Improving procedural 2D map Generation based on multi-layered cellular automata and Hilbert curves

Yuri P. A. Macedo* Luiz Chaimowicz†

Universidade Federal de Minas Gerais, Department of Computer Science, Brazil

Figure 1: Close-up of a procedural level generated through the methodology presented in this work.

ABSTRACT

Procedural Content Generation (PCG) for Games is a field that in the past few years has seen both extensive academic study and practical use in the games industry. One of its common uses being the generation of maps for levels within games that rely on replayability. While Cellular Automata is a PCG technique widely used for the creation of minor graphical systems, it has not yet seen much practical use in the generation of levels, part due to its inherent stochastic nature. With the purpose of presenting a malleable approach for improving levels created through Cellular Automata, this work presents a methodology that guides the generation process through the use of fractals, specifically Space-filling Curves. The product Automata of this process are implemented and polished on the Unity game engine, as to present their potential for generating procedural levels. Results show that this methodology can be used for the generation of organic, cohesive game levels.

Keywords: Procedural Content Generation, Evolutionary Algorithms, Cellular Automata, Fractals, Space-filling Curves, Hilbert Curves

1 INTRODUCTION

As the video game industry grows, creating sufficient content to satisfy the customers' demands proves itself to be a struggle that for some companies might as well be impossible [6]. The cost of manually generating better content outpaces sales and revenue, to the point where some companies are unable to follow the race for games of greater scale [30]. This struggle to create ever more expensive games has lead large companies to search alternatives for generating content. On the other hand, Indie developers face their own struggles with creating more for their games with fewer resources. Although their conflict is not a predatory competition, the similar lack of resources and small teams can be the catalyst for the search for 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).

One of the first documented cases of PCG for games (PCG-G) [13] might as well have started by the end of the 70's. When the designers' objective was not to thoroughly optimize monetary expenses, but instead, overcoming storage limitations. Enter Telengrad [18], an Atari/DOS dungeon crawler with enemies, treasures, and dungeons with millions of explorable rooms. This six-digit amount of content was stored in just about 50 kb of memory (earlier versions of the game required as little as 8 kb). Being able to achieve having 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 Procedural Content Generation (PCG) 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 [19]; Finally, having the game create unique content each time it is played could potentially bring new, innovative experiences every time.

After the success of games such as Diablo, Left 4 Dead,

*e-mail: ypamacedo@gmail.com
†e-mail: chaimo@dcc.ufmg.br

Minecraft, Spelunky, and many others that implemented some form or another of PCG, the topic rose in interest. Some game developers even stated their use of Procedural Content Generation as a highlight for promoting game sales. Even so, Procedural Content Generation is deceptively hard to do right [29]. Game systems that rely on procedural content are often considered fail to deliver on their promise by their target demographic. Describing procedural games as 'boring', 'repetitive', or even 'disappointing' became a common opinion amongst gamers, as both them and developers realized that Procedural Content may not bring all that is desired of it if not properly done. So much so that a wave of hatred towards procedural content surged within part of the game community.

For the game developers, a procedural content methodology has proven not to be as much as a boon for reducing costs. Instead PCG introduced several trade-offs: On one hand, procedural content can reduce workload and consequently costs for generating content. However, it can only do so if the amount of it supersedes the cost of creating a highly specialized and robust generation framework; Second, if the target game for the PCG algorithm will not make full use of dozens, hundreds, or thousands of said content, then it might be cheaper assigning 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 is deemed 'good', one must still ponder if the having more of said content compensates for the loss of that human touch.

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

One of the topics of PCG that has seen great application within the industry [13] is the generation of Game Space: A category of PCG-G that consists of crafting the environment in which the game takes space. For most games that allow the player to influence a character's movement within in a intractable area, this Game Space is the 'Map' within a 'Level' (the whole of all game elements within that space). There are a plethora of academic works discussing methodologies for Map Generation, yet we feel that many still refrain from discussing how playable, or 'game-like' resulting maps can be. Another topic that is rarely touched upon is how to adapt the proposed methodology to distinct game genres and archetypes.

As an effort to grow upon the field of generating procedural maps for game levels, we propose in this paper a combination two existing distinct procedural systems to generate maps for games. This paper also proposes a methodology through which existing studies for generating general purpose procedural maps can be expanded upon to generate full game levels.

Our work furthers the discussion over the generation of procedural game maps. As a theoretical framework, we build upon the existing discussion of PCG through Cellular Automata (CA): A discrete model that has seen uses in many areas, including vegetation [5], ecology [5, 14], and human population dynamics [21]. Our contribution is two-fold: 1. We present a methodology for generating organic game maps, proposing improvements to existing methodologies; 2. We develop a methodology and proof-of-concept example that is applicable to both to the procedural generation methods described in this work, as well as others based of Cellular Automata. The proof-of-concept illustrates within the Unity [2] Game Engine our methodology's applications and introduces the concept of multiple layers for 2d procedural generation based of CA.

The remainder of this work is divided as follows: Section 2 briefly covers few of the studies that analyze PCG as a field of research and as an industry tool. It also reviews some methods that have been proposed or related to the generation of maps/levels; Section 3 goes over the basic concepts of the research areas from which our methodology draws from; Section 4 describes in full detail our map generation process, the algorithms involved in it, it can be improved. Section 5 briefly discusses the generated maps, providing a plethora of examples for the reader to form his or her own opinions. Section 6 proposes how to expand the proposed methodology pertaining the creation of additional content upon the foundation of the map. Section 7 presents our final thoughts and conclusions on this work, discussing future improvements to both our methodology and its experimental evaluation.

2 Related Work

This section will provide a brief overview and provide references for related works. First, refers to works that classify, survey and otherwise offer a comprehensive entry point for understanding the basics of Procedural Content Generation. Second, it goes over the current state of academic research on procedural map generation as presented by the works of the last fifteen years. Third, it acknowledges the works of an industry game that in many ways inspired us to tackle generating procedural maps as we did.

2.1 PCG as a field of research

The evolution of PCG has been documented, analyzed, and streamlined [29, 13, 33] for those interested in dabbling with its concepts, and for veteran researchers alike. Togelius et al. [31] proposes three types of PCG oriented design that are defined by how much procedural content influences the game it was designed for. It also defines nine challenges that range from the generation of resources for diverse themes, to the inherent flaws to procedural content generation, such as the modularization of generator systems, the lack of integration, or the blandness of the content it can create. As an overview in most types of Procedural Generation, the PCG Book [29] summarizes various works within the area, comparing and contrasting them. Hendrikx et al. [13] covers the practical uses of procedural content generation, defining a six-layered taxonomy that intends to cover all types of procedural content. It also analyzes commercial games that use PCG, discerning the types of procedural generator implemented by each game.

2.2 Procedural Map Generation

Fractals [10], such as the Space-filling Curves described further in this work, are widely used as computationally efficient methods for procedural generation, as well as impressive models for topology and erosion simulations [25]. They however are not as effective for parametrized generators such as what should be desired from a level generator. This is mainly due to fractals requiring and accepting only a random seed as parameter to generate their shapes, restricting any alterations to be done exclusively after the generation process.

Search based procedural content generation is a methodology that attempts to return the 'best' combination of procedural elements, given a set of parameters. It does so by searching a subset from all the possible combinations of elements that create a procedural system. This approach is greatly adaptable to distinct types of content, including map/level configurations [12] that supports quests and narratives, and attempts to generate maps that fulfill multiple objectives [32] for strategy games.

In the context of integrating map and story generation, Matthews and Malloy [22] present, design, and implement a technique that utilizes flood-fill algorithms. A designer's document of restrictions is interpreted for generating over-world maps with cities, towns, and other landmarks that define large scale maps in geographical

cohesion with a story. Another work in this context by Valls-Vargas et al. [34] designs a story driven map generator focused on the representation of Plot Points and their causal relationship with the map in which they occur. It then generates, and evaluates configurations of maps that support a given story through planning algorithms.

Many procedural maps face the problem of generating untraversable, unexplorable, and/or unreachable areas. This is usually tackled during the generation process, with path-finding algorithms. In 3D environments, the landscape has to be tweaked to avoid these flaws, but could still be too bland for the player to navigate in. Biggs et al.[7] presents a combination of architectural techniques for generating landmarks that guide the player, helping him or her identify their location within a larger 3d map.

Generative Grammars have been applied to map generation by representing a map's structure as rules of a graph grammar [3]. Representing maps this way, by itself, 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.[28] 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 based of Quad-trees, connecting the rooms created by the partitioned units in order; The second being an agent that 'digs out' a dungeon by traversing a space while creating rooms.

Johnson et al. [15] presents a computationally efficient approach to utilizing Cellular Automata as a basis for infinite procedural 2D top-down perspective maps. The cave generation algorithm is proven to execute in polynomial time, its complexity being defined mostly by the map's width and height. It is effective enough to be run on-line (during the game's execution). This work intends to develop the discussion on cellular automata maps, although not necessarily for the purposes of infinite ones.

2.3 Combining Cellular Automata and Space-filling Curves

Although the methodology of combining Cellular Automata with Space-filling curves has not seen documented use within academic works, it has seen an effective implementation within the commercial game Galak-Z [4], by the 17-bit Indie Studio, published by Sony Interactive. It utilizes from the process of generating subsections 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, this work 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.

3 BACKGROUND

This section will briefly cover the basics of the theoretical fundamentals of our methodology. Our purpose is to present a basic understanding of what goes behind Section 4. Each of the topics described in the following subsections are fields of study on their own, and we highly advise that, for the purposes of inquiring for more detailed information, one would gain more from reading other focused sources.

3.1 Cellular Automata

Cellular Automata is a discrete model with self-organizing properties that 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, which can be done one cell at a time (Asynchronously) on a predetermined order, or all cells at once (Synchronously). This constant shifting of values generates distinctive results depending on the starting configuration and the rules by which a cell evolves. The resulting grid after a number of 'iterations' or 'cycles' can be a stable structure or a constantly shifting landscape, occasionally looping into an evolutionary cycle. Often, Cellular Automata generate interesting shapes such as the ones presented in Figure 2.

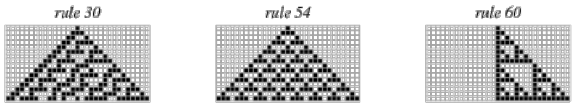

Figure 2: The Cellular Automata generated by rule 30, 54, and 60 of 'The 256 Rules' [20].

The automata's universe starts at the 'configuration' state: all cells start 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 within the automata's status quo. The state of any single cell is determined periodically by the state of it's 'neighbors' which itself is a concept that can be distinct for each automata, such as the Von Neumann and Moore Neighborhood concepts, presented in Figure 3.

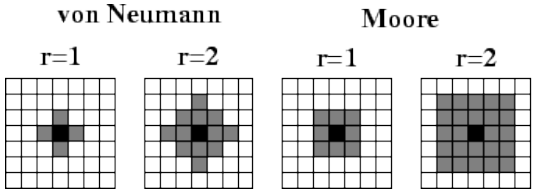

Figure 3: Von Neumann and Moore neighborhoods for a single cell (colored in black). The neighborhood's size 'r' can be increased to consider the states of additional nearby cells.

Perhaps the most common examples of Cellular Automata are Conway's Game of Life [9], and Stephen Wolfram's Elementary Cellular Automaton [11]. 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.

3.1.1 Multi-layered Cellular Automata

The concept of multi-layered cellular automata [24, 8] is one that has not seen as much academic research as the general 2-dimensional definition of the model. A multi-layered cellular automaton is a set of cellular automata where the rule-set for the cells of each automata takes into account the state of the cells from other automata. A representation of a multi-layered cellular automaton is presented in Figure 4

The concept of multi-layered automata will be of relevance to this work when introducing methods by which to improve the generation of maps in Section /refsec:proofOfConcept.

3.1.2 Synchronous vs. Asynchronous Updating

When updating Cellular Automata, one might update the state of each cell immediately upon discovering its next state, or it might update all cells at once after their following states are determined. These are Asynchronous and Synchronous updating methodologies, respectively, and yield distinct results. Yet, Asynchronous methods can have nuances of their own, specifying distinct update

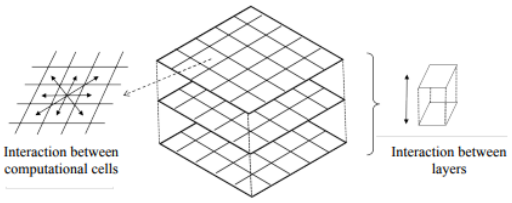

Figure 4: A Multi-layered Cellular Automaton schematic drawing as presented by [24].

orders which also yield different results. 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 6, all further automata shown follow a Synchronous updating order. Discussion on the topic of Cellular Automata Updating methodologies will hereafter be left for related works [27, 8].

3.2 Space-filling Curves

Space-filling curves, or Peano Curves[26], 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 applications such as clustering [23], and improving data structures [17, 16].

Hilbert curves are specific cases of Space-filling Curves that present interesting shapes when mapping 1D coordinates to 2D, as shown in Figure 5.

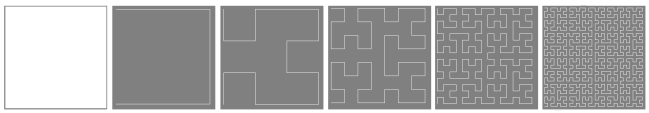

Figure 5: Hilbert Curves mapped on a NxN grid for the values of $N = 1, 2, 4, 8, 16, 32$, respectively.

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.

4 Methodology

This section describes the implementation, parameters and design decisions of the Cellular Automata and Space-filling Curves, as well as the methodology for integrating both.

4.1 Cellular Automata

Our Cellular Automata follows the cave generation method that is best introduced by Johnson et al. [15]. This procedural cave generation method is comprised of a 2-dimensional NxM grid, and a rule-set with two types of cells, true and false. The behavior of any one C cell is defined strictly by the state of it's $r = 1$ Moore Neighborhood (8 adjacent neighboring cells), 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:

- $T > 4 \Rightarrow C = true$
- $T = 4 \Rightarrow C = C$
- $T < 4 \Rightarrow C = False$

Our preliminary experiments suggest that an equal number of True and False cells ($Fill = 0.5$) presented better results, as its rules tend to converge to all cells becoming False for higher filling values ($Fill > 0.5$), or all cells except some of those adjacent to the edges of the automata being True for lower ones ($Fill < 0.5$). 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 is presented in Figure 6

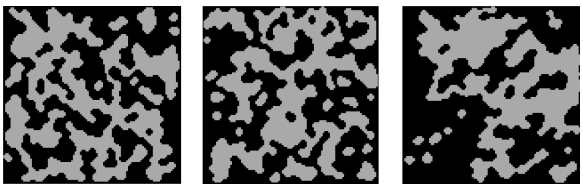

Figure 6: Three versions of the same Cellular Automaton generated from different updating methods (from left to right): Synchronous; Asynchronous, with cells updated in a random order; Asynchronous, with cells updated sequentially ordered by their position in the grid. Black cells are $true$, white cells 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'[15] 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 Space-filling Curves

The purpose of utilizing Space-filling curves is to create a guideline 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 subsection. Figure 7 illustrates the automata's changes from step 1 to 4.

4.2.1 Plotting

To place the Space-filling Curve within the automata's Grid, we start by generating a Hilbert curve that fills all of the space of the grid. Therefore, to also cover grids where M != N, the curve's order is defined by $\alpha = Max(M,N)$, generating 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 that is the basis for the tracing of the map's path.

4.2.2 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).

4.2.3 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 of which is to define a random sector from the entire curve from which to create a path upon. This is done by dividing the $(N*S)x(M*S)$ grid by the power of the Hilbert curve, resulting in a $((N*S)/\alpha)x((M*S)/\alpha)$ grid from which a random offset is selected. The curve is then Shifted to that position. Subsection 4.4 returns to the subject of introducing random variations, once the integration between the Automata and Space-filling Curve is complete.

4.2.4 Tracing a Path

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 only a few of its points remain within the grid's bounds. The ordering of the curve's path remains within a list stored from the curve's plotting, and now a new path has to be traced from what points remain.

Let η be 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. The position in the list i by which each point σ from the curve is created ranges from $0 \leq i < \alpha^2$. In order to connect the split segments η_a, η_b, at least two points $\sigma_i \in \eta_a, \sigma_j \in \eta_b$ must be connected, where $i < j < \alpha^2$.

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:

$\lambda(\sigma_i(x_i, y_i), \sigma_j(x_j, y_j)) =$

- True: $(x_i = x_j) \vee (y_i = y_j)$
- False: $(x_i \neq x_j) \wedge (y_i \neq y_j)$

Given the function for connecting points from distinct paths, for each path a and $b = a+1$, starting from $a = 0$: for each point $\sigma_j \in \eta_b$ and $\sigma_i \in \eta_a$, with $0 \leq (i = length(\sigma_a)) < (j = length(\sigma_b)) < \alpha^2$, the function $\lambda(\sigma_i, \sigma_j)$ is checked. If it returns *true*, the points become connected, which in turn connects both paths: $\eta_b = \eta_a \cup \eta_b$. If it fails, then it attempts by brute force to connect to previous points $0 \leq g < i$ from σ_a. If no pair of points can be found, then it attempts to connect other points $i < h < j$ from σ_b to each point of σ_a. This process of brute force in theory could bring a factorial worst case complexity, based of the number of points within the grid. However, due to the high number of points spread across the grid, the and the regularity of the fractal's shapes, the theoretical worst case does not apply in practice. The algorithm ends when the k number η paths equals 1, meaning all paths have been connected.

4.2.5 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 7. 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.

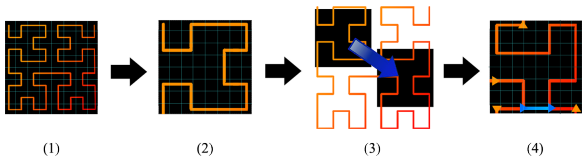

Figure 7: From left to right, the first four steps described in section 4.2 are represented.

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 apply 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 are 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.

4.3 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 it's girth is no larger than a specified 'Negative Girth' parameter N_g. This negative curve should influence as little as the automata as possible as 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 it's sides at the same time. Finally, the resulting automata setup shown in Figure 8 contains paths for the player to follow that are guaranteed not to be broken by the automata's own rules.

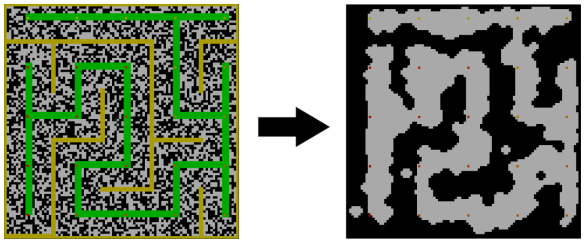

Figure 8: To the left, the Cellular Automata marked with the Space-filling Curve's path (green), as well as the negative-path curve (yellow). To the Right, the resulting automata after a number of iterations until it stabilizes. The yellow dot represents the beginning of the path, the red one represents the end. Colors in between mark the path. Parameters for this Automata are as follows: $N = 100$, $M = 100$, $Fill = 0.5$, $S = 22$, $P_g = 1$, $N_g = 1$

4.4 Polishing

Although we have achieved being able to generate the automata whilst maintaining the cohesiveness of the player's path, as shown in Figure 8, the resulting automata can still look stiff. The number of unreachable areas has been minimized, but the path still looks artificial: the points from the curve are scattered in the x and y axis in regular intervals, there are too many straight segments, and parts of the automata's topology are 'scarred' by the path curves.

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, as well as changing the physical distance between points in the map.

4.4.1 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 square 2-dimensional grid: 1) After rotating the grid, scale it down to fit the original MxN dimensions; 2) Redefine the expected grid's length 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. As a rotated version the original grid is rescaled to fit it's 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.

The second alternative is much more appealing in comparison. Instead of reducing the grid, a new larger grid is generated to fit the rotated one. Even if this means that the generated map is likely to exceed the expected bounds, it will never do so by a larger amount than the worst case of $N\sqrt{2}$ (41%) which is acceptable.

4.4.2 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, it 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 9.

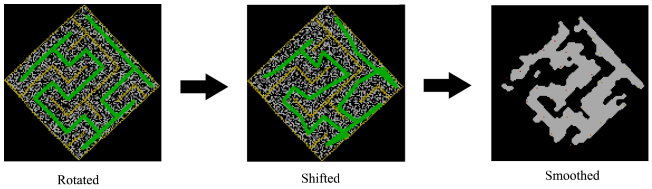

Figure 9: The same automata presented in Figure 8 after rotation and shifting operations are introduced.

As all points were shifted, the fifth step described on subsection 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 method 4.2.5 is required.

To help understand the process up to the final automata, we believe its best to help understand why each step was necessary before introducing others. Therefore, now that all steps have been proposed, an overview diagram summarizing all of the methodology is displayed in Figure 10.

Both polishing operations complete their tasks within polynomial time. In conjunction with the other methods described in this work, the process for generating maps is still accomplished within acceptable complexity to be executable online.

5 Experiments

Experiments were implemented in C# within the Unity Game Engine, and 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, as well as other minor tasks, are evenly divided across distinct game frames, through the use of Unity's 'Coroutines'. This means that, while it takes longer to be completed, the

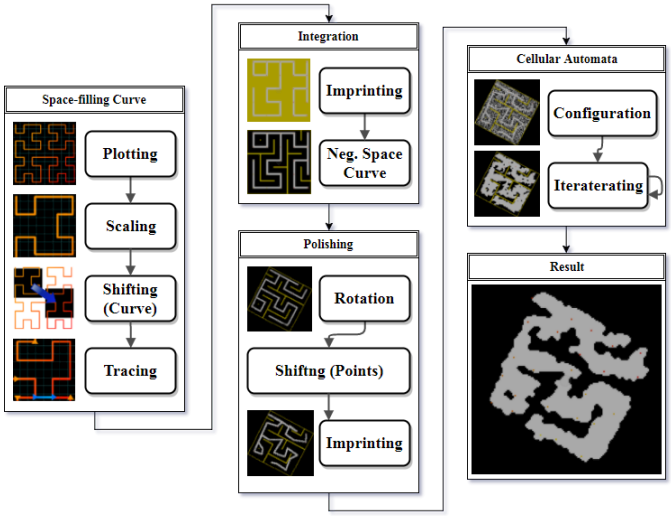

Figure 10: The complete sequence of steps for the generation of the procedural map automata. The images for the 'Space-filling' curve section are merely illustrative and do not represent the same curve-path displayed in all other images.

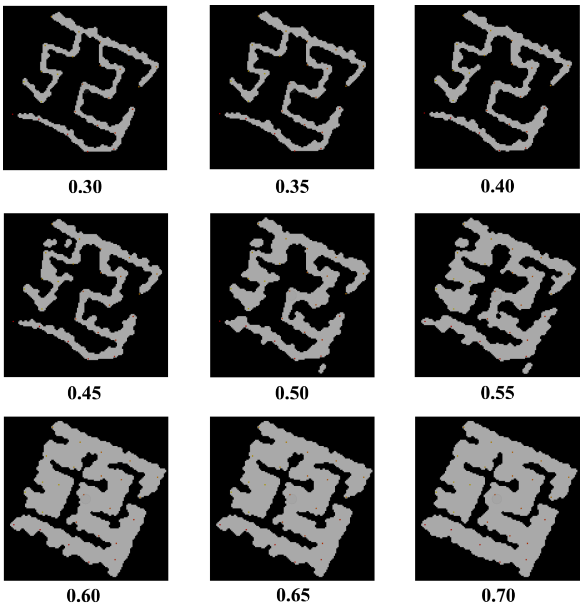

Figure 11: Experiments with the Fill parameter for different values as to find a compensation for the reduction in random cells by the imprinting of paths.

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.

For analyzing our results, our baseline for comparison are the Automata before the introduction of the curve-path and the polishing methods. A purely quantitative statistical data-driven analysis for the results would yield no significant comparison over the baseline as to measure an abstract concept as the 'quality', or 'fun' of a level. A qualitative or hybrid analysis however with playable interface and a test however would benefit our analysis greatly. Unfortunately, the framework required for such tests requires a great amount of preparation, as does the process of acquiring potential candidates and creating qualitative/hybrid evaluation methods, such as questionnaire's and interview scripts. Yet, this is definitely a goal we intend to pursue for future works that further the discussion of this methodology.

As it is often the case with results that are hard to evaluate quantitatively, it comes down to displaying examples while attempting to minimize the authors' opinion. To counterbalance this, we have selected a large set of maps presented in Figure 16 generated with random seeds, but the same parameters presented in Figure 8, to help the reader to formulate his or her own opinion on the 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 11. Each configuration of cellular automata 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 that has to be coped with. Pure procedural systems might generate beautiful results, but the unpredictability of those systems hamper the possibility of utilizing them in commercial games. In no way we intent do 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 it's natural beauty may be lost for it.

The general image of the Space-filling Curve, the Hilbert 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 task for future work, still within this one we introduce the concepts on how to build content upon the basis of generated maps.

6 From Maps to Levels

The proposed methodology allows the generation of an organic geometrical configuration adapted to a player's path through a level's map. 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 would require a flexible framework with which existing constructions could be interpreted, as for determining where to create additional content.

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 mentioned within the subsection 3.1. 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 'Mask-layers': grid layers whose purpose is not to act as automata, but to instead parse information between automata.

As a proof of concept for both Automata-layers and Mask-layers, we implement and present an example of each to generate mountainous topological markings and vegetation behavior within the

context of a hypothetical 2D top-down perspective game. The proof of concept presented within this section was implemented in the professional game engine Unity 2D [2], version 5.6.1f1 (64 bit), Personal edition.

6.1 Mask-layers

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 12. 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 12: 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 [28], as exemplified in Figure 13.

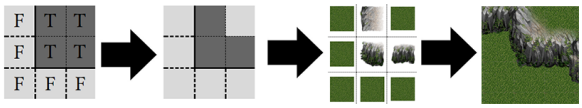

Figure 13: From the base layer, to the topology layer-mask, 2d tiles are generated depending on their position in the topology mask, creating the aspect of a continuous 'natural cliff' structure. The visual quality of the resulting tiles map (right image) is the result of editing and experimenting with or selected free visual assets[1]. As of now, we cannot present a theoretical basis as how to optimize the visual quality of matching 2d tiles.

6.2 Automata-layers

Automata-layers are the main concept by which we propose improving the systems proposed so far. 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 Mask-layers, as well as some information from the original Base automata (although these should be separated into their own Mask-layers themselves).

The benefit of having a multi-layered architecture is integrating the results of one automata with another. In this case, the tree-generating automata is integrated to the topology layer-mask and the base layer, in the sense that it does not update cells that are part of the space-filling curve path, or aree marked as 'cliffs' in the topology layer. Creating additional automata that consider restrictions from other layers ensures the product of one does not interfere with the others.

6.3 Proof of Concept

A 'Tree Generating Automata' is proposed as an example of this. This layer's automata is somewhat more complex than the base automata presented so far, but it represents vegetation dynamics [5] fairly well, and it fulfills its purpose as a demonstration of the capabilities of the methodology presented in this work. Its purpose is to simulate 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 [9]. 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 14.

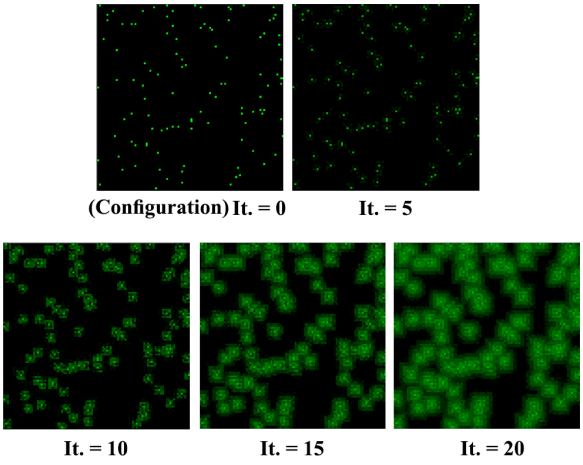

Figure 14: Evolution of the 'tree-generating automata' without the introduction of space-filling curves (Parameters: $N = 100$, $M = 100$, $Fill = 0.01$). The greener the cell, the older it is compared to the other trees. As trees need to be '5 iterations old' for them to start spawning other trees, the automata changes the most every 5 iterations. This age restriction also prevents trees from spreading wildly, instead forming small forests.

As with the layer-mask, we have set sprites for each tree: the older the tree, the bigger it's sprite becomes, trees younger than 5 iterations have alternative, smaller sprites. Examples of this are shown in 1 and 15. 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 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.

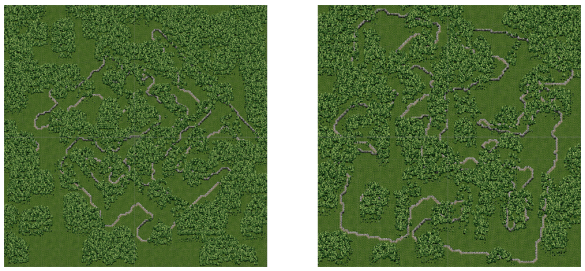

Figure 15: Examples of finalized maps.

7 Conclusion

This work proposes and implements a methodology that improves upon the generation of procedural maps through cellular automata by benefiting of the properties of Space-filling Curve Fractals. The methodology does not deviate from the low computational complexity of the existing methodologies and therefore remains viable as a robust Procedural Content Generation tool executable at runtime. This work also contributes as to offer guidelines and examples for improving maps generated through cellular automata.

A field as young as procedural generation still has a path to travel before it can safely become an industry standard for commercial games, without falling into the traps and commitments of generating procedural systems. Yet examples of games that thrive with innovative procedural experiences, and academic works that document, study, and improve upon them means we are getting ever closer to that point.

As for the methodology presented on this work, it still has areas to improve upon. A qualitative analysis of the generated content would aid greatly in finding repeating or 'boring' patterns, as with many implementations of PCG maps. This requires a framework for 'playing' the levels as well. And although building a framework and undergoing the necessary steps for a robust qualitative evaluation is time-consuming challenge, it is one that we are keen on the idea of tackling. On doing so, we plan to extend upon the concept of additional generic layers, including ones that are oriented towards game-play and intractable elements.

Acknowledgements

The authors would like to thank CAPES, CNPq and Fapemig for their financial support.

References

[1] "opengameart". www.opengameart.org/, 2017. Accessed: 2017-08-7.

[2] "unity 3d/2d game development platform". www.unity3d.com/, 2017. Accessed: 2017-08-7.

[3] D. Adams et al. Automatic generation of dungeons for computer games, 2002.

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

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

[6] J. Benson. World of warcraft team: "procedural content is totally something we've talked abfout". www.pcgamesn.com/wow/world-warcraft-team-procedural-content-totally-something-we-ve-talked-about, November 2013.

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

[8] A. Bonomi. Dissipative multilayered cellular automata facing adaptive lighting. 2009.

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

[10] D. S. Ebert. *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.

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

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

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

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

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

[16] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. Technical report, 1993.

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

[18] D. Lawrence. Telengrad. www.aquest.com/telen.htm, 1976.

[19] J. Lee. "how procedural generation took over the gaming industry". www.makeuseof.com/tag/procedural-generation-took-gaming-industry/, 2014. Accessed: 2017-08-7.

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

[21] H. Liang and Z. Wang. Optimized distribution of beijing population based on ca-mas. *Discrete Dynamics in Nature and Society*, 2017, 2017.

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

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

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

[25] J. Olsen. Realtime procedural terrain generation, 2004.

[26] H. Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[27] B. Schonfisch. Synchronous and asynchronous updating in cellular automata. 1999.

[28] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games*, pages 31–55. Springer, 2016.

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

[30] Takatsuki. Cost headache for game developers. www.news.bbc.co.uk/1/hi/business/7151961.stm, December 2007.

[31] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley. Procedural content generation: Goals, challenges and actionable steps. In *Dagstuhl Follow-Ups*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[32] J. Togelius, M. Preuss, and G. N. Yannakakis. Towards multiobjective

procedural map generation. In *Proceedings of the 2010 workshop on procedural content generation in games*, page 3. ACM, 2010.

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

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

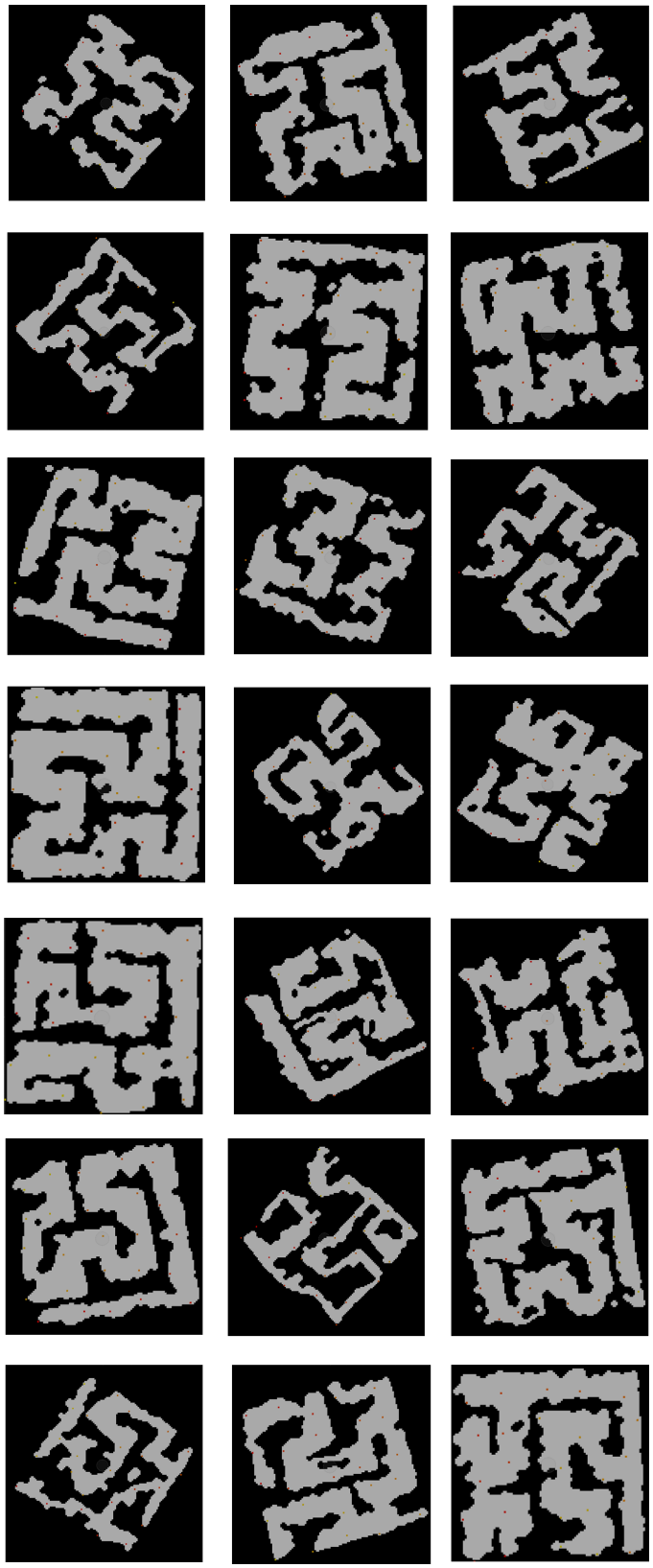

Figure 16: Different versions of the parameters used for the automata in Figure 8, executed with different random seeds.