AN INTEGRATED PLANNING AND CELLULAR AUTOMATA BASED PROCEDURAL GAME LEVEL GENERATOR

YURI PESSOA AVELAR MACEDO

AN INTEGRATED PLANNING AND CELLULAR AUTOMATA BASED PROCEDURAL GAME LEVEL GENERATOR

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtencão do título de Mestre em Ciência da Computação

ORIENTADOR: LUIZ CHAIMOWICZ

Belo Horizonte
Agosto de 2018

YURI PESSOA AVELAR MACEDO

AN INTEGRATED PLANNING AND CELLULAR AUTOMATA BASED PROCEDURAL GAME LEVEL GENERATOR

Master's Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment for the acquisition of the degree of Master in Computer Science.

ADVISOR: LUIZ CHAIMOWICZ

Belo Horizonte

August 2018

© 2018, Yuri Pessoa Avelar Macedo.
Todos os direitos reservados

Macedo, Yuri Pessoa Avelar.

M141i An integrated planning and cellular automata based procedural game level generator [manuscrito] / Yuri Pessoa Avelar Macedo. – 2018.
xxiii, 108 f. il.

Orientador:. Luiz Chaimowicz.
Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f.85-90
.
1. Computação – Teses. 2. Geração procedural de conteúdo – Teses. 3.Inteligência artificial – Teses. 4.Jogos eletrônicos – Teses. I. Chaimowicz, Luiz. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III.Título.

CDU 519.6*34(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

An Integrated Planning and Cellular Automata Based Procedural Game Level Generator

YURI PESSOA AVELAR MACEDO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. LUIZ CHAIMOWICZ - Orientador
Departamento de Ciência da Computação - UFMG

PROFA. RAQUEL OLIVEIRA PRATES
Departamento de Ciência da Computação - UFMG

PROF. BRUNO FEIJÓ
Departamento de Informática - PUC-Rio

Belo Horizonte, 22 de novembro de 2018.

This work is dedicated to everyone within my friends and family that have encouraged me to get this far. To all of you, thanks. Your guys are awesome, and you have helped to make me feel awesome too. I couldn't have done this without all of you.

Acknowledgments

Shout-outs to Dr. Luiz Chaimowicz for having an endless supply of patience to help a eighteen-year-old me who was so unsure of things, he went to the Dr.'s office to second guess his own academic choices no sooner than his third day of class. It was a rough beginning, but one that made me friends I wish to carry for life, just as they carried me through the trials of the first half of College. Only with the guidance of Chaimowicz and that of other amazing teachers such as Gisele L. Pappa, Adriano Alonso Veloso, and Renato A. Ferreira, that the Yuri from seven-and-a-half years ago who was pretty much uncertain about everything, was shaped into a Yuri that is kind of less uncertain about things...Maybe...I think...Yeah.

Jokes aside, this necessity get stressed about anything and everything has cost me and those around me dearly. A price paid with efficiency, health, sanity, and sleep. The latter of which, for those who do not know, is regular practice that humans do a little bit of. And also that, for almost two years now, I physically and mentally **couldn't**. A state triggered by an aggregate of scars left by my own uncertainties and fears. Many of which were ramifications of the very early passing of my little brother, Yan, fourteen years ago.

That which shackles me is being torn, however. And I couldn't have mustered this strength without the help and encouragement from my mom and dad: Rosemeire Pessoa, and Vladimir Avelar Macedo. They who did not only support me just about every day, but also banked medical bills longer than this here chunk of paper.

Soothing to my rage was the gentle caress of my beloved one, Alice. Who I've had the uncanny luck to meet, and the courage to call on a date only five months before everything went down. She gave me the optimism I've never had, to the point where I now can have some to call my own. I'm not even sure whether I would still be here, weren't for her and my family being by my side. A would-be gloom thought that I instead now ponder with happiness for my luck to have them with me, rather than with dread for what could have been.

Finally, of course, this two-and-a-half year work that you're about to read wouldn't be possible without the comprehension of my professor adviser, (again) Luiz Chaimowicz, who has also helped me keep my head cool whenever I've doubted my own work; To PPGCC,

who has allowed me to take this long to deliver this; And to CAPES, who's granted me the financial support that allowed me to put all of my efforts into it. Thrice had I the opportunity to share my academic works within and beyond my country over the years, and each was only possible due to the exceptional aid of these three.

I've already gone past the first page, yet I can't say any less. I guess my point is, thank you. Everyone.

Abstract

Procedural Content Generation for Games (PCG-G) is a field that in the past few years has seen both extensive academic study and practical use in the games industry. While it isn't the panacea to solve the problem of suppling the endless demand for content in games, it has proven to be a powerful tool-set that some successful games could not thrive without. PCG-G, when properly applied, is beneficial to both large and small game companies looking to optimize scalability.Likewise, when not properly applied, it can doom any project to mediocrity. Games require very distinct types of content to be designed and implemented, and the research field of procedural content generation aims to design, survey, contrast, and improve, upon methods that apply to distinct domains. One of these domains is the generation of game levels, a branch of great interest for developers looking to improve their games' replayability. With the intent of contributing to the existing discussion of procedural level generation methods, as well as improving upon the established baseline, this work proposes the designing of procedural levels through the integration of AI Planning and Cellular Automata. The product levels of this process are implemented and polished on the Unity Game Engine, as to present their potential for entertainment through qualitative evaluation. Results show that this methodology can be used for the generation of organic and cohesive game levels.

Keywords: Procedural Content Generation, Artificial Intelligence, Digital Games

Resumo

Geração Procedural de Conteúdo em jogos (PCG-G) é um campo de estudo que nos últimos anos presenciou tanto extensivo estudo acadêmico quanto uso prático na indústria de jogos. Mesmo que Geração Procedural não seja a panacéia que resolverá o problema de gerar conteúdo infinito para jogos, este paradigma mostrou-se como uma poderosa ferramenta pela qual diversos jogos só atingiram seu sucesso através dela. PCG-G, quando devidamente aplicada, é beneficial para empresas grandes e pequenas de jogos que estão em busca de otimizar a escalabilidade de seus projetos. Da mesma forma, quando indevidamente aplicada, Geração Procedural, pode fadar projetos à mediocridade. Jogos requerem tipos bem distintos de conteúdo que precisam ser desenvolvidos e implementados, e a busca por métodos de Geração Procedural visam propor, sondar, contrastar, e melhorar métodos aplicados a diferentes domínios. Um destes domínios é a criação de níveis de jogo, que é um ramo de alto interesse para desenvolvedores interessados em melhorar a re-jogabilidade de seus jogos. Com a intenção de contribuir com a discussão existente sobre métodos de geração de níveis, e também para melhorar o baseline existente, este trabalho se propõe o design de níveis procedurais por meio da integração de AI Planning e Autômatos Celulares. Os níveis de jogo criados por esse processo foram implementados e polidos na Unity Game Engine para apresentar seu potencial para entretenimento por meio de avaliações qualitativas. Resultados demonstraram que essa metodologia pode ser usada para geração de níveis de jogo orgânicos e coesos.

Palavras Chave: Geração procedural de conteúdo, Inteligência artificial, Jogos eletrônicos

List of Figures

3.1 The Cellular Automata generated by rule 30, 54, and 60 of 'The 256 Rules' by Li and Packard [1990]. 18

3.2 Von Neumann and Moore neighborhoods for a single cell (colored in black). The neighborhood's size 'r' can be increased to consider the states of additional nearby cells. 19

3.3 A Multi-layered Cellular Automaton schematic drawing as presented by Nakayama et al. [2015]. 20

3.4 With the production rules being applied M times to the axiom, a curve of length $M^2 = N$ is obtained. The following Hilbert Curves are mapped to a NxN grid, for the values of $N = 1,2,4,8,16$, and 32, respectively. 21

4.1 General structure and flow of our methodology. Beveled boxes represent different programmed modules, document shaped boxes represent text files, and rounded boxes represent resulting constructs. Each component encapsulated within the gray box is implemented and executed within the Unity Engine. Arrows from a box source that have as their destination a construct (rounded) box indicate that the source has generated that construct. All other arrows that have their destination as a non-construct box indicate that their source is used as input to their destination. 23

4.2 Example of multiple variable tree hierarchies within a 'character creator' used for our language's preliminary testing, as displayed on a custom interface programmed on Unity. 27

4.3 Example of the goal function for the 'character creator' problem represented as a tree of logical functions. 29

4.4 Example from part of a Domain Library designed for preliminary testing. In this test Domain Library, the planner must create a character from the game *Dungeons & Dragons* in accordance with the game's rules. The first image contains the group of actors that represent races. The second contains states that describe how much of the generation process has been completed. And the third image contains one of the actions that define one of the character's properties. 30

4.5 Example action in the map theme generator domain library. This planning problem action takes no input parameters (meaning also that there are no constraints to define which parameters that are acceptable). Has as a precondition function that there are no _Temperature(), and no _Vegetation() states active. 31

4.6 Example final states configuration of the planner algorithm. The result is a set of state(actor) that is a subset from all possible valid combinations of states and their non-constraint violating actors. 32

4.7 Three versions of the same Cellular Automaton generated from different updating methods (see Chapter 3). From left to right, these methods are: synchronous, asynchronous with cells updated in a random order, and asynchronous with cells updated sequentially ordered by their position in the grid. Black cells are $true$, white cells are $false$. 33

4.8 From left to right, the first four steps described in Sub-section 4.2.2 are represented: (1) Plotting, (2) Scaling, (3) Shifting, (4) Tracing. 37

4.9 To the left, the Cellular Automata marked with the Space-filling Curve's path (green), as well as the negative-path curve (yellow). To the Right, the resulting automata after a number of iterations until it stabilizes. Parameters for this Automata are as follows: $N = 100$, $M = 100$, $Fill = 0.5$, $S = 22$, $P_g = 1$, $N_g = 1$. The colored dots on the right image are the points from the Space-filling Curve. 38

4.10 The same automata presented in Figure 4.9 after rotation and shifting operations are introduced. 40

4.11 The complete sequence of steps for the generation of the procedural map automata. The images for the 'Space-filling' curve section are merely illustrative and do not represent the same curve-path displayed in all other images. 40

4.12 Stacking of multiple layers of cellular automata, each responsible for one element type in the map. 41

4.13 A base automata, a topology layer-mask generated from automata, and a mapped version of the topology automata to 2D sprites. 42

4.14 From the base layer, to the topology layer-mask, 2D tiles are generated depending on their position in the topology mask, creating the aspect of a continuous 'natural cliff' structure. The visual quality of the resulting tiles map (right image) is the result of editing and experimenting with selected free visual assets from Websites such as ope [2017]. As of now, we cannot present a theoretical basis as how to optimize the visual quality of matching 2D tiles. 43
4.15 Evolution of the 'tree-generating automata' without the introduction of space-filling curves (Parameters: $N = 100$, $M = 100$, $Fill = 0.01$). The greener the cell, the older it is compared to the other trees. As trees need to be '5 iterations old' for them to start spawning other trees, the automata changes the most every 5 iterations. This age restriction also prevents trees from spreading wildly, instead forming small forests. 44
4.16 Example of 2 layers of CA 45
4.17 C# Script component that takes references for the planner library, the planning solver, and the map generator (respectively), and filters 6 types of states from the resulting planner's solution (a previous example of a solution is as shown in 4.6). Then, actors from the planning Domain Library are mapped (a String to Prefab dictionary) onto an Automaton Prefab, which is stored on one of the project's folders. 46
4.18 Unity Component for Map Generation. Information about distinct parts of map generation are separated in drop down menus, as is the case with the open 'Layer Execution Order' segment. Graphical interfaces such as this have been made for all components of planning, map, and their integration. 47

5.1 Mosaic with different maps generated with different random seeds. Maps may have different sizes. 56
5.2 Different versions of Map Geometry achieved by variating the random seed. Parameters for these automata are as follows: $N = 100$, $M = 100$, $Fill = 0.5$, $S = 22$, $P_g = 1$, $N_g = 1$ 57
5.3 Map Geometry experiments with the Fill parameter for different values as to find a compensation for the reduction in random cells by the imprinting of paths. . . 58
5.4 Variations of the same 'theme' for a Map with different random seeds for its Automata. 59
5.5 Two maps with the same random seed. On the left map, the cliff topography layer iterates before the river generating layer. On the right map, the river generating layer iterates first. 59

5.6 A segment of the results of the tracker condensed into multiple tables. The tracker shows, for a number of instances of the map generator (in this case 1000)), what is the frequency of each layer being in the set of layers chosen for the final map. Certain groups of Layers, such as the 'Stage1Complete()' and an additional set of topography layers are used as intermediary states for the creation of the map. 60
5.7 Plotting of the time required for the generation of maps of various sizes, with distinct random seeds. 61
5.8 Unity's profiler tool measuring, to the left, the CPU Usage and Allocated Memory over time for an average sized (aprox. 75x75 cells) map. To the right is a detailed measurement of the percentage of memory consumption per type of component in the game in the current frame (marked by the white vertical line on the left side). The most important information to pull out from the profiler is it's first line: 'FixedUpdate.Physics2DFixedUpdate' is the overhead required to evaluate the physics properties of the map's objects and taxes 58.5% of Unity's allocated memory. 61

6.1 Screenshot of the developed game. 68

List of Tables

2.1 Academic works related to games that used either the qualitative analysis paradigm or used techniques common, but not exclusive, to qualitative analysis. 12

4.1 Table of notable properties of the ADL planning language, as organized by Russell and Norvig [2016] 26

6.1 Division of maps among volunteers of groups A and B (numbered 1 to 6). The maps were labeled by number. Any map starting with 'S' is a small map, and maps starting with 'M' are medium sized maps. Before any testing, maps were divided into volunteers in a way that all but 2 medium sized maps were tested by a total of 2 volunteers. 67

6.2 Data from the volunteer's answers regarding topics about their entertainment. Both are organized as 0's (no) and 1's (yes). 73

6.3 Data from the volunteers' answers, displaying averages and totals (when relevant) for each closed topic. Separated column entries for groups A, B contain the sums and averages for each entertainment related topic in Table 6.2. 73

6.4 Data from the volunteers' answers about topics regarding their opinions on the maps' aesthetics. **Found maps visually engaging** is organized as 0's (no) and 1's (yes). **Number of Remembered levels** is a question that emerged from the original protocol. It accounts the number of levels the volunteer could remember and accurately describe. 74

6.5 Data from the volunteers' answers, displaying averages and totals (when relevant) for each closed topic's answers. Separated column entries for groups A, and B contain the sums and averages for each aesthetic related topic in Table 6.4. 74

6.6 Data from the volunteers' answers about topics regarding their understanding of the maps' structure. Most information is organized as 0's (no) and 1's (yes). Two exception topics have distinct values: **Room to Explore** is an integer from 1 to 5: A value of 1 means that there was no freedom to explore, a 5 means that the amount of explorable areas was overwhelming, and a 3 being the perfect balance; **Understood the Map** is a simple grade from 1 to 5 of how well the map was understood. 75

6.7 Data from the volunteers' answers, displaying averages and totals (when relevant) for each closed topic. Separated column entries for groups A, and B contain the sums and averages for each layout related topic in Table 6.6. 75

6.8 Data from the volunteers' answers regarding the implementation of PCG on the generated maps. **Found the maps visually interesting**, **Would develop a game with this generator**, **Noticed it was about PCG**, and **Asked if it was about PCG during play** are answered as 0's (no) and 1's (yes). The remainder topics were open answers that were condensed into one or two sentences for convenience 77

6.9 Data from the volunteers' questions, displaying averages and totals (when relevant) for each closed topic. Separated column entries for groups A, and B contain the sums and averages for each PCG related topic in Table 6.8. 78

D.1 Data from the volunteers' questions. The vertical axis contains each stated opinion, while the horizontal contains answers relative to each topic. Most information is organized as 0's (no) and 1's (yes). A few exception topics have distinct values: **Room to Explore** is an integer from 1 to 5: A value of one means that there was no freedom to explore, a 5 means that the amount of explorable areas was overwhelming, and a 3 being the perfect balance; **Understood the Map** is a simple grade from 1 to 5 of how well the map was understood; **Got Lost** is a ternary scale with 0 meaning the volunteer did not get lost, with 2 meaning that they have gotten lost, and 1 meaning that, while getting lost, the volunteer did not find it to be a bothersome problem. Finally **Number of Remembered levels** is just that, an accounting of the number of levels the volunteer could remember and accurately describe. 105

D.2 Data from the volunteers' questions, displaying averages and totals (when relevant) for each topic. Separated column entries for groups A, B contain their answers' sums and averages. For a better understanding of the meaning of each average, it would be best to check the information described in Table D.1's caption. 106

D.3 Data from the volunteers' questions displayed in the same format as Table D.1. These answers regard questions exclusive to Group B. The first and last rows are closed binary answers, while the other remaining three have opinions condensed into one or two sentences. 106
D.4 Data from the volunteers' questions presented in Table D.3, displaying averages and totals for PCG related topics presented exclusively to Group B. 107

Contents

Acknowledgments **ix**

Abstract **xi**

List of Figures **xiii**

List of Tables **xvii**

1 Introduction **1**

1.1 Motivation 2

1.2 Goals 3

2 Literature Review **5**

2.1 Presenting PCG-G 5

2.2 Current state of PCG-G 7

2.3 AI Planning 8

2.4 Map Generation 9

2.5 Integrated Map Generators 10

2.6 Qualitative Analysis in Games 11

2.7 Discussion 12

3 Background **15**

3.1 AI Planning 15

3.1.1 The Basis of Planning Languages 16

3.2 Cellular Automata 18

3.2.1 Synchronous vs. asynchronous Updating 19

3.2.2 Multi-layered Cellular Automata 19

3.3 Space-filling Curves 20

4 Methodology **23**

4.1 Level Planning 24
4.1.1 Planning Language 25
4.1.2 Hierarchical Classes 26
4.1.3 ’Unknown’ Operator 27
4.1.4 Planning Algorithm 28
4.1.5 Domain Library 29
4.1.6 Example: Map Theme Generator 31
4.2 Map Automata 32
4.2.1 Cellular Automata 32
4.2.2 Space-filling Curves 34
4.2.3 Securing Negative Space 37
4.2.4 Polishing 38
4.2.5 Stacking Cellular Automata 41
4.2.6 Example 1: Topology Layer 42
4.2.7 Example 2: Tree Generating Layer 43
4.3 From Planning to Map Generation 44

5 Quantitative Analysis 49
5.1 Resulting Maps 49
5.2 Map Geometry 50
5.3 Map Generation Variety 51
5.4 Theme Planning Variety 52
5.5 Performance 53

6 Generator Analysis 63
6.1 Volunteer Selection 65
6.1.1 Consent Form & Information security 66
6.2 Map Distribution 66
6.2.1 Unity Game 67
6.3 Testing Protocol 69
6.3.1 Observation 70
6.3.2 Interview 71
6.3.3 Pilot Tests 72
6.4 Results 72
6.4.1 Engagement 72
6.4.2 Aesthetic 74
6.4.3 Layout 75

6.4.4 PCG 76
6.5 Final Observations 79

7 Conclusion **81**

Bibliography **85**

Attachment A Map Generation Planning Library **91**

Attachment B Consent Form **99**

Attachment C Interview Diagram **103**

Attachment D Qualitative collection **105**

Chapter 1

Introduction

As the games industry grows, creating sufficient content to satisfy the customers' demands is proving itself to be a struggle that for some companies might as well be impossible. Takatsuki [2007] stated that the cost of manually generating content outpaces sales and revenue, to the point where some companies are unable to follow the race for games of greater scale. As recorded by Benson [2013], even the greatest titans in the industry are not immune to this effect. The struggle to create ever more expensive games has lead large companies into searching for alternatives for generating greater volumes of content.

On the other hand, Independent developers face their own struggles with creating more for their games with fewer resources. Although their conflict is not of predatory competition, the similar lack of resources and small teams can be the catalyst for searching alternative methodologies for content creation. One such method for effectively dealing with resource limitations is the paradigm of creating content automatically through algorithmic methods, generally called Procedural Content Generation (PCG).

Procedural Content Generation is an area of computer science applied mostly on computer graphics and digital games that consists of methods and algorithms that automatically create content based on a set of rules. Within the realm of PCG, there are multiple types of content that can be generated. Some methods are centered around the creation of environments (landscapes, mountains, trees, waterfalls, etc.) that are meant to be visually impressive and diverse by emulating the rules of nature. In the context of games, procedural content can be expanded into creating resources that directly affect gameplay, such as items, abilities, enemies, Artificial Intelligence, or even the game's rules themselves.

The topic of Procedural Content Generation for Games, which was first abbreviated as PCG-G by Hendrikx et al. [2013], had one of its first documented cases by the end of the 70's. A time when most designers' objectives were not to thoroughly optimize monetary expenses, but instead to overcome storage limitations. Enter Telengrad, an Atari/DOS

dungeon crawler created by Lawrence [1976] that featured enemies, treasures, and dungeons with millions of explorable rooms. This seven-digit amount of content was stored in just about 50 KB of memory (with earlier versions of the game requiring as little as 8 kb). Being able to achieve a game with this much content was unprecedented at the time, and it still would be in this day for any type of handcrafted levels. Instead, to create more with less, Telengrad had its dungeons generated during execution time algorithmically.

Despite remaining as a low-profile development methodology and as a research topic for three decades, it was mostly within the last fifteen years that developers, both large and Indie, have turned to PCG-G as a tool for supplying the demand for diverse, scalable content. This rise in popularity can be attributed to several reasons: a PCG algorithm holds promise for being able to generate an endless amount of whatever content it specializes in; it holds the possibility of greatly reducing development costs, as documented by Lee [08 7]; finally, having the game create unique content each time it is played could potentially bring new, innovative experiences every time.

As a means to add to the discussion on procedural generation, this work implements a procedural level generator that integrates a procedural map generator module, and a classical planner module. The planner, which functions with a variation of the STRIPS and ADL languages, decides which subset of mutually cohesive elements should be added to the level. The map generator, which is based on stacked layers of Cellular Automata, takes this subset of elements instantiates each one while being mindful of their interactions. As a means to bridge the gap between academic research and game development, the proposed architecture has been implemented onto the Unity: A professional standard game engine.

1.1 Motivation

For the game developers, a procedural generation methodology has proven not to be the panacea for cost savings. Instead, PCG introduced itself as a game design paradigm that presents several trade-offs, many of which are best discussed by Shaker et al. [2016b]: on one hand, procedural content can reduce workload and consequently costs for generating content. However, it can only do so if the amount created supersedes the cost of developing a highly specialized and robust generation framework; second, if the target game for the PCG-G algorithm will not make full use of hundreds, thousands, or millions of said content, then it might be cheaper to assign resources towards having it handcrafted instead. This almost always yields better results, as the procedural constructs today are still not as expressive, detailed, or engaging as that which was created by experienced developers. Therefore, even if a generator holds prestige for its quality, one must still ponder if the having more of said

content compensates for the loss of the 'human touch'.

Having lightly gone over the dilemmas, advantages and disadvantages of Procedural Content, it is important to address the importance of PCG in games. For some developers, Procedural Content is a necessary step to meet consumer demand. For others, it is a potential tool for lowering costs or to attain better scalability. PCG is a field with potential that still has a long path to thread before it can consistently meet its expectations. As such, it is deserving of the attention it receives. The amount of related academic works is steadily growing, and so is the development of games that utilize it, both for commercial and experimental purposes.

Procedural Content Generation has been applied with critical acclaim within indie titles such as *Binding of Isacc*[1] and *Spelunky*[2]. These projects have assumed the risk to implement procedural generation as a core feature of their games and have benefited greatly from it. Gaming companies have been using procedural generation for creating environments for years, but now there are games that utilize procedural content to influence gameplay such as the encounters of *The Elder Scrolls V: Skyrim*[3], and the item generators within the *Borderlands*[4] franchise, and *World of Warcraft*[5] as well.

PCG-G as a field of research evolves as it is documented and refined taxonomies emerge, such as the ones presented by Cheong et al. [2016]; Hendrikx et al. [2013]; Togelius et al. [2011]. Different methodologies are theorized, while practical implementations are put to practice in the games industry. These examples and many others have only begun to show the potential of procedural generation as a solid paradigm for game design.

1.2 Goals

One of the sub-topics of PCG-G that has seen great application within the industry, as surveyed by Hendrikx et al. [2013], is the generation of Game Space: a category of PCG that consists of algorithmically crafting the environment in which the game takes place. For most games that allow the player to influence a character's movement within in a intractable area, this space is a 'Map' that is contained within one of the game's 'Levels' (the whole of all game elements within a section of play). There are a plethora of academic works discussing methodologies for Map Generation, yet we feel that many still refrain from discussing how playable, entertaining, or 'game-like' resulting maps can be.

Cellular Automata (CA) time and again has shown to be a promising model due to its flexibility to represent organic and populational behaviors. Case in point, CA is widely used

[1] http://bindingofisaac.com/
[2] https://www.spelunkyworld.com/
[3] https://elderscrolls.bethesda.net/skyrim
[4] https://borderlands.com/en-US/
[5] https://worldofwarcraft.com

for modeling vegetation, ecology, and human society dynamics, as documented by Balzter et al. [1998]; Hogeweg [1988] and Liang and Wang [2017], respectively. Albeit a powerful model, it still requires extensions that account for multiple types of behaviors that interact with one another and that are possibly mutually restrictive. Furthermore, as an effort to grow upon the field of generating procedural maps for game levels, this work establishes five goals:

1. Improve upon the established methods of Cellular Automata Map Generation
2. Propose the use of a procedural multi-domain tool to assist procedural generators modeled as Classical AI Planning problems.
3. Integrate the architectures developed for achieving 1 and 2.
4. Bridge the gap between game research and game development by having 3 implemented onto an industry standard game engine.
5. Conduct a qualitative evaluation of the effectiveness of 1,2 and 3 through 4.

The proposed and implemented methodologies aim to be flexible enough as to be easely adaptable to distinct domains (game genres, perspectives, themes, etc.) other than the ones chosen for the purposes of this work. By integrating the modules for goals 1 and 2, this work tackles two of the greatest challenges of PCG-G: (1) Interfacing & controllability for PCG systems; (2) Interaction & opportunistic control flow between generators. By taking the time to implement our architecture within an industry standard game engine, we intend to reinforce our methodology as a viable, design friendly, and scalable possibility for procedural generation in commercial games. Finally, by conducting a qualitative analysis, we measure the abstract concept of entertainment and fun achieved by our integrated procedural generator.

The remainder of this work is divided as follows: Chapter 2 briefly covers a few of the studies that analyze PCG as a field of research and as an industry tool. It also reviews the progress and speculation on planning languages as game AI tools, followed by some methods that have been proposed or related to the generation of maps/levels, and to what other procedural systems these can be integrated to. Chapter 3 goes over the basic concepts of the research areas from which each of our methodologies draw from. Chapter 4 describes in full detail how we have achieved goals 1 through 4, our generation processes, and the algorithms involved with it. Chapter 5 briefly discusses the generated maps, providing a plethora of examples as well as performance data for the reader to formulate their own opinions. Chapter 6 presents details on our interview protocol, script and results. Chapter 7 elaborates on our final considerations and possibilities for improvements and future works.

Chapter 2

Literature Review

This chapter will provide a brief overview and references for related works: (1) It refers to those that classify, survey or otherwise offer a comprehensive entry point for understanding the basics of Procedural Content Generation; (2) It introduces the current state of procedural methods applied to other areas; (3) It presents how planning has been introduced as a solid choice for the basis of artificial intelligent agents' in the game industry, as well as its potential for integration with procedural generators; (4) It glances over the current state of academic research on procedural map generation as presented by the works of the last fifteen years; (5) This chapter wraps up describing attempts to integrate procedural map generation with other procedural methods.

2.1 Presenting PCG-G

As a general overview for most types of Procedural Generation, Shaker et al. [2016b] summarizes various works within the area, comparing and contrasting them. It serves as a great entry point onto the world of PCG-G research, and aids in consolidating much of the sporadic taxonomy that has emerged from different works. Hendrikx et al. [2013] covers the practical uses of procedural content generation, defining a six-layered taxonomy that intends to cover all types of procedural content (space, systems, scenarios, design). It also analyzes commercial games that use PCG, discerning the types of procedural generators implemented by each.

Togelius et al. [2013] proposes three types of PCG oriented designs that are defined by how strongly procedural content influences the game it was designed for: (1) Multi-Level Multi-content generation, where multiple structures of a game *(terrain, vegetation, roads, cities, people, etc.)* are procedural; (2) PCG-based Game Design, where procedural content generation is a core mechanic by which the game could not exist without (as in games that

have 'unlimited' areas to explore); And (3), generating complete games, where even the game's underlying mechanics such as physics, win conditions, and the most basic rules are generated. Its contribution also extends to identifying the inherent problems of PCG-G, by defining nine core challenges:

1. **Non-generic, Original Content:** Design generators that do not fall into generic, uninspiring patterns.
2. **Representing Style:** Having the generator emerge with the concept of a perceivable 'style' of content creation.
3. **General Content Generators:** Dealing with the limitations of PCG systems tied to their original game's rules and domain.
4. **Search Space Construction:** Designing how changes between parameters define the space of possible constructs.
5. **Interfaces and Controllability for PCG Systems:** Allowing for control of the generated constructs through input parameters.
6. **Interaction and Opportunistic Control Flow Between Generators:** Interaction between generators.
7. **Overcoming the Animation Bottleneck:** Supplying the high number of procedural constructs with at least mediocre animations.
8. **Integrating Music and Other Types of Content:** Creating dynamic sound design that fits unforeseen game scenarios.
9. **Theory and Taxonomy of PCG Systems:** Comparing distinct approaches based on their effectiveness, methods, and domains.

For games featuring more than one type of procedural content generation, Togelius et al. [2013] suggests that blandness or incoherence can be associated to the lack of integration between generational modules. Yet, not many works are centered around integration, even though it is one of the challenges of Procedural Content Generation. The difficulty of integrating PCG systems stems from their lack of malleability, as most procedural generators can not afford to be designed for allowing significant input from the designer, or from other systems into the generative process. As a result, the difficulty of integrating procedural systems is a product of items 5 & 6 of Togelius et al. [2013] list.

2.2 Current state of PCG-G

Although procedural generation has grown to be a reliable paradigm of game design, it has acquired a general reputation for creating repetitive content within some implementations, as well as not meeting up players' expectations, such as in the recent game *No Man's Sky*. This however cannot be generalized to every PCG algorithm, due to its strict domain dependence: each procedural content algorithm has to be tailored or adapted to each specific game. The academic community and the gaming industry are aware that no general purpose generators are means by which to replace the designer, but serve as a way to increase the available content for games that rely on having vast amounts of it.

One of the classical examples of procedural structures are Fractals. Exemplified by the Space-filling Curves described further in this work, these recursive structures are widely used as computationally efficient methods for procedural generation as exemplified by Ebert [2003]'s 3D environments, as well as the impressive models for topology and erosion simulations by Olsen [2004]. Fractals, however, are not effective as parametrized systems. A characteristic that is desired for a game map generator. This is mainly due to the fact that they require and accept only a random seed as parameter to generate their shapes, restricting any alterations to be done exclusively after the generation process.

One way of optimizing the quality of generated content is to explore a batch from all possible procedural constructs, and return an instance from that group that best fits some type of heuristic. Search based procedural content generation does just that: it is a methodology that attempts to return the 'best' combination of procedural elements, given a set of parameters. It does so by analyzing a subset from all the possible combinations of elements that create a procedural system. This approach is greatly adaptable to distinct types of content, and it is most interesting when the generated content aims to fulfill multiple objectives or criteria, as in by Togelius et al. [2013]. By integrating distinct types of generated content, Hartsook et al. [2011] attempts to tackle search based methods to integrate procedural map configurations that support 'Quests' and narratives.

In exploration or role-playing games, 'Quests' or 'Missions' that orient the player's objectives are a core part of the experience. Quest generating systems have been implemented within games that require a large amount of content, such as Massive Multiplayer Online Role-Playing Games (MMORPGs). These quests however can feel like chores with no connection to what is happening in the world around them. As a means to improve the personal engagement of procedural quests, a solution proposed by Togelius et al. [2013] is to integrate them with story generation Systems. One such example of these systems being Ciarlini et al. [2005], which proposes a temporal logic model for storytelling that, when stimulated by player intervention, create dynamic plots.

Lastly, the topic of Player modeling is explored within many works, and can be used to orient procedural generators. This approach is promising, as the adaptation of content to the preferences or actions of the player increases immersion and flow when well executed, as explained in Drachen et al. [2009]. There are many kinds of systems within a game that can be procedural, and thus there are many areas that may benefit from player modeling. Currently, however, player modeling and procedural generation are mostly integrated to rule over the Artificial Intelligence of Non-Playable Characters (NPCs) as in Riedl et al. [2011], or even the game narrative itself, as in Biggs et al. [2008a].

2.3 AI Planning

Automated Planning and Scheduling, or AI planning, has seen little association with games in academic works, but it has stealthy introduced itself into the industry. Its first documented instance was within the game *F.E.A.R.*[1], which featured a *STRIPS* based planner applied to the game's enemy NPCs. Although extensive optimizations were required to plan the actions of multiple agents, the game took its players by surprise with adversaries that had realistic and unpredictable behaviors, and was critically acclaimed for its 'intelligent' AI. Since then, *F.E.A.R* has kept planning the core of its AI and has inspired other games to attempt similar planning based strategies.

In a practical study of the uses of planning in games, Champandard [2013] has conducted interviews with multiple game developers regarding the basis for their NPC AI, and has concluded that there is a spectrum between the simplistic, yet effective Behavior Trees, and planning based approaches. In broad strokes, if the project has nonlinear scripted sequences, or the decision space is vast enough that the combination of choices generates unpredictable results, it benefits from planning. Otherwise, in closed/controlled environments that present fewer choices for the AI, or games that require multiple instances of agents making quick decisions, a classical Behavior Tree would be the better choice. In between, games such as *Killzone II*[2] have found that Hierarchical Task Networks (HTN), as formalized by Erol [1996], would perform effectively as a middle ground between planning and behavior tree strategies.

As a final commentary about this work, Champandard [2013] has established that as short term expectations, we were likely to see planning being applied to games other than shooters, and to domains other than enemies. As a long term expectation, agents based on planners would generate more believable character behaviors. As of 2013, the evolution of processing power had already been considered to have overcome the cost hindrance of

[1] https://www.lith.com/games/fear
[2] https://www.killzone.com/killzone2.html

planning, and therefore planning better and for more agents could become feasible. Lastly, one prediction that is interesting in the context of procedural content generation, is that planners could be used to craft custom experiences and stories through Game A.I. Directors: a concept that has been introduced and became famous with the game *Left 4 Dead*[3], but has otherwise been poorly explored.

Even before Champandard [2013], planning has been intuitively associated with creating stories. Establishing a procedural story consists mostly of the generation of a sequential cohesive and possibly engaging narrative of events. In the context of games, it may also dictate an order of tasks which the player must undergo. As noticed by Cheong et al. [2016] this similarity in performing sound sequences of events resembles the tactics used by AI in robotics aimed towards completing labor tasks in the physical world. Story elements, actions, and the possible events and outcomes of a narrative story are likewise defined in the STRIPS framework [Fikes and Nilsson, 1971], or relatively more recent in the Action Description Language (ADL) [Gelfond and Lifschitz, 1993]. For generating complex narratives that consider the motivations of each character for their actions, Magerko et al. [2004] utilizes an example of a fully-structured plot as a partial-order plan, and Riedl and Young [2010] extensively explores the case for the generation of consistent fables, with their methodology serving as a guideline for planning based procedural story generation.

2.4 Map Generation

Procedural maps face the challenge of generating untraversable, unexplorable and/or unreachable areas. This is usually solved during the generation process, with overhead algorithms dedicated to identifying and avoiding these kinds of situations. In 3D environments, the landscape might be properly tweaked to avoid these flaws, but it might still be too bland for the player to navigate in. Biggs et al. [2008b] presents a strategy for combining PCG with architectural techniques in 3D environments, generating landmarks that guide the player by helping them to identify their location within a larger map.

Generative Grammars have been applied to map generation by Adams et al. [2002] through the representation of a map's structure as rules of a graph grammar. By itself, this representation does not allow for changing specific characteristics of the terminals (rooms), but it does allow for search algorithms to sort and determine maps appropriate for meeting criteria such as 'global size' and 'difficulty'. Shaker et al. [2016a] suggests two families of map generation for *rogue-like* dungeon maps and for platformers: one of which is based on recursively partitioning maps into segments of structures based of Quad-trees, connecting

[3]http://www.l4d.com

rooms created by the partitioned units in order; The second being an agent that 'digs out' a dungeon by traversing a closed space while creating rooms.

Presented by Johnson et al. [2010], another computationally efficient approach is to utilize Cellular Automata as a basis for infinite procedural 2D top-down perspective maps. Its cave generation algorithm is proven to execute in polynomial time, with its complexity being defined mostly by the map's width and height. It has even shown to be effective enough to run online (during the game's execution). While comparable to our work we intend to develop the discussion on Cellular Automata maps not necessarily for the purposes of Johnson et al. [2010]'s infinite ones.

Although the methodology of combining Cellular Automata with Space-filling curves has not seen documented use within academic works, other than our proposed approach, this mix of procedural systems has seen an effective implementation within the commercial game Galak-Z by Aikman [2014] from 17-bit Indie Studio, published by Sony Interactive. It generates sub-sections, or 'chunks', of the level with Cellular Automata while using Hilbert curves to design the overall dungeon layout. This methodology, while interesting and effective for the game's levels, does not fully explore the flexibility of employing Cellular Automata and Space Filling curves for procedural generation. Therefore, our approach also intends to expand upon the Procedural Generator implemented on Galak-Z: We introduce the generation of the entire map from a single automata, as well as polishing methods to increase the diversity of space-filling curves.

2.5 Integrated Map Generators

Integration of Maps and Story generative systems has been studied in Tomai [2012], Thue et al. [2007], Hartsook et al. [2011], Valls-Vargas et al. [2013], Matthews and Malloy [2011] among others. Each proposes a solution for connecting the map's generation with an underlying story, narrative, or quest, while also being restricted by the the game domain or game genre. Hartsook et al. [2011] makes a case for genetic algorithms as an approach to procedurally generating and rendering playable novel games based on a-priori unknown story structures. The generated content is meant to be integrated with role-playing games with the game level's structure being dependent on a story formed by plot points that the player must accomplish. With role-playing games, Tomai [2012] proposed the idea of expanding procedural stories that alter the state of persistent worlds such as MMORPGs, where the interference of a player on a game's space might affect the narrative experience of another.

While not directly integrating two distinct generators, Matthews and Malloy [2011] proposes and implements a technique that utilizes flood-fill algorithms. Based on a de-

signer's document of restrictions, it generates over-world maps with cities, towns, and other landmarks that characterize large scale maps. It does so in a way that is cohesive to the story that the designer intends to transmit, but also in a geographical sense. Another work in this context is Valls-Vargas et al. [2013], which presents a story driven map generator that focuses on representing Plot Points and their causal relationship with the map in which their occur, thus configuring maps through planning algorithms that support a given story and evaluate their quality.

2.6 Qualitative Analysis in Games

Qualitative Analysis is a toolbox of methods that allow for the collection of data and views that are hardly or not at all quantifiable. The intended purpose of research may be in some cases only possible to be evaluated through abstract yet universally understood concepts such as "good", "fun", "boring", "entertaining", etc. This is mostly the case for developing content or methods that invoke feelings, which is one of the goals of most entertainment media. Neuroscience has gone far in identifying and determining the foundations of human emotions, such as presented in Panksepp [2004]. All can be done by presenting a situation and collecting from our subconscious without the need for human input. Yet, were we to collect thoughts and obtain a subjective point of view, methodologies for doing so are well defined and tested through the study of Qualitative Analysis.

Works similar to Consalvo and Dutton [2006] that offer methodologies for evaluating games qualitatively are still rare. Therefore, we have conducted a survey of 12 academic papers that utilized Qualitative Analysis, which grants us a basic understanding of when and how it is being used, with our findings being presented in Table 2.6. Also presented in this table is the previously mentioned work by Thue et al. [2007] for interactive procedural storytelling. While it uses the Questionnaire method to collect the opinions of volunteers, all questions did not allow for open answers. When all answers are restricted to predefined criteria, such as opinion agreement scales, the study in itself is not considered qualitative, as it only evaluates a single axis that summarizes the subject's point of view even though it refers to the user's subjective perceptions.

What Table 2.6 suggests to us is that most qualitative methods also rely partially on answers limited to predefined spectra, as registered by the 'Mixed' approach. The usage of observational data, collected through vigilance of the subject during the intended test, is sometimes used as the only method, or used to support others. Since there is no absolute concept of validation within Qualitative Analysis, the concept of Triangulation is adhered to instead, where distinct qualitative methods are used to form a holistic view of the subject's

Paper Information		Qualitative Method Used			Evaluation
Citation	Theme	Interview	Observation	Questionnaire	Paradigm
Garner et al. [2010]	Procedural Sound Generation	X		X	Mixed
Kazmi and Palmer [2010]	Adaptive Mechanisms		X	X	Mixed
Browne and Anand [2012]	Interface			X	Mixed
Scirea et al. [2014]	Procedural Sound Generation			X	Mixed
Geurts et al. [2011]	Educational Games		X		Qualitative
Vannaprathip et al. [2016]	Educational Games	X	X		Mixed
Dekker and Champion [2007]	Biometric Information Analysis	X	X	X	Mixed
Guckelsberger et al. [2016]	Intelligent Agents		X		Qualitative
Thue et al. [2007]	Interactive Naratives		X	X	Quantitative
Raffe et al. [2011]	Interactive Environment Generation		X		Qualitative
Hash and Isbister [2011]	Procedural Reactive Animation		X	X	Mixed
Biggs et al. [2008b]	Procedural Environment Generation		X	X	Mixed

Table 2.1. Academic works related to games that used either the qualitative analysis paradigm or used techniques common, but not exclusive, to qualitative analysis.

vision.

2.7 Discussion

The challenges presented by Togelius et al. [2013] guide the development of procedural systems towards the goals we must strive to overcome. As it is, the concept of integration between procedural generators is still in its infancy. Thus furthering this discussion should also aid in tackling the subject of directing procedural methods through inputs. Cellular Automata methods for Map Generation presented by Johnson et al. [2010] hint at the need for developers to control the generated structures. A task that itself is feasible due to CA's far greater leniency for customization, when compared to other methods such as Fractals.

All discussions focused in integrating procedural methods gravitate towards the idea that all or part of the inputs could, in theory, be provided by another generator. Galak-Z proposes a commercial implementation of this by mixing Space-Filling Curves with Cellular Automata. This method, however, requires the partitioning of the former into sub-sectors, and furthermore uses them as outlines for multiple CA within a grid. We believe that an approach that would make the entire level based of a single automaton could give room for more cohesive constructs, and so that belief was one of the basis of our work.

Taking a step back from the bulk of discussion related to generating procedural maps, we have noticed an emphasis on the creation of topography, but only rare instances of studies regarding its theming (what kinds of identifiable environments are being created, and what fellings they intend to instill). If a method creates geometry for a playable space, then in most cases, the theme for it is usually fixed to a single concept like forests, mountains, caverns, etc. Again, this is a topic we believe to be feasible, but yet unexplored.

Champandard [2013] points out the advent of Planning within game development, proposing that it could be used in the near future for the creation of complex AI Directors. These managers dictate the pace and feel of the player's experience, all the while keeping it cohesive. Within this work, we follow through with this idea by having a planner dictate the theme of a map, which is then utilized to generate its layout by using a procedural generator that integrates Space-Filling Curves and Cellular Automata.

Chapter 3

Background

This chapter will briefly cover the basics of the theoretical fundamentals of our methodology. Our purpose is to present a bare understanding of the techniques behind Chapter 4. Each of the topics described in the following sections are fields of study on their own and worthy of focused research.

3.1 AI Planning

As discussed in Russell and Norvig [2016], AI Planning is the automated process of solving a problem that has been deconstructed onto multiple classes of components: states that define the world; actions that change the world; an initial set of states before any action is taken; and finally, a goal to achieve through these actions. Then, a planning algorithm determines the likely optimal course of actions that minimizes resources spent (if any), to achieve a goal by changing the world's states to a desired configuration. The simplest of planning problems are defined by: (1) A single known initial state, (2) All actions take the same discrete or null amount of time to be completed, (3) All actions are deterministic. That is, performing one action under the same variables and circumstances will always result in the same outcome, (4) Only a single agent or actor may perform any one action at any given time. Despite the simplicity of such environment, factoring the outcomes of these actions starts to become a problem once they are shown to be mutually exclusive, or that many of them do not contribute to reach the goal.

A planning problem is usually associated with a domain, which represents a world, and a task, such as a robot in a storage room pilling up boxes, or traversing through an obstacle field. Normally, a set of tasks that one person or entity must complete. This style of domain is modeled and represented onto a planning language, such as STRIPS, ADL, or the Planning Domain Definition Language (PDDL). Outside of the limitations of the language,

most planners are domain independent, meaning they can tackle distinct types of problems such as logistics, robot tasks, work-flow management, etc. This flexibility makes planners versatile problem modeling and solving tools.

That is not to say planning is a trivial task: one can intuitively comprehend how navigating through the space of all possible actions with all their possible literals (locations, obstacles, checkpoints, etc.), in all possible configurations of states, can result in a combinatorial explosion. Most planning problems can be reduced to boolean satisfiability problems. Being at least as hard to solve as other NP-Complete problems, various heuristic driven algorithms are available depending on one's domain. Prior to making this choice however there is the need to first understand the basics of how a problem should be modeled through planning.

Planning is most useful when the means to satisfy a problem's concept of 'completion' or 'goal' are hard to determine. As this objective is divided into sub-problems, a logical relationship between these sub-problems must be established. The main problem itself then becomes to determine what configuration of bits (the binary state of a sub-problem being solved or not) should satisfy a logical function. To accurately solve such a problem in polynomial time, one must first determine whether the function itself is solvable, a problem that is all too similar to the Constraint Satisfaction Problem (CSP).

Solving these sub-problems themselves is one that on a complex environment requires taking steps/actions that impede others from being taken. The restriction that one or more actions may impose on others makes it so that the problem becomes non-trivial, being itself also comparable to CSP. As Ghallab et al. [2004] presents, even the classical examples of planning are EXPSPACE-complete for determining the existence of a viable plan, and NEXPTIME-complete to determine its length.

3.1.1 The Basis of Planning Languages

The basic language for representing planning problems is the STRIPS language, developed and documented by Fikes and Nilsson [1971]. As it becomes convenient to represent problems beyond the limitations of STRIPS, or that some functionality of the original language is not required to solve a type of problem, variant languages such as ADL, PDDL are designed. Regardless, how STRIPS described the world and the means to change it, is relevant for understanding its variants.

3.1.1.1 States

Planners decompose the world into logical statements defined as states, which can be as simple as describing a person through its characteristics that are relevant to the problem

at hand. The person in question for example could be defined by a conjunction of literal states, such as $Healthy \wedge Rich$, that express that subject's relevant characteristics while making intuitively unrealistic configurations such as $Healthy \wedge Rich \wedge Sick$ unlikely. Languages based of STRIPS allow for negative states, so instead one could represent *Sick* as $\neg Healthy$. Strips and other languages also define ground and function free first-order literals for detailing those states. For example, $Rich(Tony) \wedge Rich(Bruce)$. Before any actions are taken, a planning problem is usually initialized with a set of states.

3.1.1.2 Actors

Some representations of problems are more concerned in defining the literals in states as variables, and thus define the concept of 'actors'. With the problem of procedurally creating stories, Riedl and Young [2010] have defined that the concept of an actor aids in creating states that pertain to one or more entities. An example is having an action that kills a character be defined by a conjunction such as $Killed(Vader, Ben) \wedge \neg Alive(Ben) \wedge Alive(Vader)$. And thus, languages will dedicate portions of code to defining classes of actors (Such as plane, person, killer, victim, etc.) and which literals belong to each.

3.1.1.3 Representation of Actions

A problem's set of actions defines what agency the AI planner has over the world, and how can it can change it. For an action to be taken, the planner must first meet a conjunction of preconditions, which then changes the world by changing its states in accordance to the action's effects. For example, the previous example of a conjunction of states could have likely been defined by the initial configuration being modified by one or more actions. Actions in most languages are written in a way similar the following pattern:

ACTION: $Kill(killer, victim))$
- **Preconditions:** $Alive(killer) \wedge (Alive(victim)$
- **Effect:** $Alive(killer) \wedge \neg Alive(victim)$

In languages concerned with defining types of actors, a way of defining who or what would be able to take part in an action would be to define constraints. In this example, if there are actors that are *characters*, and actors that are $objects$, it would make sense to define that both $Killer$ and *victim* are of the first type, which could be written as an requirement parameter of the action:

- **Constraints:** $characters(killer) \wedge characters(victim)$

Although *characters* is not a State, and rather a type of actor, it is effectively used as a restriction as well.

3.1.1.4 Goals

A goal is a partially specified set of states. A conjunction of literals that when reached, signals that the problem has been solved. In the example above, if the goal of 'Ben' was to be 'Killed' by 'Vader', then the goal state would be the conjunction $Killed(Vader, Ben) \wedge \neg Alive(Ben) \wedge Alive(Vader)$, which would be reached by a number of possible available actions and depending on the initial state.

3.2 Cellular Automata

Next, we shift our discussion to Cellular Automata: A discrete model with self-organizing properties. It consists of grid, which can be finite or infinite in dimension, containing cells that can exist within a finite number of states. These usually being a binary configuration such as 'On' or 'Off', 'True' or 'False', 'Alive' or 'Dead', etc. Periodically, each cell has to update its own state based on the cells around it (neighbors), which can be done one cell at a time on a predetermined order, or all cells at once. This constant shifting of values generates distinctive results depending on the starting configuration and the rules by which cells evolve. The resulting grid after a number of 'iterations' or 'cycles' can be a stable structure or a constantly shifting landscape. This structure may eventually stabilize by reaching an immutable end, or may end up looping into a perpetual cycle. Often, Cellular Automata generate interesting shapes such as the ones presented in Figure 3.1.

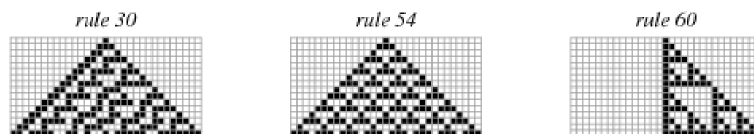


Figure 3.1. The Cellular Automata generated by rule 30, 54, and 60 of 'The 256 Rules' by Li and Packard [1990].

The automata's universe starts at the 'configuration' state: All cells starting with the same value, except by a finite predetermined number of cells that begin at different states. These usually act as the catalyst for change to the automata's status quo. The state of any single cell is determined periodically by its 'neighbors', which itself is a concept that can be distinct for each automata. The two most common being the Von Neumann and Moore Neighborhoods, presented in Figure 3.2.

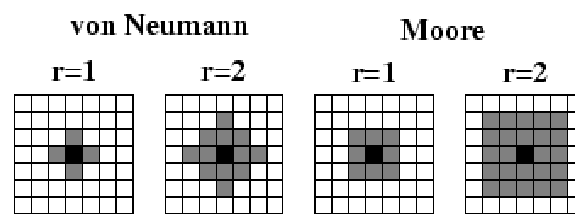


Figure 3.2. Von Neumann and Moore neighborhoods for a single cell (colored in black). The neighborhood's size 'r' can be increased to consider the states of additional nearby cells.

Perhaps the most common examples of Cellular Automata are Conway [1970]'s Game of Life, and Stephen Wolfram's Grassberger [1986] Elementary Cellular Automaton. The latter of which has been proven to be Turing-Complete. Ever since its conception in the 1940's, Cellular Automata have seen applications and studies in Biology, Computability Theory, Computer Science, Mathematics, and Physics.

3.2.1 Synchronous vs. asynchronous Updating

When updating Cellular Automata, one might update the state of each cell immediately upon discovering its immediate future, or it might update all cells at once after their following states are determined. These are called asynchronous and synchronous updating, respectively, and yield distinct results. Yet, asynchronous methods can have nuances of their own, specifying distinct update orders which also create different effects. While the effects of synchronous and asynchronous updating are clearly noticeable within the rule-set of our Automata, the individuality of the shapes generated by each methodology become significantly less noticeable as the constraints that will be presented further in this work are introduced. Therefore, while examples of different iteration methodologies in this work's automata are presented in Figure 4.7, the automata presented by our methodology will follow exclusively synchronous updating. Henceforth discussion on the topic of Cellular Automata Updating methodologies will hereafter be left for related works such as Schonfisch [1999]; Bonomi [2009].

3.2.2 Multi-layered Cellular Automata

The concept of multi-layered cellular automata is one that has not seen as much academic research as the general 2-dimensional definition of the model, save a few exceptions such as Nakayama et al. [2015], and Bonomi [2009]. A multi-layered cellular automaton is comprised of a set of cellular automata where the rule-set for the cells of each automaton takes

into account the state of the cells from other automata. A representation of a multi-layered cellular automaton is presented in Figure 3.3

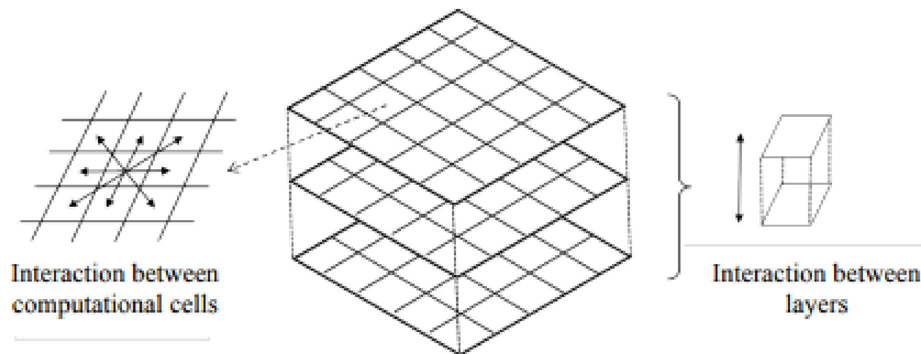


Figure 3.3. A Multi-layered Cellular Automaton schematic drawing as presented by Nakayama et al. [2015].

The subject of multi-layered automata will be of relevance to this work when introducing methods by which to improve the generation of maps in Chapter 4.

3.3 Space-filling Curves

Space-filling curves, or Peano Curves [Sagan, 2012], are a concept in mathematical analysis first discovered by the end of the 19th century, as a special case of fractal constructions. It pertains to curves whose range contains all the available space within a discrete n-dimensional space. That is, it maps a multi-dimensional space grid (e.g. 2D) into one-dimensional space (1D), like a continuous thread that visits every cell exactly once.

A useful property of these curves is that their tracing is generated through a function that maps information contained within one dimension to another (e.g 1D to 2D) while preserving locality. This property allows Space-filling curves to be applicable in Computer Science topics such as Moon et al. [2001]'s clustering, and improving data structures by Kamel and Faloutsos [1993b,a]. The Hilbert curves are an example of the specific case of Space-filling Curves that present interesting shapes when mapping 1D coordinates to 2D, as shown in Figure 3.4. As a Lindenmayer System Mishra and Mishra [2007], the Fractal shape for the Hilbert Curve is defined as below. In this representation, F means 'draw forward', 'l' means 'turn left 90°', 'r' means 'turn right 90°', and 'A' and 'B' left are ignored during drawing.

Aphabet: A, B
Constraints: $F\ r\ l$
Axiom: A
Production Rules:

- $A \rightarrow lBFrAFArFBl$
- $B \rightarrow rAFlBFBlFAr$

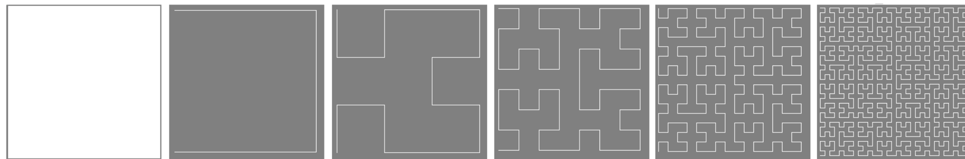


Figure 3.4. With the production rules being applied M times to the axiom, a curve of length $M^2 = N$ is obtained. The following Hilbert Curves are mapped to a NxN grid, for the values of $N = 1, 2, 4, 8, 16$, and 32, respectively.

Although the process described in this work could be used for any space-filling curve, our experiments shall be limited only to Hilbert curves, as to further the discussion started by the Galak-Z game. Exploring the diversity in the generated content by choosing from a repertoire of distinct Space-filling Curves is something that will be explored in future work.

Chapter 4

Methodology

This chapter describes the methodology pertaining to each of our procedural modules. As a quick overview, figure 4.1 presents a diagram that encompasses the whole of our methodology. As the reader progresses, the description of this diagram should become more understandable. Furthermore, even if at a first reading of this image is not entirely clear we recommend checking back on it.

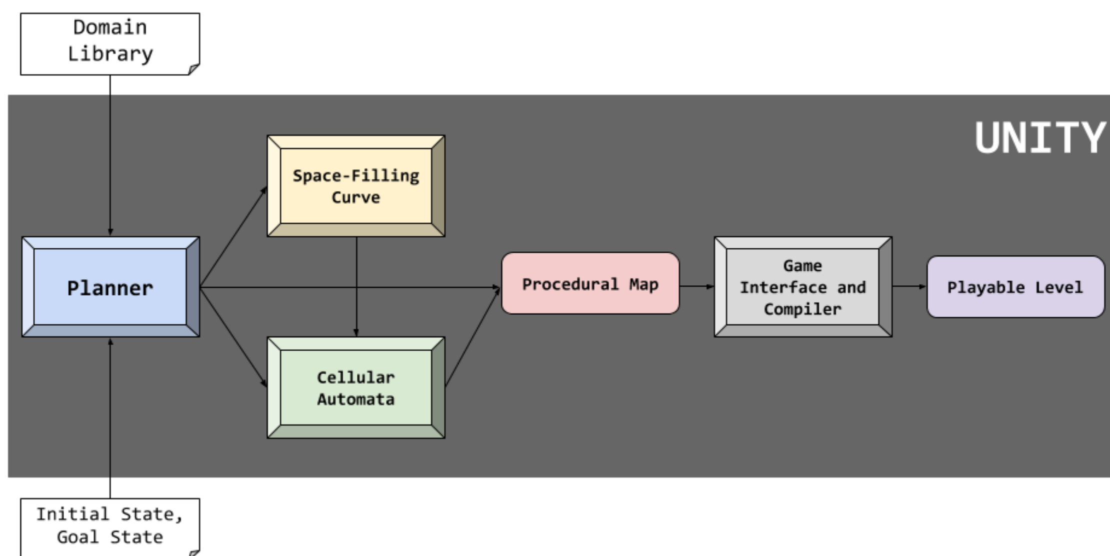


Figure 4.1. General structure and flow of our methodology. Beveled boxes represent different programmed modules, document shaped boxes represent text files, and rounded boxes represent resulting constructs. Each component encapsulated within the gray box is implemented and executed within the Unity Engine. Arrows from a box source that have as their destination a construct (rounded) box indicate that the source has generated that construct. All other arrows that have their destination as a non-construct box indicate that their source is used as input to their destination.

We feel that it would be appropriate to address some of the decisions regarding the

general implementation of our work. First, about our choice of game engine: in the debate for the optimal game engine to use, most discussions end with using the Unity Engine, the Unreal Engine, or crafting your own game engine. Engines that are created for specific types of games excel at their job, but are far from the multi-domain approach we have aimed for, and are expensive time and costwise and therefore a counter productive option. On the choice of Unity vs Unreal, we have opted for Unity simply based on our previous experience, and we firmly believe both are equally capable of delivering the procedural designs we propose. Thus, this work's implementation has begun on the Personal Edition Unity 2D version 5.6.1f1 (64 bit) going all the way to 2018.1.6f1 in its completion. All methods were programmed entirely with the C# programming language.

Second, addressing the use of 2D versys 3D assets: while it is true that Cellular Automata are usually exclusively envisioned in the realm of procedural map generation as 2D map generating methodologies, that is not necessarily true. Lague [2015] exemplifies the use of Cellular Automata that is then converted into three dimensional geometry. Nevertheless, 3D asset libraries are not as extensive and accessible as 2D ones. Being that much of procedural generation as a whole is based of creating permutations and subsets of content from a vast library of generic assets, the choice for 2D sprites becomes much more accessible and appealing.

Having gone over these two major design choices, the three following sections thoroughly describe the process for the development of the AI Planning Module, the extensive Map Generator Automata module, and the Planning to Map module.

4.1 Level Planning

Attempting to make decisions that ultimately generate a cohesive environment is not too different than making a set of sequential actions towards a goal. In an efficiency point of view, the problem is looser by there being no cost for taking actions, other than the time the planning algorithm itself takes. In this case, the goal is to achieve the completion of a map concept. The actions taken are the decisions that lead to this goal, such as deciding which elements to include in the environment. And the states are these elements themselves.

The idea of a completion/goal itself for any planning problem must be divisible by a set of one or more states. In our map planning case, a state is the consequence of an action taken to determine characteristics of the map. Therefore, a state is either a characteristic of the map, an intermediary variable to be considered by an action at some time either to determine another state that is itself a characteristic, or to determine the value of another variable.

Finally, the process to generate the theme of a map itself is a matter of choosing an

appropriate approach to modeling the selection of any and all possible characteristics available to a game map. This decision process and its domain is one that is highly dependent on the amount of available game elements that can be placed, and how complex the relationship between such elements is.

Due to the limited graphical assets and the time available to both create the language and the programs required for its testing and use, as well as developments regarding Cellular Automata, its integration with these planning systems, and also our quantitative and qualitative testing, the planning problem we have intended to generate for now is quite simple. One that is easily solvable by an Heuristic-oriented planner with no current need for efficiency optimizations, other than the bare basic.

4.1.1 Planning Language

So far we have mentioned the notion of a language needing to be implemented for tackling our problem, and that is due to Unity's non-existent integration to any existent planning languages, as of the beginning of this work. Being that our goal was always to have an implementation available to be used in commercial games, this predicament has required us to temporarily step out of our way to first create the means by which we may model and solve our planning problem.

PDDL, while a powerful universal language for expressing most planning problems, has no implementation for game engines, as of the writing of this work. While planning is ascending as a Game AI principle, no implementations that we are aware of were available for Unity. Having an, in-engine, interpreter and executing planning outside of the engine is a feasible option, however we have intended to avoid external plug-ins to a methodology that otherwise functions exclusively with the engine's resources.

Thus, this opportunity does allow us to have our intended language be one specialized to the task at hand. Although it required a great amount of time to code and implement, it is a tool that certainly brings benefits for this and further projects. By comparing existing planning languages and their purposes, we have determined what characteristics should be best suited to planning game map themes.

As described in Section 2.3, most planning languages in games are rather focused towards their use in creating believable agents. Two of the most iconic of these languages are PDDL and the Game Description Language (GDL). These focus their structure on modeling game scenarios that allow agents to understand the current game's state, and furthermore make informed decisions and actions upon it. Both languages have their nuances in implementation and use. In a few words, while GDL is focused towards modeling game scenarios, PDDL is meant to be an universal tool that takes from STRIPS, ADL, and others. Both

ADL
Positive and negative literals in states: $Rich \wedge Famous$
Open World assumption: Unmentioned literals are unknown.
An effect $P \wedge \neg Q$(set a state P to true and another to false Q) means add P and Q and delete $\neg P$ and Q
Quantified variables in goals: $\exists x\, At(P_1, x) \wedge At(P_2, x)$ is the goal of having P_1 and P_2 in the same place.
Goals allow for conjunction and disjunction: $\neg Poor \wedge (Famous \vee Smart))$
Conditional effects are allowed: **When** $P : E$ means E is an effect only if P is satisfied.
Equality predicate $(x = y)$ is built in.
Variables can have types, as in $(p : Plane)$

Table 4.1. Table of notable properties of the ADL planning language, as organized by Russell and Norvig [2016]

PDDL and GDL offered us insights as how we should define our language's syntax and specifications, even if the basis of our planning language is mostly taken from ADL's functionalities, shown in Table 4.1. Our developed language shares many of these specifications, save two major changes elaborated upon below.

4.1.2 Hierarchical Classes

Variables do not only have types, but each literal is a leaf node in a hierarchy tree of variable types, as exemplified in Figure 4.2. This essentially means that variables may be of multiple types of distinct levels of abstraction. For example, a Vegetation node, with two child nodes Vegetation/Trees & Vegetation/Plants which have their own leaf literals. A combination of States and Actions/Literals may only be set if the literals themselves are leaves in the types hierarchy tree, as there would not make sense to specify something such as $Exists(Tree)$ without specifying exactly which tree is to be added to the map.

In this test example, he have intended to fiddle with the complexity of our language by having it design characters for the *Dungeons & Dragons* tabletop role-playing game. This presents a level of complexity that is rather easier to model than map generation, and relieves us of the burden of designing a test example that does not rely on graphical assets, and that is purely hypothetical.

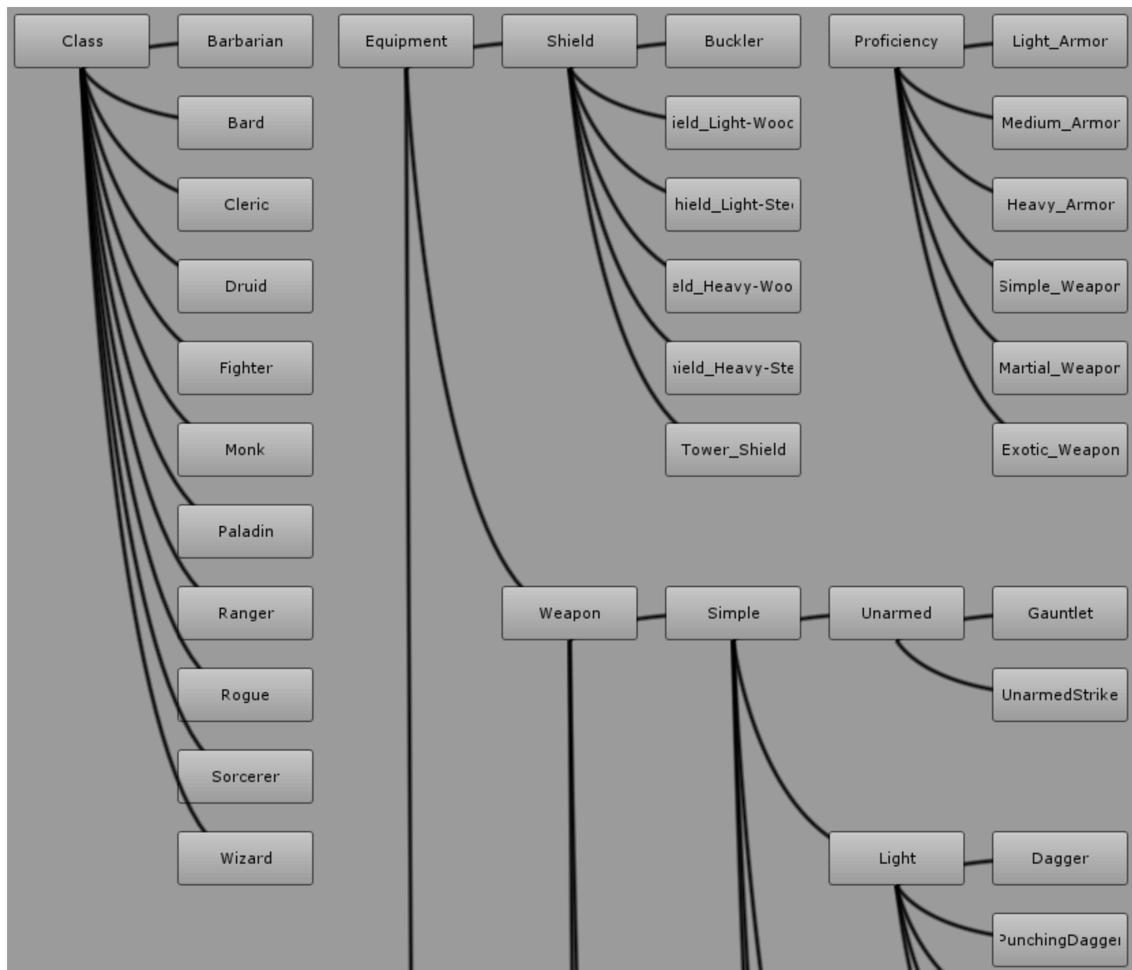


Figure 4.2. Example of multiple variable tree hierarchies within a 'character creator' used for our language's preliminary testing, as displayed on a custom interface programmed on Unity.

Although this resource has been barely used on our initial modeling of the map theme generator, it has only been so due to the limitations in graphical assets that have halted the creation of complex elements. Further work with our generator is sure to explore the avenue of complex class/variable relationships.

4.1.3 'Unknown' Operator

Our language allows the use of a prefix 'unknown' unary operator, much like the 'not' operator. This 'unknown' operator returns true if a combination of states and literals/actors has not yet been set to a value within the world of current active states. For example, a precondition could check for $(UNK)Exists(PineTree)$ to see whether a map has been set to contain the presence of one or more Pine Trees, or even $(UNK)Exists(Tree)$ to check the existence of

any tree at all. This is mostly useful for allowing the planner to understand the difference between something that has been set not to appear on a map, from one that could, but it is still unknown. Additionally, allowing preconditions to check whether a state with empty variable parameter literals is not unknown, meaning the state exists with at least one literal, does help us in checking what elements are still to be added in the map. Another example would be checking a state $Exists(T : Tree)$ that expects a literal T of the $Tree$. By doing this we may be able to check whether any type of tree has been set to be an asset on the map, and perhaps place one accordingly.

4.1.4 Planning Algorithm

Determining a plan is exponentially complex, and thus requires the use of specialized and optimized algorithms when dealing with larger problems. However, our preliminary experiments are, for now, easy and short enough that a complex approach for tackling them is still not necessary. Therefore, for the purposes of the work, a greedy approach is sufficient, and, as shown in section 6.4, not too expensive computationally.

The basis of our initial planning algorithm is simple enough to explain in a single paragraph: From all possible actions within the current configuration of states, choose the one which maximizes the amount of subproblems completed. As mentioned, the goal of a problem is divisible into multiple, smaller problems that are themselves logic functions. Being that this function is one of multiple logic unary operators (Not, and in our case, $Unknown$) as well as binary operators (And,Or, and all other derivable operators based on them). The measure of the completeness of a plan is determined by how many of these subproblems are themselves complete. One intuitive heuristic then is attempting to act in a way that maximizes the completion of these problems, even if doing so may not always work. A representation of how the algorithm perceives the completion of a problem is presented in Figure 4.3.

In this example, the goal function of having a character fully equipped with weapons and armor is satisfied if the character has: A suit of armor; a ranged weapon; and either a two-handed weapon or two one handed weapons. As this goal tree has been organized, choosing the character's suit of armor solves 50% of the problem, selecting a ranged weapon solves 25%, choosing a two-handed weapon or two one-handed weapons solves the remaining 25%, with each one-handed weapon solving 12.5%. Therefore, by organizing the goal's logical tree, one may determine which objectives hold priority over others.

A certain degree of stochastic decision making is required with creating concepts that are not common in classical planning. The goal is not to build a character as efficiently as possible, but to simply generate diverse characters. If left to simply look for the most

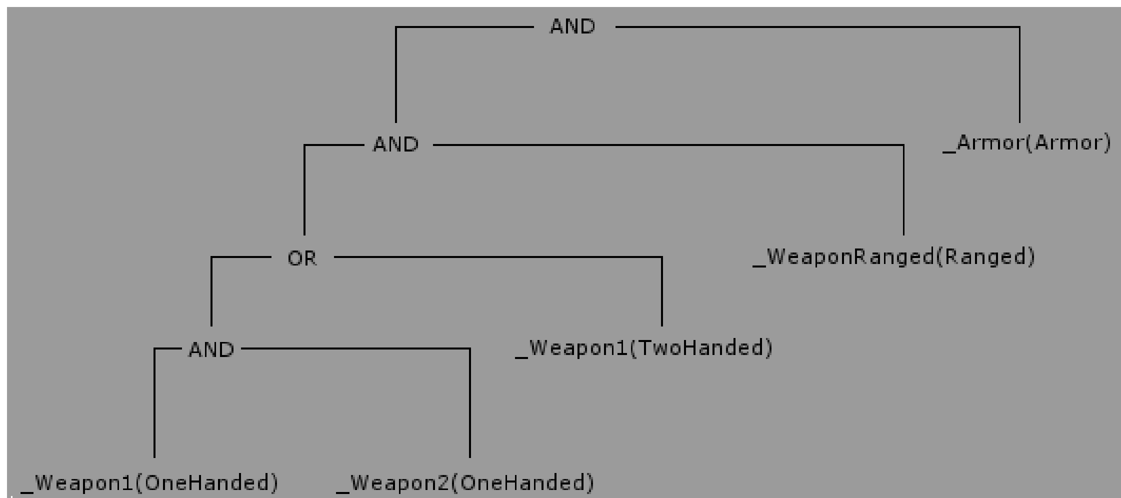


Figure 4.3. Example of the goal function for the 'character creator' problem represented as a tree of logical functions.

effective way to conclude the problem, priority will be always given to choices closer to the tree's root. Thus, and a touch of random is required to create choice diversity.

A greedy planner is prone, however, to falling into the pitfalls of the most complex scenarios of planning. Some of which include states that require a continuous sequence of actions to be realized for it to be set true, possibly creating a deadlock where no actions can be taken. Or worse yet, a group of actions creating cyclical configuration of states. As our search space is currently rather small, we have tackled this through four simple measures: (1) Once a deadlock state is reached, our algorithm backtracks one step; (2) Each configuration of states generated by a set of actions is stored, as to identify which steps have created a deadlock, and henceforth avoiding taking these same steps again; (3) A user defined parameter R determines, from a sequence of actions, the maximum number of deadlocks that sequence may create before the planner completely restarts its plan, while keeping the list of actions that have generated a deadlock; (4) User defined parameter T that determines the amount of time the planner may take before performing only the best actions to conclude the plan. This is implemented as to avoid having the planner taking longer than a desirable amount of time to complete.

4.1.5 Domain Library

To determine how can a planning language be used to solve the problem of generating maps, we must first discern the difference between a planning problem and its domain: a problem's domain contains every action, state, and entity within the abstraction of the world in which the planning problem resides. A document containing every information on this world is its

'Domain Library', and it contains every Action, Entity, Precondition, State, and Effect that characterize that single domain, serving as building blocks for creating a cohesive interpretation of the world, and therefore allowing the solving of problems within it. These problems themselves being characterized by a starting configuration of the world's entities, and the goal configuration which solves the problem.

For example, if a plan's goal is to have a robot carry boxes around multiple points, then that Domain Library would have to define several fundamental concepts: (1) The robot, boxes, and places, which would likely be actors/literals; (2) states that determined the current status quo of an actor. For example, a state 'IsAt(object,location)' to be used to both store the positions of boxes and the robot's; (3) Actions that define what changes can be done to the environment. Every possible change and configuration of the environment is described by a combination of elements of the library. Another example, but this time regarding the character generator mentioned previously is presented in figure 4.4

```
<<type>>

>Race
Human
Dwarf
Elf
Gnome
Half-Elf
Half-orc
Halfling
```

```
<<state>>
_Race(Race)
_Size(Size)
_Class(Class)
_MagicCastingType(Magic:Type)
_MagicMaximumLevel(Magic:MaximumLevel)
_Armor(Armor)
_Weapon1(Weapon)
_Weapon2(Equipment)
_WeaponRanged(Weapon:Ranged)
_WeaponFullyArmed()
_ProficientType(Proficiency)
_Proficient(Equipment)
```

```
Action = Set_Class_Bard()
    Constraints  =
    Precondition = {UNK}_Class(Class)
Effect = _Class(Bard)
         _ProficientType(Light_Armor)
         _ProficientType(Simple_Weapon)
         _Proficient(Longsword)
         _Proficient(Rapier)
         _Proficient(Sap)
         _MagicMaximumLevel(6th)
<ACT_END>
```

Figure 4.4. Example from part of a Domain Library designed for preliminary testing. In this test Domain Library, the planner must create a character from the game *Dungeons & Dragons* in accordance with the game's rules. The first image contains the group of actors that represent races. The second contains states that describe how much of the generation process has been completed. And the third image contains one of the actions that define one of the character's properties.

Then a second file, the Planning Problem, presents the starting configuration of states for its given domain (where the boxes and the robot are at, in this case). And an end goal: The configuration of states by which the actions taken by the planner must strive to somehow achieve. A Domain, and by extension its Library, are independent from the problems they are used to solve, but not vice-versa. A Domain by itself simply represents a problem which its solution requires a set of possibly sequential, but most times containing mutually exclusive actions. This strong position of a Planner solver as a generalized problem solver is the basis by which many problems requiring a cohesive set of instructions may be modeled as a planning problem.

4.1.6 Example: Map Theme Generator

The Planning Domain Library we have used defines a map's theme in sets of goal states which we call layers. Further in this we work, we present the concept of Multi-layered Cellular Automata, and use the same nomenclature ('layer') to define an individual automata as a layer. This is intentional, as understanding both the planner's goal states and individual automatons as layers will make comprehending the integration between Planner and Generator much easier. The planning library for this example is presented within Appendix A.

The first step of this planning problem is comprised of subsetting abstract concepts such as the map's climate, and then broad concepts such as its hydrography, topography, and vegetation density, to be options that could be placed in the map. For example, a 'Set_Forest()' action requires that the temperature and vegetation density of the map have not been set, so that it may set these parameters itself. If allowed to execute, this action will set a number of states regarding the possible configurations of layer types that could be placed. As with the 'Set_Forest()' example, it sets the 'Hot', 'Temperate', and 'Cold' temperatures to be possible choices, and the only option for vegetation being 'Dense', as shown in Figure 4.5.

```
Action = Set_Forest()
    Constraints  =
    Precondition = [[{UNK}_Temperature(Temperature)] {AND} [{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureHot)
         _Temperature(TemperatureTemperate)
         _Temperature(TemperatureCold)
         _Vegetation(VegetationDense)
<ACT_END>
```

Figure 4.5. Example action in the map theme generator domain library. This planning problem action takes no input parameters (meaning also that there are no constraints to define which parameters that are acceptable). Has as a precondition function that there are no _Temperature(), and no _Vegetation() states active.

Once these layers and the map's size have been all determined. Another action 'Set_Stage1Complete()', that is only possible when those concepts are defined, marks the first stage of the map generation to be complete. From then on, actions define the specific layers of content to be added to the map's theme, until the goal state of having a number of layers defining all aspects of a map is achieved. One such example of all layers being defined at the end of this planning problem is presented at Figure 4.6.

Many states other than those desired to the completion of the goal function will end up active, as a product of the path of actions taken to reach the plan's end. As we shall explain further, on Section 4.3, only a few of those layers are needed and considered for the integration of the planner's solution, and the cellular automata maps.

Current Active States

```
_Temperature(TemperatureHot)
_Temperature(TemperatureTemperate)
_Temperature(TemperatureCold)
_Vegetation(VegetationDense)
_Hidrography(HidrographyCoastal)
_MapSize(MapSizeMedium)
_Topography(TopographyDoubleUp)
_TopographyChoice(TopographyDoubleUp)
_Stage1Complete()
_Tree(TreeRegular)
_Water(WaterCoastal)
_Ground(GroundGrass)
_Cliff(CliffEarth)
```

Figure 4.6. Example final states configuration of the planner algorithm. The result is a set of state(actor) that is a subset from all possible valid combinations of states and their non-constraint violating actors.

4.2 Map Automata

This section describes the implementation, parameters and design decisions behind our Cellular Automata and Space-filling Curves, as well as the methodology for integrating both. We interpret the idea of a good automaton for designing a 2D, top-down game map, as one that has: (1) A cohesive path structure that guides the player through the areas of the map, while allowing for an appropriate number of parallel roads to explore; (2) These paths are not made pointless by shortcuts; (3) The automaton's configuration around has an organic feel to it; (4) The automaton's structure looks and feels distinct in comparison to other constructs by the same generator. Furthermore, our design decisions are taken towards achieving these criteria.

4.2.1 Cellular Automata

Our Cellular Automata for generating map elements starts off following the cave generation method that is best introduced by Johnson et al. [2010]. This procedural cave generation method is comprised of a 2-dimensional *MxN* grid, and a rule-set with two values for cells, true and false. A cell C has its value defined by its Moore Neighborhood of $r = 1$ (See Section 3.2), from which T is the number of true cells, and F is the number of false ones. For a cell that is on the grid's edge, positions outside of the grid count as true cells. The Cellular Automata's Grid is initialized with semi-random distribution of true and false cells which

guarantees that a percentage 'Fill' of cells are false.

Most Cellular Automata delegate to their own cells a timer with which the cell updates itself based upon the state of its neighbors. Should all cells operate on the same timer, as it is the case for our Automata, then all cells are updated instead within universal steps, which are hereafter refereed to as 'Iterations'. One iteration of our Cellular Automata updates each cell in accordance to the following rules:

$$C = \begin{cases} true, & \text{if } t > 4. \\ false, & \text{if } t = 4. \\ false, & \text{if } t < 4. \end{cases} \tag{4.1}$$

Our preliminary experiments suggest that an equal number of True and False cells ($Fill = 0.5$) presented better results, as the rules tend to converge to all cells becoming False for higher filling values of $Fill > 0.5$ (meaning cells are mostly false), or all cells except some of those adjacent to the edges of the automata being True for $Fill < 0.5$. This rule-set converges within a reasonable linear number generations as its rules are simple enough not to fall into a looping sequence of states. Examples of the resulting automata are presented in Figure 4.7

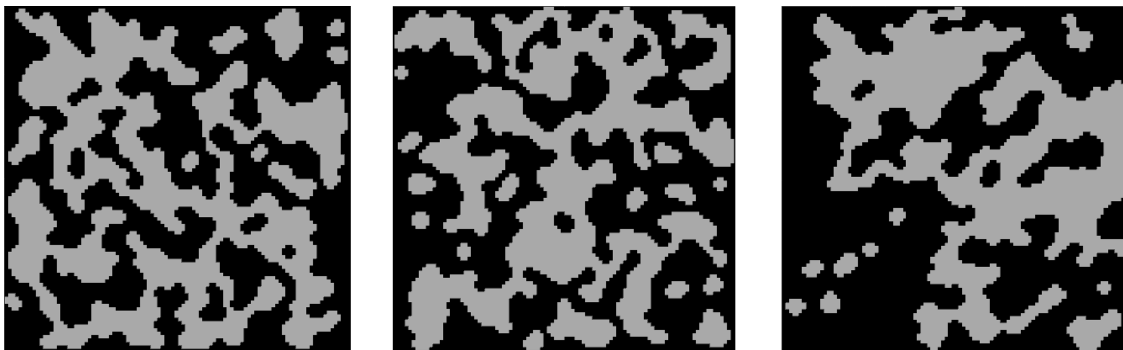


Figure 4.7. Three versions of the same Cellular Automaton generated from different updating methods (see Chapter 3). From left to right, these methods are: synchronous, asynchronous with cells updated in a random order, and asynchronous with cells updated sequentially ordered by their position in the grid. Black cells are *true*, white cells are *false*.

Simple as it is, this methodology could already be converted to a 2D game map with a top-down perspective. As with Johnson's 'Cave Crawler' [Johnson et al., 2010], which features an additional step for generating continuous, infinite maps with adjacent grids. All it would take is to map True cells to floor tiles such as grass, and False cells to unwalkable tiles such as walls (or vice-versa). Yet, even should another specialized algorithm place game elements within this map such as items, coins, enemies, etc., its design is likely to still be

lacking: there are a great number of unaccessible areas, and the geometry of the path hardly represents progression.

To better mold and orient the generative process, we propose the introduction of space-filling curves, such as Hilbert Curves, as a guide for the creation of paths within the map. This step is introduced prior to iterating the automata, during the 'configuration' step.

4.2.2 Space-filling Curves

Space-filling curves are guaranteed to generate a path that is linear (no parallel intersections) through a grid that is at least as large and wide as the square root of the path's length. For us, the purpose of utilizing Space-filling curves is to create a guide-line by which the automata updates its cells around it, while preserving the curve's shape. This guideline could represent the path by which the player must explore the map, and the curve's inherent properties assist in creating areas to place content and to explore. The methodology for generating a path out of the points of any type of Space-filling Curve, as well as adapting it for usage with the automata are summarized as the following steps:

1. **Plot** the curve within the Automata's grid.
2. **Scale** the curve to increase distance between its points.
3. **Shift** the origin of the grid to randomize which of the curve's shapes remains in the grid.
4. **Trace** a path through the points within the grid's limits.
5. **Imprint** the curve into the Grid

These steps are further explained within this sub-section. Figure 4.8 illustrates the automata's changes from step 1 to 4.

4.2.2.1 Definitions

Before describing the required steps, let's first define a few terms, while giving a basic overview of the following steps. A Cellular Automaton's Matrix is henceforth refered to as a 'Grid' of dimensions M by N. This grid also contains a Space-filling curve whose greatest dimension is defined as its 'Order' and represented by ($\alpha = Max(M,N)$). This curve is used to draw or 'Trace', as we call it, the player's 'Path' along the map. When we refer to 'Scaling' a curve, it means to multiply the coordinate of every point in the curve by an

amount S, which furthers the gap from every point of the curve within the grid. Finally, since each point of the curve has been split by an equal sized gap, by 'Shifting', we simply mean to increase the offset or origin of the curve in the X and Y coordinates by randomly determined amounts. Therefore, 'shifting' which region of the expanded curve that exists within the Grid.

4.2.2.2 Plotting

To place the Space-filling Curve within the automata's Grid, we have begun by generating a Hilbert curve that fills all of the space of the grid. Therefore, to also cover grids where $M \neq N$, we generate a curve of length α^2 into a $\alpha x \alpha$ grid. The path of the 1-dimensional representation of the curve is stored into a list data structure that is the basis for the tracing of the map's path.

4.2.2.3 Scaling

The Space-filling curve that occupies all of the grid's space is then rescaled by an integer amount $1 < S < \alpha$ as to increase the distance between the points of the curve (eg.: $S = 2$ would imply a distance of 1 cell between points of the path).

4.2.2.4 Shifting

Although fractals have proven to be valuable tools for generating procedural content, their inherent predictability can be a problem for generating distinct content. For the purposes of utilizing fractal curves as paths for game maps, some measures have to be taken to attain the desired diversity of procedural content. The first being the definition of a random sector from the entire curve from which to create a path upon. This is done by dividing the $(N \times S)x(M \times S)$ grid by the curve's order, resulting in a $((N \times S)/\alpha)x((M \times S)/\alpha)$ grid from which a random offset is selected. The curve is then shifted to that position. Section 4.2.4 returns to the subject of introducing random variations, once the integration between the Automata and Space-filling Curve is complete.

4.2.2.5 Tracing

The steps taken to alter the Space-filling curve up to this point have it not filling the entirety of the grid's space. Instead, as the curve has been scaled, only a few of its points remain within the grid's bounds. Furthermore, as shown on (3) and (4) in Figure 4.8, what remains of the curve within the grid, are split segments that are not connected within the bounds of the grid. To find a way to connect these segments, the order by which each point of the

curve has been created (and therefore the original 'path' created) is stored within a list that is created as the curve is plotted. From the remaining points, a new path must be traced.

Along with the previously stated definitions, let's define η as the set of k split segments from the Space-filling curve that remain within the grid. To create a traversable path for the player, all k paths from η must be combined into a single path. Each point σ that is stored within the list has an index that indicates how early it was created, and ranges from 0 (the first point) to α^2 (the last point). In order to connect two split segments η_a, η_b, at least two points $\sigma_i \in \eta_a, \sigma_j \in \eta_b$ must be connected, where i and j are indexes, and $i < j < \alpha^2$.

To define a criteria for connecting two points, a function $\lambda(\sigma_i, \sigma_j)$ must be established. In our case, for preserving the non existence of diagonal paths, our function for verifying if two points are connectible checks whether they are on the same X or Y axis in the grid. Formally:

$$\lambda(\sigma_i(x_i, y_i), \sigma_j(x_j, y_j)) = \begin{cases} true, & \text{if } (x_i = x_j) \vee (y_i = y_j). \\ false, & \text{if } (x_i \neq x_j) \wedge (y_i \neq y_j). \end{cases} \tag{4.2}$$

Given the function for connecting points from paths k_a and k_b, for each path a and $b = a + 1$ starting from $a = 0$, each point $\sigma_i \in k_a$ and $\sigma_j \in k_b$, with $0 \leq (i = length(k_a)) < (j = length(k_b)) < \alpha^2$ tries the function $\lambda(\sigma_i, \sigma_j)$. If it returns $true$, the points become connected, which in turn connects both paths: $k_b = k_a \cup k_b$. If it fails, then it attempts by brute force to connect to previous points g from k_a $(0 \leq g < i)$. If no pair of points can be found, then it attempts the same with k_b, but instead trying to k points moving further on the curve $(i < h < j)$. This process of brute force in theory could bring a factorial worst case complexity, based on the number of points within the grid. However, due to the high number of points spread across the grid, and the regularity of the fractal's shapes, the theoretical worst becomes unlikely in practice. The algorithm ends when the k number of k paths within η equals 1, meaning all paths have been connected.

4.2.2.6 Imprinting

The resulting curve from all previous steps is one that represents a linear path that may have generated dead-ends and cycles, but mostly follows a longer, main path, as the one presented in Figure 4.8. These possible dead-ends and cycles are not detrimental to the map's design, and in fact, they are useful as a basis to expand as secret or optional areas to explore. One possibility of expanding the concept of the curve map's path could be to connect more segments, as to create alternative ways to reach the same place or complete the level's map.

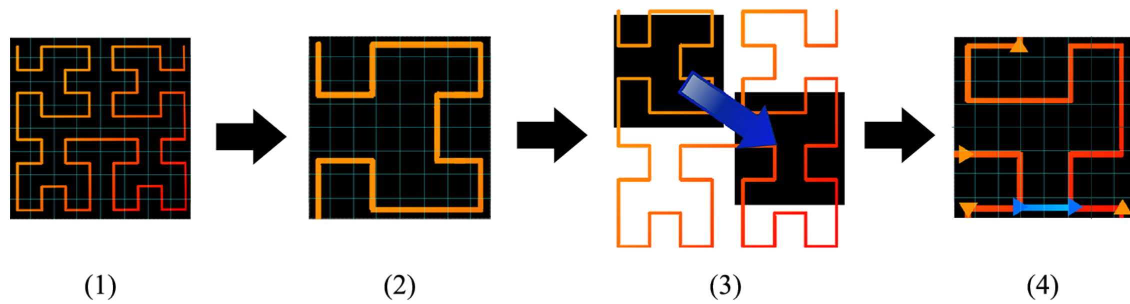


Figure 4.8. From left to right, the first four steps described in Sub-section 4.2.2 are represented: (1) Plotting, (2) Scaling, (3) Shifting, (4) Tracing.

To have the cellular automata's topology integrated with the curve, the configuration step (before any iterations) must be altered to set all cells that are covered by the path to a value. Which in the case of our rule-set is $false$. This however still presents a problem, as the automata's rules could cause part of the path to be erased depending on the random configuration of the remaining cells.

To avoid this, there are two safety measures that will come in handy for future improvements: 1. Introduce the concept of 'locking' cells, so that their values are not changed during iterations, and applying it to all cells within the path; 2. Redefine which cells belong to the the curve's path, introducing the concept of a path's 'girth'. Once all cells from the curve's path are defined, each other cell within a Moore distance of P_g of at least one cell within the path is synchronously added to it. A value of $P_g = 1$ already guarantees that the path will not be broken, as the cells from the original path are all adjacent to $false$ cells.

4.2.3 Securing Negative Space

While guaranteeing paths between two points is possible with the tracing of paths through the Cellular Automata, there is nothing stopping the automata into generating alternative, unintentional paths which could effortlessly lead the player from the start to the end of the map. While some games would not be bothered by this, levels that present challenges to the player as he or she traverses from the beginning to the end of the level's map must restrict these alternative paths from emerging. Therefore, a safety lock to avoid unintentional paths is required.

By the same logic that we orient the automata into organize itself around the Space-filling Curve path, we generate a 'Negative Space' path: a second curve that covers all the areas of the grid that the Hilbert Curve does not. This area is then trimmed so that its girth is no larger than a specified 'Negative Girth' parameter N_g. This negative curve should

influence the automata as little as possible to optimize the amount of random True and False cells the automata can work with.

The algorithm for trimming negative space is a simple one: it begins by splitting all the cells not covered by the curve path being added to a subset of nodes γ, and all cells that do to another subset δ. Then, for each cell $c \in \gamma$, if there is at least one cell $d \in \delta$ within a Moore neighborhood of 1, then c is moved from subset γ to δ. This algorithm updates synchronously each cell within γ for $(\frac{S}{2} - P_g - N_g)$ iterations, where S is the scaling of the Space-filling curve. For the desired girth to be obtained, S must be an even number, as the Negative-path is trimmed from both its sides at the same time. Finally, the resulting automata setup shown in Figure 4.9 contains paths for the player to follow that are guaranteed not to be broken by the automata's own rules.

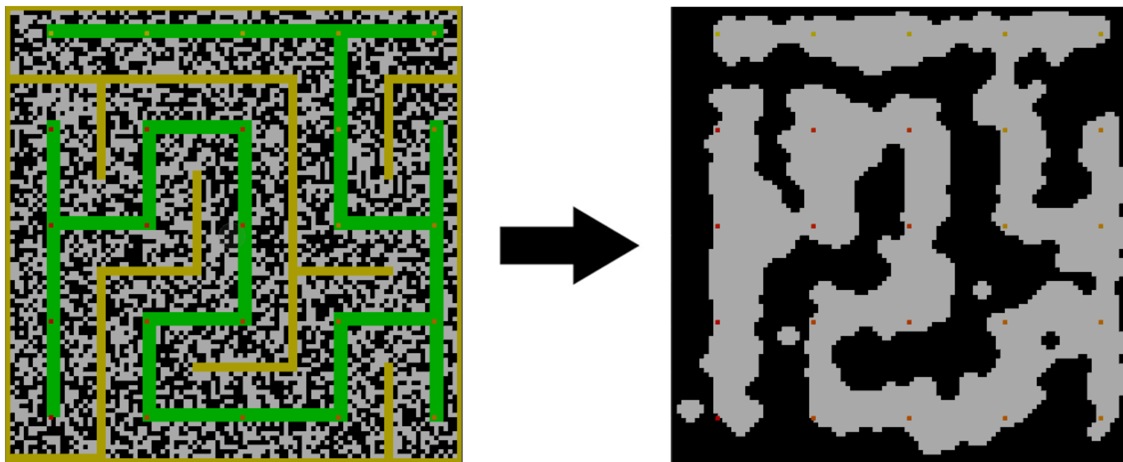


Figure 4.9. To the left, the Cellular Automata marked with the Space-filling Curve's path (green), as well as the negative-path curve (yellow). To the Right, the resulting automata after a number of iterations until it stabilizes. Parameters for this Automata are as follows: $N = 100$, $M = 100$, $Fill = 0.5$, $S = 22$, $P_g = 1$, $N_g = 1$. The colored dots on the right image are the points from the Space-filling Curve.

4.2.4 Polishing

We have been able to generate the automata whilst maintaining the cohesiveness of the player's path, as shown in Figure 4.9. The number of unreachable areas has been minimized, but the path still looks artificial: (1) The points from the curve are scattered in the x and y axis in regular intervals, (2) There are too many straight segments, (3) Parts of the automata's topography become marked by the straight line segments of the path.

During the configuration of the automata, two additional steps are introduced before the first iteration, in an effort to minimize the interference of the curves where it is not

intended: The first is to 'rotate' the grid, as to break the monotony of completely straight paths; the second is to shift each point on the path by a reasonable amount and then retrace the path through the connected points. This visually distorts the recognizable fractal shapes of the Space-filling curves, and changes the physical distance between points in the map.

4.2.4.1 Rotating

Rotating the map's grid presents a dilemma that requires design concessions. As the automata still requires a *MxN* grid, having it rotate means that one of two alternatives has to be taken to maintain it a 2-dimensional square grid: (1) After rotating the grid, scale it down to fit the original *MxN* dimensions; (2) Redefine the expected grid's proportions in each axis, depending on the rotation angle. Regardless of the alternative, rotating the grid is likely to cause some information loss during floating point to integer conversions.

The first alternative introduces a myriad of problems which caused us to avoid it: when a rotated version the original grid is rescaled to fit its original proportions, the number of random cells it has available to generate the automata could be greatly reduced. The same is true for the proportions of the curve-path and the negative-path: as both of them are reduced to fit the grid, it is not guaranteed that their girths will remain near their specified values. Assuming the worst case $M = N$, and a rotation of $\frac{n\pi}{2} + \frac{\pi}{4}$ radians (45°, 135°, 225°, 315°, etc. degrees), the length in the x and y axis of the grid would become $N\sqrt{2}$, meaning a 29% reduction of the available map space. Having less cells to work with causes makes it harder for the automata to create balanced, organic shapes, with reinforces the 'scarring' effect caused by the path curves.

4.2.4.2 Shifting

Rotating the Grid breaks some of the 'sameness' of the curve-path shapes, but the characteristic fractal appearance remains noticeable. As a second polishing step, each point within the curve-path is shifted randomly by an amount $\varepsilon = \frac{S}{2} - P_g - N_g$. This value is the same as the number of iterations needed to trim the negative space, and represents the closest distance between the curve-path and the negative-path. For each point $\sigma(x,y)$ in the curve, its new position is determined as $\sigma(x + Rand(-\varepsilon,\varepsilon), y + Rand(-\varepsilon,\varepsilon))$. The final result of both polishing steps is presented in Figure 4.10.

As all points were shifted, the fifth step described on Section 4 would have to be repeated in order to imprint onto the automata the shifted path. The original tracing of the unshifted path is still required for the tracing of the negative-path. Therefore, the order of the steps cannot simply be rearranged, and a second iteration of the printing sescribed in section

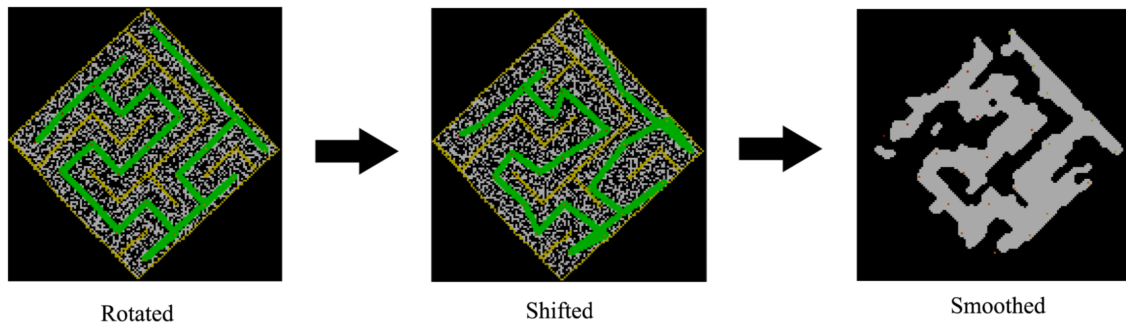


Figure 4.10. The same automata presented in Figure 4.9 after rotation and shifting operations are introduced.

4.2.2.6 is required. An overview diagram summarizing all of the methodology is displayed in Figure 4.11.

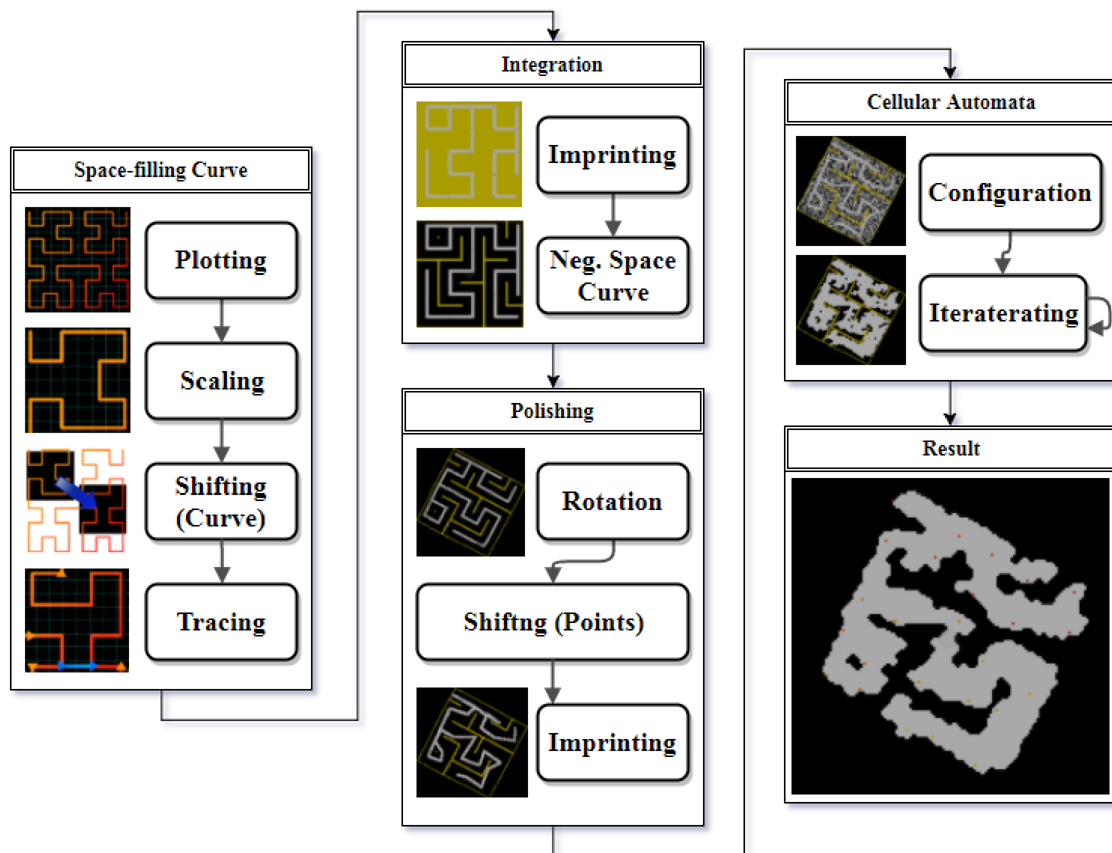


Figure 4.11. The complete sequence of steps for the generation of the procedural map automata. The images for the 'Space-filling' curve section are merely illustrative and do not represent the same curve-path displayed in all other images.

Both polishing operations complete their tasks within polynomial time. In conjunc-

tion with the other methods described in this work, the process for generating maps is still accomplished within acceptable complexity to be executable online.

4.2.5 Stacking Cellular Automata

As a theoretical background for expanding upon the concept of Cellular Automata as a basis for map generation, this work draws from the definition of Multi-layered Cellular Automata. We propose the concept of having additional layers of content being created by semi-independent 'Automata-Layers' that include within their rule-sets specifications for interacting with other 'Automata-Layers'. A visual example of this concept is presented in Figure 4.12.

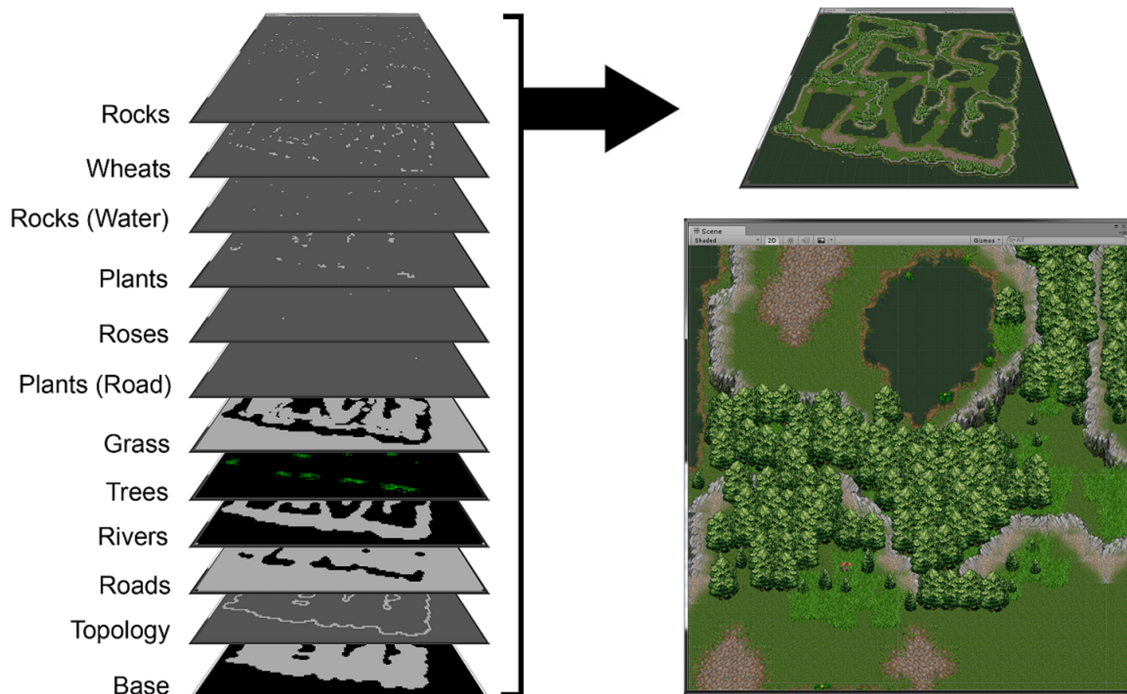


Figure 4.12. Stacking of multiple layers of cellular automata, each responsible for one element type in the map.

Their functioning is as simple as having additional cellular automata generating other features of the level such as vegetation, objects, roads, among other structures that could be modeled. These structures then use information from the original base automata, as well as other Automata-Layers

The benefit of having a multi-layered architecture is integrating the results of one automata with another. In this case, one could create multiple interdependent concepts, such as a 'tree-generating automata' and a 'cliff-generating automata' or a 'topology layer', as it will be refered to henceforth. By integrating these with the already established automaton,

we define the 'base' layer: one layer can use information from other layers to limit their own generative process. As an example, we next provide a description for the implementation of two types of layers that have been used on the qualitative testing of our maps.

4.2.6 Example 1: Topology Layer

While the resulting Automata conveys information by itself, it could be useful to draw specific types of information from it. While an algorithm that requires a specific type of information from a layer could implement a method to do so by itself, it would be far more organized to have this information stored within an accessible structure, assuming it is relevant for a number of other algorithms.

As an example of this, if the designer were to determine the positions within the grid where *True* cells change into *False* cells (borders), as to generate cliffs on the intersections of cell types, having a representation of the grid that contains only that information would be very helpful. In this example, a 'topological mask' could check these intersections and store it in a separate grid (or, for the sake of efficiency, store distinct classes of values within each cell). This type of mask is exemplified in Figure 4.13. And its implementation is as simple as determining which *True* (black) cells on the automata contain at least one *False* cell within a Moore Neighborhood of 1.

Figure 4.13. A base automata, a topology layer-mask generated from automata, and a mapped version of the topology automata to 2D sprites.

This topological mask by itself is already enough to generate cliffs from 2D tiles through a simple grammar-based approach [Shaker et al., 2016a], as exemplified in Figure 4.14.

This layer defines a prominent geographical feature of the map, and thus an approach of having additional layers such as the 'Tree Generating Layer' detailed below is the basis of our map generation. All we would need is to have both layers be mutually exclusive. Which means no single cell may have a non-zero value at each of those layers at the same time.

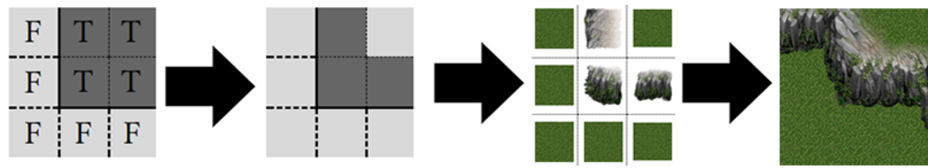


Figure 4.14. From the base layer, to the topology layer-mask, 2D tiles are generated depending on their position in the topology mask, creating the aspect of a continuous 'natural cliff' structure. The visual quality of the resulting tiles map (right image) is the result of editing and experimenting with selected free visual assets from Websites such as ope [2017]. As of now, we cannot present a theoretical basis as how to optimize the visual quality of matching 2D tiles.

4.2.7 Example 2: Tree Generating Layer

A 'Tree Generating Automata' is proposed as an example of an Automata Layer. This construct is somewhat more complex than the base automata presented so far, but it represents vegetation dynamics fairly well, as exemplified by Balzter et al. [1998]. It fulfills its purpose as a demonstration of the capabilities of the methodology presented in this work. It simulates the growth of trees within the map, while taking into account the player's path and topology.

The automata of this layer has cells whose integer values range from 0 to 255: cells with a value of 0 do not have a tree created at their position; Cells with values between 1 and 4 are spaces that are 'growing'; Cells with values of 5 and higher have grown into a tree. Each cell with a non-zero value increases its own value by 1 at each iteration, making so that the value of the cell represents the 'age' of the tree.

The updating method for each cell c that does not already contain a tree takes into account the age of each cell d nearby that has a tree, which is represented by $v_d = value_d - 4$. When updating a c cell it has a $1-(0.95^{v_d})$ probability of having its value change from 0 to 1, for each other within a Moore Neighborhood of 2 with $v_d > 0$. This automata presents no conditions for eliminating trees, as most automata that model organic behaviors, such as the iconic *Game of Life* by Conway [1970]. Therefore, if left to update during a sufficiently high number of iterations, the Automata would converge and stabilize once all positions on the grid are filled with trees. As it is defined, the designer would have to determine a number of iterations for the expected amount of vegetation desired, as shown in Figure 4.15.

As with the topology layer, we have set sprites for each tree: the older the tree, the taller its sprite image becomes, trees younger than 5 iterations have alternative, smaller sprites. An example of this are shown in 4.16.

With as little as 2 layers of content, the resulting map already shows promise. Many more layers can be added to extend one's desired concept of the generated level, and those presented here are merely examples and suggestions. Other types of environments such

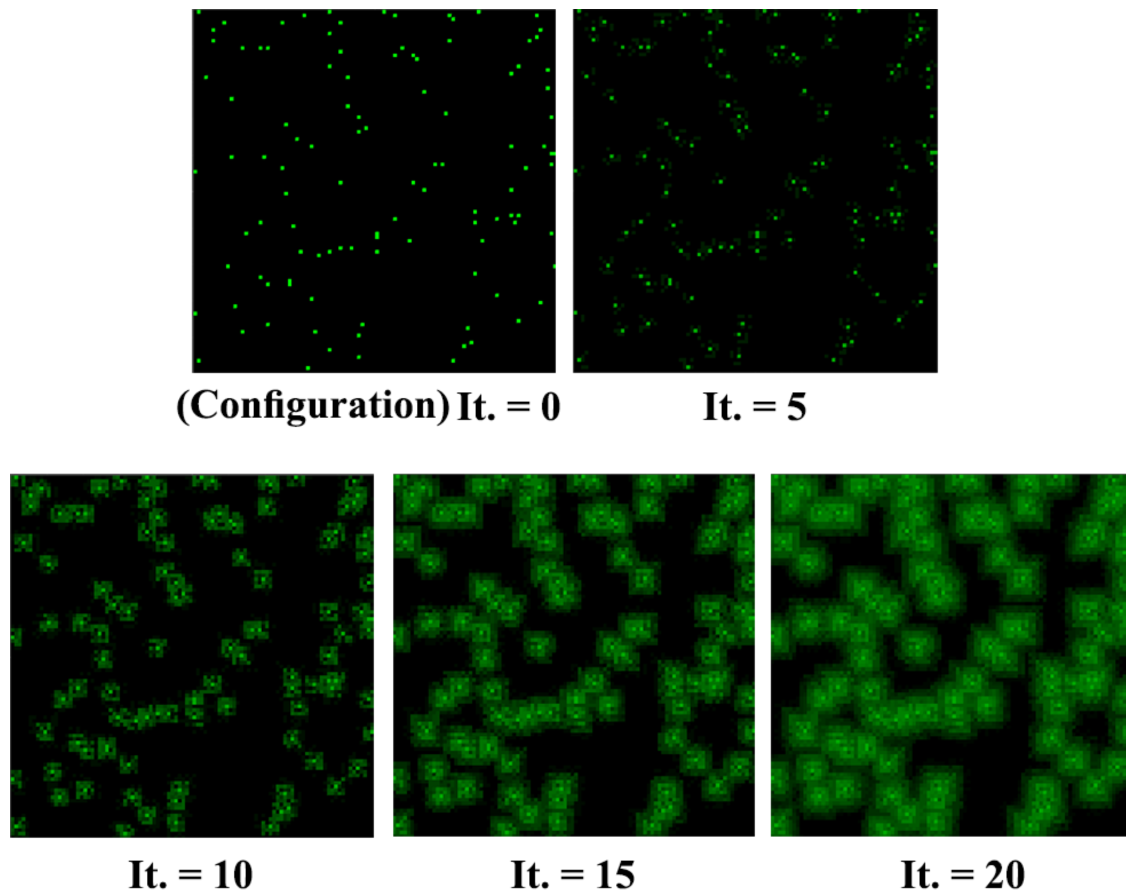


Figure 4.15. Evolution of the 'tree-generating automata' without the introduction of space-filling curves (Parameters: $N = 100$, $M = 100$, $Fill = 0.01$). The greener the cell, the older it is compared to the other trees. As trees need to be '5 iterations old' for them to start spawning other trees, the automata changes the most every 5 iterations. This age restriction also prevents trees from spreading wildly, instead forming small forests.

as caves, islands, mountains, and other organic environments can be designed from this methodology, as long as their characteristics can be reasonably modeled by the principles of Cellular Automata, which have proven in this and many other works to be extremely flexible. The entire process from the generation of the space-filling curve, to the end of the tree-generating automata takes an average of 8.6 seconds.

4.3 From Planning to Map Generation

With a cohesive subset of states, and a collection of possible automata to stack, what is left is the integration between both systems. As the planner generates a description of a map, all that is needed for the map generator is to select a number of states from that description, and create layers of Cellular Automata accordingly. As it will be shown in Chapter 5 the

Figure 4.16. Example of 2 layers of CA

placement and updating order of layers is a synchronous process that may cause the final aspect of a map to greatly change, as some layers of the stack might be limiters to the development of others.

The technical basis for converting the plan result to a map is one that is fundamentally easy, with a few caveats to address. Within the Unity Game Engine, the solution's implementation is equally simple: so far, every single method and algorithm previously mentioned to be a part of the procedural process has been implemented within Unity. In turn, this means that every single type of automaton is stored within a Class, and so is the case for every planning state, which is stored within a 'Solver' class. On Unity's Component based design paradigm, these generative constructs can be simply passed as parameters to other classes.

As a result, an interface class is implemented that takes the resulting states from the planner program, filters for those that are determined by the designer to be the ones that define which layers of Cellular Automata are supposed to be added. Then, each actor within these actions is mapped to an Automaton 'Prefab': a generic type of object that the Unity engine uses to instantiate almost anything, and is organizable by its intuitive drag and drop interface, as shown in figure 4.17.

Layers are iterated through sequentially, in the order by which they are placed. Meaning the first layer placed is effectively a single Cellular Automaton, while the second Layer onwards behave as Multi-layered Automata. While the effects of having all layers iterate at the same time is more interesting when dealing with integrated components, each layer would have to be crafted in a way that is too time consuming for the scope of this work.

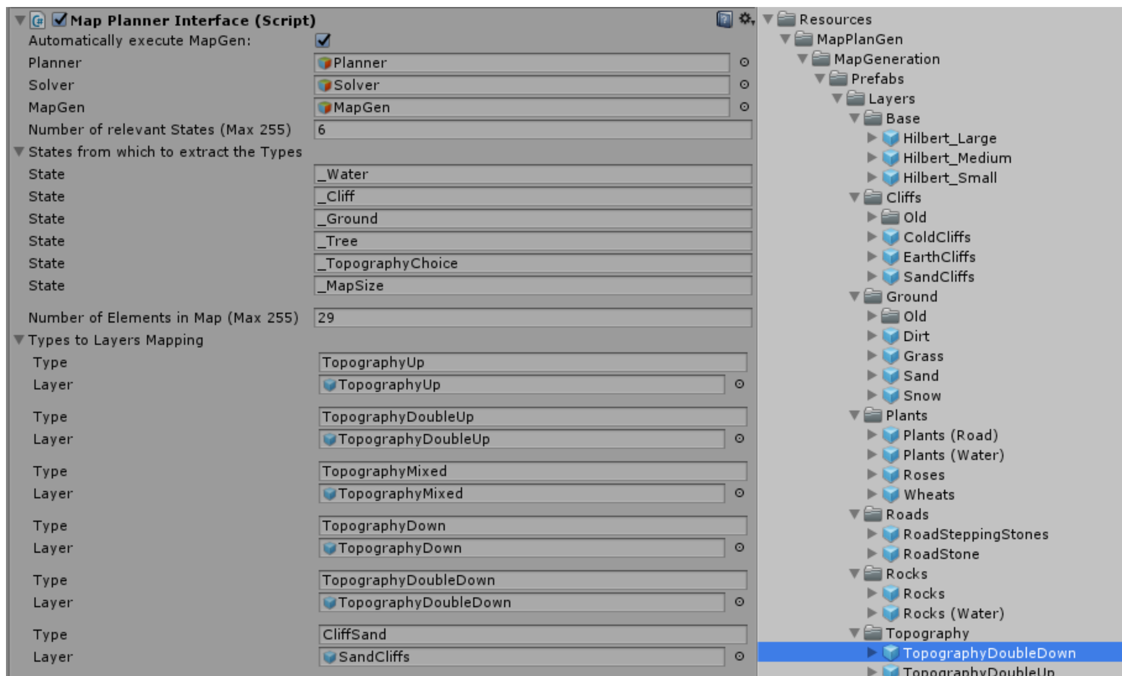


Figure 4.17. C# Script component that takes references for the planner library, the planning solver, and the map generator (respectively), and filters 6 types of states from the resulting planner's solution (a previous example of a solution is as shown in 4.6). Then, actors from the planning Domain Library are mapped (a String to Prefab dictionary) onto an Automaton Prefab, which is stored on one of the project's folders.

The way it is, each layer has a 'priority' parameter ranging from 0 to 1, determining how likely this layer is to be set up first, as layers are chosen. Also, a second 'ignore probability' parameter guarantees that, once all layers without this parameter are determined in order, layers that ignore this random sorting are placed above all layers of lower priority. This setting is useful for creating environments where a layer is too significant to be allowed to iterate last. For example, an island themed map would likely need a 'Water_Island' layer that needs priority over other layers that fill a lot of space, such as the 'Tree' layers. Figure 4.18 shows the planner Unity C# script component that handles all placement and iteration of the layers, as well as their organization after generating a sample map.

Having the integration between planning solutions and Automata complete, the next step for expansion of this methodology is to improve on the execution and design of its components. A generated map should be only as good as the quality of the planning Domain Library and Layers that define its creation. As the design and expansion of our Domain Library is an arduous task that requires a shift from a research perspective to a design one, we have refrained from investing even more time into improving the library presented at Appendix A. Furthermore, from here onwards, we shall discuss the results and testing of all the methodology has been already defined so far.

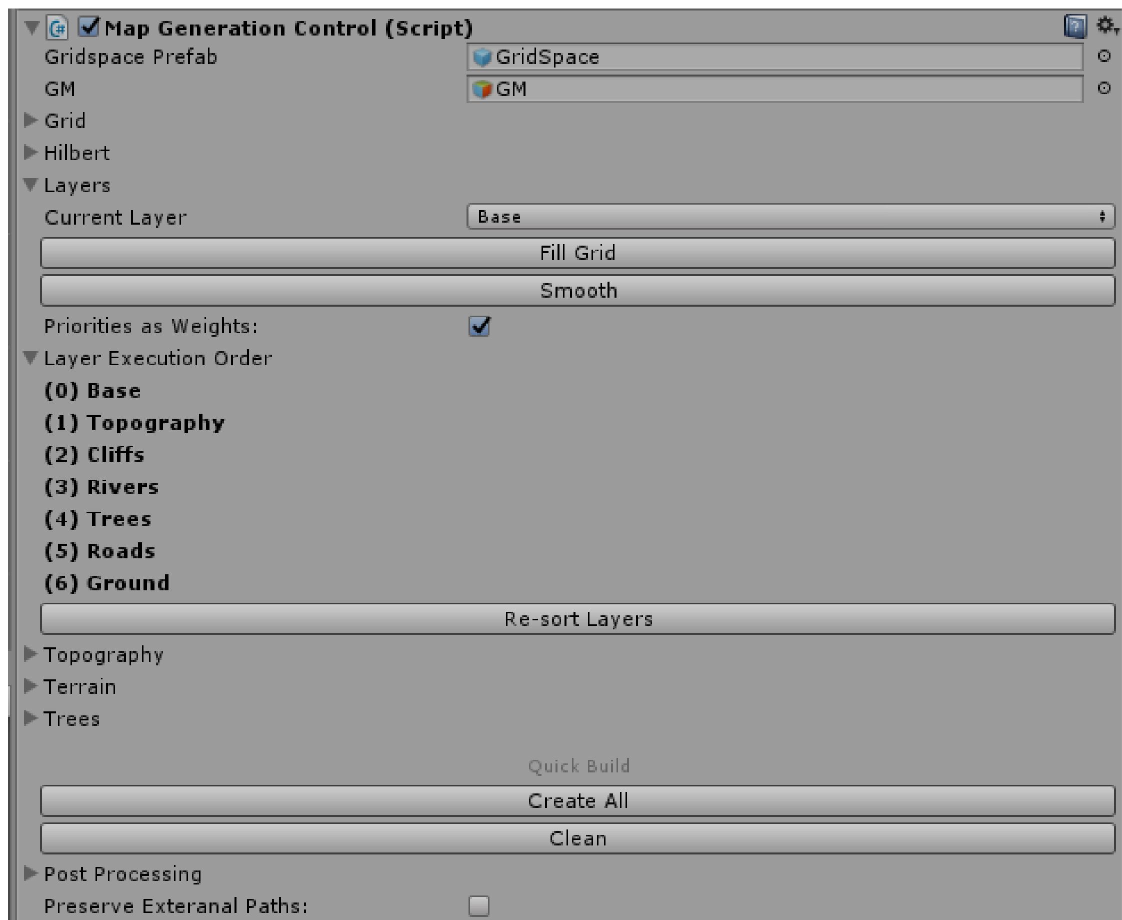


Figure 4.18. Unity Component for Map Generation. Information about distinct parts of map generation are separated in drop down menus, as is the case with the open 'Layer Execution Order' segment. Graphical interfaces such as this have been made for all components of planning, map, and their integration.

Chapter 5

Quantitative Analysis

The end goal of the procedural map generator is to create maps that feel interesting. It makes it so that evaluating the users' opinions is our main indicator of quality. Furthermore, the Quantitative Analysis of our results focuses on their mathematical expressiveness, and how reliably these are results achieved. In other words, to guarantee that our generator fulfills it's purpose, it is necessary to evaluate its correctness and performance. Hence, this section discusses the resulting maps' consistency, variety, among other factors. In order, we first present the design of the maps used for both Qualitative and Quantitative testing. Then, the performance costs of generating these maps, and the mathematical variety and consistency of Map and Theme generation.

5.1 Resulting Maps

The perceived quality of our generator regarding playability is directly proportional to the designs created with it. It would be hard to argue in favor of the quality of a tool without examples of its use. Yet, to develop a tool and to learn and master designing for it is a task too great for our limited scope. As it is, we have designed a total of 5 layers of functional Automata, as described below.

1. **Base**: The initial layer containing the base cellular automata set up presented in Section 4.2.
2. **Grass**: A trivial layer that has a singular rule of setting all its cells to one.
3. **Cliffs**: The Topology Layer presented in Section 4.2.6.
4. **Trees**: The Tree Generating Layer presented in Section 4.2.7.

5. **Water**: A Flood-Fill algorithm based automaton layer which generates bodies of water, and then shrinks and morphs that body within the confined available spaces limited by other layers.

6. **Roads**: A random filling and trimming algorithm based automata layer which uses the same smoothing techniques as the base automata layer to generate roads along some of the Hilbert Curve's path as a means to guide the player through the curve.

Each layer, as described in Section 3.2.2 is able to interact with others. In this case, it can be summarized to trees not being able to grow or germinate on water, cliffs not being able to be placed on trees or water, roads only being placed around the Hilbert Curve, and similar interactions.

The proposed methodology allows the generation of an organic geometrical configuration adapted to a player's path through a level's map. A completely procedural progression for the player to explore is generated with both a path from a starting area to the goal, and parallel ones for exploration. Furthermore, these are not immediately noticeable by the player, even if they are given the opportunity to see the map in its entirety. This is only possible due to the versatile self-organizing properties of Cellular Automata. Even by adding as few layers of content as we have, we are able to generate interesting configurations.

However, as each game has its own features, a general purpose generator can only get so far: a decent, complete, game level requires many other resources such as enemies, items, power-ups, and other intractable elements. It would also require other topological and environmental elements to bring an interesting, visually appealing game. All of these domains of content being specific to the archetype of game in question. Furthermore, introducing additional elements to the map requires a flexible framework with which existing constructions could be interpreted for determining where to create additional content. For a first impression of the visual aspect of the generated maps, we have provided a mosaic with distinct generated maps in display, presented on figure 5.1.

5.2 Map Geometry

As a preliminary evaluation of our results regarding the map geometry, it is often the case with results that are hard to evaluate quantitatively to come down to displaying examples while attempting to minimize the authors' bias. To counterbalance this, we have selected a large set of maps presented in Figure 5.2 generated with random seeds, but the same parameters presented in Figure 4.9, to help the reader to formulate their own opinion on the basic patterns of generated Automata.

One repercussion of adding the curve-path that we did not initially account for was how some maps would become more claustrophobic with few open areas. This occurs mainly due to there being less cells and space to generate topology. To compensate for this, we have performed tests varying the number of cells that are generated as True during the configuration step, as shown in Figure 5.3. Each configuration of the base cellular automaton was generated at an average of 0.8 seconds.

While it is undeniable that some of the charm of pure cellular automata is lost, we believe is one problem that for now has to be coped with. Pure procedural systems might generate beautiful results, but their unpredictability hamper the possibility of utilizing them in commercial games. In no way do we intend to critique or demotivate the development of other methods that accept the randomness of pure automata or other procedural systems. On the contrary, we are eager to see alternatives to our methodology. But we do acknowledge that our intent is to better harness procedural systems, and that some of its natural beauty may be lost for it.

The general image of the Space-filling Curve is still identifiable in some of the shapes. It is a sign that more controlled random factors need to be introduced within the curve-path algorithm, as an attempt to introduce new, alternative paths. Yet, the generated map shapes follow a visible progression and allow for use as a base for a completely procedural level design. While improving upon this system is also a task for future work, still within this one we have introduced the concepts on how to build content upon the basis of generated maps.

5.3 Map Generation Variety

One of the most important influencing factors on the map's final appearance is the random seed that defines the results of every purely random and of every stochastic process within the Map's Generation. How easy it is to replicate a previous result is a quality check-mark for any procedural generator, and it should almost always be solvable by simply reusing the same random seed. Our procedural method achieves this level of consistence for all its components (planner, automata, and integration), however this raises an specific question based of the effects of locking the results of one, and then allowing the other to experiment with another seed. Fundamentally, it is interesting for us to see the visual aspect of generating a set of maps that, while receiving the same configuration of states from the Planner module, execute their automata with distinct seeds.

As shown in Figure 5.4, by changing the map's random seed while keeping its theme, we are able to generate maps of similar visual appearance, but distinct geographical configurations. Meaning our generator is consistent both with its individual components and with

their combined system.

Another crucial factor that changes the map's geometry once all layers are placed, is the order by which they are executed. As many layers that impose movement restrictions on the player have limitations in occupying the same cells in the map. Thus by having a layer be chosen to be executed first, the visual aspect of the maps can drastically change. A purely demonstrative example is presented in figure 5.5.

In this example, a we have changed the iteration order of two layers that create movement restricting obstacles, and it has caused the overall theme of the map to shift from what we would call 'a forest with rivers' to 'a small island'. Regarding the distinct shapes of the player's path despite the same random seed being reused, the random system utilized in our process utilizes a queue of a randomly organized distribution of numbers, instead of being reliant on time, or other mutable aspects that could be affected by performance hiccups. By changing the layers that first take from this queue, all further non-deterministic decisions are influenced.

5.4 Theme Planning Variety

A simplistic estimation of the amount of content that can be generated by a Map Generation oriented Domain Library would be to consider the number of permutations of content that could be selected from each group of layers. For example, in the library shown so far (Appendix A, we have 3 distinct map sizes, 3 types of cliffs, 3 types of ground, 4 topographical configurations, 5 vegetation configurations (including no vegetation), and 4 configurations of hydrography. This estimation would suggest that there are, considering only the theme, $3 \times 3 \times 4 \times 5 \times 4 \times 4 = 2880$ distinct map themes. In practice, this number is much smaller, due to the very nature of planning problems.

While planning, there will be always mutually exclusive actions that may be spread over any interval of time. Meaning that an action may create a configuration of states that restrict other actions from ever being immediately executed, and so can happen that the restricting action could be the first, and the restricted one could have been the last. On planning problems that allow for cycles, that is, sequences of actions that have a configuration of states be created, undone, and then remade, verifying these exclusive relations becomes much harder. Worse yet, some planning problems might see themselves having actions that can never be performed. Not because the action is too restrictive by itself, but that the system generated by the sum of all the domain's actions and their repercussions make it so that these actions become impossible.

For us, guaranteeing that every created automaton layer has a way of being chosen, and

therefore appearing in the final map, requires us to only make sure that the action that sets that layer is guaranteed to be possible, regardless of how much its restrictions, or restrictions from other actions hamper it from showing up in the final maps. To verify this, we have also developed a 'Tracker' module within Unity that measures over *N* iterations of planning how often certain state appears on the final configuration. Meaning that with at least these *N* versions of maps, each layer is guaranteed to appear at least once, and therefore, at least one action that could determine its appearance is not impossible. Further work experiments should also track action calls, monitoring if each action has been chosen or not. For now, our focus is to have every layer present on the map, and therefore it is only necessary to monitor their incidence. A partial example of this state tracker is shown by Figure 5.6.

Still on this figure, there are cases of layers that are very likely to be set. For example, a 'HidrographyCoastal' layer has a 26,4% percentage of chance of appearing on the final map based of the total of solutions verified, while others like the 'HidrographyIsland' having a low chance, at 7,2% chance of appearing. That may be due to the set of actions that define what type of tree will appear being too biased, or that the possible combinations of states taken until these actions are taken are biased instead. Likely, it is a combination of the two possibilities.

When developing a map with this in mind, it is up to the designer to decide whether having a layer be a rare occurrence is desired. An expanded version of our concept could include types of treasures to be placed within the map, and by that example, having a layer being unlikely to appear could be an interesting take on the design of the game's planning domain library. Likewise, experimenting with the actions and testing how it affects the end outcome of a problem that is exponentially hard is left for future work, be it academic, or within a functional game.

5.5 Performance

Experiments were performed on a 3.20 GHz 64 bits AMD FX-8320E Eight-Core Processor with 8 GB RAM, and a Nvidia GeForce GTX 1060 6G graphics card. For the purposes of testing the process adaptability to be executed during a game's runtime, each iteration of any Cellular Automata, planning, as well as other minor tasks, are partially divided across distinct game frames. This means that, while it takes longer to be completed, the methodology can be executed in the game's background. While a considerable slowdown is still present, we firmly believe it could be improved through a more effort on optimizing the division of tasks across game frames.

Besides optimizing as development went, no attempts were made into developing a

basic overhead optimizer to the map generation, or even a post processing cleaner to avoid the performance cost of the high number of objects generated. As mentioned, we did put effort into reducing the workload of a single frame by splitting object creation along multiple frames, though not enough that no considerable slowdown will be present. In this alpha implementation of our generator, performance is a heavy burden that requires a heavier investment.

While analyzing the time cost in relation to the size of each map, we have determined that, as expected, the amount of time required to generate a map is proportional to its size. However, there were a high enough number of outliers to indicate that size by itself has only a partial influence on the overall time cost. The results over the testing of 33 maps are presented on Figure 5.7, where a function reliant on the system's time is used to capture the time required to create maps without being influenced by system slowdowns and hiccups.

As shown in this Figure, the time required for the Planning of the map's theme has, in all cases, been lower than 1 second, being even hard to see in the graph, and therefore negligible when comparing to the amount of time required for the the remainder of the generation process. The integration module that maps the planner's results to the layers of automata to be created also does so a very short time. What then remains is divided into two groups: (1) Automata; (2) Other. The former represents the time required from configuration step to completion of all Automata layers. The latter represents a number of processes required to initialize, maintain, and terminate other systems for generating the map. The worst culprit of all, being the calls to instantiate and delete the Game Objects required to form the map.

Within the Unity Engine, every object that exists in physical space is a member of the 'GameObject' class, with a number of restrictions, such as having a 'Transform' component that represents its position within game space. The act of instantiating and deleting objects is a costly one, and without optimizations, our generator is guilty of creating a high number of objects. For every grid space, a single object is created. Within it, for every layer that creates an object on that position (often by having the value of the cell being greater than 1 for that layer), one additional object is created. The process of positioning these objects and adjusting them in other parts of the generation procedure also creates and destroys objects.

This could be resolved easily with optimization steps specific to each layer. For example, a ground or water layer does not need one tile per grid space it occupies, instead it should be created only if the contents of that layer are visible. A solution that would require handiwork is to make certain tiles stretch if they occupy convex groups of adjacent tiles, in order to reduce the number of title game objects being instantiated. This among other optimizations, such as reducing the number of physics simulating colliders that stop the player's movement in the map using the same logic as reducing tiles would greatly aid our performance costs presented in figure 5.8.

Most of our memory consumption, ranged from roughly 30 Mb to 100 Mb from smaller to larger maps. A substantial amount of memory is allocated specifically for the map's components. The maintenance of physics required by the number of colliders from trees, cliffs, and water costs over half of the total memory allocated, with roughly 30% of being used when monitoring the entire map all at once. Again, Unity provides methods for reducing this physics maintenance cost and there exists a certain 'hygiene' for creating physics objects while also reducing this burden. As the generator evolves and grows, optimizing these criteria is likely to be a high priority.

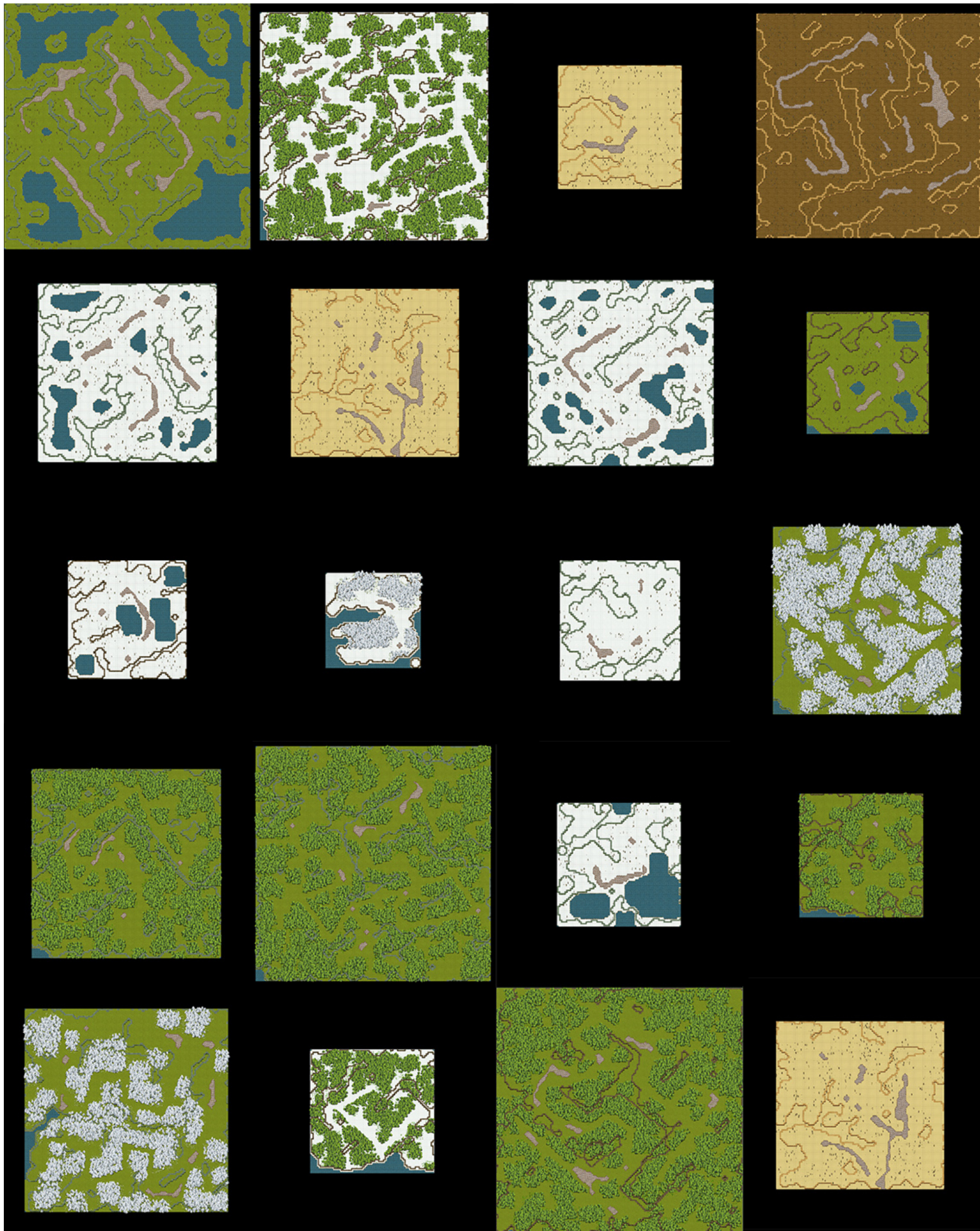


Figure 5.1. Mosaic with different maps generated with different random seeds. Maps may have different sizes.

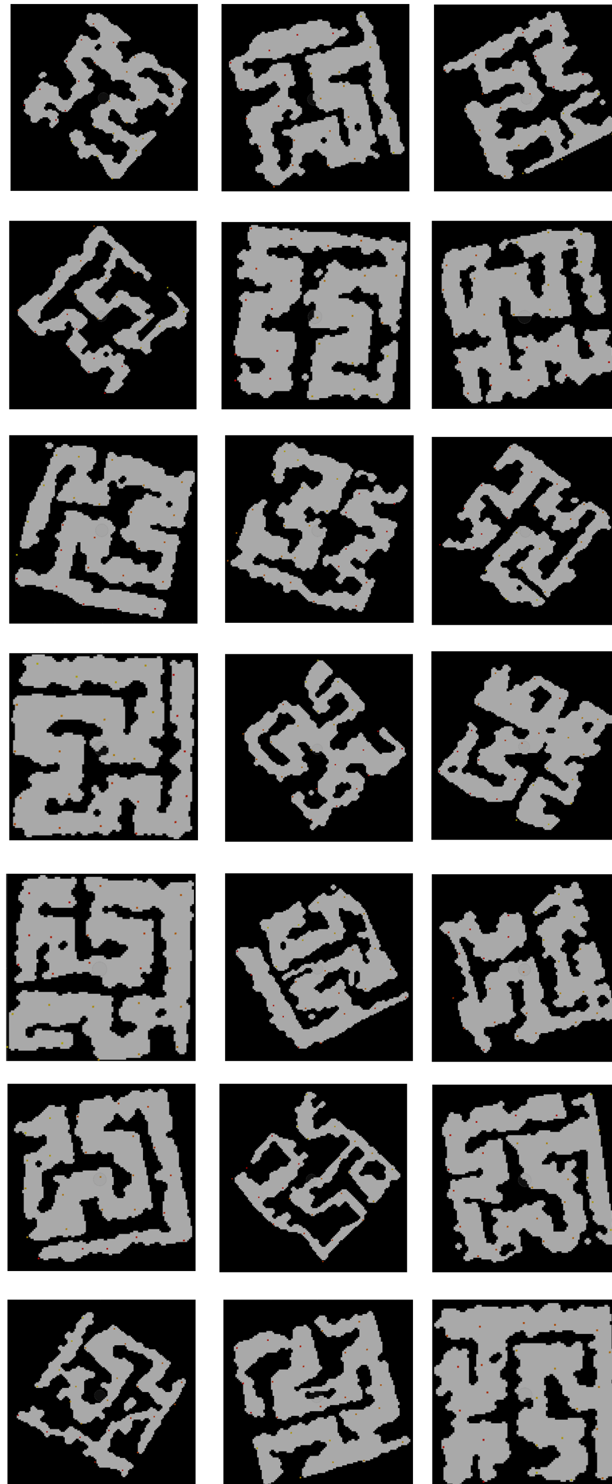


Figure 5.2. Different versions of Map Geometry achieved by variating the random seed. Parameters for these automata are as follows: $N = 100$, $M = 100$, $Fill = 0.5$, $S = 22$, $P_g = 1, N_g = 1$

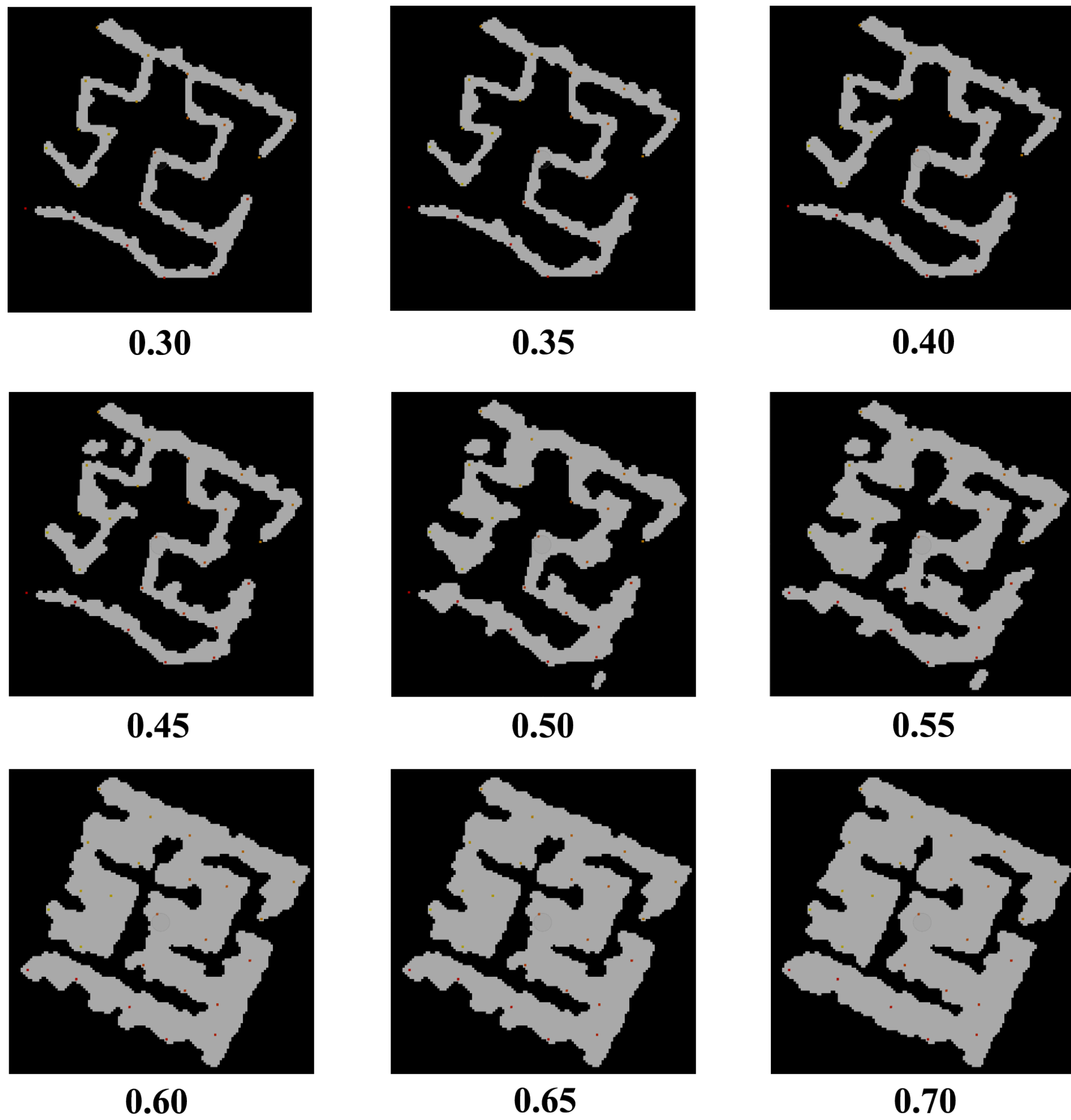


Figure 5.3. Map Geometry experiments with the Fill parameter for different values as to find a compensation for the reduction in random cells by the imprinting of paths.

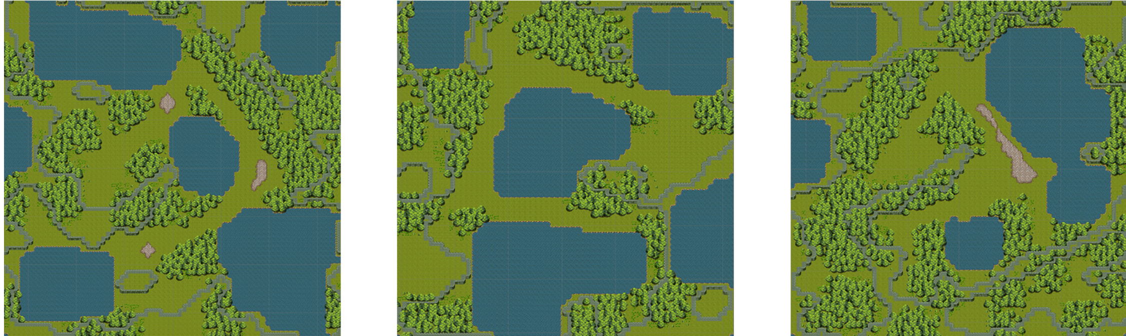


Figure 5.4. Variations of the same 'theme' for a Map with different random seeds for its Automata.

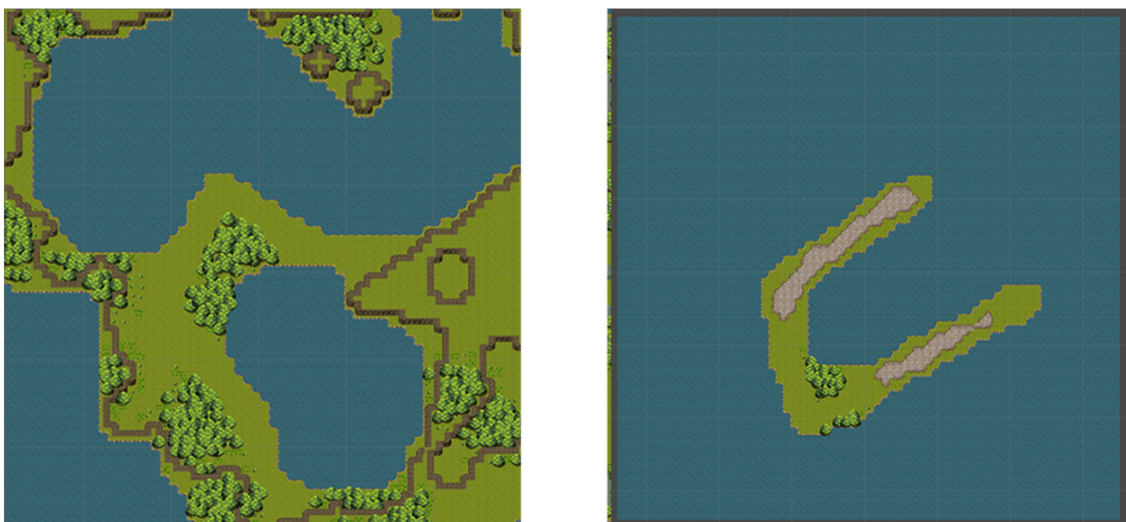


Figure 5.5. Two maps with the same random seed. On the left map, the cliff topography layer iterates before the river generating layer. On the right map, the river generating layer iterates first.

Other	Frequency
Stage1Complete	100.00%

Topography (Chosen type)	Frequency
TopographyDoubleUp	33.50%
TopographyUp	22.90%
TopographyMixed	2.80%
TopographyDown	7.00%
TopographyDoubleDown	33.80%

Topography (Types that can be chosen)	Frequency
TopographyDoubleUp	26.20%
TopographyUp	25.50%
TopographyMixed	7.30%
TopographyDown	14.70%
TopographyDoubleDown	26.40%

Map Size	Frequency
MapSizeSmall	32.80%
MapSizeMedium	34.20%
MapSizeLarge	33.00%

Temperature	Frequency
TemperatureScalding	5.90%
TemperatureHot	31.40%
TemperatureTemperate	25.60%
TemperatureCold	30.90%
TemperatureFreezing	6.20%

Hidrography	Frequency
HidrographyDry	24.90%
HidrographyPonds	24.90%
HidrographyRivers	16.60%
HidrographyCoastal	26.40%
HidrographyIsland	7.20%

Vegetation	Frequency
VegetationNone	16.30%
VegetationSporadic	22.50%
VegetationSparse	22.50%
VegetationFocused	14.30%
VegetationDense	24.50%

Figure 5.6. A segment of the results of the tracker condensed into multiple tables. The tracker shows, for a number of instances of the map generator (in this case 1000)), what is the frequency of each layer being in the set of layers chosen for the final map. Certain groups of Layers, such as the 'Stage1Complete()' and an additional set of topography layers are used as intermediary states for the creation of the map.

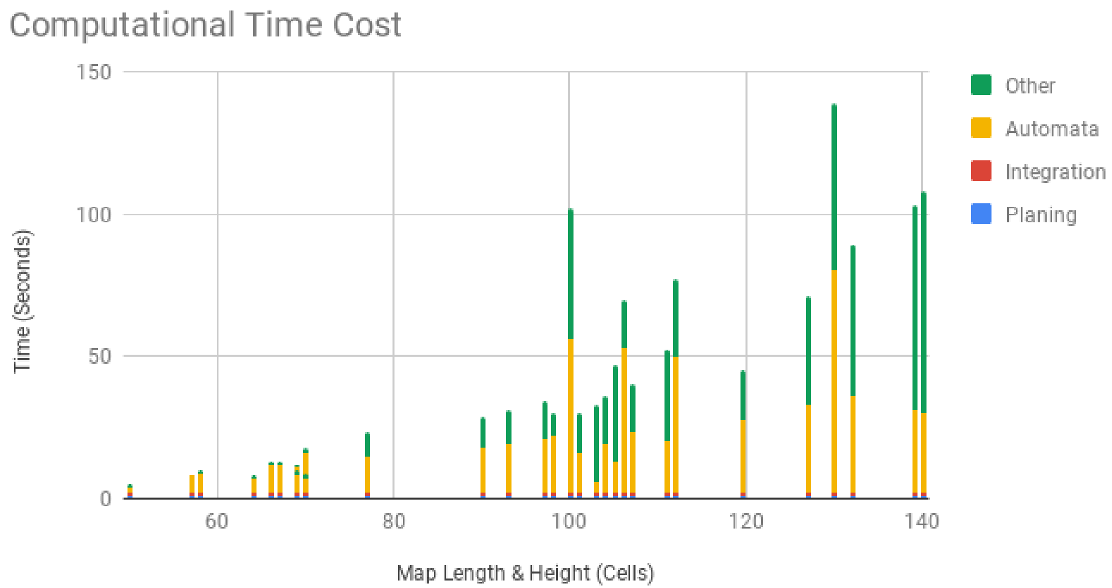


Figure 5.7. Plotting of the time required for the generation of maps of various sizes, with distinct random seeds.

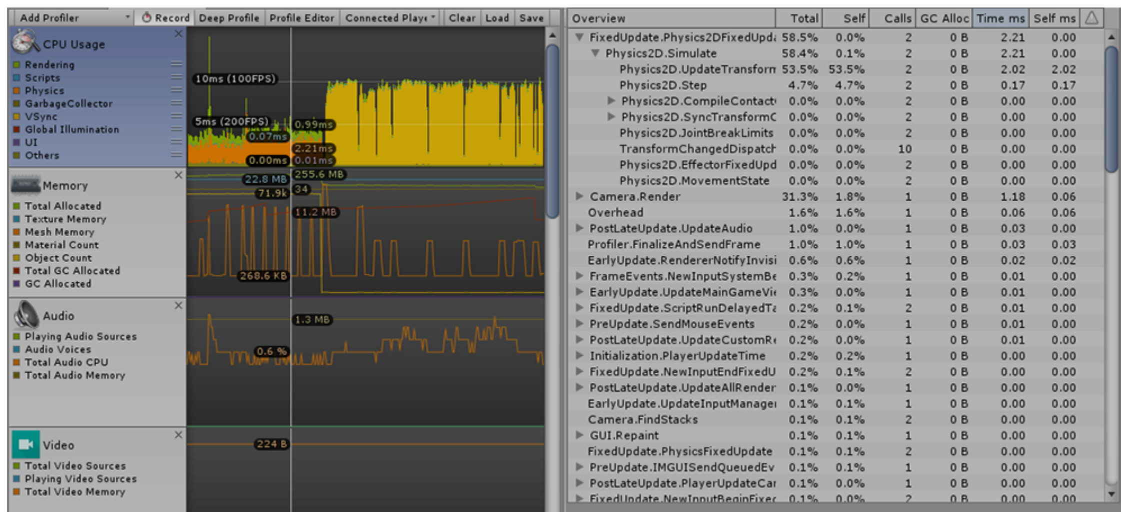


Figure 5.8. Unity's profiler tool measuring, to the left, the CPU Usage and Allocated Memory over time for an average sized (aprox. 75x75 cells) map. To the right is a detailed measurement of the percentage of memory consumption per type of component in the game in the current frame (marked by the white vertical line on the left side). The most important information to pull out from the profiler is it's first line: 'FixedUpdate.Physics2DFixedUpdate' is the overhead required to evaluate the physics properties of the map's objects and taxes 58.5% of Unity's allocated memory.

Chapter 6

Generator Analysis

In this Chapter, we present our objectives for the qualitative analysis of the generated maps, and which tests we have chosen to perform to properly address each topic. The main purpose of these tests is to have human users navigate through the generated environment, then report back and elaborate on their views. Offering a player an environment to simply move across is not very entertaining unless the the environment itself is fine crafted to invoke sensations, such as awe, curiosity, tension, dread, etc.. Procedural systems have long ways to go before reaching this level of craftsmanship, and in the scope of this research, it is simply not achievable.

Therefore, an intermediary interface has to be placed between the player and the generated maps. One that gives the player meaning and purpose for their traversal of the environment. In lieu of this matter, we have developed a simple game in Unity that puts the player in the role of a character looking to find an objective within the map while facing basic challenges to keep them engaged with the experience. To offer each subject incentive to explore areas parallel to the main level path, and thus explore the map generation's own ability to create divergent paths, several collectibles that aid the player in reaching their goal are placed in each stray path generated by the Space Filling Curve. Further details of the developed game are presented in Section 6.2.1.

To give the player a brief time and experience to navigate through the maps with ease, three different maps are provided to each subject. The first two consist of $M = N = 50$ sized grids, with the last one being of size $M = N = 75$. The remaining parameters as referenced in Section 4.2 are the same for all maps: $Fill = 0.5$, $S = 22$, $P_g = 1$, $N_g = 1$. By having the player navigate through paths that are essentially linear, we may prepare the player to navigate with confidence on a more complex environment. Whilst many tests show that a $M = N = 100$ map is feasible, a map this large takes even for us the designers too long to travel for the purposes of experiments with multiple subjects.

Concerning the Qualitative Method to use, we have come to the conclusion that the choice of performing interviews with open questions would be most beneficial to our purposes. Interviews themselves have their own variations that allow narrower or wider room for volunteers to express their opinions, and some questions may be designed to target a specific demographic of the volunteer set. For us, it is interesting to get both the perspectives of future users of our system, and that of other designers. Therefore, we have split our interview questions into two groups: the first being directed at players, and second being directed at designers who, by also being players, are addressed the questions in the first group as well. Questions other than those directed at designers shun from mentioning procedural generation as to avoid bias toward the repetitiveness stigma of procedural generation. Instead, if such feelings of bore are provoked by our experiment, our interview protocol looks to determine this through other indirect questions. Furthermore, our semi-structured interview method was intended to be malleable enough that our open questions could be changed during the interview to shift the player's answer closer to covering their whole thoughts about the generated maps. Although the interview protocol had numbered questions, these numbers were never enunciated to the subject. In case the subject did not touch on the desired answer topics, a second variation of a question was asked. One that was closer to having the player express their opinions on the maps without giving out our intentions to evaluate procedural maps.

Our interview protocol and its questions were designed in a bottom-up approach, meaning we first decomposed the criteria and topics that were relevant to us, and then defined our methods. This interview method was triangulated through the Observational method, which consisted on metrics collected during play time by the interviewer regarding the player's performance, tendency towards exploration, difficulty, completion time, etc. The contribution of the observation during the analysis is twofold: (1) To assist in validating a player's claims and verbalized thoughts during the interview; (2) To register specific events during the player's experience that they might not remember while answering the interview's questions. And then to remind them of these events.

The player's opinion of the game is inevitably going to influence the way they perceive the experience as a whole, as well as their thoughts on the generated maps. We have addressed this bias by having the first questions cover almost exclusively the volunteer's opinions of of the game. These are also some of the open ended questions of our interview protocol, and they serve to have the player express their general thoughts as early as possible on the interview, allowing us to obtain this information as early as possible. This method had been tested on the pilot control group, and has achieved its purpose on the test interviews. Finally, all the topics introduced above fit within the following steps:

1. Select a group of possible volunteers.
2. Generate a subset of maps for testing.
3. Place game elements into a map as to make them playable.
4. Have the volunteer grant their written consent for the usage of collected data.
5. Present the player with 3 Maps (two small and one medium sized) in sequence, by having them play a simple game on each of the maps.
6. While the player navigates through the maps, collect observational data.
7. After the play session, conduct the interview.

In order, the following sub-sections will address each of these items, as well as establishing all requirements and technical specifications for each.

6.1 Volunteer Selection

Based on the scope of this research and its testing, a total of 14 users was determined as the ideal feasible goal, 2 of which were assigned to the pilot tests. To address both the views of the players of the procedural systems, as well as to collect game designers' perspectives on the generation process, we divide these 12 test volunteers into 2 groups: (1) Six of them that play games as a hobby (hereafter refereed to as group A); (2) And 6 that actively work on game development (hereafter refereed to as group B). The volunteers' age ranged from 23 to 32 years old, from which 12 were male, and 2 were female.

We were interested in looking for subjects that, at least seldom, critique the games they follow or play, so that they may be comfortable in giving their inputs on the testing experience. As to determine the concept of 'casual' and 'hobbyist' players with average or higher playing experience, we had to define a favorable criteria for establishing a threshold for the time each subject spends playing video games.

Thankfully, Andrejkovics [2016] divides professional eSports players into categories (Amateur, Professional, Determined, Fanatic) based on their average daily and weekly play time. This work also suggests that the hobbyist digital games player enjoys up to 8 hours a week of playing video games. Although encompassing the hobby category into a single 0 to 8 hour weekly playtime does not inform us of the the distinct degrees of hobbyists, it does help us to guess how to determine the type of game player we are interested in. Based on the average hobbyist, we have then determined that our ideal volunteers adhere to the following profile:

- Spends an average of 4 to 8 hours weekly playing digital games.
- Consider themselves able to critique to at least some extent the video game media, including their own experiences.
- **Spends an average of 20 hours weekly developing digital games, or assets intended for digital games.**

The last requirement is only limited to the 6 subject portion that work at least part time in game development, as determined by the minimum of a half time work schedule.

6.1.1 Consent Form & Information security

For each volunteer that qualified by the selection criteria above, a consent form was granted to that subject. No monetary, financial, or any other type of valuable reward was offered to any subject. This consent form is annexed both in the Portuguese and English language as it was presented to volunteers in appendix B. The form explicitly allowed the volunteer the right to have all their personal information, as well as the very fact that they have participated in this experiment, to be kept confidential.

After agreeing to the established terms, Volunteers then received a copy of the consent form with the Interviewer's contact information. And the testing procedure began by having them play the three game levels that take place on the generated maps.

6.2 Map Distribution

To instill a sense of progression, as well as allowing the player time do adjust to the game before undertaking a larger map, players were given 2 small maps with $N = M = 50$, sequenced by a medium sized map with $N = M = 75$. The first two maps contained a more restrained set of challenges, as well as presenting them at a more forgiving pace.

Maps for testing were initially generated one by one, and then stored onto Unity's 'scenes', which are essentially distinct levels. Once all maps had been generated, we had the game elements described on Section 6.2.1 placed manually. For each subject, a number of maps were created in a way that a sufficiently ample range of map configurations could be tested, but rather than assigning each player a unique triplet of maps, as part of the experiment we have chosen to form a distribution that repeats maps along distinct subjects, allowing us to investigate on convergent or divergent opinions. This distribution is illustrated in table 6.1.

Per the choice of maps to be presented, a certain level of 'cherry-picking' was required in selecting which maps, from all those that had been created would be selected for testing, in

Volunteer	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	M1	M2	M3	M4	M5	M6	M7	M8
VA1	X	X													X							
VA2			X	X												X						
VA3					X	X											X					
VA4		X			X													X				
VA5							X	X											X			
VA6	X					X									X							
VB1									X	X										X		
VB2			X								X					X						
VB3												X	X								X	
VB4										X		X										X
VB5								X						X				X				
VB6									X				X							X		

Table 6.1. Division of maps among volunteers of groups A and B (numbered 1 to 6). The maps were labeled by number. Any map starting with 'S' is a small map, and maps starting with 'M' are medium sized maps. Before any testing, maps were divided into volunteers in a way that all but 2 medium sized maps were tested by a total of 2 volunteers.

a sense that there had to be a certain variety in the themes of maps that would be presented. As shown before in our discussion about tracking the probability of each state appearing in the finalized map (Figure 5.3), certain layers had a much lower chance of showing up in finalized maps. Therefore, we had to generate a higher number of maps than those presented, and select sets of maps that presented overall different themes (combinations of layers). To force the existence of rare layers, the state that generated that layer was introduced into the starting state of the planning problem, in such a way that it minimized its influence on other layers and guaranteed that it would appear on the final goal configuration of states.

6.2.1 Unity Game

Each player was instructed to play three consecutive stages of a top down action shooter where they, under control of a character wielding a gun, must find a Helicopter to escape to the next level. Movement was performed with the W,A,S,D keys to move Up, Left, Down, and Right respectively, or in any combination of the four. With the mouse, players could direct the character to aim at a certain direction and fire their gun with the left mouse button. The choice for a Top-down action shooter came at the necessity for a quick game that allowed the player to navigate through a map while having a sense of urgency and need to explore. Once the player reached the helicopter, a cut-scene of the helicopter moving away would play, fading out and into the next stage of the three (if any).

The game has only one type of enemy: A zombie with the simplistic behavior of chasing a player within range, and then periodically dealing damage to the player if too close to it. The zombie moves slower than the player, allowing them to avoid zombies if they felt necessary. These non-playable characters would often be placed in waves of 3 to 6 zombies, or in special cases like the end of a stage, hordes of 12 to 20 zombies. The number of enemies

found at each point were progressively increased on later maps. An image representing the overall visual aspect of the game is presented in 6.1.

Figure 6.1. Screenshot of the developed game.

Four types of weapons are available to the player: (1) a straight shooting pistol with endless ammo; (2) a machine gun with a hundred shots that fires quickly but with a large random spread; (3) a shotgun that fires only once per second, but in a cone burst of eight bullets all at once; (4) a Sniper rifle with very slow shooting speed, but with a large bullet that passes through damaging all zombies in its path. The pistol is given to the player as their starting weapon, and the remainder are scattered across the level, positioned in parallel paths generated by the Hilbert Curve for the player to find. Health and ammunition replenishing pickups were also spread across the level to aid the player in finding the helicopter, and to serve as reward for exploring. Once a player obtained a new weapon, then their old weapon would be lost. Whenever they would reach a new level, their character would be reset, being given full health and regaining their pistol.

Game elements were placed over generated maps as thought to be needed. Unfortunately, to have the generator automatically place the game elements is feasible and desirable, but not within the time and scope of this work. This however is but a minor setback, for the paths generated by the Hilbert Curves automatically generate points at which we may place game elements. The player and the Helicopter were placed at the points furthest from

each other, and zombies, weapons, and other pickups being placed at intermediary or parallel points.

6.3 Testing Protocol

Between the 6th and 17th of June 2018, a meeting was scheduled for each volunteer at a time and location of their preference, which for most game developers it was on their own studios during work hours. Once that volunteer had confirmed themselves to be within the requirements previously stated. The only specification for a desirable location was that it should be reasonably silent, as to allow the interview to be recorded without excessive background noise that would make its transcription difficult. After introductions took place, and the volunteer appeared to be comfortably set to a position where they could properly play the game on a portable computer brought in by the interviewer, they were orally given the following introduction:

1. *'You will play a total of three levels of a simple game where you play as a survivor on a zombie apocalypse, and must reach a helicopter to escape.'*

2. *'You control your character through a top-down perspective. You move around with the W,A,S,D keys, and aim your gun with your mouse, firing by pressing the left mouse button.'*

3. *'There are items such as different weapons and supplies scattered around the level that may help you reach the goal. If you grab a weapon, you lose your previous one. If you obtain a weapon, you still start the following stage with your starter weapon. Your starter weapon has unlimited ammunition.'*

4. *'Should you feel like skipping the level, or if you feel like you can't beat the level, you may skip the current level by pressing the 'P' keyboard key. There's absolutely no problem in skipping the level if you feel like it.'*

5. *'Should you feel like restarting the current level, you may do so by pressing the 'O' keyboard key.'*

6. *'During your gameplay session, I may ask you questions regarding your play experience. If you are comfortable in talking while you play, feel free to communicate your opinion on the game, or any topic in general.'*

Once all doubts regarding the instructions were settled, players were also given a headset, keeping one ear covered to hear the game's audio, and another one uncovered to hear the interviewer.

6.3.1 Observation

While players interacted with the game, the interviewer filed an observational document with information regarding that game session. This document's topics were mostly open, registering player's statements about their session, or reminders of noteworthy actions taken by the player, as well as behaviors that deviated from the norm. In addition to these open notes, the Observation documented presented four log data topics, referring to the game session's information:

- **Completion Time**: Time taken by the player to complete each level.
- **Retries**: Number of additional attempts made by pressing the 'O' key on each level.
- **Given Up**: For each level, a level is marked if the player had given up playing the level.
- **Deaths**: Number of deaths by the player on each level.

Also, following these numeric and check-mark log data topics, we defined another four perceptive data topics that, while had no absolute answers and were graded based on the interviewer's opinion, would be later useful for understanding the player's point of view, or to identify opinions that contradicted the registered data:

- **Difficulty**: A discrete grade system from 1 to 5. This topic defined a grade for the perceived difficulty the player had with the game. A player that had at least one death was scored with a 3, while players with more deaths or no deaths at all were split onto each respective side of the axis.
- **Engagement**: The most self explanatory topic that, after all interviews, had proven to be not very effective due to how open to interpretation it was. Players had very distinct ways of representing engagement, and other than in cases where a player would openly complain or make excited remarks, this topic has not been as helpful as we had planned.
- **Exploration**: A discrete grade system from 1 to 5. If the player went out of their way to look for weapons or supplies during all 3 levels at least once, that player was scored

with a 3, while players that went out of their way to explore additional areas when possible, or players that did not look around unless it was necessary to find the goal were split onto each respective side of the axis.

- **Got Lost**: A check-mark for each level registering if the player had wandered off for roughly 15 seconds without knowing their way back to the path they were before.

Each notable observation regarding any of the topics above was then brought up during the interview whenever the interviewer thought it to be appropriate.

6.3.2 Interview

Questions for our interview process that was performed after each volunteer's play session are laid out in Appendix C. Each interview had its audio recorded and then transcribed. When elaborating these questions, we have chosen to highlight eight question topics representing criteria that help define the quality of our generator, and the goals we aimed to achieve with it. All questions we have had to design for our interview adhered to at least one of these topics, in a way that an answer to a question addresses its associated ones, or could possibly address others, depending on the volunteer's answer. These topics are as defined below:

- **Repetitive**: Do two or more mathematically distinct constructs seem to be too much alike?
- **Consistent**: Interesting results appear more often than not. And it follows the intended rules more often than it doesn't.
- **Diverse**: Is the measure of repetitiveness of the procedurally generated content, low enough to justify its implementation.
- **Cohesive**: Do the generated elements make sense with one another? Are there things that are intuitively not supposed to happen?
- **Perceptible**: Are the Quasi-random variations of the procedural system perceptible to the end-user?
- **Scalable**: Able to be improved? Is it based upon a strong foundation? Does it allow room for content, diversity, and exploration?

Additionally, there are two extra topics about the bias derived from the player's impression of the game, and another addressing the technical opinion of game designers on the generative process, Starting with a brief discussion where the interviewer reveals that the purpose of this experiment was to evaluate a Procedural Generator. The former is tackled within the first two, and last, questions of the interview, as shown in Appendix C. By having

the first minutes of the interview be dedicated to commenting about the game, the remainder could be more focused towards questions regarding the maps themselves. The last question within the first group is meant to drag the designer player's final thoughts on the game before the questions dedicated to their group, as well as the non-designer player's insights on what we've could have done better.

6.3.3 Pilot Tests

Before any testing could be conducted and additional volunteers selected, 2 tests of the interview protocol described above were performed: one for group A's questions, and one for both group A and B's questions. During these pilot tests, the recorded total time for play sessions had been stored, and then used to inform volunteers for the actual tests about how much time their play session would likely take. These pilot interviews assisted us in many aspects of the Qualitative Evaluation, especially in correcting minor issues with our Interview Protocol.

6.4 Results

This section encompasses all that was collected from our Qualitative Analysis, as well as our impressions on our successes and failures into new categories that were derived from the analysis. The data collected, as well as select opinions for each volunteer, are all annexed sequentially within Attachment D. Our interpretation of this collection will be dissected by the following sub-sections, with subtables from this data being split as relevant, and individual opinions being diluted onto our description of each topic, not necessarily being directly attributed to their respective volunteer.

6.4.1 Engagement

As early as the pilot tests, our interview protocol had managed to hit our 6 target topics. Game sessions took 4 to 13 minutes, and the interviews took 12 to roughly 18 minutes. Within the first 2 to 7 minutes of interview, the volunteers' answers began gravitating toward the game levels' maps. The major factor that determined this 5 minute difference was the volunteer's criticism regarding the game, that, contrary to our expectations, was not as exclusive to group B as we thought it would be.

Players from both groups reported the game to have too few features, with many players calling it a prototype. Regardless, once most of the volunteer's opinions about the game were expressed, there was rarely a case where they would return to address the same views

about the game. One topic that has been frequent enough to be worth discussion was that, due to the game's balancing, a considerable number of player's thought that there was no reason to explore the maps. Mostly due to there being no collectibles worth chasing. Also, many have stated that their starting weapon, which was designed to be enough to allow the player to engage the level, was considered to be the best of the 4 designed weapons.

Even with these critical opinions, most players considered the experience to be fun. Which points towards our first favorable point regarding the generator, since the experience of navigating through the map was not deemed to be frustrating. Given that most players had negative impressions of the game's completeness, and that all time spent playing was also spent in roaming around the environment. Tables 6.2 and 6.3 show our results regarding the entertainment factor collected from our interviews.

Volunteer\Topic	VA1	VA2	VA3	VA4	VA5	VA6	VB1	VB2	VB3	VB4	VB5	VB6
Had Fun	1	1	1	1	1	1	0	1	1	0	1	1
Stated that there was no reason to explore	1	1	1	1	0	0	1	0	0	0	0	1

Table 6.2. Data from the volunteer's answers regarding topics about their entertainment. Both are organized as 0's (no) and 1's (yes).

Summation\Topic	Sum (A)	Sum (B)	Total	Averages (A)	Averages (B)	**Total**
Had Fun	6	4	**10**	1.000	0.666	**0.833**
Stated that there was no reason to explore	4	2	**6**	0.666	0.333	**0.500**

Table 6.3. Data from the volunteers' answers, displaying averages and totals (when relevant) for each closed topic. Separated column entries for groups A, B contain the sums and averages for each entertainment related topic in Table 6.2.

One of our initial goals was to obtain a helpful answer as to the 'repetitiveness' factor of our generated content. Though, as even pointed out by a volunteer, the feeling of repetitiveness may only kick in after hours of play, and could be likely to occur considering the amount of content in the game. Our testing is far from being able to pinpoint whether we descend into this pitfall. Nevertheless, as a bare-bones game experience, there would still be much to be done to have the generated product be refined and presentable as a commercial game level, and our interview analysis shows this to be the case. This predicament would also come into play regarding the volunteers' opinions on the maps' (and by extension, the generator's) aesthetic.

6.4.2 Aesthetic

Our layers consisted of 1 to 4 distinct 'subtypes' that are defined by the planner. Each being defined to appear by the planner based on other established parts of the plan. For example, our the Tree Generating Layer included four possible types of trees: cacti, snowy trees, dead trees; and green trees. This, and the small variety of aesthetic, as well as all possible layers appearing in almost all maps, have shown not to be enough to make the maps visually pleasing, as noted by our results within Tables 6.4 and 6.5.

Volunteer\Topic	VA1	VA2	VA3	VA4	VA5	VA6	VB1	VB2	VB3	VB4	VB5	VB6
Found maps visually engaging	1	0	1	1	1	0	0	0	0	0	1	1
Number of remembered levels	2	3	3	3	3	3	3	2	1	2	1	2

Table 6.4. Data from the volunteers' answers about topics regarding their opinions on the maps' aesthetics. **Found maps visually engaging** is organized as 0's (no) and 1's (yes). **Number of Remembered levels** is a question that emerged from the original protocol. It accounts the number of levels the volunteer could remember and accurately describe.

Summation\Topic	Sum (A)	Sum (B)	Total	Averages (A)	Averages (B)	**Total**
Found maps visually engaging	4	2	**6**	0.666	0.333	**0.500**
Number of remembered levels	-	-	**-**	2,833	1,833	**2,333**

Table 6.5. Data from the volunteers' answers, displaying averages and totals (when relevant) for each closed topic's answers. Separated column entries for groups A, and B contain the sums and averages for each aesthetic related topic in Table 6.4.

As shown, a little under half the volunteers did not think the maps to be visually interesting. The most common opinions categorized the maps' appearance as being 'too basic', 'too few distinct game elements', and 'having no uniqueness', with some volunteers even being able do discern all visual element types that were present. These opinions were prevalent in Group B, but one interesting observation is that two of the volunteers from group A, that pointed the maps not to be aesthetically pleasing are graduated graphic designers.

This seems to show is that, from the critical point of view of those frequently engaged with graphic or game designing, the lack of refinement from our generator is glaring and evident. Whilst the concepts for improving our design are there, our lack of focus on refining our graphical elements, as well as the lack of graphic designers allocated for development on our part has been spotted and called out. Another consequence of this, is that our maps have not shown to be very memorable, as also shown in our collected opinions describing the lack of unique elements.

On average, players could remember only 2 of the three maps each of them played, with volunteers from Group A remembering on average 2,8 maps, and volunteers from Group B remembering 1,8 maps. As an additional note, the two graphic designers from Group A were able to remember all three maps. These overall results could have suggested that the Procedural Generator itself could have needed changes, but analyzing the improvements suggested by Group B on their dedicated questions shows that this is likely to be mostly a consequence of our lack of designers and the lack of unique elements, and not an inherent flaw to the methodology of our generator.

6.4.3 Layout

Results on questions regarding the player's impressions on our maps' layouts suggest that we must focus on further investment regarding the generation of natural hints meant to guide the player toward the goal. As it is, the only elements guiding the player are the spacing filling curve, and the 'Road' layer that is placed and iterated over it. This is shown in Tables 6.6 and 6.7, with just over half our players noticing these hints.

Volunteer\Topic	VA1	VA2	VA3	VA4	VA5	VA6	VB1	VB2	VB3	VB4	VB5	VB6
Noticed guiding tips	1	0	0	1	1	0	0	1	0	1	1	0
Intuitive layout	1	0	0	1	1	0	0	1	0	1	1	0
Room to Explore	3	4	4	3	3	4	3	3	3	2	4	4
Understood the map	4	4	3	2	4	3	5	4	5	3	4	5
Got lost	1	1	2	1	0	1	1	0	1	0	1	2

Table 6.6. Data from the volunteers' answers about topics regarding their understanding of the maps' structure. Most information is organized as 0's (no) and 1's (yes). Two exception topics have distinct values: **Room to Explore** is an integer from 1 to 5: A value of 1 means that there was no freedom to explore, a 5 means that the amount of explorable areas was overwhelming, and a 3 being the perfect balance; **Understood the Map** is a simple grade from 1 to 5 of how well the map was understood.

Summation\Topic	Sum (A)	Sum (B)	Total	Averages (A)	Averages (B)	**Total**
Noticed guiding tips	3	3	**8**	0.500	0.500	**0.500**
Intuitive layout	3	3	**7**	0.500	0.500	**0.500**
Room to Explore	-	-	**-**	3.500	3.166	**3.333**
Understood the map	-	-	**-**	3.333	4.333	**3.833**

Table 6.7. Data from the volunteers' answers, displaying averages and totals (when relevant) for each closed topic. Separated column entries for groups A, and B contain the sums and averages for each layout related topic in Table 6.6.

Players understood well which areas were designed to be inaccessible, with any doubts or unclarified perceptions of accessible areas being almost all situations solved by trial and

error within the first map, or less commonly on the second. Although even with this comprehension the map's structure, save three of all volunteers, players have stated to have gotten lost somewhere during play. This lack of direction is more associated by the players as there being no direct hint as to where the goal was, than it being a bigger problem with the map's design. From our observation, players were less likely to be lost on more confined levels, with large lakes, trees, and/or roads than with levels with open areas. This we believe to be due to confined areas allowing for less room for the players to get lost.

Another interesting observation is that most of the players that have stated to have gotten lost have also stated that they did not feel as if being lost was a problem, and that they were merely attempting to find the path by moving around. This, in retrospect, is more of a flaw in our game's direction than with the generator. It was considered at one point of development to add coins (or any other form of collectible) to point the player straight to the goal or to areas they would find loot, as is the case with many games. And perhaps it could be better to stick to this concept on future works.

Our methodology of using the space filling curve to generate a main and parallel paths has been well evaluated by our interview's answers. On a 1 to 5 scale about how much room the player has found to explore on the maps. As it was explained to the players, 1 meant that there was no room to explore, being a completely linear path, and 5 meant that there were too many explorable areas, hampering the player's progression, we have attained an average answer of 3 in both groups. The highest answer has been a 4, the lowest has been a 2, and just under half of the answers being 3's. This could mean that we have struck a good balance of the number of divergent, optional paths.

6.4.4 PCG

Covered mostly by our second batch of questions, directed exclusively at Group B, was the designers' opinions on the generative process. As a starting point, the number of volunteers that spontaneously had the impression that the maps were procedural was very considerable, even if it was the minority of players. A total of 6 players, 3 from Group A, and 3 from Group B have noticed that they were playing a procedural map, as shown in Tables 6.8 and 6.8. They have communicated this perception either through openly asking if there were algorithms involved, or by a post interview discussion (for group A), or when the interviewer revealed that the experiment was focused on the procedural maps (for group B).

The most critical factors that gave away the procedural generation were: (1) The occasional glitches that in rare cases had gone unnoticed by us had elements be placed out of cohesion. Even a single appearance of these glitches can reveal at once that there is an algorithm, rather than a human, involved in creating the maps; (2) The number of elements

Volunteer \Topic	VA1	VA2	VA3	VA4	VA5	VA6	VB1	VB2	VB3	VB4	VB5	VB6
Found the Procedural maps interesting	-	-	-	-	-	-	1	1	1	0	1	1
Closest associated PCG method	-	-	-	-	-	-	Connect begin and end, Dungeon Generator	Spelunky	Spelunky	Generate separate areas that limit movement	Couldn't think of a method	Generate outline and fill around
Perceived procedural disadvantages	-	-	-	-	-	-	No pacing	None	Uncoherent elements are aesthetically unpleasant	It lacked memorable segments	Confusing and frustrating level design	It requires refinement but it doesn't present major problems
Suggested improvements	-	-	-	-	-	-	Multiple types of encounters and optional rewards. Handmade encounters	More visual elements	Work in the art	Use proper algorithms for individual elements	Add more interest points	More visual elements
Would develop a game with this generator	-	-	-	-	-	-	0	1	1	1	1	1
Noticed it was about PCG	1	1	0	0	0	0	1	0	0	1	1	0
Asked if it was about PCG during play	0	1	0	0	0	0	0	1	0	0	1	0

Table 6.8. Data from the volunteers' answers regarding the implementation of PCG on the generated maps. **Found the maps visually interesting**, **Would develop a game with this generator**, **Noticed it was about PCG**, and **Asked if it was about PCG during play** are answered as 0's (no) and 1's (yes). The remainder topics were open answers that were condensed into one or two sentences for convenience

that did not seem to have a purpose, and thus generated several cases of negative possibility space (explorable areas within reach, but with no reward); (3) The labyrinthine design of the maps themselves, which is a common trope of Procedural Maps.

Not all players that had noticed the PCG have been vocal about it during either the interview or their play-session. Those who did have openly asked if the the maps they were playing were procedural, 'automatic', or 'randomly' generated. There were, however, two interesting cases in Group B worth discussing on their own: The first being where one designer player had been convinced that there was no procedural generation involved due to our decision of having the maps not being different at each retry. Which is another common trope of procedural games that have opted for a roguelike style, where death meant the game would be completely reset, including the levels' generation. The second was more of a logistic deduction where the designer player asked if the maps were Procedurally Generated, but did not have the impression that the maps were generated themselves. When asked about it, the volunteer stated that considering each level's size and number of game elements, it

Summation \Topic	Sum (A)	Sum (B)	Total	Averages (A)	Averages (B)	**Total**
Found the Procedural maps interesting	-	5	**6**	-	0,833	**0,833**
Would develop a game with this generator	-	5	**6**	-	0,833	**0,833**
Noticed it was about PCG	2	3	**6**	0.333	0.500	**0.416**
Asked if it was about PCG during play	1	2	**3**	0.166	0.333	**0.250**

Table 6.9. Data from the volunteers' questions, displaying averages and totals (when relevant) for each closed topic. Separated column entries for groups A, and B contain the sums and averages for each PCG related topic in Table 6.8.

would make more sense to have it generated though algorithms, rather than by hand.

When asked about their opinions on how the designers thought that the maps were generated, most of them have matched their descriptions with methods known and discussed within Chapter 2. Out of the seven answers, one was indecisive. As expected, mostry answers have associated the map generator to another known methods: two of them have associated it with the level generation of *Spelunky*, another two have associated it with the Dungeon Generator of Shaker et al. [2016a], and the last 2 have come closer to identifying our process by elaborating on the basic idea of revolving the generation process around a main path that has content created around it.

Before, many of the players have stated to have noticed the bare bones nature of our generator's assets and game, as well as the lack of engagement with its visual. The designers' opinions regarding improvements reflect on this. From all 7 designers, 4 have focused on the notion that the generator requires investment on the art assets used, and the 3 remaining designers have stated that it needs more unique elements to make maps more memorable.

For a closing discussion point to our results section, as it was for the designer's interviews, we have inquired the volunteers whether they would be interested in designing games using the generator described to them, to which the response was overwhelmingly positive. Of all the designers, only one has stated that they would not design a game with this generator, they have done so under the argument that, even with improvements regarding the art, design, and style, they would not be interested in playing a game that used a generator such as this. In this particular case, this designer's overall opinion had been notably negative. And while it was initially positive about the development of a game with our generator, our observational data suggested the interviewer to further inquire about this topic during the interview, without attempting to steer the volunteer's answer either way. After pondering about it longer, that designer had chosen to switch their answer to 'no'.

6.5 Final Observations

Though our Qualitative Analysis had suffered due to factors beyond our designs for our Integrated Procedural Generator, such as the game's simplicity, poor game design decisions, and the lack of dedicated art design, our protocol and analysis conduction has yielded very interesting results. These are be both valuable in validating the efficiency of our generator, as well as heading us toward the most important areas in need improvement. To which, we now shall now discuss as we present our final thoughts and conclusions.

Chapter 7

Conclusion

We have now come to a close on this version of our Integrated Procedural Map Generator, and we are pleased with the results and many achievements conquered over the course of its development that meaningfully evolve the discussion of procedural levels on PCG-G. On the topic of AI Planning, we have provided scalable means by which to describe a theme, and presented the results of a prototype of a toolkit for designing Planing problems and domains. With polish and a proper interface, our methodology could be published for use in commercial games. On the field of Multi-layered CA-Map Generation with Space-filling curves, we have proposed an alternative that improves on the standard CA-Map Generation methods that offers better control.

Overall, combining Planning, Cellular Automata, and Space Filling curves, we have presented an alternative for expanding upon the concept of generating procedural maps, and thus evolving the discussion pertaining the use of each of these methods for that purpose. Prior to the completion of this work, part of our contribution had been already been acknowledged by the scientific community, as some of it was presented in Macedo and Chaimowicz [2017]. The methodology we have presented has yielded a strong and complete foundation for generating themed maps that allows great room for accessible expansion and improvement, as shown by our Qualitative and Quantitative analyses.

Still our work is one that requires further revisions before it is able to be used on commercial grade games. In the topic of improving structure, cohesion, and layout, it will require us to return to the basics of our path generation: To work on the organization of the Multi-layered Cellular Automata. Improving on its layers, and how they are laid out to guide the player toward their goal while offering alternative and optional paths. In part, this could be first investigated by experimenting different space filling curves, and also by designing layers crafted to iterate in parallel with others, rather than sequentially.

To correct specific mistakes and unfortunate flaws common to Procedural Map Gener-

ators. The best alternative should be not to nag on small imperfections of unpredictable systems, but to have additional overhead monitoring algorithms and/or post processing methods to identify and correct purposeless and/or empty areas that both make it noticeable that they are based of Procedural Generation, and that create negative possibility space. Despite the effort required to oversee the many necessary optimizations our work needs, we believe that our focus, as we work on the growth of this generator, perhaps should be directed less on the basis of our methodology, and more on refining the resources created with it. The measure of its potential require examples of its use, both for allowing us to discover its limitations, and for others, who will experiment and find features and flaws we have not.

As commercial games reliant on PCG-G spread across the spectrum of success and failure have shown, the safest and most effective approach to computerize the birth of interesting interactive experiences is to mix it with the occurrence of well crafted hand-made content, be it sporadic or integral. As such, one less academic path that is directed towards the growth of our generator as a product, is the creation of more memorable set-pieces to be placed by automata layers, as well as refining the planning domain library that allows them to be placed on the map in a cohesive way. Perhaps through chunks of interesting handmade content with small chances to be added to each map, or with fewer general purpose automata, and more with unique rule-sets.

One investment that would aid us greatly in having more asset resources available for our generator, both with the intent of testing, and for the intent of commercial usage, is the development of dedicated modular art assets designed for fitting with the generation of our procedural maps. Additional resources open doors to for allowing different ideas, to experiment with more rule-sets of automata, and for presenting a better image of what our generator can do. Thus, expanding on the art expands our domain of possible map themes.

Finally, testing all further change will require further qualitative play-testing. As our Qualitative Analysis has shown, the hypothetical filter that is the reason and motivation to travel through the environment is bound to heavily impact player opinions. Thus, it should be polished. It cannot be impactful enough to detract the user's attention from the maps, but it should not be so simple that it bothers the player. This will require the next iteration of qualitative testing to use a better developed game with proper time and resources, which should also allow us the opportunity to have the selection and positioning of game elements also be procedural, as the state-of-the-art AI directors have set a precedent for.

In conclusion, our Integrated Planning-Automata Procedural Map Generator has areas to improve, but overall has yielded promising results. Which, coupled with its solid theoretical foundation, and potential for expansion, should allow it to be refined past a point to be applied to commercial Digital Games. PCG-G requires a heavy load of effort equal to, or even surpassing that of producing original content. Designing for scale is the ideal motivation

for investing in procedural systems, and we have felt this weight in our own development. Creating a solid basis, being aware and proactive in solving the system's practical faults, and adding the spice of hand-crafted design seems to be the most promising approach to procedural content. Whilst we have not obtained a concrete answer as to how the sum of our methodology fares in avoiding the challenges and pitfalls of procedural content generation, our experiments assure us that we are on the right path. And thus it is with confidence that we explore new horizons, within related future work.

Bibliography

(2017). "opengameart". www.opengameart.org/. Accessed: 2017-08-7.

Adams, D. et al. (2002). Automatic generation of dungeons for computer games.

Aikman, Z. (2014). "unite 2014 - generating procedural dungeons in galak z". https://www.youtube.com/watch?v=ySTpjT6JYFU. Accessed: 2017-08-7.

Andrejkovics, Z. (2016). The invisible game. In *The Invisible Game: Mindset of a Winning Team*. CreateSpace Independent Publishing Platform.

Balzter, H., Braun, P. W., and Köhler, W. (1998). Cellular automata models for vegetation dynamics. *Ecological modelling*, 107(2):113--125.

Benson, J. (2013). "world of warcraft team: Procedural content is totally something we've talked about". www.pcgamesn.com/wow/world-warcraft-team-procedural-content-totally-something-we-ve-talked-about.

Biggs, M., Fischer, U., and Nitsche, M. (2008a). Supporting wayfinding through patterns within procedurally generated virtual environments. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 123--128. ACM.

Biggs, M., Fischer, U., and Nitsche, M. (2008b). Supporting wayfinding through patterns within procedurally generated virtual environments. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 123--128. ACM.

Bonomi, A. (2009). Dissipative multilayered cellular automata facing adaptive lighting.

Browne, K. and Anand, C. (2012). An empirical evaluation of user interfaces for a mobile video game. *Entertainment Computing*, 3(1):1--10.

Champandard, A. J. (2013). Planning in games: An overview and lessons learned.

Cheong, Y.-G., Riedl, M. O., Bae, B.-C., and Nelson, M. J. (2016). Planning with applications to quests and story. In *Procedural Content Generation in Games*, pages 123--141. Springer.

Ciarlini, A. E., Pozzer, C. T., Furtado, A. L., and Feijó, B. (2005). A logic-based tool for interactive generation and dramatization of stories. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 133--140. ACM.

Consalvo, M. and Dutton, N. (2006). Game analysis: Developing a methodological toolkit for the qualitative study of games. *Game Studies*, 6(1):1--17.

Conway, J. (1970). The game of life. *Scientific American*, 223(4):4.

Dekker, A. and Champion, E. (2007). Please biofeed the zombies: Enhancing the gameplay and display of a horror game using biofeedback. In *DiGRA Conference*.

Drachen, A., Canossa, A., and Yannakakis, G. N. (2009). Player modeling using self-organization in tomb raider: Underworld. In *2009 IEEE symposium on computational intelligence and games*, pages 1--8. IEEE.

Ebert, D. S. (2003). *Texturing & modeling: a procedural approach.* Morgan Kaufmann.

Erol, K. (1996). *Hierarchical task network planning: formalization, analysis, and implementation*. PhD thesis.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189--208.

Garner, T., Grimshaw, M., and Nabi, D. A. (2010). A preliminary experiment to assess the fear value of preselected sound parameters in a survival horror game. In *Proceedings of the 5th Audio Mostly Conference: A Conference on Interaction with Sound*, page 10. ACM.

Gelfond, M. and Lifschitz, V. (1993). Representing action and change by logic programs. *The Journal of Logic Programming*, 17(2):301--321.

Geurts, L., Vanden Abeele, V., Husson, J., Windey, F., Van Overveldt, M., Annema, J.-H., and Desmet, S. (2011). Digital games for physical therapy: fulfilling the need for calibration and adaptation. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, pages 117--124. ACM.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: theory and practice*. Elsevier.

Grassberger, P. (1986). Long-range effects in an elementary cellular automaton. *Journal of Statistical Physics*, 45(1):27--39.

Guckelsberger, C., Salge, C., and Colton, S. (2016). Intrinsically motivated general companion npcs via coupled empowerment maximisation. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1--8. IEEE.

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

Hash, C. and Isbister, K. (2011). Reactive animation and gameplay experience. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, pages 328--330. ACM.

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

Hogeweg, P. (1988). Cellular automata as a paradigm for ecological modeling. *Applied mathematics and computation*, 27(1):81--100.

Johnson, L., Yannakakis, G. N., and Togelius, J. (2010). Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 10. ACM.

Kamel, I. and Faloutsos, C. (1993a). Hilbert r-tree: An improved r-tree using fractals. Technical report.

Kamel, I. and Faloutsos, C. (1993b). On packing r-trees. In *Proceedings of the second international conference on Information and knowledge management*, pages 490--499. ACM.

Kazmi, S. and Palmer, I. J. (2010). Action recognition for support of adaptive gameplay: A case study of a first person shooter. *International Journal of Computer Games Technology*, 2010:1.

Lague, S. (2015). "[unity] procedural cave generation (tutorials)". https://www.youtube.com/watch?v=AsR0-wCTJl8. Accessed: 2017-08-7.

Lawrence, D. (1976). Telengrad. www.aquest.com/telen.htm.

Lee, J. (2014, howpublished = www.makeuseof.com/tag/procedural-generation-took-gaming-industry/, note = Accessed: 2017-08-7). "how procedural generation took over the gaming industry".

Li, W. and Packard, N. (1990). The structure of the elementary cellular automata rule space. *Complex Systems*, 4(3):281--297.

Liang, H. and Wang, Z. (2017). Optimized distribution of beijing population based on camas. *Discrete Dynamics in Nature and Society*, 2017.

Macedo, Y. P. A. and Chaimowicz, L. (2017). Improving procedural 2d map generation based on multi-layered cellular automata and hilbert curves. In *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 116--125. IEEE.

Magerko, B., Laird, J., Assanie, M., Kerfoot, A., and Stokes, D. (2004). Ai characters and directors for interactive computer games. *Ann Arbor*, 1001(48):109--2110.

Matthews, E. A. and Malloy, B. A. (2011). Procedural generation of story-driven maps. In *Computer Games (CGAMES), 2011 16th International Conference on*, pages 107--112. IEEE.

Mishra, J. and Mishra, S. (2007). *L-system Fractals*, volume 209. Elsevier.

Moon, B., Jagadish, H. V., Faloutsos, C., and Saltz, J. H. (2001). Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on knowledge and data engineering*, 13(1):124--141.

Nakayama, A., Yamamoto, T., Morita, Y., and Nakamachi, E. (2015). Development of multi-layered cellular automata model to predict nerve axonal extension process. In *VI International Conference on Computational Bioengineering*.

Olsen, J. (2004). Realtime procedural terrain generation.

Panksepp, J. (2004). *Affective neuroscience: The foundations of human and animal emotions*. Oxford university press.

Raffe, W. L., Zambetta, F., and Li, X. (2011). Evolving patch-based terrains for use in video games. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 363--370. ACM.

Riedl, M., Thue, D., and Bulitko, V. (2011). Game ai as storytelling. In *Artificial Intelligence for Computer Games*, pages 125--150. Springer.

Riedl, M. O. and Young, R. M. (2010). Narrative planning: balancing plot and character. *Journal of Artificial Intelligence Research*, 39(1):217--268.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Sagan, H. (2012). *Space-filling curves*. Springer Science & Business Media.

Schonfisch, B. (1999). Synchronous and asynchronous updating in cellular automata.

Scirea, M., Cheong, Y.-G., Nelson, M. J., and Bae, B.-C. (2014). Evaluating musical foreshadowing of videogame narrative experiences. In *Proceedings of the 9th Audio Mostly: A Conference on Interaction With Sound*, page 8. ACM.

Shaker, N., Liapis, A., Togelius, J., Lopes, R., and Bidarra, R. (2016a). Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games*, pages 31--55. Springer.

Shaker, N., Togelius, J., and Nelson, M. J. (2016b). *Procedural Content Generation in Games*. Springer.

Takatsuki (2007). Cost headache for game developers. www.news.bbc.co.uk/1/hi/business/7151961.stm.

Thue, D., Bulitko, V., Spetch, M., and Wasylishen, E. (2007). Interactive storytelling: A player modelling approach. In *AIIDE*, pages 43--48.

Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M., and Stanley, K. O. (2013). Procedural content generation: Goals, challenges and actionable steps. *Dagstuhl Follow-Ups*, 6.

Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172--186.

Tomai, E. (2012). Towards adaptive quest narrative in shared, persistent virtual worlds. *Eighth Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2012)*, pages 51--56.

Valls-Vargas, J., Ontanón, S., and Zhu, J. (2013). Towards story-based content generation: From plot-points to maps. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1--8. IEEE.

Vannaprathip, N., Haddawy, P., Suebnukarn, S., Sangsartra, P., Sasikhant, N., and Sangutai, S. (2016). Desitra: a simulator for teaching situated decision making in dental surgery. In *Proceedings of the 21st International Conference on Intelligent User Interfaces*, pages 397--401. ACM.

Attachment A

Map Generation Planning Library

Starting State

<<start>>
None

Goal

<<goal>>
[_Cliff(Layer/Cliffs) {AND} _Ground(Layer/Ground)] {AND} [_Tree(Layer/Trees) {AND} _Water(Layer/Water)]

Actors

<<type>>

>MapSize
MapSizeSmall
MapSizeMedium
MapSizeLarge

>Temperature
TemperatureScalding
TemperatureHot
TemperatureTemperate
TemperatureCold
TemperatureFreezing

>Topography
TopographyUp
TopographyDoubleUp
TopographyMixed
TopographyDown
TopographyDoubleDown

>Hidrography
HidrographyIsland
HidrographyCoastal
HidrographyRivers
HidrographyPonds
HidrographyDry

>Vegetation
VegetationDense
VegetationFocused
VegetationSparse
VegetationSporadic
VegetationNone

>Layer/Cliffs
CliffSand
CliffEarth
CliffCold

>Layer/Ground
GroundSand
GroundDirt
GroundGrass
GroundSnow

>Layer/Trees
TreeCactus
TreeRegular
TreeLeafless
TreeSnowy

>Layer/Water
WaterIsland
WaterCoastal
WaterRivers
WaterPonds
WaterDry

States

<<state>> _Temperature(Temperature) _Topography(Topography) _TopographyChoice(Topography) _Hidrography(Hidrography) _Vegetation(Vegetation) _MapSize(MapSize)	 _Stage1Complete() _Cliff(Layer/Cliffs) _Ground(Layer/Ground) _Tree(Layer/Trees) _Water(Layer/Water)

Actions

<<action>>

Action = Set_Grassland()
Constraints =
Precondition = [[{UNK}_Temperature(Temperature)] {AND} [{UNK}_Topography(Topography)]] {AND} [[{UNK}_Hidrography(Hidrography)] {AND} [{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureHot)
_Temperature(TemperatureTemperate)
_Temperature(TemperatureCold)
_Topography(TopographyUp)
_Topography(TopographyMixed)
_Topography(TopographyDown)
_Vegetation(VegetationSparse)
_Vegetation(VegetationSporadic)
_Vegetation(VegetationNone)
_Hidrography(HidrographyRivers)
_Hidrography(HidrographyPonds)
_Hidrography(HidrographyDry)
<ACT_END>

Action = Set_Rainforest()
Constraints =
Precondition = [[{UNK}_Temperature(Temperature)] {AND} [{UNK}_Topography(Topography)]] {AND} [[{UNK}_Hidrography(Hidrography)] {AND} [{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureHot)
_Temperature(TemperatureTemperate)
_Topography(TopographyUp)
_Topography(TopographyDown)
_Vegetation(VegetationDense)
_Vegetation(VegetationFocused)
_Hidrography(HidrographyCoastal)
_Hidrography(HidrographyRivers)
<ACT_END>

Action = Set_Tundra()

```
            Constraints  =
            Precondition = [[{UNK}_Temperature(Temperature)]] {AND} [[{UNK}_Hidrography(Hidrography)] {AND}
[{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureFreezing)
                         _Temperature(TemperatureCold)
                         _Hidrography(HidrographyRivers)
                         _Hidrography(HidrographyPonds)
                         _Hidrography(HidrographyDry)
                         _Vegetation(VegetationSparse)
                         _Vegetation(VegetationSporadic)
                         _Vegetation(VegetationNone)
<ACT_END>

Action = Set_Chaparral/Savannah()
            Constraints  =
            Precondition = [{UNK}_Temperature(Temperature)] {AND} [[{UNK}_Hidrography(Hidrography)] {AND}
[{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureHot)
                         _Temperature(TemperatureTemperate)
                         _Hidrography(HidrographyPonds)
                         _Hidrography(HidrographyDry)
                         _Vegetation(VegetationSparse)
                         _Vegetation(VegetationSporadic)
<ACT_END>

Action = Set_HotDesertDesert()
            Constraints  =
            Precondition = [{UNK}_Temperature(Temperature)] {AND} [[{UNK}_Hidrography(Hidrography)] {AND}
[{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureScalding)
                         _Temperature(TemperatureHot)
                         _Hidrography(HidrographyCoastal)
                         _Hidrography(HidrographyPonds)
                         _Hidrography(HidrographyDry)
                         _Vegetation(VegetationSparse)
                         _Vegetation(VegetationSporadic)
                         _Vegetation(VegetationNone)
<ACT_END>

Action = Set_Polar()
            Constraints  =
            Precondition = [{UNK}_Temperature(Temperature)] {AND} [[{UNK}_Hidrography(Hidrography)] {AND}
[{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureFreezing)
                         _Hidrography(HidrographyIsland)
                         _Hidrography(HidrographyCoastal)
                         _Vegetation(VegetationNone)
<ACT_END>

Action = Set_Taiga()
```

```
            Constraints  =
            Precondition = [[{UNK}_Temperature(Temperature)] {AND} [{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureCold)
            _Vegetation(VegetationDense)
            _Vegetation(VegetationFocused)
<ACT_END>

Action = Set_Forest()
            Constraints  =
            Precondition = [[{UNK}_Temperature(Temperature)] {AND} [{UNK}_Vegetation(Vegetation)]]
Effect = _Temperature(TemperatureHot)
            _Temperature(TemperatureTemperate)
            _Temperature(TemperatureCold)
            _Vegetation(VegetationDense)
<ACT_END>

Action = Set_Mountain()
            Constraints  =
            Precondition = {UNK}_Topography(Topography)
Effect = _Topography(TopographyDoubleUp)
<ACT_END>

Action = Set_Canyon()
            Constraints  =
            Precondition = {UNK}_Topography(Topography)
Effect = _Topography(TopographyDoubleDown)
<ACT_END>

Action = Set_Island()
            Constraints  =
            Precondition = [{UNK}_Hidrography(Hidrography)] {AND} [[{UNK}_Topography(Topography)] {OR}
[_Topography(TopographyUp)]]
Effect = _Hidrography(HidrographyIsland)
            _Topography(TopographyUp)
<ACT_END>

Action = Set_Coast()
            Constraints  =
            Precondition = {UNK}_Hidrography(Hidrography)
Effect = _Hidrography(HidrographyCoastal)
<ACT_END>

Action = Set_TopographyChoice(?t)
            Constraints  = Topography(?t)
            Precondition = [{UNK}_TopographyChoice(Topography)] {AND} _Topography(?t)
Effect = _TopographyChoice(?t)
<ACT_END>

Action = Set_MapSize(?s)
            Constraints  = MapSize(?s)
```

```
            Precondition = [{UNK}_MapSize(MapSize)]
Effect = _MapSize(?s)
<ACT_END>

##STAGE 1 END##

Action = Set_Stage1Complete()
            Constraints  =
            Precondition = [[{UNK}_Stage1Complete()] {AND} [[_Temperature(Temperature) {AND}
_TopographyChoice(Topography)] {AND} [_Hidrography(Hidrography) {AND} _Vegetation(Vegetation)]]] {AND}
[_MapSize(MapSize)]
Effect = _Stage1Complete()
<ACT_END>

###STAGE 2 START

Action = Set_CliffSand()
            Constraints  =
            Precondition = [[[{UNK}_Cliff(Layer/Cliffs)] {AND} _Stage1Complete()] {AND}
[[_Temperature(TemperatureScalding)] {OR} [_Temperature(TemperatureHot)]]]
Effect = _Cliff(CliffSand)
<ACT_END>

Action = Set_CliffEarth()
            Constraints  =
            Precondition = [[[{UNK}_Cliff(Layer/Cliffs)] {AND} _Stage1Complete()] {AND}
[[{UNK}_Temperature(TemperatureFreezing)] {AND} [{UNK}_Temperature(TemperatureScalding)]]]
Effect = _Cliff(CliffEarth)
<ACT_END>

Action = Set_CliffCold()
            Constraints  =
            Precondition = [[[{UNK}_Cliff(Layer/Cliffs)] {AND} _Stage1Complete()] {AND}
[[_Temperature(TemperatureCold)] {OR} [_Temperature(TemperatureFreezing)]]]
Effect = _Cliff(CliffCold)
<ACT_END>

#Ground

Action = Set_GroundSand()
            Constraints  =
            Precondition = [[[{UNK}_Ground(Layer/Ground)] {AND} _Stage1Complete()] {AND}
[_Temperature(TemperatureScalding)]]]
Effect = _Ground(GroundSand)
<ACT_END>

Action = Set_GroundDirt()
            Constraints  =
            Precondition = [[[{UNK}_Ground(Layer/Ground)] {AND} _Stage1Complete()] {AND}
[[_Temperature(TemperatureScalding)] {OR} [_Temperature(TemperatureHot)]]]
```

```
Effect = _Ground(GroundDirt)
<ACT_END>

Action = Set_GroundSnow()
	Constraints  =
	Precondition = [[[{UNK}_Ground(Layer/Ground)] {AND} _Stage1Complete()] {AND}
[[_Temperature(TemperatureCold)] {OR} [_Temperature(TemperatureFreezing)]]]
Effect = _Ground(GroundSnow)
<ACT_END>

Action = Set_GroundGrass()
	Constraints  =
	Precondition = [[[{UNK}_Ground(Layer/Ground)] {AND} _Stage1Complete()] {AND}
[[{UNK}_Temperature(TemperatureFreezing)] {AND} [{UNK}_Temperature(TemperatureScalding)]]]
Effect = _Ground(GroundGrass)
<ACT_END>

#Trees

Action = Set_TreeSnowy()
	Constraints  =
	Precondition = [[{UNK}_Tree(Layer/Trees)] {AND} _Stage1Complete()] {AND}
[[_Vegetation(VegetationDense) {OR} _Vegetation(VegetationFocused)] {AND} [_Temperature(TemperatureCold)]]
Effect = _Tree(TreeSnowy)
<ACT_END>

Action = Set_TreeCactus()
	Constraints  =
	Precondition = [[[{UNK}_Tree(Layer/Trees)] {AND} _Stage1Complete()] {AND}
[[_Temperature(TemperatureHot) {OR} _Temperature(TemperatureScalding)] {AND} [[_Vegetation(VegetationSparse)
{OR} _Vegetation(VegetationSporadic)]]]] {AND} [_Ground(GroundDirt) {OR} _Ground(GroundSand)]
Effect = _Tree(TreeCactus)
<ACT_END>

Action = Set_TreeRegular()
	Constraints  =
	Precondition = [[{UNK}_Tree(Layer/Trees)] {AND} _Stage1Complete()] {AND}
[[_Vegetation(VegetationDense) {OR} _Vegetation(VegetationFocused)] {AND}
[[{UNK}_Temperature(TemperatureFreezing)] {OR} [{UNK}_Temperature(TemperatureScalding)]]]
Effect = _Tree(TreeRegular)
<ACT_END>

Action = Set_TreeLeafless()
	Constraints  =
	Precondition = [[{UNK}_Tree(Layer/Trees)] {AND} _Stage1Complete()] {AND}
[[_Vegetation(VegetationSparse) {OR} _Vegetation(VegetationSporadic)] {OR} [[_Temperature(TemperatureFreezing)
{OR} _Temperature(TemperatureScalding)]]]
Effect = _Tree(TreeLeafless)
<ACT_END>
```

```
#Action = Set_TreeNone()
#		Constraints  =
#		Precondition = [{UNK}_Tree(Layer/Trees)] {AND} [_Vegetation(VegetationNone)]
#Effect = _Tree(TreeNone)
#<ACT_END>

Action = Set_WaterIsland()
		Constraints  =
		Precondition = [[{UNK}_Water(Layer/Water)] {AND} _Stage1Complete()] {AND}
[_Hidrography(HidrographyIsland)]
Effect = _Water(WaterIsland)
<ACT_END>

Action = Set_WaterCoastal()
		Constraints  =
		Precondition = [[{UNK}_Water(Layer/Water)] {AND} _Stage1Complete()] {AND}
[_Hidrography(HidrographyCoastal)]
Effect = _Water(WaterCoastal)
<ACT_END>

Action = Set_WaterDry()
		Constraints  =
		Precondition = [[{UNK}_Water(Layer/Water)] {AND} _Stage1Complete()] {AND}
[_Hidrography(HidrographyDry)]
Effect = _Water(WaterDry)
<ACT_END>

Action = Set_WaterPonds()
		Constraints  =
		Precondition = [[{UNK}_Water(Layer/Water)] {AND} _Stage1Complete()] {AND}
[_Hidrography(HidrographyPonds)]
Effect = _Water(WaterPonds)
<ACT_END>

Action = Set_WaterRivers()
		Constraints  =
		Precondition = [[{UNK}_Water(Layer/Water)] {AND} _Stage1Complete()] {AND}
[_Hidrography(HidrographyRivers)]
Effect = _Water(WaterRivers)
<ACT_END>
```

Attachment B

Consent Form

Termo de Consentimento

1. Você está sendo convidado/a para participar de um procedimento de avaliação qualitativa para a dissertação de mestrado de “Yuri Pessoa Avelar Macedo” pelo Departamento de Ciência da Computação da Universidade Federal de Minas Gerais (UFMG).

2. Você foi selecionado/a pois a seu perfil foi considerado de ser desejado para os testes da pesquisa em questão. Sua participação não é obrigatória, e a qualquer momento você pode desistir de participar e imediatamente retirar seu consentimento. Sua recusa não trará nenhum prejuízo em sua relação com o pesquisador ou com a instituição.

3. Os objetivos deste estudo são obter informações sobre sua experiência com um jogo simples. Sua participação nesta pesquisa consistirá em jogar um jogo durante 10 a 30 minutos e descrever suas experiências de acordo com perguntas que serão feitas pelo pesquisador .

4. À sua vontade, informações obtidas através dessa pesquisa poderão ser confidenciais. Neste caso, asseguramos o sigilo sobre sua participação, e as informações coletadas não serão divulgadas de forma a possibilitar sua identificação.

5. Você receberá uma cópia deste termo onde consta o telefone e o endereço do pesquisador principal, podendo tirar suas dúvidas sobre o Projeto de Pesquisa de sua participação, agora ou a qualquer momento.

DADOS DO PESQUISADOR

____

Nome

____

Assinatura

____

Endereço completo

____

Telefone

Declaro que entendi os objetivos, riscos e benefícios de minha participação na pesquisa e concordo em participar.

Belo Horizonte, _____ de ________________ de ______

____

Participante da pesquisa

Consent Form

1. You're invited to participate in a qualitative evaluation procedure for the master's thesis of "Yuri Pessoa Avelar Macedo", from the department of computer Science (DCC) of the Universidade Federal de Minas Gerais (UFMG).

2. You've been chosen due to your profile being considered desirable for this research's tests. Your participation is not obligatory and you may stop it at any time and remove your consent. Your refusal will not bring any loss to your relationship with the researcher and/or with their institution.

3. This study's objective is to collect information based on your experience with a simple game. Your participation will consist of you playing a game for ten to thirty minutes, and then to describe your experiences based on questions asked by the researcher.

4. At your will, information acquired through this research may be confidential. At which case, we assure secrecy about your participation. The collected information will then not be divulged in a way that allows or assists in your identification.

5. You'll receive a copy of this form that contains the researcher's phone number and home address, allowing you to query about this research project and your participation in it. Now, or at any time.

RESEARCHER'S INFORMATION

____
Name

____
Signature

____
Full home address

____
Contact phone

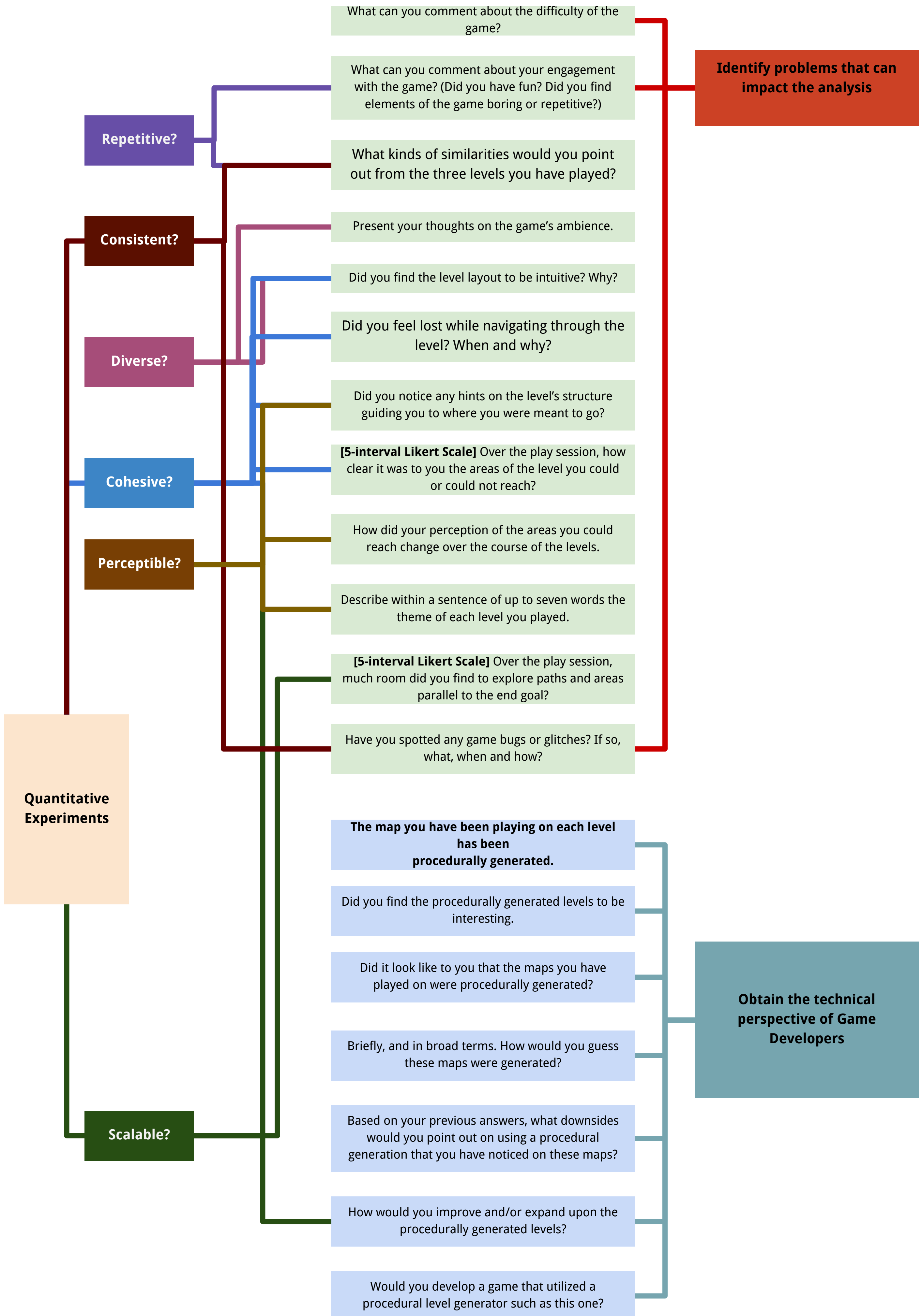
I hereby declare to understand the objectives, risks, and benefits of my participation in this research, and agree to participate.

Belo Horizonte, _______________ (Month) ____ (day), ______ (year)

____
Participant's Signature

Attachment C

Interview Diagram



What can you comment about the difficulty of the game?
What can you comment about your engagement with the game? (Did you have fun? Did you find elements of the game boring or repetitive?)
Identify problems that can impact the analysis
Repetitive?
What kinds of similarities would you point out from the three levels you have played?
Consistent?
Present your thoughts on the game's ambience.
Did you find the level layout to be intuitive? Why?
Did you feel lost while navigating through the level? When and why?
Diverse?
Did you notice any hints on the level's structure guiding you to where you were meant to go?
[5-interval Likert Scale] Over the play session, how clear it was to you the areas of the level you could or could not reach?
Cohesive?
How did your perception of the areas you could reach change over the course of the levels.
Perceptible?
Describe within a sentence of up to seven words the theme of each level you played.
[5-interval Likert Scale] Over the play session, much room did you find to explore paths and areas parallel to the end goal?
Have you spotted any game bugs or glitches? If so, what, when and how?
Quantitative Experiments
The map you have been playing on each level has been procedurally generated.
Did you find the procedurally generated levels to be interesting.
Did it look like to you that the maps you have played on were procedurally generated?
Obtain the technical perspective of Game Developers
Briefly, and in broad terms. How would you guess these maps were generated?
Based on your previous answers, what downsides would you point out on using a procedural generation that you have noticed on these maps?
Scalable?
How would you improve and/or expand upon the procedurally generated levels?
Would you develop a game that utilized a procedural level generator such as this one?

Attachment D

Qualitative collection

Includes the results from the pilot tests' volunteers.

Volunteer\Topic	VPA1	VA1	VA2	VA3	VA4	VA5	VA6	VPB1	VB1	VB2	VB3	VB4	VB5	VB6
Noticed guiding tips	1	1	0	0	1	1	0	1	0	1	0	1	1	0
Intuitive layout	0	1	0	0	1	1	0	1	0	1	0	1	1	0
Room to Explore	2	3	4	4	3	3	4	2	3	3	3	2	4	4
Understood the map	4	4	4	3	2	4	3	5	5	4	5	3	4	5
Got lost	1	1	1	2	1	0	1	0	1	0	1	0	1	2
Had Fun	0	1	1	1	1	1	1	1	0	1	1	0	1	1
Stated that there was no reason to explore	0	1	1	1	1	0	0	0	1	0	0	0	0	1
Found maps visually engaging	0	1	0	1	1	1	0	0	0	0	0	0	1	1
Number of remembered levels	2	2	3	3	3	3	3	2	3	2	1	2	1	2
Noticed it was PCG	0	1	1	0	0	0	0	0	1	0	0	1	1	0
Asked if it was about PCG during play	0	0	1	0	0	0	0	0	0	1	0	0	1	0

Table D.1. Data from the volunteers' questions. The vertical axis contains each stated opinion, while the horizontal contains answers relative to each topic. Most information is organized as 0's (no) and 1's (yes). A few exception topics have distinct values: **Room to Explore** is an integer from 1 to 5: A value of one means that there was no freedom to explore, a 5 means that the amount of explorable areas was overwhelming, and a 3 being the perfect balance; **Understood the Map** is a simple grade from 1 to 5 of how well the map was understood; **Got Lost** is a ternary scale with 0 meaning the volunteer did not get lost, with 2 meaning that they have gotten lost, and 1 meaning that, while getting lost, the volunteer did not find it to be a bothersome problem. Finally **Number of Remembered levels** is just that, an accounting of the number of levels the volunteer could remember and accurately describe.

Group A, Pilot Test Volunteer

- *'Zombies always come from the same places.'*
- *'The zombies' positioning should be randomly generated.'*
- Noticed cohesion between level elements.

Summation\Topic	Sum (A)	Sum (B)	Total	Averages (A)	Averages (B)	**Total**
Noticed guiding tips	4	4	**8**	0.571	0.571	**0.571**
Intuitive layout	3	4	**7**	0.429	0.571	**0.500**
Room to Explore	-	-	**-**	3.286	3.000	**3.143**
Understood the map	-	-	**-**	3.529	4.429	**3.929**
Got lost	1	2	**3**	1.000	0.714	**0.857**
Had Fun	6	5	**11**	0.857	0.714	**0.786**
Stated that there was no reason to explore	4	2	**6**	0.571	0.286	**0.429**
Found maps visually engaging	4	2	**6**	0.571	0.286	**0.429**
Number of remembered levels	-	-	**-**	2,714	1,671	**2,143**
Noticed it was PCG	2	3	**6**	0.285	0.429	**0.357**
Asked if it was about PCG during play	1	2	**3**	0.143	0.286	**0.214**

Table D.2. Data from the volunteers' questions, displaying averages and totals (when relevant) for each topic. Separated column entries for groups A, B contain their answers' sums and averages. For a better understanding of the meaning of each average, it would be best to check the information described in Table D.1's caption.

Volunteer\Topic	VPB1	VB1	VB2	VB3	VB4	VB5	VB6
Found the Procedural maps interesting	1	1	1	1	0	1	1
Closest associated PCG method	Dungeon Generator	Connect begin and end. Dungeon Generator	Spelunky	Spelunky	Generate separate areas that limit movement	Couldn't think of a method	Generate outline and fill around
Perceived procedural disadvantages	None, it seems like a map a person would make	No pacing	None	Uncoherent elements are aesthetically unpleasant	It lacked memorable segments	Confusing and frustrating level design	It requires refinement but it doesn't present major problems
Suggested improvements	Cover more empty areas with loot. Improve the art	Multiple types of encounters and optional rewards. Handmade encounters	More visual elements	Work in the art	Use proper algorithms for individual elements	Add more interest points	More visual elements
Would develop a game with this generator	1	0	1	1	1	1	1

Table D.3. Data from the volunteers' questions displayed in the same format as Table D.1. These answers regard questions exclusive to Group B. The first and last rows are closed binary answers, while the other remaining three have opinions condensed into one or two sentences.

Group A, Volunteer 1

- Noticed a sense of progression with the maps' structures.
- Noticed no relation of themes between different maps.
- Noticed the Procedural Generation through elements that did not seem coherent or have a purpose.

Volunteer \Topic	Sum Total	Average Total
Found the Procedural maps interesting	6	0,857
Would develop a game with this generator	6	0,857

Table D.4. Data from the volunteers' questions presented in Table D.3, displaying averages and totals for PCG related topics presented exclusively to Group B.

Group A, Volunteer 2

- During Question 3, the volunteer managed to identify all the Layers while answering.
- Noticed the Procedural Generation through elements that did not seem coherent or have a purpose

Group A, Volunteer 3

- Believes that there was no replayablity (game) due to it being too simple.
- Noticed that there was a pattern, but couldn't figure out what.
- Has called the game an 'obvious beta'.

Group A, Volunteer 4

- Noticed the road layer guiding to the the main path.

Group A, Volunteer 5

- *'If the game was 10 stages long, it would be repetitive.'*
- Thought two or more levels to be too much alike.

Group A, Volunteer 6

- Thought two or more levels to be too much alike.
- Noticed the PG due to it being a labyrinthine Stage, with many paths that lead nowhere, as well as the object's formations.

Group B, Pilot Test Volunteer

- *'I Did not identify a lore behind the levels and the game.'*
- *'The map and the mechanics could have been applied to any game.'*
- *'In retrospect, the empty areas could have been generated by an algorithm.'*

Group B, Volunteer 1

- *'I Don't believe (improving the content) would make it (the experience) better.'*
- Did think the levels were cohesive
- Thought exploring was counter productive, for there were no worthwhile rewards.

Group B, Volunteer 2

- Supposed Procedural Generation had been used since it would be easier to generate maps this large and with this many sprites the to create it by hand.
- Missed a city environment for a zombie game.

Group B, Volunteer 3

- Missed a city environment for a zombie game.

Group B, Volunteer 4

- Thought the ambiance to be consistent.
- Identified that the generator defined a theme to the map.
- Found nonsensical formations to be identifiable as procedural.

Group B, Volunteer 5

- Thought the ambiance to be too generic.

Group B, Volunteer 6

- *'The graphic style pushed me away from thinking of procedural generation'*