Procedural Game Map Generation using Multi-leveled Cellular Automata by Machine learning

Zhixuan Wu†

College of Arts and Science
New York University
New York, NY, United States
† Corresponding author
zw1887@nyu.edu

Yuwei Mao

Department of Performance
University of California, Santa Cruz
Santa Cruz, CA, United States
maoyuwei99@gmail.com

Qiyu Li

York School
Monterey, CA, United States
lik2022@york.org

Abstract

The concept of Procedural Content Generation (PCG) has been intensively applied in the game industry for its capability of producing infinite game maps without any human effort. Innumerable games, such as *Minecraft*, *Terraria*, and *No Man's Sky*, successfully employed this technique to create unpredictable yet playful gaming experiences. While randomness is essential to adding engaging elements to a game, complete randomness may hurt the outcome of map generations by making a chaotic scene. To address this issue, this paper introduces an effective way of "tweaking" the randomness to generate flexible, endless, natural-looking game maps by machine learning.

CCS CONCEPTS

Applied computing • Life and medical sciences • Bioinformatics

Keywords

Cellular Automata, Procedural Content Generation, Cave Generation Algorithm, Roguelike Game, medical deep learning

1. Introduction

Games, especially rogue-like games, require a large number of levels and maps to create an engaging, story-telling play experience. While the maps indeed can be created manually, these "man-made" levels can be expensive in 2 ways: first, for games that need an endless supply of level designs, creating maps for every single level takes a lot of time and energy; second, the large number of maps that are pre-designed and loaded into the game take up tons of memory footprint. Procedural Content Generation avoids these two problems by automatically generating maps on the fly so that no static storage is needed and no two maps are the same. Through handling the lengthy, repetitive work using an effective algorithm, game developers can instead focus on more critical design problems and output more intriguing games. Therefore, an algorithm that is able to generate infinite and reliable environments can cast a direct influence on the quality of the entire game.

While there has already been a lot of studies and interesting algorithms developed for procedural map generation such as *the Tunneling algorithm* and *Drunkard's Walk*, this work focuses on a way of producing customizable maps with rich elements and natural twists and curves. In short, this paper will introduce a reliable, flexible algorithm based on multi-leveled cellular automata and adjusted probability to generate infinite, realistic maps enriched by various landforms such as lakes, islands, forests, and deserts.

ISAIMS 2021, October 29–31, 2021, Beijing, China

ACM ISBN 978-1-4503-9558-8/21/10…$15.00
https://doi.org/10.1145/3500931.3500962

2. Background

2.1 Roguelike Games

Roguelike games are games with strong focuses on intricate content and replayability. This genre of game is usually characterized by pixel art visual style and endless replayable levels. More importantly, randomization plays an important role in replayability. One of the characteristics defined by Erdi Igzi from Charles University is that "Major parts of the world in which the game is played are generated using a random maze/dungeon generation algorithm. Thus, every game is different than the others, and this feature makes the level playable many times [1]." The concept described above is called procedural generation, and it provides a different experience each time playing the game regardless of the simplex style.

2.2 Cellular automata: Game of life

John Conway's game of life is a type of cellular automata in which there are multiple alive and dead cells. According to Alexander Gellel and Penny Sweetser in their paper *A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels*, "cellular automata (CA) are spatial, discrete time models represented on a uniform grid, which can be used to model different aspects of game environments [2]." The rules are as follows:

> *1. Survivals. Every counter with two or three neighboring counters survives for the next generation.*

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

3. Births. Each empty cell adjacent to exactly three neighbors--no more, no fewer--is a birth cell. A counter is placed on it at the next move [3].

A simple pattern will be created according to the rules. The pattern also changes after rerunning the automation each time.

To further exemplify this concept, this work included a simple simulation of the Game of Life with python. The first randomized map is shown in Figure 1. Figure 2 shows the state of the cells after ten rounds of automation following these same rules. Overall, this is a conceptual idea for the generation of game maps.

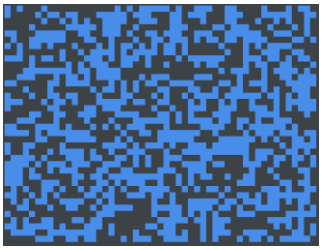

Figure 1 Initial 2D grids filled by random values

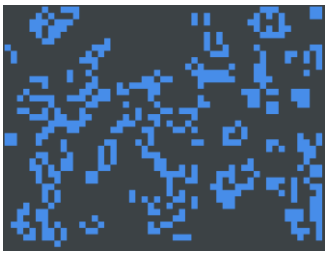

Figure 2 The automaton map after ten life stages

2.3 Cave Generation Algorithm

The given rules of Conway's game of life are not sufficient to create an inner connected map for potential caves or islands. In Michael Cook's article *Generate Random Cave Levels Using Cellular Automata*, he presents an alternative set of rules that avoids the overly sporadic pattern of automata generations and makes actual map-like looking scenes. Instead of having the exact number of neighbors dead or alive to change the state of the cell, a birth limit and a death limit are created for the rule of automation:

The rules are simpler than the Game of Life - this program has two special variables, one for birthing dead cells (birthLimit), and one for killing live cells (deathLimit). If living cells are surrounded by less than deathLimit cells they die, and if dead cells are near at least birthLimit cells they become alive [4].

Figure 3 and 4 displays the visualization results of the new set of rules. Through observation, the initial map in Figure 3 is very similar to the one in Figure 1. But after running the cave generation algorithm with the limits, the result in Figure 4 is very different from the result in Figure 2 and is more suitable to be the basis of a game map.

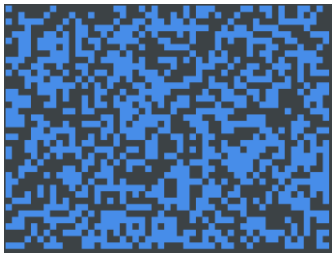

Figure 3 Initial map for cave generation

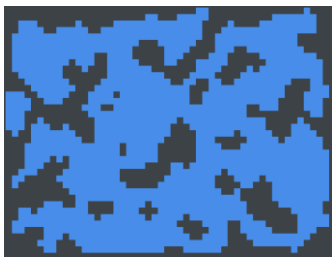

Figure 4 After 3 rounds of automations

2.4 Procedural Generation

In general, the process that generates data with algorithms instead of generating them manually is called procedural generation.

"Procedural generation is usually used to create content for video games or animated movies, such as landscapes, 3D objects, character designs, animations, or non-player character dialogue. One famous example of this is the planets generated in "No Man's Sky." In this game, players can explore 18 quintillion (18,000,000,000,000, -000,000) unique planets and moons, which were generated algorithmically by computers [5]." The example of "No Man's Sky [6]" better shows the importance of procedural generation as it is impossible for human beings to generate such a large amount of data manually. It brings infinite possibilities to a game, especially Roguelike games that rely on this infinite possibility.

In addition to the possibilities, the data generated cannot be completely random. Shown by the difference between Figure 2 and Figure 4, specific rules of randomization would affect the result of procedural generation and the experience of gameplay. This problem will be discussed further in the paper.

3. Methodology

3.1 Rules and Representation

The simple rules of Cellular Automata have the desirable nature of generating infinite and randomized patterns. However, as the patterns are too sporadic to be considered as maps, this research project adopts the adjustment made by Michael Cook in his article *Generate Random Cave Levels Using Cellular Automata* as introduced in 2.3.

The calculation of the map is performed on a set of 2D arrays with different numbers as markers representing different landforms. To evaluate the final result, visualizations of the map are generated using the Pygame library.

3.2 Multi-leveled Cellular Automata Approach

The Cave Generation Algorithm and Cellular Automata enable us to generate ocean and island-like shapes with natural curves and turns. Combining with adjusted parameters, the algorithm can form the first layer of the game map: sea and island. To create a more engaging and playable map, however, we need to enrich the content with more elements. This is where multi-leveled cellular automata come into place.

Consider the maps in real life, or clear high angle shots of islands, the elements that composed the whole picture are far more than simple island-shaped terrains and water - there are mountains and forests, freshwater lakes inside the ocean-surrounded islands, dryer and sandy lands that are of brighter color than the brownish soil, etc.

One approach to creating such natural maps with rich elements is through procedurally generating levels of the landscape. For instance, after the generation of the sea, the islands start to form; after the formation of islands, trees grow and mountains arise; and after mountains and forests, deserts and beaches appear. Figure 5 and 6 illustrates a high-level view of this multi-leveled structure.

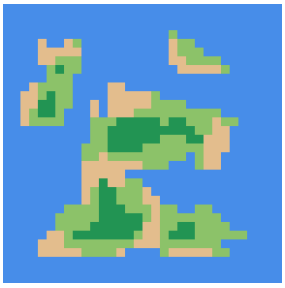

Figure 5 Final Completed Example Map

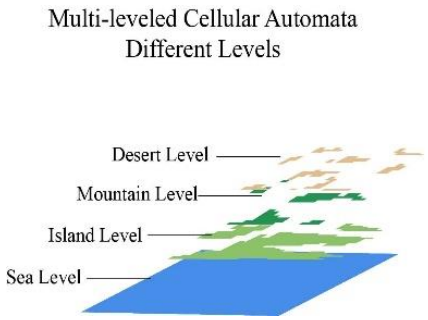

Figure 6 Different Levels of Multi-leveled Cellular Automata

For different landscapes, of course, there are different limits and constraints: for example, mountains cannot appear on top of water alone, deserts seldomly appear in a forest, beaches more frequently appear by the ocean side than at the inner island, etc. Section 3.3 will introduce in detail how the algorithm calibrates these features and constraints and finally output a realistic game map.

3.3 Developing Different Landscape Layers

3.3.1 Setup

To start with, a map populated by random values of 1 and 0 with a set probability needs to be generated. This set probability is called "Probability of Island Generation (PI)". It specifies the rate at which a map is filled by islands rather than the ocean. For example, if PI = 70, then each cell initially has a probability of 70% of becoming an island cell.

3.3.2 Sea and islands

Based on the random array generated in the last step, we perform the rules of the *Cave Generation Algorithm* on each cell. The set of rules will in turn trigger a higher probability for living cells that are connected to stay alive, and deceased cells in the middle of other dead cells remain dead. In the following pseudocode, 1 represents the island, and 0 represents the ocean.

```
island_simulation(map, death_limit, birth_limit):
    for each cell in map:
        count = number of alive neighbors
        if cell = 1:
            return (count < death_limit) ? 0 : 1
        else return (count > birth_limit) ? 1 : 0
```

3.3.3 Mountains and Forests

Mountains and Forests are essential to simulate realistic landscapes. Nevertheless, we want to limit their generation such that they only appear in lands and never in the water. Hence, for this level, the algorithm needs to first ensure that the generation only takes place on top of islands.

One way to set this constraint is by distinguishing the markers generated in the previous level. With that added, the algorithm generates mountain cells determined by another set of death_limit and birth_limit parameters so that the mountain cells only appear if the area is already occupied by island cells.

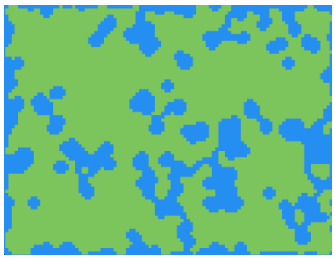

Figure 7 The Sea and Islands Level

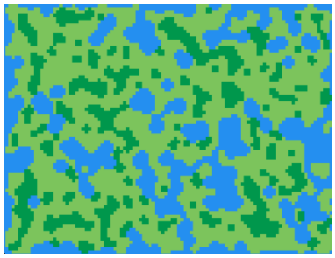

Figure 8 Map with the Mountain and Forests Level added

3.3.4 Sand and Desert

Sand and desert are not allowed to birth alone in the middle of the ocean. In addition to that, the rules for generating sand and desert are slightly different - mountains and forests, in our common sense, may appear anywhere on an island. However, this is not the case for deserts. Normally, the probability of seeing sands near the sea is higher than in the interior of the island. This is because beaches often appear near the sea and seldomly show at the center of a forest.

```
sand_simulation(map, desert_rate):
    for each cell in map:
        r = random number between [1, 100]
        rate = desert_rate
    If (i, j) is near ocean:
        rate *= 2
    if r <= rate & cell is island not mountain:
        calculate surroundings and mark cell
```

4. Experiments

Experimentation and visual comparisons are required for concluding the most appropriate combination of different parameters. As the property and landscape settings of games vary, the desired resulting maps can also be extremely different. This part will introduce the visualization results of adjusting parameters for birth limit and death limit(B&D), the number of automata life stages(N), and generation probability for mountains, islands, and deserts (PI, PM, PD).

Figure 9 - 13 shows the visualization results of different parameters after each experiment.

4.1 Birth Limit and Death Limit (B&D)

Birth Limit and Death Limit influence the amount of area that is occupied by islands. They together limit the number of alive neighbors one cell is allowed to have for it to remain alive. For instance, according to the rules of the *Cave Generation Algorithm*, when B = 4 and D = 3, a dead cell only becomes alive if it is surrounded by 4 or more cells, and a living cell is dead if it is surrounded by less than 3 cells.

In the array representation of a map, a cell is typically surrounded by 8 neighbors with exception of border cases (5 neighbors) and corner cases (3 neighbors). Hence, B and D are restricted to numbers equal to or less than 8 and greater than 1 in the first place.

Nevertheless, as the chance in which a cell has all 8 neighbors alive is only 0.4%, setting B to 8 severely deducts the possibility for cells to birth, while setting D to 8 makes all cells almost doomed to die after the first iteration. And vice versa when B & D each is set to 1 - the cells will all be alive then the landscape will flood the whole picture.

Therefore, for a map to contain a proper amount of landscapes other than water, one should adjust B & D to an intermediate number between 2 to 7. Below is a form that shows the visualization of different Birth Limits and Death Limits applied.

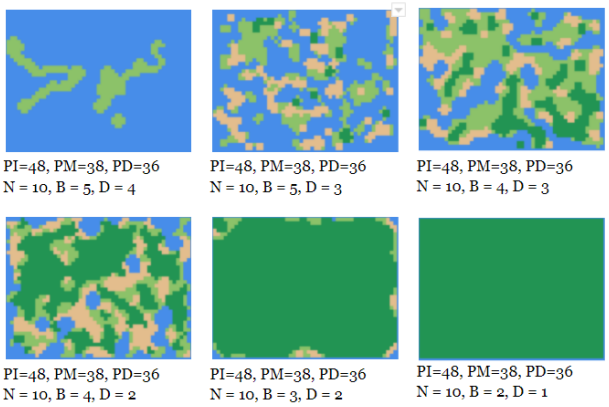

Figure 9 Visual Result of Different B&D Parameters

From the visualization above, B = 4 and D = 3 show a desirable outcome for map generation. One can also adjust D to lower if they need a map filled with more land than water.

4.2 Number of automata life stages(N)

The number of automata life stages(N) determines how sporadic the result is. When N is lower, the islands appear to be small pieces that are yet to be connected. When N is higher, the initially small islands gather together to form larger entities. However, as the set of rules defined for the island level stops permuting after several iterations, there is no need to set N to an extremely large number.

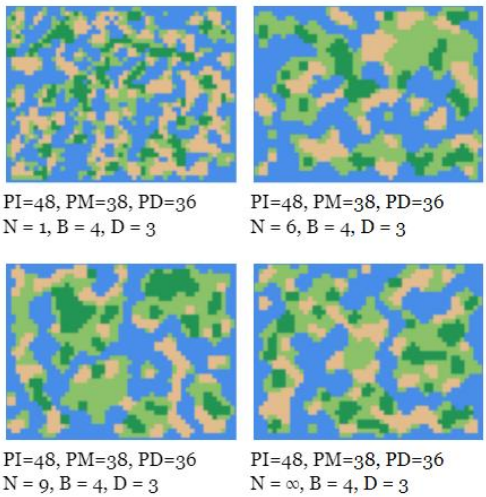

Figure 10 Visual Result of Different Number of Automata Life Stage (N)

The above visualization shows that the shapes of levels become rounder and squarer after increasing the parameter N. 6-10 is a good range, but the most appropriate value depends upon different game contexts.

4.3 Probability(P)

The generation probability of different landscapes is essential to creating varied game experiences. Depending on the game contexts, lower or higher probabilities can be applied to create desirable outcomes. These parameters altogether assist with creating flexible game maps.

4.3.1 Probability of Island Generation (PI)

The probability of island generation (PI) determines the number of islands a map contains. The larger PI is, the more area on the map is filled by lands. Below are several examples of different PI parameters being applied. One can adjust this variable to tweak the land-sea proportion of the results.

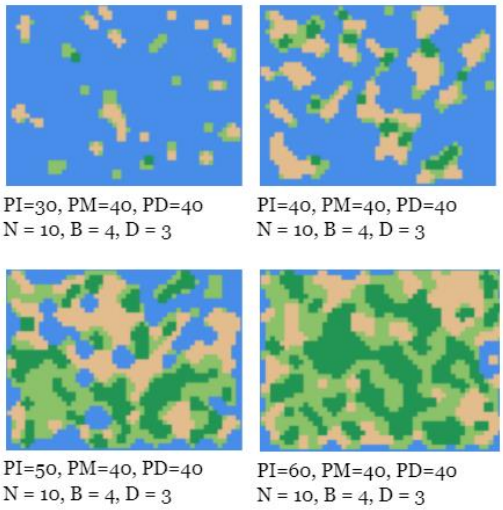

Figure 11 Visual Result of Different Probability of Island (PI)

4.3.2 Probability of Mountain Generation (PM)

The probability of mountain generation (PM) determines the proportion of mountains that are on the islands. Similar to PI, the larger PM is, the more the mountains appear in the final result.

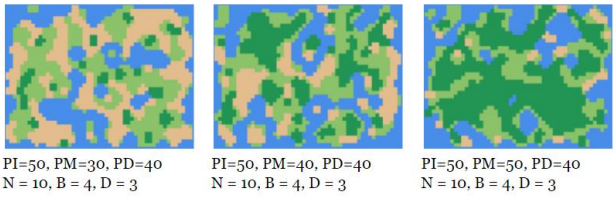

Figure 12 Visual Result of Different Probability of Mountain (PM)

4.3.3 Probability of Desert Generation (PD)

The probability of desert generation (PD) sets how many areas of the islands are occupied by sands. Below shows several outcomes of different PD parameters.

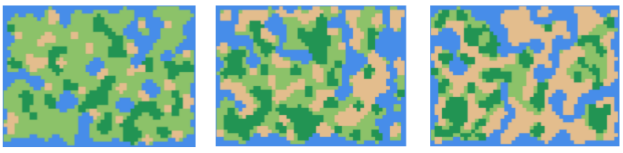

Figure 13 Visual Result of Different Probability of Desert (PD)

The visualization shows that the larger PD parameters correspond to larger areas of drylands.

5. Evaluation

The final result of the multi-leveled cellular automata approach shows the potential to output natural game scenes with randomness and replayability. Moreover, the maps generated using combinations of different parameters display highly varied visual appearances and land features. Although there are a lot of existing algorithms focusing on the random generation of game environments, this algorithm further presents more customizable and elastic adjustments for procedural map generations.

6. Conclusion

This paper proposes a reliable way of generating randomized game maps enriched by various landforms. The visualization results of this algorithm prove to be highly flexible, and reusable for different types of game contexts. One can procedurally generate highly varied game environments by adjusting different parameters and generation probabilities.

For now, there are two major next steps to look into. First, we will continue to polish the probability and generation rate mechanics. And second, a 3D implementation of this algorithm is highly possible and worthwhile to be examined. In addition to that, playtesting of the maps is also needed to further evaluate the viability of the algorithm.

REFERENCES

[1] Izgi, E. (2018) *Framework for Roguelike Video Games Development.*

[2] Gellel, A., Sweetser, P. (2020) *A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels.*

[3] Gardner, M. (1970) The fantastic combinations of John Conway's new solitaire game "life". Scientific American, 233: 120–123.

[4] Cook, M. (2013) Generate random cave levels using cellular automata. https://gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664.

[5] Brummelen, J. V., Chen, B. (n.d.) *Procedural generation: creating 3D worlds with deep learning.* http://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html.

[6] Hello Game. (2016) No Man's Sky. https://www.nomanssky.com