to give the graph more details.



(a) Three octaves of sine noise.

(b) Superposition of three sine octaves.

Figure 3.2: Superposition of Noise

Figure 3.1b shows an example where I interpreted the noise values produced by *Perlin Noise* as a height $h$, coloring them black to blue for water in the range of $0 \leq h \leq 0.4$ and green to yellow for flat lands in the range of $0.4 \leq h \leq 0.8$. Finally gray for mountains when $0.8 \leq h \leq 0.95$ with white as the mountain top when $0.95 \leq h \leq 1.0$, yielding a natural looking textures of what could be seen as a landscape. This approach is used in the terrain generation example (see section 7.3).

## 3.3 L-SYSTEMS

In 1968, Aristid Lindenmayer wrote about *Mathematical models for cellular interaction in development* [27], to help describe the complex behavior of cells in plants. In the paper, she presents a formal system today known as Lindenmayer-System (L-System). It uses a set of rules, describing how objects should be rewritten after a generation. Originally Lindemayer used her system to model the growth of red alga, Callithamnion roseum Harvey. A simple example of her system would be a set of rules rewriting two variables.

| | |
|---|---|
| VARIABLES | $L\ R$ |
| START | $L$ |
| RULES | $(L \to LR)$, $(R \to L)$ |

Table 3.1: Simple L-System rule set

When creating a new generation every L gets rewritten as LR and every R gets rewritten as L. When applying five generations of rewriting we get:

| GENERATION | STATE |
|---|---|
| 0 | L |
| 1 | LR |
| 2 | LRL |
| 3 | LRLLR |
| 4 | LRLLRLRL |

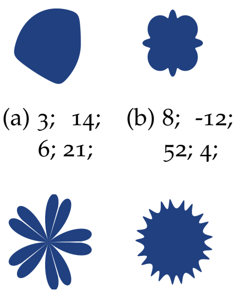
Table 3.2: Five L-System generations

If you allow for parametrization in a L-System, it may be referred to as parametric L-System. Contemporary research shows further interesting uses. In *L-system based interactive and lightweight web3D tree modeling* [43] Hang Qi et al. presents a 3D Virtual Reality web-application, which allows the user to interactively model a modified version of a parametric L-System to describe the generation of trees. A similar concept to the rewriting of an L-System are shape-grammars. They are often used in the procedural generation of buildings (see section 4.3).

## 3.4 SUPERELLIPSOIDS, SUPERQUADRICS AND THE SUPERFORMULA

*Superllipses are also known as Lamé Curve.*

Superellipses are geometric figures in a Cartesian coordinate system formulated by Gabriel Lamé. The shapes can range from a rectangle with rounded edges, a rhombus to a four-armed star with concave sides. In the 2003 publication *A generic geometric transformation that unifies a wide range of natural and abstract shapes* [18] author Johan Gielis described a generalization of Superellipses (see figure 3.4) capable of producing *Supershapes* (see figure[1] 3.3). The shapes generated by his so called *Superformula* are based on his assumptions that "many geometrical forms, both in nature and culture, can be interpreted as modified circles" [18]. Superellipses and Supershapes both exist primarily in two-dimensional space. If you elevate Superellipses into three-dimensional space, you get what is commonly called a Superquadrics. There has been some research on using the wide variety of shapes produced by the Superformula to produce three-dimensional models, showing promising shapes [19]. The upcoming game No Man's Sky features the Superformula [25] to procedurally generate shapes to populate its universe with flora and fauna derived from a single seed value. Supershapes will also be featured in one of the examples in the form of Supercylinders (see 7.2).



(a) 3; 14; 6; 21; (b) 8; -12; 52; 4; (c) 10; -6; 6; -1; (d) 21; 9; 5; 16;

Figure 3.3: Supershapes (m; n1; n2; n3; / a = b = 1)

$$r(\varphi) = \left( \left|\frac{\cos\left(\frac{m\varphi}{4}\right)}{a}\right|^{n_2} + \left|\frac{\sin\left(\frac{m\varphi}{4}\right)}{b}\right|^{n_3} \right)^{-\frac{1}{n_1}}$$

Figure 3.4: Superformula in polar coordiantes

1 Shapes were rendered with a tool I developed available at `http://web.archive.org/web/20160206142844/https://github.com/BlurryRoots/js13k-2015/releases/tag/1.0.1`

## 3.5 BÉZIER CURVES

Named after Pierre Bézier, a french engineer using curves based on the Bernstein polynomial to design car frames for Renault in 1962, a Bézier Curve (see figure 3.5) consists of two nodes $P_0$ and $P_1$, potentially augmented with a set of control point.

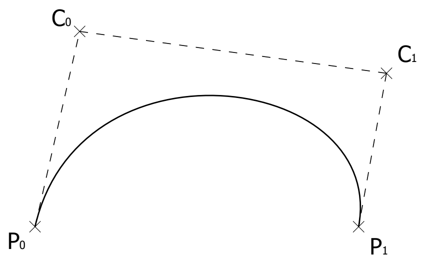


Figure 3.5: Concpet of a Cubic Bézier curve

If present, control points [2] are used to give the curve between the two nodes its curvature. This is done via linear interpolation between all points. If we would use no control points, the curve would just consist of two nodes, which would result in a straight line, sometimes called a *Linear Bézier Curve*. Obtaining a position on this curve is done via linear interpolation. The position $R$ can be obtained by specifying a normalized fraction $t$ between $P_0$ and $P_1$ in the interval $0 \leq t \leq 1$. The result can be calculated via $R(t) = P_0 + t(P_1 - P_0)$ which can also be simplified to $R(t) = t(1-t)P_0 + tP_1$. For example, when specifying $t = 0.5$ we will get the point $R$ which has the same distance to $P_0$ and to $P_1$; right in the center between the two points. By using one control point, we get what is called a *Quadratic Bézier Curve*. When using two control points $C_0$ and $C_1$, we get what is commonly known as a *Cubic Bézier Curve*, which is used in the Supercylinder example (see section 7.2). The position $R(t)$ can be calculated as shown in figure 3.6.

$$R(t) = (1-t)^3 P_0 + 3(1-t)^2 t C_0 + 3(1-t)t^2 C_1 + t^3 P_1$$

Figure 3.6: Simplified Cubic Bézier Curve

2 Sometimes called handles.

# 4 APPLICATIONS OF PCG

## 4.1 PROLOGUE

Not only game developers are interested in the problems and capabilities which accompanies Procedrual Content Generation (PCG). In this chapter I want to take a look at some examples from current research and some commercial applications using PCG. Rather than exhaustively listing all current research, I picked a set of references I considered interesting in the context of PCG in video games and for the topics discussed in this thesis.

## 4.2 VEGETATION

Traditionally based on recursive models, like fractals [35] and rewriting systems like the L-System [41], procedural generation of vegetation, especially trees, has been studied considerably. More recent approaches, discuss algorithms capable of taking things like free space and and available light into account when growing a tree or shrub [36]. There is also some research in systems capable of analyzing already existing models of trees and "allows developmental stages to be generated from a single input and supports animating growth between these states" [39]. It is even shown how environmental factors like wind can be used to shape such developmental stages of growth [40].

## 4.3 BUILDINGS AND CITIES

Current research has shown interesting applications of PCG techniques to generate urban environment, cities and buildings. Citigen [1], proposed in a 2007 paper [22] by George Kelly et. al. and later developed in more detail in his thesis [21] focuses on a tree step generation process. In the first iteration a rough street network is created, followed by a more detailed iteration to create subsection road network and lastly the generation of the buildings populating the spots between the streets. Generating buildings based on a modified version of L-Systems called shape grammars - using geometric shapes instead of variables - has also shown promising results [58]. Approaching the generation of buildings by reinterpreting a 2D city map with a straight skeleton algorithm has also been examined [48].

1 http://web.archive.org/web/20150428035052/http://www.citygen.net/

Using a combination of modular building parts procedurally assembled at runtime and alternatively baked into a new model, if the number of instance would be to high [20] has been presented by the development team behind the second title in the *Mirrors Edge* series as an effective approach to generate city landscapes.

## 4.4 NARRATIVE

Procedurally generating stories, interaction of characters or even narrative puzzles in games have been subject to research. In a 2012 publication [13] the authors describe a system called *Puzzle-Dice* capable of generating puzzles in the style of adventure games, where the player has to interact with objects or characters to progress the story. Procedurally generating whole stories with believable characters by using roles and constraints [7], as well as working together with an algorithm to write stories [44] have also been subject to recent studies.

## 4.5 ANIMATION

Procedurally generated animations are used for example by particle systems to visualize smoke or fire in real-time or animate cloth and hair. It is also possible to animate whole characters. Giving for example, a biped character "life-like, responsive, and non-repetitive" [23] animation has been also subject to research in the field of procedural generated animations. One prominent example would be the creature generator in SPORE (see section 2.2.7), where procedural animation was used to bring user created characters to life. But not only in the context of video games are procedural animations showing promising applications. Film studios have created solutions for animating large numbers of digital actors. Engineers from Walt Disney Animation Studios published a paper describing their technology [4] used in the recently released movie *Big Hero 6*. Another example would be the crowd-related visual effects technology used in the film *Lord of the Rings: Return of the King* called MASSIVE. This software is capable of, based on pre-recorded animation clips, procedurally animating thousands of independent actors. These approaches and technologies used in film, might also be interesting for the use in video games.

## 4.6 MUSIC

Music in video games is an essential part of storytelling and often conveys the mood of a character or setting. Using procedurally generated music to underline a certain mood has been examined by a publication [46] from 2014. In it the authors describe a game which uses procedural music to foreshadow story events. In many video games, it is also important to keep the player engaged and not have her loose immersion by repetitive looping music. I a 2012 publication [2] the authors take a look at techniques to introduce variance into game music.

## 4.7 TOOLS

One notable software would be Houdini, which tries to combine many already existing Procedrual Content Generation (PCG) techniques into one software. These techniques include procedural modeling, as well as animation, particles and physics. Developers of Houdini also published a paper [62] about the role of procedural generation in the visual effects pipeline in game development and describing the concepts behind the PCG system used in Houdini.

Part II

# VELVET: A MODULAR PCG CONCEPT

In this chapter, I will present a design to a modular workflow for PCG. Starting with a short introduction on where I got the inspiration for the concept and what problems I tried to address, I proceed with a description of the core parts of the design. To evaluate the capabilities of the proposed design, I will describe a prototypical implementation in form of a Unity plugin.

# 5 CONCEPT

## 5.1 PROLOGUE

The inspiration for this concept has mainly been drawn from a blog post titled *Procedural Content Generation: Thinking With Modules* [14] by developers at Dejobaan Games. In this article the authors present the way they structured their Procedrual Content Generation (PCG) code base for the *AaaaaAAaaaAAAaaAAAAaAAAAA!!!* [1] series. They highlight a major problem, when working with the premise of creating a sufficiently expressive algorithm, to aid them in their endeavor to procedurally generate content for their game.

> "It's really easy to create an algorithm that generates a simple level – but as we made things more complex, implementation became disproportionately more difficult." [14]

To solve this problem, they tried to break apart the different elements of their procedural generation process into different modules; each with their own decoupled responsibility.

> "In the three years we've been working on Ugly Baby, we haven't solved all of these problems, but we've had some success when combining simple, modular concepts to produce complex results." [14]

With this motivation, I want to present my approach to a modular PCG system. Additionally I will focus on the ability to build composite structures with these modules. Furthermore I want to find a way to parameterize modules as well as the composite structures. The system will be build as plugin for the Unity Engine (see chapter 6).

1 See company page for further details `http://web.archive.org/web/20160115155547/http://www.dejobaan.com/`

## 5.2 ARCHITECTURE

Every part of the blueprint for your procedural generation is contained within a *module*. A set of modules can be arranged sequentially called a *chain*. *Chains* themselves are to be considered a *module*, so they too can be a part in another *chain*. In the end, this system produces a graph data structure, in particular a tree of modules. Each module implements a common function to yield its result. This design is also known as the *composite design pattern* (see figure 5.1). The motivation to use this pattern is its transparency. While traversing

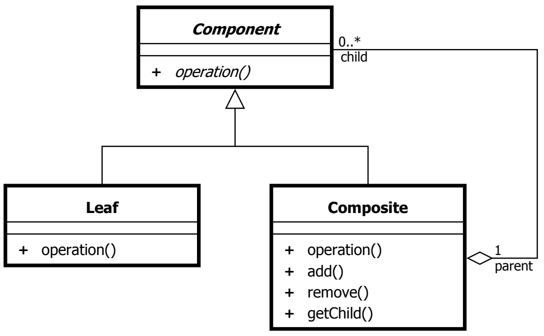


Figure 5.1: UML class diagram for the composite design pattern

the tree to process all its nodes, the part responsible for starting the processing, does not need knowledge of the particular type of node it processes. A leaf node simply produces a direct result, while a composite node would have to start a new traversal process to eventually yield its result. The tree of modules can be traversed and therefor processed with different strategies (see section 5.2.4). Each module should manage a certain task. While developing the system I found several categories with a specific set of responsibilities which will be described in section 5.2.1.

### 5.2.1 *Module types*

GENERATOR A generator module is responsible for generating any unit of information used in the system to eventually assemble a piece of content. This could be for example a the generating a noise texture, create/load a sound effect, creating set of vertices, to later be combined into a model or simply instantiate a set of pre-designed models.

COLLECTOR A special type of generator is called a collector. A collector module is responsible for taking a set of units of information, e.g. a set of game objects and creates a new unit of information with them, e.g. a root game object containing all the previous generated game objects.

SELECTOR Selector modules are responsible for taking units of information and select a subset of this information according to the parameters provided by the user. This might be a sub-model of a composite model, a vertex, a certain texture or an instrument in a modular piece of music.

MANIPULATOR The job of a manipulator module should be the transformation of units of information provided to the module. This might be the translation of a vertex, the rotation of a model, the recoloring of a texture, or pitching of a sound effect.

### 5.2.2 *Chain link*

The chain element is defined as a generic interface (see figure 5.2). The motivation for generically typing the input and output value of the process function is to allow basically any type of data to be processed by a module implementing the `IChainLink` interface. Although generic, the only type being processed in this thesis is a list of Unity game objects.

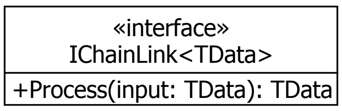


Figure 5.2: UML IChainLink

### 5.2.3 *Chain*

The `Chain` is used to link elements together. A chain itself implements the `IChainLink` interface. The link method adds a new element to the chain. The process method, will either produce a direct result or call the process method of each child module contained by this node. While processing, the results from the previous module gets fed into the next module. The method can be called with a start value which will be provided to the process method of the first element.

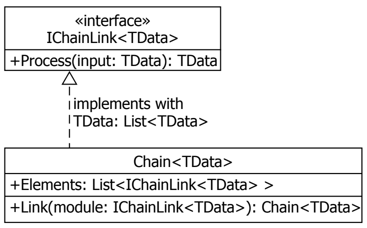

«interface»
IChainLink<TData>
+Process(input: TData): TData
implements with
TData: List<TData>
Chain<TData>
+Elements: List<IChainLink<TData> >
+Link(module: IChainLink<TData>): Chain<TData>

Figure 5.3: UML Chain

### 5.2.4 *Processing Order*

Each chains can be configured to process its modules in two ways.

| | |
|---|---|
| Pre | Processes its own modules first<br>and funnels results into nested modules. |
| Post | Processes nested modules first<br>and funnels results into own modules. |

Table 5.1: Options for processing order enumeration type

# 6 IMPLEMENTING THE UNITY PLUGIN VELVET

## 6.1 PROLOGUE

Beginning with a few words on working with the Unity Engine and how it is structured. This will be followed by short introduction on what Entity-Component System (ECS) are. After this, I will proceed with describing the usage and inner workings of the prototypical plugin implementation of the concept described in the previous chapter called VELVET.

### 6.1.1 *Unity*

First released in 2005, Unity[1] gained a lot of traction and is by today one of the leading technologies when it comes to game development. It provides a complete set of tools for developing interactive 3D as well as 2D application. Unity bundles a custom written engine with a useful set of tools for managing assets and streamlining the build process. It also comes with a powerful visual editor. You can extend the basics of the game engine as well as the editor tools, asset pipeline or even the build process used to package your application. Currently there are 23 target platforms supported by Unity.

COMPONENTS When working with Unitys game engine, most of your code directly relating to Unity functionality is maintained in *components*. This is because the engines architecture is based on the concept of an Entity-Component System (ECS) (see section 6.1.2).

When creating a component, you inherit from a base class called `MonoBehaviour` [2]. Classes inheriting from `MonoBehaviour` can be associated with a game object and presented in the inspector window (see section 6.4) of the visual editor. When compiling the scripts, Unity looks for the presence of a set of methods with specific signatures. To, for example, execute code when a component is create, you would implement the method `void Awake()`. If you would want to calculate values each frame you would implement a method called `void Update()`.

1 http://web.archive.org/web/20160223065954/https://unity3d.com/

2 Unity uses the Mono framework to run its scripts; a re-implementation of .NETs compiler and runtime.

CONVENTION VS. CONFIGURATION In many cases Unity prefers convention over configuration. If you want to develop editor extensions for example, you would have to create a directory called *Editor* within the project directory. When scanning the project Unity would then recreate the Visual Studio/Monodevelop solution file to include a separate project for the editor extension featuring all source files within the *Editor* directory.

### 6.1.2 *Entity-Component System*

As early as 2002 Scot Bilas, the leading engineer at Gas Powered Games gave a talk titled *A Data-Driven Game Object System* [3] at GDC San Jose. In his talk he presents their technology used to create Dungeon Siege, which is based on the concept of dissolving game objects into components to allow game designers to reassemble them at will with no software engineer required. He may be considered the first person presenting the concept of an Entity-Component System (ECS). In 2007 Adam Martin published a series of blog posts [28] presenting his thoughts on data driven engine design discussing how data in game engines should be structured and maintained in the context of Massive Online Multiplayer (MMO) game development.

Academic publications on the concept of this architecture have also been starting to appear shortly there after. For example the 2012 published *Gear2D: an extensible component-based game engine* [10]. Today there are a multitude of approaches and names like: Entity System, Entity-Component System (ECS) (which will be the preferred name in this thesis) or Component-based entity systems. These names all try to describe the idea of a data driven architecture seperating the data from its processing. Coming from a background of object oriented programming, we are used to the idea of having an object taking care of its own data and functionality. ECSs split objects into three parts.

ENTITY An entity is nothing more than an ID. This ID is very similar to an ID in a relational database. Actually the analogies to relational databases are pretty much how entities in an Entity-Component System (ECS) can be understood.

> “Programming *well* with Entity Systems is very close to programming with a Relational Database. It would not be unreasonable to call ES’s a form of “Relation Oriented Programming”.” [28]

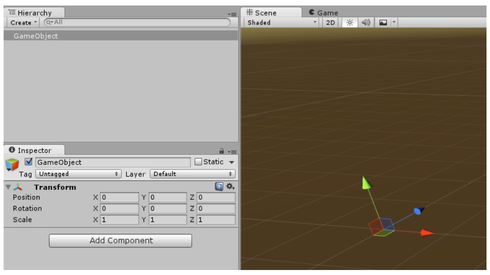
3 `http://web.archive.org/web/20160127013253/http://scottbilas.com/games/dungeon-siege/`

COMPONENT To give an Entity any meaning, it can be associated with data. This data or component could be anything, from the current position of the Entity within the 3D world, player attributes like health, strength or agility, or the mesh data used to render a character onto the screen. Components can be added, removed or manipulated during runtime.

SYSTEM The last part of an ECS is the System. A System has knowledge on how to process a specific component. So for example, if you would like to have a basic physics system, you would need a position component, and a physics component (containing e.g. mass and velocity), which could then be processed by a movement system, changing the position of an entity according to its components data. Due to the distributed nature of ECSs it is vital to use some sort of event system, to assure the communication between systems. In the physics example, such an event would be for example, the input system registering the button for throttle, in turn notifying the physics system to apply a given force to an object, resulting in a new velocity vector depending on the objects mass. Alternatively you could reference systems or components directly, which would create strong coupling.

### 6.1.3 *Entities in Unity*

Unity has come up with its own approach to the idea of an ECS. On interesting difference is the missing separation of data and its processing. Components inheriting from `MonoBehaviour` can implement methods which will be called every frame or in a fixed amount of time, so data contained in the component can be updated.



GameObject
Tag Untagged Layer Default

| Transform | X | Y | Z |
|---|---|---|---|
| Position | 0 | 0 | 0 |
| Rotation | 0 | 0 | 0 |
| Scale | 1 | 1 | 1 |

Figure 6.1: Entity in Unity called GameObject

An entity in Unity is a class called `GameObject` and represents the basic object within a scene, combining an *ID* and has one fixed *component*; the `Transform`. A purpose of a `Transform` is to hold the position, rotation and scale for the game object within the 3D world of Unity (see figure 6.1). Once an entity has been instantiated, further components can be associated with it.

### 6.1.4 *Scene Graph*

Additionally to Unitys approach to structuring game data through an Entity-Component System (ECS) architecture, game objects are also arranged in a scene graph. A scene graph is a tree structure used to express dependencies between game objects. This is especially handy if you want to construct complex objects comprised of multiple sub-objects. Each sub-object is manipulated relative to its parent object, which enables independent manipulation of sub-objects, but also allows for a coherent manipulation of all child objects when its parent object is manipulated. For example, if we were to construct an island full of trees, bushes and gold chests, each of the aforementioned objects could be modeled as separate game object. The island would be the parent object, harboring all the trees and bushes, as well as the chests. If the position of the island would be changed in the game world, all its attached child objects would move and transform relative to the manipulation of its parent. This helps managing complex interactions between different transforms of game objects.

### 6.1.5 *Library*

The basic of the concept is implemented as a separate library called *UnityUtil.Runtime.Procedural*, as part of a collection of libraries called UNITY UTIL (see section 6.1.6). VELVET only uses one class and one interface from *UnityUtil.Runtime.Procedural*. This is the base interface for chain links (see section 5.2.2) and an implementation of a chain (see section 5.2.3) based on a simple list. During the development of the plugin I found several other interesting designs for chains which I will discuss in the conclusion (see section 9.3.1).

### 6.1.6 *Unity Util*

In this set of libraries I collect useful code [4] commonly shared between most of the projects I do in Unity. One major part is the class `BlurryBehaviour`, which extends Unitys `MonoBehaviour` (see section 6.1.1) and makes sure all magic methods are available by offering the class extending it the possibility of overriding virtual functions with slightly different names, but a more consistent naming scheme. Every method called by Unity - for example when a frame starts and the component can update its values - begins with *On*. So for example the `Update()` function normally implemented in a `MonoBehaviour` is called `OnUpdate()` in a `BlurryBehaviour`.

4 Source code available at `https://github.com/BlurryRoots/UnityUtil`.

### 6.1.7 *Reflection*

The plugin makes use of a technique called reflection. In the publication *A tutorial on behavioral reflection and its implementation* the authors define reflection as:

> "[...] ability for a program to observe or change its own code as well as all aspects of its programming language (syntax, semantics, or implementation), even at runtime [...]" [29]

It is used by factories (see figure 6.8), to obtain information about types available at runtime and instantiate objects of these types dynamically. This approach has the advantage of not having to manipulate the code which produces objects, when a new type which is interesting to the factory is introduced by a user. This will be further elaborated in section 6.3.4.

## 6.2 OVERVIEW OF VELVET

Due to the concept of convention over configuration applied by Unity, VELVET is split into two main parts. The core (see section 6.3) part includes the adaptation of the chain and chain links to Unitys component system. It also implements a variable binding system (see section 6.3.3), featuring custom variable randomization (see section 6.3.4). The second part is the editor extension (see section 6.4) and is comprised of a set of scripts extending Unitys editor to use custom written modules from within the visual editor instead of plain code. It is loosely based on the architectural design pattern called Model-View-Controller (MVC) (see 6.4.4).

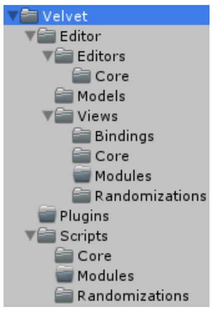


Figure 6.2: Structure of VELVET

## 6.3 CORE

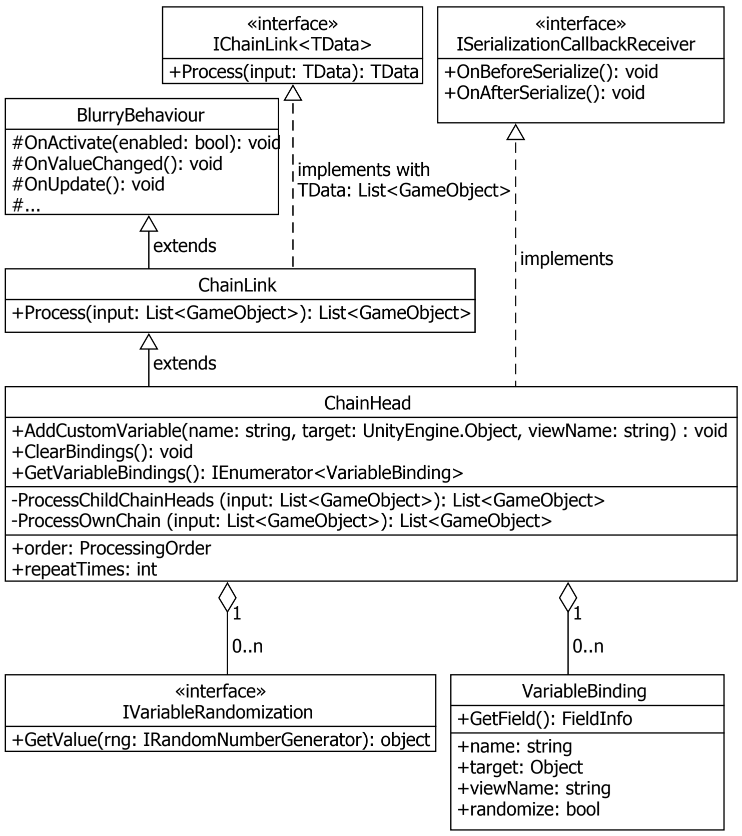

«interface»
IChainLink<TData>
+Process(input: TData): TData
«interface»
ISerializationCallbackReceiver
+OnBeforeSerialize(): void
+OnAfterSerialize(): void
BlurryBehaviour
#OnActivate(enabled: bool): void
#OnValueChanged(): void
#OnUpdate(): void
#...
implements with
TData: List<GameObject>
extends
implements
ChainLink
+Process(input: List<GameObject>): List<GameObject>
extends
ChainHead
+AddCustomVariable(name: string, target: UnityEngine.Object, viewName: string) : void
+ClearBindings(): void
+GetVariableBindings(): IEnumerator<VariableBinding>
-ProcessChildChainHeads (input: List<GameObject>): List<GameObject>
-ProcessOwnChain (input: List<GameObject>): List<GameObject>
+order: ProcessingOrder
+repeatTimes: int
1
0..n
1
0..n
«interface»
IVariableRandomization
+GetValue(rng: IRandomNumberGenerator): object
VariableBinding
+GetField(): FieldInfo
+name: string
+target: Object
+viewName: string
+randomize: bool

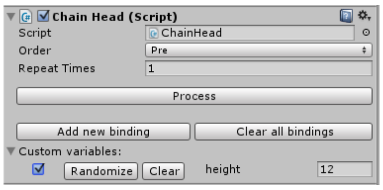
Figure 6.3: UML of ChainLink and ChainHead

### 6.3.1 *Chain Link*

The class `ChainLink` (see section 6.3) extends `BlurryBehaviour` and implements the `IChainLink` interface specifying the generic data type to be a list of game objects. It serves as the base for every custom module used in a chain.

### 6.3.2 *Chain Head*

The core module is called `ChainHead` (see figure 6.3). It inherits from `ChainLink`, making it possible to have a chain head part of a chain. Additionally `ChainHead` implements Unity `ISerializationCallbackReceiver` interface for properly serializing all data. The head is responsible for managing all modules contained within its chain, as well as managing all custom variable bindings (see section 6.3.4).

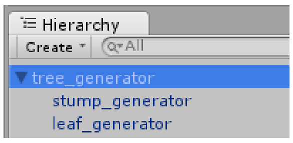


Script: ChainHead
Order: Pre
Repeat Times: 1
Custom variables:
height: 12

Figure 6.4: Inspector view of a chain head

To create a chain, a game object must be created and associated with a chain head component. Every custom module component attached to this game object will then be taken into consideration, when the chain head is instructed to process its chain. This can either be the user pressing the *Process* button or invoking the process function by code. In the architecture overview, I mentioned the goal of not only having modules in a sequencial order, but also in a tree-like structure. To archive this, I made use of the already available scene graph structure in Unity. A game object associated with a chain head can have child game objects. These child objects can then in turn, have their own chain head component as well as several custom modules.

When the user presses the *Process* button, the chain head will first check in which order the child chain heads and the modules of its own chain should be processed. This is done via a simple enumeration type called `ProcessingOrder`, presented to the user in a drop down menu (see section 5.2.4). After this, the chain head initiating the process, queries its object hierarchy for all game objects which are direct descendants and have a chain head component associated with them. The game objects will be kept in the same order as they appear in the scene graph (see figure 6.5). These heads will then be told



```
tree_generator
    stump_generator
    leaf_generator
```

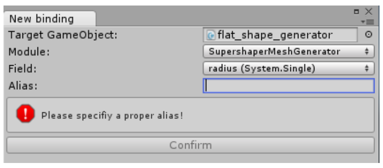
Figure 6.5: Hirarchy overview of a chain generating trees

to process their chains one after another. If one of these chain heads happen to have child objects, these children would be processed according to the strategy configured in their `ProcessingOrder`.

With this approach, all chain head components get processed via a depth-first strategy. Changing the position of a game object associated with a chain head component within the hierarchy will thereby change the result.

### 6.3.3 *Variable Bindings*

To allow developers and designers to use the created chain as one unit, it would be very handy to have all necessary variables in one place. This can be archived through binding variables. A binding is created via the chain head editor. When clicking the button *Add new binding*, the developer is presented with an input mask (see section 6.6). Here she can specify which game object should be selected. Once a game object is selected, a list of attached modules will be shown. After selecting a module, the user will be able to select a public variable of the selected module to be bound. After typing in a descriptive name, the binding can be confirmed.


New binding

| | |
|---|---|
| Target GameObject: | flat_shape_generator |
| Module: | SupershaperMeshGenerator |
| Field: | radius (System.Single) |
| Alias: | |

Please specifiy a proper alias!

Confirm

Figure 6.6: Window with mask to add new binding

From now on, every time the user views the inspector view for this chain head component, it will display the created binding. The toggle button, deactivates or activates the randomization (see section 6.3.4) of the bound variable. If a binding should not be necessary any more, it is possible to delete it via the *Clear* button, or dispose of all bindings via the button *Clear all bindings*. To make binding variables possible, I developed a set of editor extensions coupled with runtime type reflection[5]. The first step is initiated by the user via the interface by dragging a game object associated with a chain head and modules into the game object field presented by the *New binding window* (see figure 6.6).

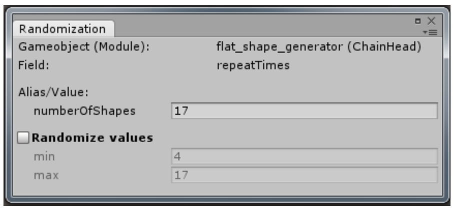
When a game object is dropped into the according field, it is examined to determine if it is actually associated with a chain head component. If this is the case the algorithm then asks the chain head to retrieve all modules, associated with its game object. This list is then rearranged to be visualized as drop down menu. After the user selects a module, it gets examined via reflection for any publicly available field. All fields get collected into a list and presented to the user in yet another drop down menu.

5 Reflection is the ability of a program to inspect or manipulate its runtime meta information, such as types.

After having selected a field to be bound, the user is then asked to specify a descriptive name to be displayed when the binding is completed. You can now create the binding by clicking the *Confirm* button. When confirmed, the editor window then asks the chain head which requested the binding to be made to save a new `VariableBinding`. This value object contains all information necessary for the chain head to save, serialize and visualize the binding. Serialization turned out to be a bit cumbersome when working with a custom data structure decoupled from Unitys object system. See section 6.3.5 for more information on serialization. Additionally after creating the binding, the system tries to determine how the bound variable might be randomized.

### 6.3.4 *Randomizations*

By specifying a binding, a designer should basically be able to build a coherent structure, or blueprint for her creation. To have random variations within this blueprint and therefor allowing for a diverse set of results, custom variable bindings can be initialized by custom value randomization. After a binding has been created, it can be randomized, by clicking the button *Randomize*. When editing the randomization options a window is shown, presenting the options available to the designer (see figure 6.7). The randomization system is based on specific implementations for each data type, the developer wishes to be randomized.



Gameobject (Module): flat_shape_generator (ChainHead)
Field: repeatTimes

Alias/Value:
numberOfShapes 17

☐ **Randomize values**
min 4
max 17

Figure 6.7: Randomization options for an integer value

As discussed previously all publicly available fields on a module can be bound. So if a developer writes a custom module using a data type currently not supported, she has to also provide an implementation of a interface `IVariabelRandomization`. This implementation will then be used by the binding system to describe the process of initializing a bound variable with random values. The type of variable which can be randomized with the implementation has to be specified via a class level attribute called `CustomRandomizer`. Randomizations get created when a new binding is set up. To instantiate a randomization depending on the type of variable ought to be bound, the factory method pattern is used (see figure 6.8).

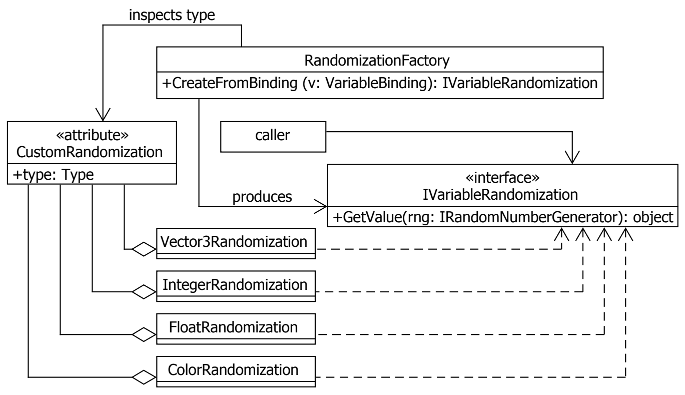


Figure 6.8: Factory pattern for creating randomizations

When asking the `RandomizationFactory` to create a randomization, it will use reflection to search the namespace *Velvet.Randomizations* for classes implementing the `IVariabelRandomization` and holding attribute `CustomRandomizer`. It then examines the type of the bound variable provided and looks for a randomization responsible for this type. This allows custom randomizations to be written without the need to manually extend the factory method.

### 6.3.5 *Serialization*

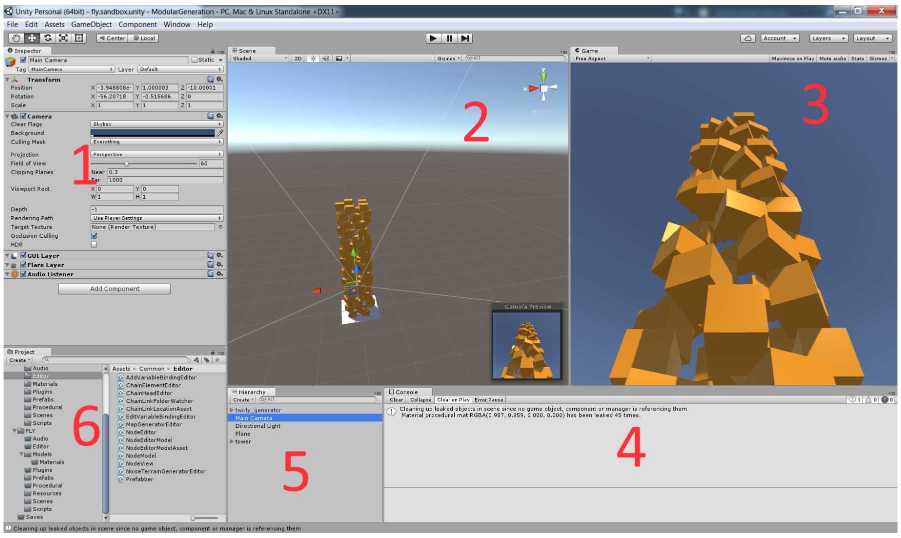
Serialization is the process of taking runtime data and converting it to a different representation to be stored on hard-drive or transmitted over the network. To be able to save the chains including all randomization options, it is necessary to do some manually serialization. Normally Unity does a pretty good job of serializing all relevant data from components, game objects, prefabs or assets you create or import. There is however a problem when it comes to serializing Unitys standard types like a `Vector3` or a `Color`. To serialize a class or struct in a convenient way, it has to have a class level attribute called `Serializable`, provided by the .NET framework. Unfortunatly most of Unitys standard types do not contain this attribute. In order to circumvent that problem, I created helper classes which have this attribute and mirror all public fields of its target class to serialize all necessary information. For example the `Color` value type has fields for red, green, blue and alpha. These fields are also provided by the class `ColorSerializable`. When Unity is about to serialize its data all chain heads get notified via the implemented method `OnBeforeSerialize()` provided by the `ISerializationCallbackReceiver` (see figure 6.3) interface. It then serializes each randomization into a binary format[6], converting it to string and storing it into a list of `VariableRandomizationSerializable`.

6 The binary format used here is base64.

This list is a private field and marked with an attribute provided by Unity called `SerializeField`. Normally only public fields get serialized by Unitys built-in serializer. If a less visible field is marked with the `SerializeField` the built-in serializer will include this field in the serialization and deserialization process. When Unity is about to deserialize its data, the chain heads will also get notified via a second method called `OnAfterDeserialize()`, also provided by the serialization callback interface. In this method each element in the list of `VariableRandomizationSerializable` will be deserialized and stored in its runtime representation.

## 6.4 EDITOR EXTENSION

The second part is the extension of Unitys visual editor, enabling the user to add, remove and control modules within a chain and create bindings to variables from modules within a chain. All code extending Unitys editor has to be contained in a directory named *Editor*. When scanning the project directories Unity will then collect these scripts and create a separate project from the normal source code. Unity allows developers to redefine to some extend how data associated with your application is present, created or manipulated. The basic structure of Unitys interface are windows (see figure 6.9).



```
Cleaning up leaked objects in scene since no game object, component or manager is referencing them
 Material procedural mat RGBA(0.987, 0.959, 0.000, 0.000) has been leaked 45 times.
```

Figure 6.9: Numbered Window overview of Unitys standard interface

A window is a resizable canvas which can be placed freely within the main application window of Unity. It can be tapped, horizontally stacked or vertically aligned.

WINDOWS SHOWN IN FIGURE 6.9:

INSPECTOR (1) The Inspector Window displays the currently selected game object and its associated components.

SCENE (2) In the Scene Window the user can move through the currently loaded scene, create, select and manipulate game objects them.

GAME (3) The Game Window shows the game from the perspective of the currently active camera of the scene.

CONSOLE (4) The Console Window shows errors, warnings and log outputs to help debug your game.

HIERARCHY (5) Displays all objects contained in the current scene, the Hierarchy Window allows the developer to create, select and manipulate game objects directly though a textual representation of the scene view.

PROJECT (6) In the Project Window, all directories and game assets are displayed, can be created, renamed, moved or deleted.

### 6.4.1 *Custom Property Drawer*

When creating a custom component, a developer is able to tell Unity how this component should look in the inspector view. The simplest way would be to create a custom property drawer. As far as Unity is concerned, a property is any public field on a component (not to be confused with C# properties!). When loading a component Unity looks for publicly available field and tries to draw a representation for it. Every built-in simple type [7] provided by the basic .NET framework and most of Unitys core library types can be drawn by the default `PropertyDrawer`. If you need more than customized ways of displaying public fields you would have to implement a custom editor.

7 See `http://web.archive.org/web/20160219110405/https://msdn.microsoft.com/en-us/library/s1ax56ch.aspx` for more on simple types.

### 6.4.2 *Custom Editor*

When creating a complex component, which should be presented in a special way, developers can create a custom inspector editor. This editor is responsible for rendering a component when viewed though the inspector window (see section 6.9). To create such an editor, developers can extend a base class called `Editor`. To tell Unity how this editor should be used, the class level attribute `CustomEditor` has to be applied, which expects the class type of a component for which the custom editor is responsible for. You can now override a function called `OnInspectorGUI`, which gets called whenever Unity wants to redraw parts of its interface. For interface elements like labels, buttons, dropdown menus or checkboxes Unity provides a couple of static classes exposing functions to draw a certain element. If a developer wants to present text box with an editable floating point number, she could used the following function:

```
value = EditorGUILayout.IntField ("value", value);
```

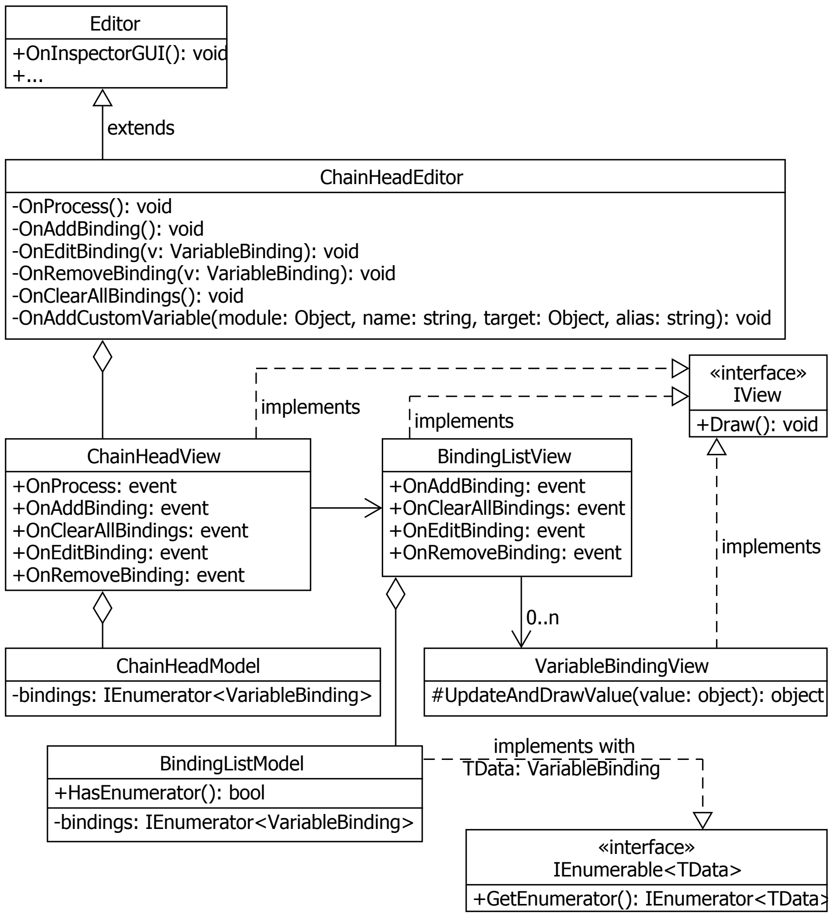
The functions provided by `EditoGUILayout` will be positioned by Unity dynamically when drawing the interface in contrast to just `EditoGUI`. This is very handy if you have interface elements which might change their dimensions depending on the value they currently have.

### 6.4.3 *Custom Window*

Being the corner stone of Unitys visual editor, developers can also create their own custom editor windows. This is used in Velvet for the window to add new bindings, as well as the window to edit a variable randomization. To create a custom window, a class has to inherit from `EditorWindow`. It then can override a method called `OnGUI` which will be called when Unity is about to redraw its interface. To open a window, a generic factory method provided by Unity called `GetWindow` has to be called.

### 6.4.4 *UI Architecture*

The design for the user interface extension part of the plugin is based on the idea of separating presentation and logic. One architectural pattern often used in such a case is the MVC pattern, on which I loosely based this architecture (see figure 6.10).



Editor
+OnInspectorGUI(): void
+...
extends
ChainHeadEditor
-OnProcess(): void
-OnAddBinding(): void
-OnEditBinding(v: VariableBinding): void
-OnRemoveBinding(v: VariableBinding): void
-OnClearAllBindings(): void
-OnAddCustomVariable(module: Object, name: string, target: Object, alias: string): void
implements
implements
«interface» IView
+Draw(): void
ChainHeadView
+OnProcess: event
+OnAddBinding: event
+OnClearAllBindings: event
+OnEditBinding: event
+OnRemoveBinding: event
BindingListView
+OnAddBinding: event
+OnClearAllBindings: event
+OnEditBinding: event
+OnRemoveBinding: event
implements
0..n
ChainHeadModel
-bindings: IEnumerator<VariableBinding>
VariableBindingView
#UpdateAndDrawValue(value: object): object
implements with TData: VariableBinding
BindingListModel
+HasEnumerator(): bool
-bindings: IEnumerator<VariableBinding>
«interface» IEnumerable<TData>
+GetEnumerator(): IEnumerator<TData>

Figure 6.10: Architecture overview

A model is responsible for storing data, while the view is concerned with how the data should be presented to the user. The controller is where the interaction logic is implemented. Instead of having a dedicated controller, the basis for managing views and models will be the classes inheriting from `Editor` or `EdiorWindow`.

CHAIN HEAD EDITOR The `ChainHeadEditor` is the central part for interacting with a chain head component through the inspector window of the unity interface. It inherits from Unitys `Editor` base class and overrides `OnInspectorGUI()`.

Specifying the class level attribute `CustomEditor` with the type of `ChainHead`, it instructs Unity to use `ChainHeadEditor` when rendering a chain head component instead of the standard view.

CHAIN HEAD VIEW When the chain head editor is asked to draw, it uses a view called `ChainHeadView`. This class knows how to display a `ChainHead` and has several events to which `ChainHeadEditor` can subscribe. This would be for example when the *Process* button is pressed. The chain head view in turn splits the rendering of its different areas into multiple sub-views.

BINDING LIST VIEW After showing the variables for processing order and repeat times as well as rendering the *Process* button, the chain head view instructs `BindingListView` to show the list of currently bound variables. The binding list view first shows the button to add a new binding as well as a button to clear all bindings. If there is at least one binding, the binding list creates a view for the variable and repeats until all variables have been drawn. This is done via the help of the iterator pattern. C# has a build in interface called `IEnumerator` which is used by the foreach loop. Such an enumerator or iterator can be obtained via the method `GetEnumerator` which is part of the `IEnumerable` interface.

VARIABLE BINDING VIEW Each binding gets drawn by a view extending `VariableBindingView` (see figure 6.11).

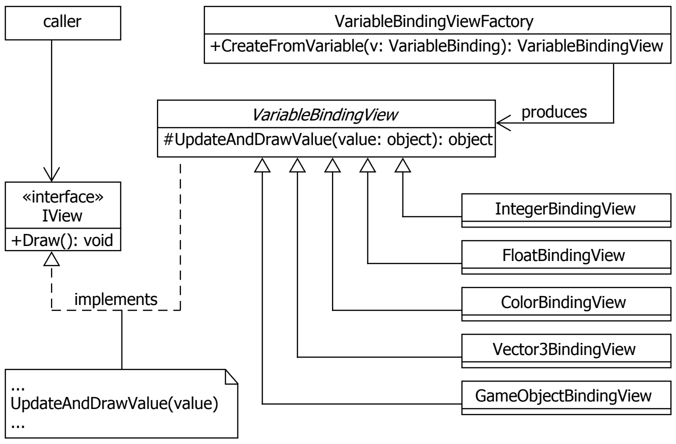


Figure 6.11: Specializations to draw different types of variables

This is because in order to draw a floating point number you have to use different parts of Unitys editor api than for example for drawing a color value or a string. By implementing the protected abstract method `UpdateAndDrawValue` each specialization can make use of whatever api is needed to get the value displayed and retrieve a new value in case it has been changed via the user interface.

The method then gets internally called when the public draw method is invoked. This design is based on the template method pattern.

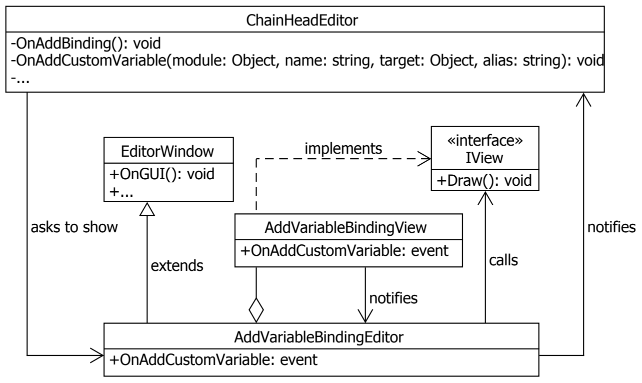


Figure 6.12: UML for adding a new binding

ADDING A BINDING Variable bindings are created through a custom editor window (see figure 6.12). This window is created by the chain head editor when the user presses the *Add new binding* button. This click invokes the event handler method `OnAddBinding` which instructs the `AddViariableBindingEditor` to show its view (see figure 6.6). When all options are specified and the user presses the *Confirm* button, the chain head editor is notified. It in turn will then instruct the chain head to save a new binding.

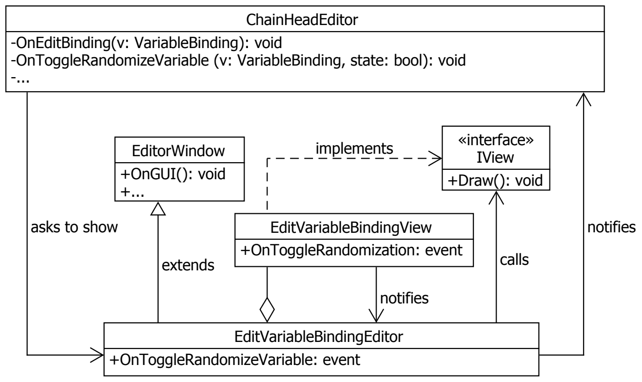

ChainHeadEditor
-OnEditBinding(v: VariableBinding): void
-OnToggleRandomizeVariable (v: VariableBinding, state: bool): void
-...
EditorWindow
+OnGUI(): void
+...
implements
«interface»
IView
+Draw(): void
asks to show
EditVariableBindingView
+OnToggleRandomization: event
notifies
extends
calls
notifies
EditVariableBindingEditor
+OnToggleRandomizeVariable: event

Figure 6.13: UML for editing a binding

EDITING A BINDING After a binding has been created, it is possible to configure its randomization options (see figure 6.7) by pressing the *Randomize* button. The structure and flow is modeled after the same concept applied to add a new binding. The chain head editor tells the `EditVariableBindingEditor` to show its window.

The edit variable binding editor, will then use its view to display the randomization and notifies the chain head editor in case a value has been changed. To visualize the custom randomization used for a specific data type, I created a extendable system; again based on the factory method pattern.

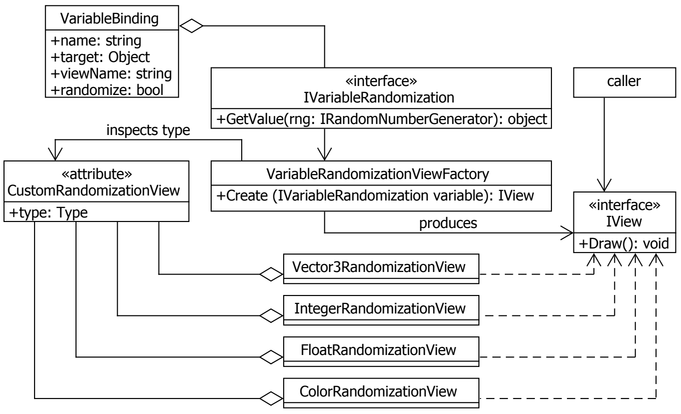


Figure 6.14: Factory of randomization views

RANDOMIZATION VIEW To display a custom randomization in the randomization option window (see figure 6.7), a similar system to the binding factory and binding view factory is applied here too. When a randomization is about to be displayed it examines the type of randomization and searches for a class holding a `CustomRandomizationView` attribute specifying that type of randomization.

# 7 EXAMPLES

## 7.1 PROLOGUE

To test drive the plugin, I created two examples using VELVET. The first one uses Bézier Curves and the Superformula to create organic looking cylindrical shapes. The following example creates a block world based on Perlin noise. And the third shows the re-use of some components of the two examples before to create curved and twisted tunnels.

## 7.2 SUPERCYLINDERS

In this examples I made use of the Superformula to create polygons and a Bézier spline to describe a curve on which I want to position said polygons. After placing the polygons I went through all vertices and build a mesh based on a cylinder; hence the name Supercylinders. I split the responsibility into three modules.

### 7.2.1 *Modules*

POLYGON GENERATOR Generating a flat polygon on the basis of the Superformula is done via the `SupershapePolygonGenerator`. It can be parameterized with the radius in Unity units, the amount of vertices used to describe the polygon and the five parameters used by the Superformula (see figure 3.4). It produces a set of game objects holding a mesh with vertices representing a Supershape polygon.

POSITION ALONG BÉZIER PATH MUTATOR This module takes a set of game objects and positions them along a Bézier curve. After setting the positions the game objects get also rotates them so their forward vector faces the tangent of the curve point. It can be parameterized by setting the positions of $P_0$ and point $P_1$ as well as the control points of the Bézier Curve. Additionally I added an editor extension for this module to show custom gizmos [1] in the scene view.

1 A gizmo is an visual aid like the position handle to help manipulate objects in the scene view.

With this gizmos it is possible to control the two points and their respective control points of the curve. You can also preview how the points along the curve will be position via gizmos showing the normal in red and tangent in yellow (see figure 7.1).

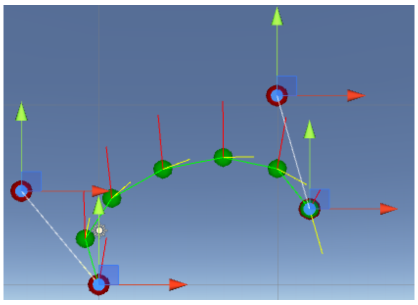


Figure 7.1: Bézier curve gizmos

MESH GENERATION Generating the complete model from polygons is the responsibility of this module (see figure 7.2). It fetches the vertices of the mesh associated with a game object and creates triangles on the assumption that the resulting shape is cylindrical in nature. This is done by first creating quads[2] between every set of two adjacent vertices of polygon $P_i$ and $P_{i+1}$.

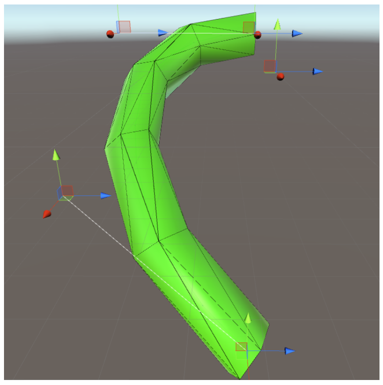
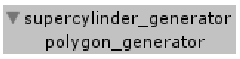


Figure 7.2: Shaded wireframe view of cylindric mesh

The quad can then be split into two triangles respectively. After triangulating the walls of the cylinder the generator also triangulates the front polygon $P_0$ and back polygon $P_{i+n-1}$ so the cylinder is closed off. The algorithm used here just takes the first vertex $V_0$ of the polygon $P$ and triangulates with two vertices $V_k$ and $V_{k+1}$ where $k \geq 1$ and is incremented by 2 while $k \leq (|V| - 1)$. I used this approach, due to its simple implementation, but it will unfortunately lead to strange results when having concave or star-shaped polygons.

2 A quad is a polygon with four vertices. It is sometimes used as the base element of surfaces of complex polygons instead of triangles.

### 7.2.2 *Generation process*

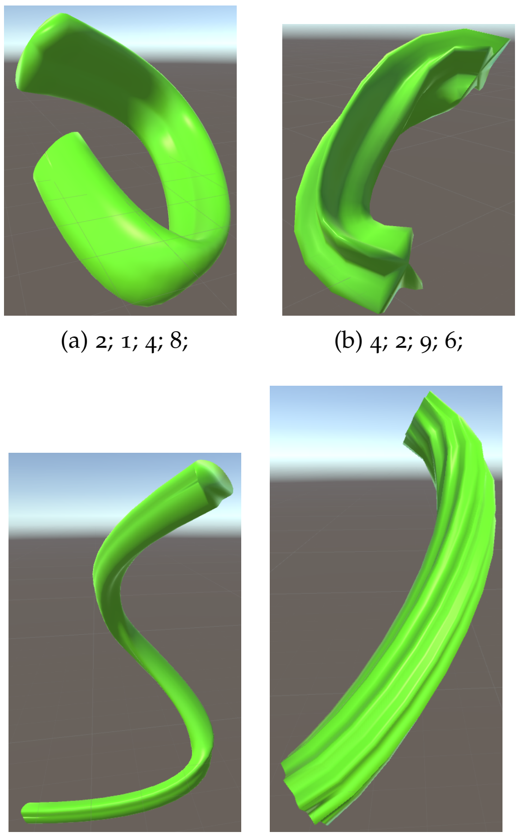


▼ supercylinder_generator
polygon_generator

Figure 7.3: Supercylinder generation tree

To control the generation process I created a nested structure. The *`polygon_generator`* object contains a chain head an the polygon generator module. I separated this module in its own chain so I can repeatedly generate polygons to control the amount of resolution I get in the cylinder. Containing the *`polygon_generator`* object *`supercylinder_generator`* holds a chain head the Bézier curve module and the mesh generator module. By setting the process order to post, I make sure the child object is processed first. In this case this will first produce the desired amount of polygons, which will then be passed to the Bézier module and finally to the mesh generator module. The figure 7.4 shows four examples of Supercylinders I created using this chain.

### 7.2.3 *Results*



(a) 2; 1; 4; 8;

(b) 4; 2; 9; 6;

(c) 3; 3; 14; 2;

(d) 9; 9; 8; 9;

Figure 7.4: Supercylinders based on Supershapes (m; n1; n2; n3; a = b = 1)

## 7.3 BLOCK WORLD TERRAIN

In this example, voxels in the shape of cubes get placed on the basis of an height map. This height map is produced by sampling a Perlin noise function multiple times as described in section 3.2.

### 7.3.1 *Modules*

PREFAB INSTANCE GENERATOR This generator module is responsible for instantiating game objects on the basis of a prefab. A prefab is like a recipes, used by Unity to remember how a game object with a specific set of components should be created. In this case it would be the voxel. A voxel is just a game object containing a cube mesh component and a mesh renderer component.

TERRAIN GENERATOR The terrain generator takes a set of game objects and arranges them on the basis of a height map. It will also color each game object on the basis of its height on the height map. This is done through the use of a noise texture generated by a Perlin noise function provided by Unity. When the texture is generated, it then will be traversed and its values, which range from 0 to 1 get interpreted as heights. Each height range gets assigned a color corresponding to water (blue), flat land (yellow and green), mountains (gray) and mountain tops (white).

REPARENT COLLECTOR This module takes a set of game objects and attaches them to a newly created parent game object. This allows for a easier handling of composite structures so the scene graph doesn't get cluttered.

### 7.3.2 *Generation Process*

All modules are contained within on chain. The first module generates all voxel needed to form the terrain. These generated voxels will then be modified by the second module which applies a height offset and color depending on the generated height map. The reparent module will then take the modified voxels and attaches them to a common parent, so the map can be handles as one object.

### 7.3.3 *Results*

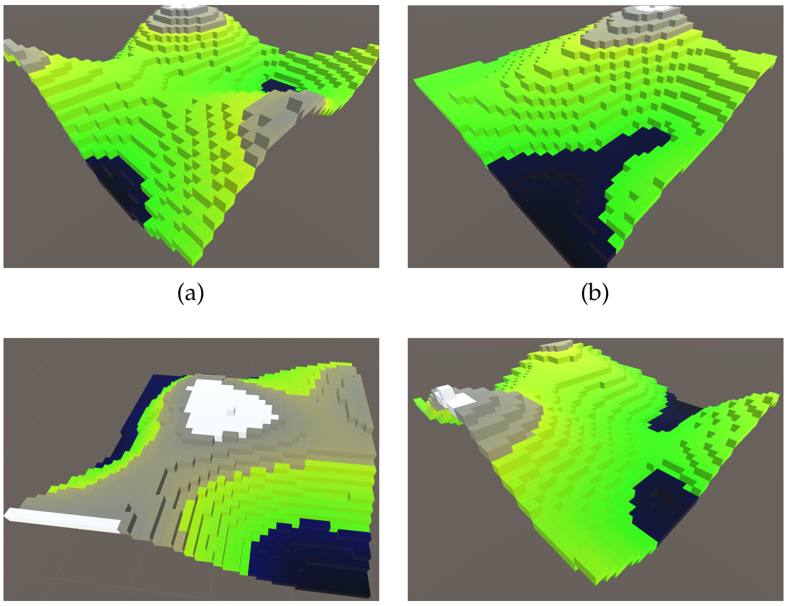

(a) (b) (c) (d)

Figure 7.5: Generated voxel landscapes

## 7.4 VORTEX TUNNELS

To examine the reusability of modules, this examples also makes use of the Prefab Module, Bézier Path Module and Reparent Module to generate curved and twisted vortex-like tunnel structures.

### 7.4.1 *Modules*

PREFAB INSTANCE GENERATOR See Block World example (section 7.3.1).

SCALE MUTATOR Changes the scale factor of a game object to the scale defined in the parameters of this module.

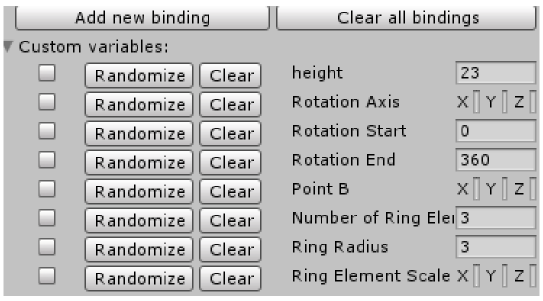
COLOR MUTATOR Changes the color of a game object to the color specified in this modules parameters.

CIRCULAR PLACEMENT MUTATOR This module takes its input game objects and arranges them on a circular path. It is possible to specify the radius of the circle used to create the path.

REPARENT COLLECTOR See Block World example (section 7.3.1).

POSITION ALONG BÉZIER PATH MUTATOR See Supercylinder example (section 7.2.1).

PARTIAL ROTATION MUTATOR With this module, all input game objects get rotate a certain angle. This angle depends on the parameters *Start Angle* and *End Angle*, as well as the game objects position in the input list. So if the start is at $0^\circ$ and the end at $360^\circ$ a game object in at index 1 in a list of 3 objects would have the rotation $180^\circ$. It is also possible to specify if the rotation should happen in the game objects local space or in global space and which angle should be used to rotate around.



| | |
|---|---|
| height | 23 |
| Rotation Axis | X Y Z |
| Rotation Start | 0 |
| Rotation End | 360 |
| Point B | X Y Z |
| Number of Ring Elei | 3 |
| Ring Radius | 3 |
| Ring Element Scale | X Y Z |

Figure 7.6: Vortex generator bindings

### 7.4.2 *Generation Process*

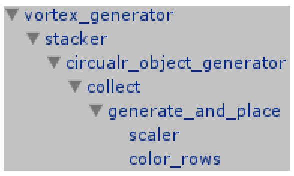
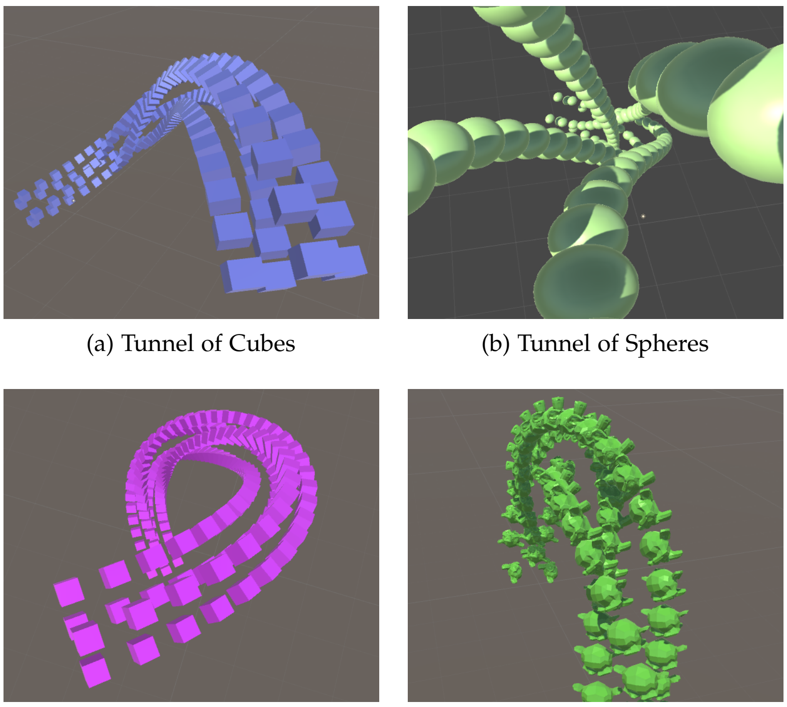

▼ vortex_generator
  ▼ stacker
    ▼ circualr_object_generator
      ▼ collect
        ▼ generate_and_place
            scaler
            color_rows

Figure 7.7: Vortex generation tree

The generation of this shape is managed by several nested chains. First a set of objects defined by a prefab are generated by `generate_and_place`. They then get handed down to `scaler` to changed the scale factor and `color_rows` to re-color the objects. The instances then get handed up to `collect`, placed on a circular path and re-parented to form a group. The object `circular_object_generator` acts as a separator to allow the generation of as many circularly placed object groups as needed. After this `stacker` has modules to place the groups along a Bézier curve and rotated them described in section 7.4.1, as well as yet again re-parented to form a single object. Finally vortex_generate acts as a hub object and holds all bindings to the important parameters of the modules within the tree (see figure 7.6).

### 7.4.3 *Results*


(a) Tunnel of Cubes (b) Tunnel of Spheres

(c) Tunnel of Cubes (d) Tunnel of Suzannes

Figure 7.8: Vortex Tunnels along a Bézier Curve

Part III

# CLOSING WORDS

Following the concept and its prototypical implementation, this chapter will be about taking a look on how well the goals set in the introduction of this thesis have been met. I will then give a short summary of what has been discussed in this thesis and what further work could be done on the basis of this work.

# 8 EVALUATION

## 8.1 PROLOGUE

In this chapter I will take a look at how well I met my goals set out in the introduction of this thesis. I will also try to give an impression on what advantages and disadvantages the proposed concept and in turn the prototypical implementation has.

## 8.2 DEVELOPING MODULES

The goals described in section 1.4.1 state the importance of not having inter-dependencies and information sharing between modules. After implementing the prototype and developing a set of modules to produce the examples presented in section 7, it turned out to be very hard to really have no assumptions or knowledge about the information in the input certain modules will receive. In the Supercylinder (see 7.2) example, I developed a module concerned with construction a mesh from a set of vertices representing flat Supershape planes. In it I had to assume the shapes come in a circular polygon without holes. I guess it is necessary to have certain requirements on the information fed to a module for it to really be able to solve a specific problem. Chainability, as stated in the goals section (see 1.4.2) as well as nestability (see 1.4.3) have been archived through the design of the plugin based on the composite pattern in combination with Unitys scene graph.

## 8.3 DEVELOPMENT OF VELVET

Another goal set out in the beginning of this thesis, was the integration of the module workflow into the Unity eco-system. This has been realized by developing the plugin VELVET which adapts the library to Unitys ECS and makes use of the scene graph already available in Unity. To allow for a more convenient use of the generation system, I also created a set of editor extension within VELVET allowing designers to bind variables of modules, allowing for the parametrization (see 1.4.6) of a chain.

## 8.4 CONTROLLING THE GENERATION PROCESS

Structuring the generation process through the use of Unitys scene graph proves to be valuable when creating a hierarchy of responsibilities. This approach also allows for an easy way of saving chains through the use of prefabs. Because all modules are components attached to a game object, this game object can just be linked as a prefab, enabling the user to use it as a predefined chain in more than one place. Changes made in the prefab object will then be applied to all instances. Being able to tell a chain head how many times it should be processed, as well as defining the order of execution of its own chain and child modules helped building a flexible system. Initializing bound variables with random values through the use of custom variable randomizations has also proven very valuable to get variety on the content created through the use of a chain. There are however certain limitations on controlling the flow of the generation. It is for example not possible to hold state. If a chain head is invoked multiple times, it has no knowledge how many times it has been process before. This might be conquered by implementing a custom set of selector modules. Although practical and functional, the use of game objects and components to represent the generation tree feels a bit cumbersome. I think the use of a node based editor would have greatly benefited the user experience.

## 8.5 USER INTERFACE

Initially I though of a node editor as the basis for handling the creation of chains. Comparable to Blenders [1] node editor (see figure 8.1) the user would have connect modules and build up a generation graph that way.

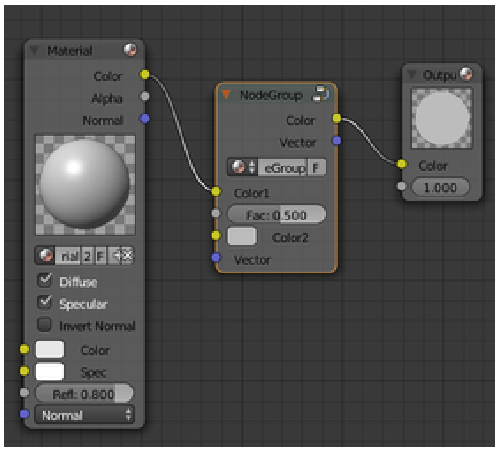


Figure 8.1: Blender 2.67 Node Editor

1 Blender is an free and open source 3D modeling, rendering and animation tool. http://web.archive.org/web/20160222121343/https://www.blender.org/

Due to a multitude of challenges, like the integration of node visualizations into the Unity editor or the concept of having input from more than on module being fed into another module led me to abandon this approach. There are however interesting commercial Plugins in Unitys asset store, featuring node based editing [2], which could come in handy for future work. However in place of the node editor, I created a UI system which made use of the basic editor elements already in place: the inspector view and custom editor windows. Combined with the scene graph, this approach turned out to work very well. With a reasonable amount of work, basically any type of module of any type of data can be displayed in the inspector view by writing custom editor scripts for the modules and extending the view system of VELVET for the visualization of bindings and randomizations. Although I think the system to create bindings is functional, the fact that a user has to drag and drop the game object into the new binding window feels a bit cumbersome. It also bears the risk of having a binding to a module which is not necessary in the same process tree as the head to which it should be bound. I think this might have been better implemented as a tree view offering the user a selection of modules which are actually contained within the object tree.

## 8.6 PERFORMANCE

Due to the fact, that the primary data type used in the prototype is a list of game objects, the base unit of information used by chains has to be contained within a game object. This might lead to an substantial amount of game objects being instantiated, which could impact memory and computational resources. It might be interesting to use a more varied set of types to use in the generic chain link system to see if there are advantages to CPU processing time or memory consumption.

2 https://www.assetstore.unity3d.com/en/#!/content/29435 (visited 28.02.2016)

# 9 CONCLUSION

## 9.1 PROLOGUE

Summarizing what has been discussed in this thesis as well as giving an outlook of what future work might be done with Velvet will be the topic of this chapter.

## 9.2 SUMMARY

In this thesis I looked at some examples of how Procedrual Content Generation (PCG) has been used in video games and also what recent research has found in some specific areas of PCG. I also examined a set of techniques and examples of areas of application. This is followed by the presentation of a modular workflow concept for procedurally generating content and the description of Velvet, a prototypical plugin implementation for Unity. To show the how to work with Velvet a set of examples have been discussed.

## 9.3 FURTHER WORK

### 9.3.1 *Different chain implementations*

Currently chains can only process their modules in one go. When trying out different combinations of modules, it would have been interesting for a chain to process its modules over time. This would have allowed for deferred generation, or possibly even procedural animation. In generation trees where nested chains could work concurrently, a chain type featuring asynchronous processing might increase performance.

### 9.3.2 *Modularization*

I think it might be interesting to investigate further into the topic of modularization in the context of procedural generation. Evaluating how beneficial the splitting of work in a complex generation system is and what value it might bring to the developer or designer. Maybe there are specific scenarios where modularization is very fitting, like for example the generation of worlds or levels described in the section about Dwarf Fortress (see section 2.2.5). There might however be tasks which do not lend themselves very well to separation. Nevertheless, in my opinion, separating parts of a system into smaller modules could benefit the way one approaches procedural generation, by not thinking of a system as one unchangeable recipe, but as a set of responsibilities; potentially exchangeable without loosing the overall concept.

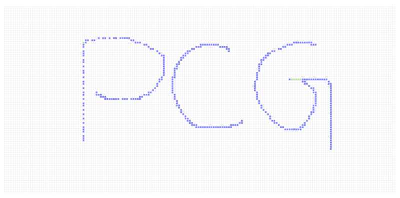
Part IV

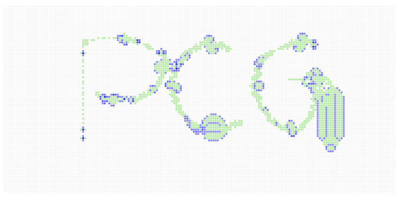
# APPENDIX

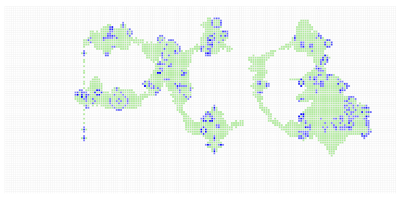
# A APPENDIX

## A.1 LIFE

The rules for LIFE, though up by mathematician John Horton Conway, are based on a celluar automaton. A concept developed by John von Neuman and published in 1966 [34]. Figure A.1 shows four snapshots over 2644 generations of LIFE [1].


(a) Generation 0


(b) Generation 6


(c) Generation 42

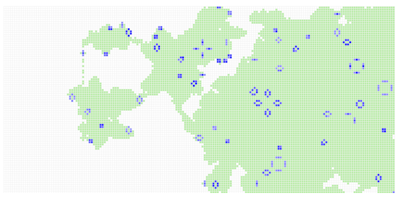

(d) Generatio 2644

Figure A.1: Snapshots over 2644 generations of "life"

The board is made up of cells, which in turn can have 3 different states making up a generation (see table A.1).

1 Screenshots from a javascript implementation [56] by Pedro Verruma `http://web.archive.org/web/20160109155314/http://pmav.eu/`

The game is played according to three simple rules[17]:

| STATE | COLOUR | MEANING |
|---|---|---|
| Untouched | Grey | Has never been alive in any genration before. |
| Alive | Blue | Is alive in the current generation. |
| Dead | Green | Has been alive in a generation before the current one. |

Table A.1: States of a generation in LIFE

SURVIVALS Every counter with two or three neighboring counters survives for the next generation.

DEATHS Each counter with four or more neighbors dies (is removed) from overpopulation. Every counter with one neighbor or none dies from isolation.

BIRTHS Each empty cell adjacent to exactly three neighbors–no more, no fewer–is a birth cell. A counter is placed on it at the next move.

Although simple, Conway later (originally in 1982) published [1] proof for the existence of a universal constructor [2]. Later research [45] on LIFE lead to the conclusion of computational universality equal to a Turing Machine. In his publication *On computable numbers, with an application to the Entscheidungsproblem* Turing states:

> "It is possible to invent a single machine which can be used to compute any computable sequence. " [54]

The Turing machine a mathematical concept named after its inventor Alan Turing, showing the fundamental capabilities and limitations of a computation device.

2 A non-trivial self-reproducing machine [30].

## A.2 FIBONACCI LIKE SERIES

Below are the listings used to generate the statistics mentioned in section 2.2.2.1.

### A.2.1 *Basic Algorithm*

```
# splits off the last digit of any given number n
def get_last_digit (n):
    _k, r = divmod (n, 10)
    return r

# generates a list of numbers in a fibonacci fashioned series
# in the form [a, b, (a, b), (b + (a + b)), ...], except each
# entry correspons only to the last digit of its original number
def generate_elite_fibo_list (a, b, length):
    x = get_last_digit (a)
    y = get_last_digit (b)
    z = 0
    r = [x, y]
    for i in range (0, length):
        z = get_last_digit (x + y)
        r.append (z)
        x = y
        y = z
    return r
```

Listing A.1: Fibonacci like series used in Elite

### A.2.2 *Generating the data*

```python
import elite_fibonacci as efib

# generates a list of fibo lists
def generate_samples (al, bl):
    n = 10
    samples = []
    for a in range (0, al):
        for b in range (0, bl):
            l = efib.generate_elite_fibo_list (a, b, n)
            samples.append (l)
    return samples

def get_stats ():
    samples = generate_samples (10, 10)
    occurences = dict ()
    for i in range (0, 10):
        occurences[i] = 0
    for sample in samples:
        for number in sample:
            occurences[number] = occurences[number] + 1
    total = float (sum (occurences.values ()))
    percentages = dict ()
    for number in occurences:
        percentages[number] = float (occurences[number]) / total
    return occurences, percentages
```

Listing A.2: Statistic utilty for elite fibonacci

B

# REFERENCES

Here you'll find references to the figures, tables, listings and acronyms used throughout the thesis.

## LIST OF FIGURES

## LIST OF TABLES

## LISTINGS

## ACRONYMS

| | |
|---|---|
| PCG | Procedrual Content Generation |
| MASSIVE | Multiple Agent Simulation System in Virtual Environment |
| PRNG | Pseudo-Random Number Generator |
| BAM | Beneath Apple Manor |
| ECS | Entity-Component System |
| L-System | Lindenmayer-System |
| MVC | Model-View-Controller |
| MMO | Massive Online Multiplayer |
| MMVE | Massively Multi-user Virtual Environments |

[1] Elwyn R Berlekamp, John H Conway, and Richard K Guy. Winning ways for your mathematical plays, volume 4. *AMC*, 10:12, 2003. (Cited on page 72.)

[2] Axel Berndt, Raimund Dachselt, and Rainer Groh. A survey of variation techniques for repetitive games music. In *Proceedings of the 7th Audio Mostly Conference: A Conference on Interaction with Sound*, AM '12, pages 61–67, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1569-2. doi: 10.1145/2371456.2371466. URL http://doi.acm.org/10.1145/2371456.2371466. (Cited on page 27.)

[3] Emma Boyes. Q & a: David braben - from elite to today, 2006. URL http://web.archive.org/web/20140322152829/http://www.gamespot.com/articles/qanda-david-braben-from-elite-to-today/1100-6162140/. (Cited on page 13.)

[4] Dong Joo Byun, Henrik Falt, Ben Frost, Mir Ali, Eric Daniels, Peter De Mund, and Michael Kaschalk. Procedural animation technology behind microbots in big hero 6. In *ACM SIGGRAPH 2015 Talks*, SIGGRAPH '15, pages 40:1–40:1, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3636-9. doi: 10.1145/2775280.2792533. URL http://doi.acm.org/10.1145/2775280.2792533. (Cited on page 26.)

[5] Alessandro Canossa. Give me a reason to dig: Qualitative associations between player behavior in minecraft and life motives. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 282–283, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1333-9. doi: 10.1145/2282338.2282400. URL http://doi.acm.org/10.1145/2282338.2282400. (Cited on page 18.)

[6] B. Carter. *The Game Asset Pipeline*. Charles River Media Game Development. Charles River Media, 2004. ISBN 9781584503422. URL https://books.google.de/books?id=Sr2HKC46ImcC. (Cited on page 7.)

[7] Sherol Chen, Adam M. Smith, Arnav Jhala, Noah Wardrip-Fruin, and Michael Mateas. Rolemodel: Towards a formal model of dramatic roles for story generation. In *Proceedings of the Intelligent Narrative Technologies III Workshop*, INT3 '10, pages 17:1–17:8, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0022-

3. doi: 10.1145/1822309.1822326. URL http://doi.acm.org/10.1145/1822309.1822326. (Cited on page 26.)

[8] Arthur Charles Clarke. *The exploration of space*. Harper, 1959. (Cited on page v.)

[9] Kate Compton, James Grieve, Ed Goldman, Ocean Quigley, Christian Stratton, Eric Todd, and Andrew Willmott. Creating spherical worlds. In *ACM SIGGRAPH 2007 Sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM. doi: 10.1145/1278780.1278879. URL http://doi.acm.org/10.1145/1278780.1278879. (Cited on page 17.)

[10] Leonardo G. de Freitas, Luiggi Monteiro Reffatti, Igor Rafael de Sousa, Anderson C. Cardoso, Carla Denise Castanho, Rodrigo Bonifácio, and Guilherme N. Ramos. Gear2d: An extensible component-based game engine. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, pages 81–88, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1333-9. doi: 10.1145/2282338.2282357. URL http://doi.acm.org/10.1145/2282338.2282357. (Cited on page 36.)

[11] David (grue) DeBry, Henry Goffin, Chris Hecker, Ocean Quigley, Shalin Shodhan, and Andrew Willmott. Player-driven procedural texturing. In *ACM SIGGRAPH 2007 Sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM. doi: 10.1145/1278780.1278878. URL http://doi.acm.org/10.1145/1278780.1278878. (Cited on page 17.)

[12] Herman Arnold Engelbrecht and Gregor Schiele. Koekepan: Minecraft as a research platform. In *Proceedings of Annual Workshop on Network and Systems Support for Games*, NetGames '13, pages 16:1–16:3, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-2961-0. URL http://dl.acm.org/citation.cfm?id=2664633.2664652. (Cited on page 18.)

[13] Clara Fernández-Vara and Alec Thomson. Procedural generation of narrative puzzles in adventure games: The puzzle-dice system. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, pages 12:1–12:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1447-3. doi: 10.1145/2538528.2538538. URL http://doi.acm.org/10.1145/2538528.2538538. (Cited on page 26.)

[14] Dejobaan Games. Procedural content generation: Thinking with modules, 2012. URL http://web.archive.org/web/20150820154248/http://www.gamasutra.com/view/feature/174311/procedural_content_generation_.php. (Cited on page 31.)

[15] GameSpot. How does no man's sky actually work? - reality check, 2014. URL https://www.youtube.com/watch?v=ZVl1Hmth3HE. (Cited on pages 4 and 5.)

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612. URL http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1. (Cited on page 8.)

[17] Martin Gardner. Mathematical games the fantastic combinations of john conway's new solitaire game "life", 2009. URL http://web.archive.org/web/20090603015231/http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm. (Cited on pages 6 and 72.)

[18] Johan Gielis. A generic geometric transformation that unifies a wide range of natural and abstract shapes. *American journal of botany*, 90(3):333–338, 2003. (Cited on page 22.)

[19] Johan Gielis, Bert Beirinckx, and Edwin Bastiaens. Superquadrics with rational and irrational symmetry. In *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, SM '03, pages 262–265, New York, NY, USA, 2003. ACM. ISBN 1-58113-706-0. doi: 10.1145/781606.781647. URL http://doi.acm.org/10.1145/781606.781647. (Cited on page 22.)

[20] Daniel Johansson and Jan Schmid. Building the city of glass in mirror's edge&trade;. In *ACM SIGGRAPH 2015 Talks*, SIGGRAPH '15, pages 65:1–65:1, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3636-9. doi: 10.1145/2775280.2775282. URL http://doi.acm.org/10.1145/2775280.2775282. (Cited on page 26.)

[21] G KELLY and H McCABE. An interactive system for procedural city generation. *Institute of Technology Blanchardstown*, 2008. (Cited on page 25.)

[22] George Kelly and Hugh McCabe. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007. (Cited on page 25.)

[23] Ben Kenwright. Generating responsive life-like biped characters. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, pages 1:1–1:8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1447-3. doi: 10.1145/2538528.2538529. URL http://doi.acm.org/10.1145/2538528.2538529. (Cited on page 26.)

[24] Rilla Khaled, Mark J. Nelson, and Pippin Barr. Design metaphors for procedural content generation in games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 1509–1518, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2466201. URL http://doi.acm.org/10.1145/2470654.2466201. (Cited on page 3.)

[25] Raffi Khatchadourian. World without end, 2015. URL http://web.archive.org/web/20160206152806/http://www.newyorker.com/magazine/2015/05/18/world-without-end-raffi-khatchadourian. (Cited on page 22.)

[26] E. Lengyel. *Game Engine Gems 2*. EBL-Schweitzer. CRC Press, 2011. ISBN 9781439869772. URL https://books.google.de/books?id=ujfOBQAAQBAJ. (Cited on page 7.)

[27] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280 – 299, 1968. ISSN 0022-5193. doi: http://dx.doi.org/10.1016/0022-5193(68)90079-9. URL http://www.sciencedirect.com/science/article/pii/0022519368900799. (Cited on page 21.)

[28] T machine aka Adam Martin. Entity systems are the future of mmog development, 2007. URL http://web.archive.org/web/20160128043231/http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/. (Cited on page 36.)

[29] Jacques Malenfant, Marco Jacques, and Franois Nicolas Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of the Reflection*, volume 96, pages 1–20, 1996. (Cited on page 39.)

[30] Genaro Juarez Martinez. Introduction to rule 110, 2004. URL http://web.archive.org/web/20160118113708/http://uncomp.uwe.ac.uk/genaro/Papers/Papers_on_CA_files/introRule110/node10.html. (Cited on page 72.)

[31] Merriam-Webster. Dictionary definition of procedural, 2016. URL https://web.archive.org/web/20160114163401/http://www.merriam-webster.com/dictionary/procedural. (Cited on page 4.)

[32] Merriam-Webster. Dictionary definition of procedure, 2016. URL http://web.archive.org/web/20160114164249/http://www.merriam-webster.com/dictionary/procedure. (Cited on page 4.)

[33] Erica Naone. Creating creatures, 2008. URL https://www.technologyreview.com/s/410281/creating-creatures/. (Cited on page 17.)

[34] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966. (Cited on page 71.)

[35] Peter E. Oppenheimer. Real time design and animation of fractal plants and trees. *SIGGRAPH Comput. Graph.*, 20(4):55–64, August 1986. ISSN 0097-8930. doi: 10.1145/15886.15892. URL http://doi.acm.org/10.1145/15886.15892. (Cited on page 25.)

[36] Wojciech Palubicki, Kipp Horel, Steven Longay, Adam Runions, Brendan Lane, Radomír Měch, and Przemyslaw Prusinkiewicz. Self-organizing tree models for image synthesis. *ACM Trans. Graph.*, 28(3):58:1–58:10, July 2009. ISSN 0730-0301. doi: 10.1145/1531326.1531364. URL http://doi.acm.org/10.1145/1531326.1531364. (Cited on page 25.)

[37] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972. ISSN 0001-0782. doi: 10.1145/361598.361623. URL http://doi.acm.org/10.1145/361598.361623. (Cited on pages 8 and 9.)

[38] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985. ISSN 0097-8930. doi: 10.1145/325165.325247. URL http://doi.acm.org/10.1145/325165.325247. (Cited on page 20.)

[39] Sören Pirk, Till Niese, Oliver Deussen, and Boris Neubert. Capturing and animating the morphogenesis of polygonal tree models. *ACM Trans. Graph.*, 31(6):169:1–169:10, November 2012. ISSN 0730-0301. doi: 10.1145/2366145.2366188. URL http://doi.acm.org/10.1145/2366145.2366188. (Cited on page 25.)

[40] Sören Pirk, Till Niese, Torsten Hädrich, Bedrich Benes, and Oliver Deussen. Windy trees: Computing stress response for developmental tree models. *ACM Trans. Graph.*, 33(6):204:1–204:11, November 2014. ISSN 0730-0301. doi: 10.1145/2661229.2661252. URL http://doi.acm.org/10.1145/2661229.2661252. (Cited on page 25.)

[41] P. Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990. ISBN 0-387-97297-8. (Cited on page 25.)

[42] Psittacine. Beneath apple manor, 2007. URL https://web.archive.org/web/20110715125304/http://psittacine.com/beneath-apple-manor/. (Cited on page 11.)

[43] Hang Qi, Ruichao Qiu, and Jinyuan Jia. L-system based interactive and lightweight web3d tree modeling. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '11, pages 589–592, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1060-4. doi: 10.1145/2087756.2087871. URL `http://doi.acm.org/10.1145/2087756.2087871`. (Cited on page 21.)

[44] Aaron A. Reed. Sharing authoring with algorithms: Procedural generation of satellite sentences in text-based interactive stories. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, pages 14:1–14:4, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1447-3. doi: 10.1145/2538528.2538540. URL `http://doi.acm.org/10.1145/2538528.2538540`. (Cited on page 26.)

[45] Paul Rendell. A turing machine in conway's game life, 2001. URL `http://web.archive.org/web/20160114200316/http://www.cs.unibo.it/~babaoglu/courses/cas00-01/papers/Cellular_Automata/Turing-Machine-Life.pdf`. (Cited on page 72.)

[46] Marco Scirea, Yun-Gyung Cheong, Mark J. Nelson, and Byung-Chull Bae. Evaluating musical foreshadowing of videogame narrative experiences. In *Proceedings of the 9th Audio Mostly: A Conference on Interaction With Sound*, AM '14, pages 8:1–8:7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3032-9. doi: 10.1145/2636879.2636889. URL `http://doi.acm.org/10.1145/2636879.2636889`. (Cited on page 27.)

[47] F. Spufford. *Backroom Boys: The Secret Return of the British Boffin*. Faber & Faber, 2003. ISBN 9780571214969. URL `https://books.google.de/books?id=efp2QgAACAAJ`. (Cited on pages 13 and 14.)

[48] Kenichi Sugihara. Automatic generation of 3-d building models by straight skeleton. In *SIGGRAPH Asia 2011 Sketches*, SA '11, pages 24:1–24:1, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1138-0. doi: 10.1145/2077378.2077408. URL `http://doi.acm.org/10.1145/2077378.2077408`. (Cited on page 25.)

[49] Claudia Szabo and Yong Meng Teo. Post-mortem analysis of emergent behavior in complex simulation models. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pages 241–252, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1920-1. doi: 10.1145/2486092.2486123. URL `http://doi.acm.org/10.1145/2486092.2486123`. (Cited on page 16.)

[50] .theprodukkt. Website of .kkrieger, 2011. URL http://web.archive.org/web/20110717024227/http://www.theprodukkt.com/kkrieger. (Cited on page 15.)

[51] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0872-4. doi: 10.1145/2000919.2000922. URL http://doi.acm.org/10.1145/2000919.2000922. (Cited on page 3.)

[52] Julian Togelius, Jim Whitehead, and Rafael Bidarra. Guest editorial: Procedural content generation in games. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES*, 2011. (Cited on page 7.)

[53] Michael C. Toy and Kenneth C. R. C. Arnold. A guide to the dungeons of doom, 2003. URL http://web.archive.org/web/20030803003006/http://home.wanadoo.nl/loche/rogue/guide.txt. (Cited on page 11.)

[54] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546, 1938. doi: 10.1112/plms/s2-43.6.544. URL http://plms.oxfordjournals.org/content/s2-43/1/544.short. (Cited on page 72.)

[55] Valve. Store page for 'ftl: Faster than light', 2014. URL http://web.archive.org/web/20160216173733/http://store.steampowered.com/app/212680/. (Cited on page 12.)

[56] Pedro Verruma. Conway's game of life, 2009. URL http://web.archive.org/web/20150318020910/http://pmav.eu/stuff/javascript-game-of-life-v3.1.1/. (Cited on page 71.)

[57] Greg Walsh, Craig Donahue, and Emily E. Rhodes. Kidcraft: Co-design within a game environment. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, pages 1205–1210, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3146-3. doi: 10.1145/2702613.2732921. URL http://doi.acm.org/10.1145/2702613.2732921. (Cited on page 18.)

[58] T. Watzl. Procedural modeling of buildings based on patterns in the form of Shape-Grammar. 2015. (Cited on page 25.)

[59] Glenn R. Wichman. A brief history of "rogue", 1997. URL http://web.archive.org/web/20071217204920/http:

//www.wichman.org/roguehistory.html. (Cited on pages 11 and 12.)

[60] Dwarf Fortress Wiki. World generation, 2014. URL http://web.archive.org/web/20160205101531/http://dwarffortresswiki.org/index.php/World_generation. (Cited on page 16.)

[61] Andrew Willmott. Fast object distribution. In *ACM SIGGRAPH 2007 Sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM. doi: 10.1145/1278780.1278877. URL http://doi.acm.org/10.1145/1278780.1278877. (Cited on page 17.)

[62] Ken Xu and Damian Campeanuy. Houdini engine: Evolution towards a procedural pipeline. In *Proceedings of the Fourth Symposium on Digital Production*, DigiPro '14, pages 13–18, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3044-2. doi: 10.1145/2633374.2633378. URL http://doi.acm.org/10.1145/2633374.2633378. (Cited on page 27.)

## COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*. `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured at:

http://postcards.miede.de/

*Final Version* as of March 2, 2016 (`classicthesis` version 4.2).

# DECLARATION

I hereby declare to have worked on this thesis on my own and to have marked sources appropriately.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe.

*Hamburg, March 2016*

Sven Freiberg