Procedural Map Generation by Energy Propagation

Mauro Victor Pimentel Dias[1], **Artur de Oliveira da Rocha Franco**[1], **Jose Wellington Franco da Silva**[1], **José Gilvan Rodrigues Maia**[1]

[1] Universidade Federal do Ceará (UFC) – Fortaleza, CE – Brazil

maurovictor2704@gmail.com,arturfranco@ufc.br, wellington@crateus.ufc.br, gilvan@virtual.ufc.br

Abstract. Introduction*: Procedural map generation plays a key role in popular game genres such as role-playing games and roguelikes, adding variety and dynamicity that leverages replayability or assists in level design tasks.* ***Objective:*** *We introduce a novel algorithm based on energy propagation for generating maps consisting of connected cells (rooms).* ***Methodology or Steps****: Starting from the concept of energy propagation, the algorithm simulates energy distribution across multiple rooms connecting an origin cell to a destination cell. We carried out an extensive experimental evaluation of this algorithm using a two-dimensional grid layout.* ***Results:*** *The algorithm executes in linear time on the number of cells, making it efficient enough to be applied in real-time scenarios. The proposed algorithm can be controlled to yield variable structures, ranging from tree layouts to fully interconnected grids.*
Keywords *PCG, map generation, dungeon, energy propagation, roguelike.*

1. Introduction

Procedural Content Generation (PCG) is a highly relevant topic in computing, as it enables computers to create content from an initial information set using algorithms, computational models, or other automation approaches [Shaker et al. 2016]. PCG offers significant benefits for digital games, contributing to increasing variety, replayability, and efficiency in producing content such as maps, levels, puzzles, and narratives. Our work was inspired by titles such as *The Binding of Isaac* (2011) and *Enter the Gungeon* (2016), variants of roguelike, a game genre that resorts to maps created by PCG to bring dynamism and greater unpredictability to players. In these games, each playthrough is unique, as items, enemy placement, and other elements are randomized[1]. Roguelikes also feature permanent death (*permadeath*), resource management, non-linear gameplay, and turn-based tactical combat [Parker 2017].

We present a new method for creating grid-structured maps represented as a matrix of connected cells. Differing from recent approaches, our algorithm does not employ methods based on GANs, LLMs [Nasir e Togelius 2023], or other trending AI models. We tested our algorithm in two distinct games to demonstrate its ability to generate varied maps for different contexts and game types. Additionally, the algorithm uses intuitive parameters and performs in $\mathcal{O}(n)$ time, where $\boldsymbol{n}$ denotes the number of map cells.

The remainder of this article is organized as follows. Section 2, *Related Work*, explores existing studies on PCG for games and general procedural generation. Section 3, *Energy Propagation Algorithm* (EPA), details the algorithm developed, which constitutes

[1] https://blog.slashie.net/on-the-historical-origin-of-the-roguelike-term/

the core of this work. Section 4, *Performance & Applicability Analysis*, presents an analysis of the algorithm's performance through benchmarking and demonstrates its applicability in two game prototypes for map and layout generation. Finally, Section 5, *Conclusions*, summarizes the main findings and implications of the research.

2. Related Work

[Togelius et al. 2016] defined PCG as how computer software can create game content on its own or with human influence, being developed or played by players. The ability to automatically generate content offers a powerful alternative to traditional manual creation methods, offering significant advantages in terms of scalability, adaptability, and innovation potential. However, PCG is not limited to games: studies like [Grossmann 2024] and [Zhou et al. 2025] applied PCG to educational content and large-scale scenes in Blender, respectively. Moreover, the use of PCG techniques in games is not new. As early as 1980, *Rogue: Exploring the Dungeons of Doom* used procedural techniques for dungeon generation [Smith 2024].

Several works have combined traditional algorithms with machine learning models to enhance content generation capabilities, such as in [Summerville et al. 2018]. [Viana e dos Santos 2019] published a survey on *Procedural Dungeon Generation* (PDG), pointing out research opportunities for the generation of 3D levels, blocking player progression, and mixed-initiative design (i.e., humans contribute to PCG content). [Liu et al. 2021] accounted for the fast-paced, growing interest on *Machine Learning* (ML), especially *Deep Learning* (DL), for PCG. They also pointed out limitations and research potential regarding mixed-initiative design, style transfer, personalization, small data, general models, and content orchestration.

[Werneck e Clua 2020] tackled the challenge of avoiding unbalanced, undesired, or mischaracterized PCG content. They resorted to ML to map relevant game design patterns and apply them in the generation, so their approach could generate reliable dungeons that respected room positioning design choices and maintained the game's characterization. A player survey suggested that PCG maps are more enjoyable and replayable than human-made maps.

[da Rocha Franco et al. 2022, da Rocha Franco et al. 2023] combined generative grammars and genetic algorithm search to generate 2D maps. These authors also proposed tools for integrating content for the popular RPG Maker MV (RMMV) game engine, including scenario layouts for JRPGs, map pathways, and cutscene animations. However, using genetic algorithms to balance the generation rules for generative grammars may be tricky, as the authors reported did not reach an ideally balanced result.

[So et al. 2024] presented Dungeoncide, a game using Genetic Algorithms (AG) and *Particle Swarm Optimization* (PSO) for PDG that features a medieval knight fighting monsters. They described a development process resorting to players' feedback on the content's quality and feasibility: generated levels were well-received but considered poorly balanced, and less challenging and immersive than hand-made levels. These outcomes differ radically from those reported by [Werneck e Clua 2020], but highlight similar limitations of GANs [Silva et al. 2025], which, despite being trending and powerful methods, can yield poor results for 2D games.

3. Energy Propagation Algorithm

To the best of our knowledge, EPA differs from other approaches by focusing on generating grid-based maps with a more "fluid" distribution and allowing for fine-grained control over how "closed" or interconnected the internal paths of the map are.

The algorithm operates through recursive functions that distribute a specified amount of energy across the matrix. Such "energy" refers to an input value related to the spread of rooms, such that the number of rooms is intrinsically linked to the initial amount of energy. However, the term was chosen due to the lack of a more appropriate expression, as the value decreases as it spreads through the rooms. This behavior is fundamental to stopping the recursion when the energy value reaches zero.

3.1. Algorithm & Complexity Analysis

The proposed generation technique is implemented by Algorithm 1, which can be divided into two parts. The first is the generation of a $H \times W \times 2$ matrix. This initial matrix is passed as an argument to the energy propagation function to determine the room distribution recursively.

Algorithm 1: Pseudocode of the `generate_map()` function

```
1  Function generate_map(width, height, energy, x, y):
2      Create an empty list called matrix_row
3      for j ← 0 to height − 1 do
4          Add the pair [0, 0] to matrix_row[j]
5      end
6      Create an empty list called matrix
7      for i ← 0 to width − 1 do
8          Add a copy from matrix_row list to matrix[i]
9      end
10     map ← propagate_energy(matrix, x, y, energy)
11     return map
```

The data structure used in this work is a three-dimensional matrix M of dimension $n = H \times W$, where W and H correspond, respectively, to the horizontal and vertical dimensions of the discrete space representing the map. So, n corresponds to EPA's input size for algorithmic complexity analysis purposes. The third index of the matrix contains two positions for each cell (i, j):

- $M[i][j][0]$ stores the energy state of the cell, represented by an integer value, indicating whether the cell has been activated and the intensity of the propagated energy it received.
- $M[i][j][1]$ an auxiliary value encoding the configuration of connections with neighboring cells, used to track the directions in which energy has propagated.

Lines **2** to **9** implement an optimization for creating an empty matrix M in $\mathcal{O}(n)$ time. This approach can be very effective for initializing a grid with default values. Variations of EPA can set some $M[i][j][0]$ to negative values to inhibit certain rooms (i, j) from being connected.

The matrix generated by the `generate_matrix` function is then passed as a

parameter to the `propagate_energy` function (see Algorithm 2), along with a starting position in the matrix and an initial energy value.

Algorithm 2: Pseudocode of the `propagate_energy()` function

```
1  Function propagate_energy(matrix, x, y, energy) returns int:
2      int nw ← number of rows in matrix
3      int nh ← number of columns in matrix
4      if x < 0 or x ≥ nw or y < 0 or y ≥ nh then
5          return 0
6      end
7      if energy ≤ 0 then
8          return 0
9      end
10     if matrix[x][y][0] > 0 and random_number_between_0_and_1() <
         CLOSE_GRID then
11         return 0
12     end
13     matrix[x][y][0] ← 1
14     int v ← energy
15     int a ← propagate_energy(matrix, x − 1, y, energy − 1)
16     int b ← propagate_energy(matrix, x + 1, y, energy − 1)
17     int c ← propagate_energy(matrix, x, y − 1, energy − 1)
18     int d ← propagate_energy(matrix, x, y + 1, energy − 1)
19     cells_around(x, y, x - 1, y, a, matrix)
20     cells_around(x, y, x + 1, y, b, matrix)
21     cells_around(x, y, x, y - 1, c, matrix)
22     cells_around(x, y, x, y + 1, d, matrix)
23     matrix[x][y][0] ← max(v, a, b, c, d)
24     return matrix[x][y][0]
```

The `propagate_energy` function manipulates the matrix to propagate energy across the cells, with this propagation acting as the main stopping criterion for the recursion. Based on this, the code between lines **2** and **9** performs boundary and termination checks.

In lines **10**, **11**, and **12**, a random noise factor is introduced into the matrix structure. The use of a random function in the presented code is intended to simplify the understanding of the algorithm. For more controlled map generation, this randomness can be replaced by customized functions to modify the conditions used.

The value `CLOSE_GRID` is a control variable (i.e., a threshold) containing a float number that determines how "closed" the connections will be. If the value is too small, the number of interconnections will be very large.

In line 13, the current cell is marked as visited, indicating the beginning of the local propagation process. Next (line 14), the current energy is stored in the variable `v`.

Between lines **15** and **18**, the function then recursively propagates the energy to each of the four immediate neighbors (left, right, up, and down), decrementing the energy

by one unit at each step. The results of these recursive calls are stored for subsequent aggregation

In lines **19** to **22**, the `cells_around` function (see Algorithm 3) is called to update the auxiliary connection data $M[x][y][1]$, encoding which of the neighboring cells received the propagated energy. This representation enables the tracking of the directional energy flow between adjacent cells.

Finally, in line **23**, the accumulated value of the central cell is updated with the maximum between the initial energy `v` and the values returned by the recursive calls (`a`, `b`, `c`, `d`), reflecting the highest energy intensity between the origin and the possible propagation paths. The function then returns this updated value (line **24**).

The `cells_around` function performs a series of transformations and data processing steps to convert energy values into connections, which are stored in positions $M[i][j][1]$.

The code snippet presented implements a logic of influence propagation between adjacent cells in a three-dimensional matrix. This propagation starts from an origin cell (x, y) toward a neighboring cell (lx, ly), considering an input value `energy` that represents the intensity of the propagation.

Algorithm 3: Update propagation between neighboring cells

```
1  Function cells_around((x, y, lx, ly, energy, matrix)) returns void:
2      if energy = 0 then
3          return
4      factors ← multiply_around(x, y, lx, ly);
5      if factors[0] = −1 or factors[1] = −1 then
6          return
7      if matrix[x][y][1] = 0 then
8          matrix[x][y][1] ← 1
9      if energy ≥ 1 and level[lx][ly][1] < 1 then
10         matrix[lx][ly][1] ← 1
11     vxy ← matrix[x][y][1] × factors[1];
12     vlxly ← matrix[lx][ly][1] × factors[0];
13     if verify_factor(vxy) and verify_factor(vlxly) then
14         matrix[x][y][1] ← vxy;
15         matrix[lx][ly][1] ← vlxly;
```

The function `cells_around`$(x, y, lx, ly, energy, matrix)$ is responsible for updating the second channel of the three-dimensional matrix `matrix`[x][y][1] based on multiplicative and conditional rules associated with the neighboring cell (lx, ly).

The function `multiply_around`(x, y, lx, ly) is called to determine two multiplicative factors associated with the relative position between the cells. If either of the factors is invalid (i.e., equals -1), the function also returns.

The function `verify_factor` in line 13 is used to validate both products. If

both are valid, the updated values are written back to the `matrix` matrix. In practical terms, the `verify_factor` function checks whether the given number is part of the permutation of possible values derived from the product of the prime factors $2 \times 3 \times 5 \times 7$. In other words, it verifies if the number is included in the following list of values: $[2,\ 3,\ 5,\ 6,\ 7,\ 10,\ 14,\ 15,\ 21,\ 30,\ 35,\ 42,\ 70,\ 105,\ 210]$.

Algorithm 4: Determine multiplicative factors based on relative position

Function `multiply_around`$((x, y, lx, ly))$ ***returns*** *list of integers*:
  $POS\_MAP \leftarrow \begin{bmatrix} -1 & 3 & -1 \\ 2 & -1 & 5 \\ -1 & 7 & -1 \end{bmatrix}$
  $dx \leftarrow x - lx + 1$;
  $dy \leftarrow y - ly + 1$;
  if $dx = 0$ ***or*** $dx = 2$ **then**
    return $[POS\_MAP[dx][dy],\ POS\_MAP[2 - dx][dy]]$;
  if $dy = 0$ ***or*** $dy = 2$ **then**
    return $[POS\_MAP[dx][dy],\ POS\_MAP[dx][2 - dy]]$;
  return $[-1,\ -1]$;

The function `multiply_around`(x, y, lx, ly) determines the multiplicative factors based on the relative position between the cell (x, y) and its neighbor (lx, ly) (see Algorithm 4). A fixed mapping matrix, `POS_MAP`, associates spatial directions with specific integer constants—typically prime numbers or their multiples. This setup was determined experimentally after testing the outcomes in actual games.

For example, a cell to the left might be associated with factor 2, and one to the right with factor 5. This approach enables efficient representation of connection permutations using only integer multiplication, avoiding the need for more complex data structures.

Prime numbers are used intentionally, as the product of distinct primes represents their Least Common Multiple (LCM). For example, a value of 210 represents connections in all four directions because $2 \times 3 \times 5 \times 7 = 210$. In contrast, a value of 6 (i.e., 2×3) would mean connections only to the left and the top.

This encoding strategy allows for fast computation and logical reasoning through common operations such as multiplication and division. If the direction is invalid or outside the bounds of the mapping matrix, then the function returns $[-1, -1]$, signaling an invalid direction.

The algorithm's performance can be analyzed as the sum of two main components. First, the creation of the matrix $M[i][j]$ has a time complexity of $\mathcal{O}(n)$. Second, the recursive behavior of the `propagate_energy` function is aware of previously visited cells. As such, the propagation step can be approximated as $\mathcal{O}(\text{energy} \times \log(\text{energy}))$. However, this execution is already bound by the $\mathcal{O}(n)$ matrix dimensions. The remaining functions in the algorithm are constant time.

3.2. Controlling the Generated Maps

Although the algorithm allows for a more stable distribution of connections, it limits precise and directed control over the resulting map. Essentially, the algorithm's output can be influenced by three main parameters: the size of the grid, the initial energy input — which, if set too high, leads to a filled map — and the degree of interconnectivity between the map's sub-connections, which is controlled by the `CLOSE_GRID` variable.

4. Performance & Applicability Analysis

The evaluation focused primarily on performance aspects, through a benchmark of the algorithm implemented in Python to analyze its execution time with different input sizes and its integration into game prototypes. Usability and acceptance analyses were intentionally avoided due to divergent results found in the literature. Moreover, such results can be strongly influenced by the context of the product or game itself, and how well the PCG technique is incorporated into the specific design and purpose of the game.

4.1. Performance Benchmark

All experiments were conducted on a personal computer with the following specifications: Windows 11, AMD64 architecture, AMD Ryzen processor (Family 23, Model 104, Stepping 1), 7.35 GB of available RAM, and Python 3.10.5. Our Python implementation of EPA is publicly available at GitHub[2].

To evaluate the performance of EPA, a benchmarking experiment was conducted using 100 different input configurations. Each configuration used an odd integer in the range $[1, 199]$, which was simultaneously applied to the `width`, `height`, and `energy` parameters. This allowed a consistent scaling of the input grid while keeping the problem symmetric and centered. The initial starting point of the algorithm was fixed in the center of the grid for all executions.

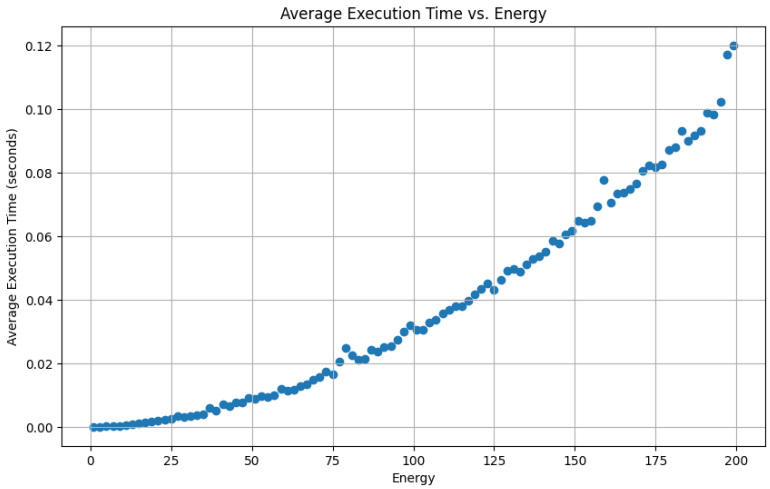


Figure 1. Average Execution Time vs. Energy. Each point represents the mean execution time (over 100 runs) for a given odd value of width = height = energy.

[2]`https://github.com/maurovictor027/py-epa`

The benchmarking results are summarized in Figure 1. As expected, the average execution time increases with the energy value, which in this setup also determines the size of the map. Since the grid is implemented as a matrix, its initialization unavoidably incurs a practical $\mathcal{O}(n)$ cost, as each cell must be explicitly allocated and initialized. This is a fundamental limitation of using an array structure — regardless of optimization, the memory layout requires "touching" all cell positions. The curve in the benchmark results confirms this, showing a near-linear execution time as the matrix size grows.

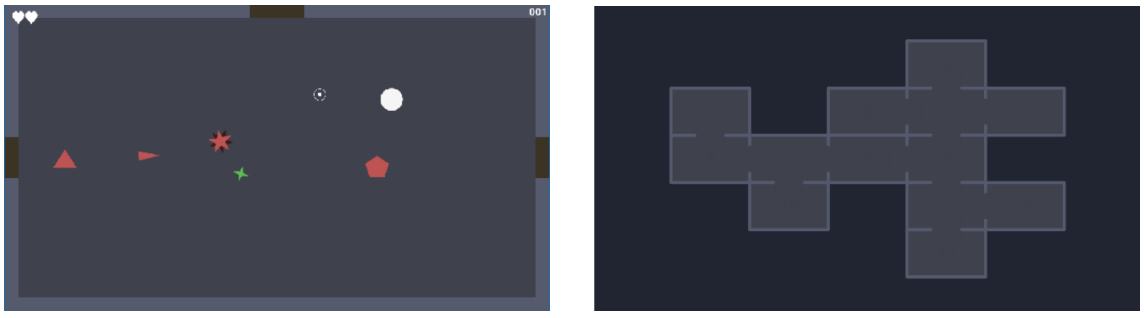
Memory usage was not explicitly measured in this experiment, but the algorithm remained within acceptable memory limits on all input sizes tested (up to 199×199 grids). Future iterations of this benchmark could benefit from explicit memory profiling tools such as `memory_profiler` or `tracemalloc` in Python.

4.2. Prototype Games

Two games were developed based on the algorithm. The first one, *Shape Dungeon*, applied the algorithm for procedural map generation. The second game, *Grid Shift*, aimed to test the algorithm's applicability in a puzzle game context.

4.2.1. Shape Dungeon

The project features different types of enemies, the possibility of acquiring upgrades for the player character, and a final boss. The implementation of the algorithm in this game was straightforward: it only required taking the resulting matrix and converting the data into rooms. This was achieved by transcribing the values of $M[i][j][1]$ into doors that connect the rooms. The rooms' structure and connections were created dynamically, ensuring replayability and increasing complexity as the player progressed.



(a) (b)

Figure 2. In image A a print of Game Screen, with player (white) and enemies (red). In image B, a visual representation of the game map

Moreover, the map generation process benefited from the ease of controlling the map size using the variables `width`, `height`, and `energy`, allowing for increasingly larger maps as the player progressed through the levels: 4×4 for level 1, 5×5 for level 2, and 6×6 for the final level. To ensure more densely connected maps, the energy value was always set higher than the map width — especifically 5, 6, and 8 for each level.

It is important to highlight that, due to the lack of time to implement a minimap for the game, the possibility of interconnection between rooms after generation was removed;

this was done by setting the CLOSE_GRID variable to a value greater than 1.0. As a result, the dungeon rooms were generated in a structure analogous to a tree — that is, only the "parent" rooms determine the connections with their "child" rooms.

4.2.2. Grid Shift

In this game, the algorithm was responsible for generating layouts composed of rotated pieces, starting like image A in Figure 3. The player's goal is to find the correct rotation for each piece to connect all the grid elements successfully. The idea for the game emerged as a way to test the algorithm's applicability in the context of a puzzle generator.

As such, there were no major issues in integrating the algorithm, with only a few parameter adjustments required—particularly regarding the CLOSE_GRID variable—, to generate maps with some closed areas, as illustrated in images B and C of the figure below. However, this behavior was fine-tuned to avoid excessive occurrence. After several tests, a value of 0.92 was chosen. The game grid has a size of 10×6, and an energy value of 16 is applied to a random cell, excluding the map's borders, to generate the puzzle layout. Below, images of different levels generated by the algorithm:

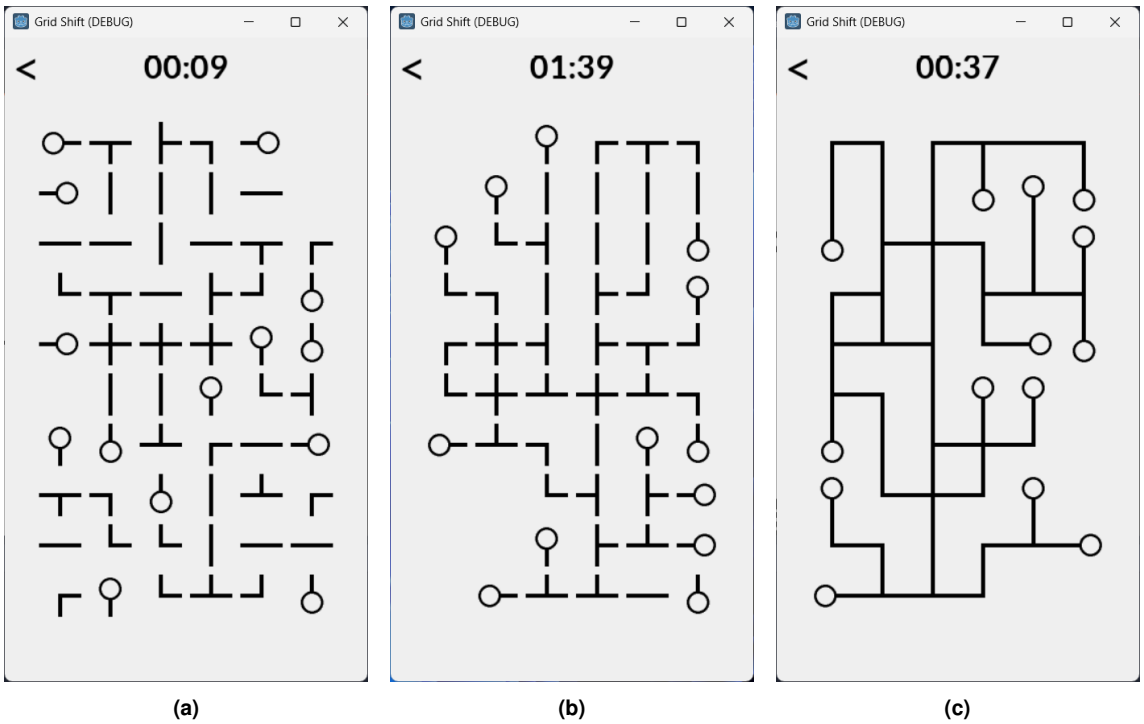


Figure 3. In image A it is possible to see a game initially, in image B an almost completely resolved layout and in C after connecting all the cells

4.3. EPA for 3D Scenarios

Adapting the algorithm for a 3D operation represents a natural evolution of 2D map generation. This required expanding the dimensions of the matrix to $W \times H \times D$, while preserving the basic principles of energy and connectivity. Furthermore, the auxiliary matrix POS_MAP was updated to support three-dimensional orientations by expanding

the matrix as a $3 \times 3 \times 3$ tensor and incorporating two new prime numbers (11 and 13) in the two new depth layers. This tensor leverages permutation values, as illustrated in Figure 4a.

For presentation purposes, we performed initial tests by developing a basic 3D map visualizer in the Godot Game Engine, so each map cell is represented by a cube. The preliminary results of this 3D conversion are depicted in Figure 4b.

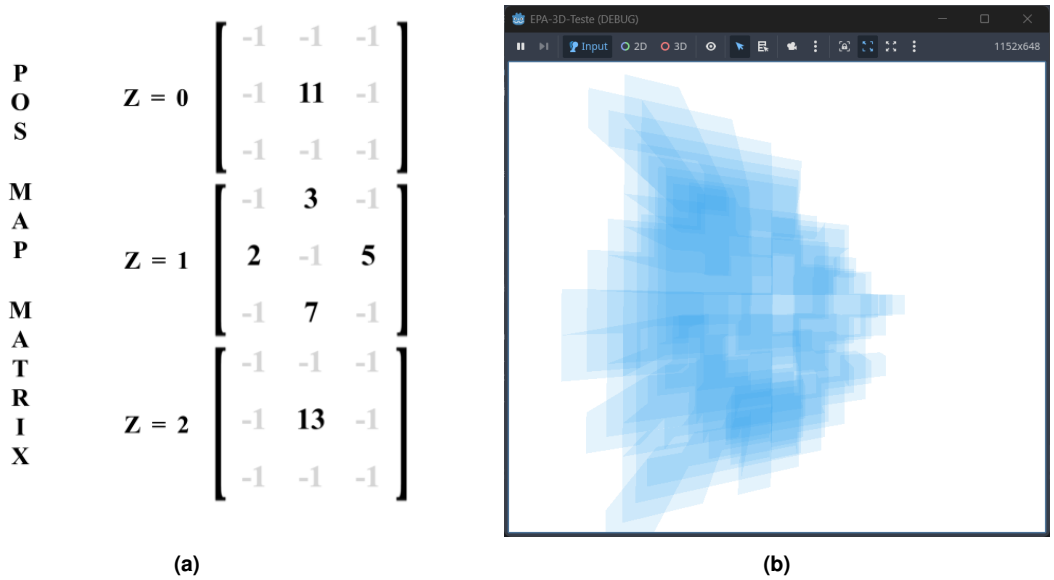


Figure 4. A visualization of the POS_MAP matrix used for operations in 3D spaces (left). The grid generated by the three-dimensional variation of the algorithm for a $12 \times 12 \times 12$ map with $energy = 18$ (right).

5. Conclusions

Despite the promising results regarding the EPA's applicability in games, it is important to highlight that the main focus was to address the algorithm's potential, plus its innovative and peculiar characteristics. As it is a new idea, no usability or user acceptance tests were reported with the game prototypes. It is reasonable to admit that usability results may be questionable, as they can be influenced by the quality of implementation — regardless of the algorithm's inherent potential.

Moreover, the works presented in this paper were limited to two-dimensional games with four possible directions. Other approaches still need to be explored in future work, such as the application of the algorithm to hexagonal layouts.

References

da Rocha Franco, A. d. O., da Silva, J. W. F., Maia, J. G. R., e de Castro, M. F. (2023). Harnessing generative grammars and genetic algorithms for immersive 2d maps. *Entertainment Computing*, 47:100595.

da Rocha Franco, A. d. O., Franco, W., Maia, J. G. R., e Franklin, M. (2022). Generating rooms using generative grammars and genetic algorithms. In *2022 21st Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 1–6. IEEE.

Grossmann, L. (2024). A literature review on the educational use of procedural content generation across disciplines.

Liu, J., Snodgrass, S., Khalifa, A., Risi, S., Yannakakis, G. N., e Togelius, J. (2021). Deep learning for procedural content generation. *Neural Computing and Applications*, 33(1):19–37.

Nasir, M. U. e Togelius, J. (2023). Practical pcg through large language models. In *2023 IEEE Conference on Games (CoG)*, pages 1–4.

Parker, R. (2017). The culture of permadeath: Roguelikes and terror management theory. *Journal of Gaming & Virtual Worlds*, 9(2):123–141.

Shaker, N., Togelius, J., e Nelson, M. J. (2016). *Procedural content generation in games*. Springer.

Silva, D. F., Torchelsen, R. P., e Aguiar, M. S. (2025). Procedural game level generation with gans: potential, weaknesses, and unresolved challenges in the literature. *Multimedia Tools and Applications*, pages 1–27.

Smith, M. (2024). 6 earliest games that used procedural generation. `https://gamerant.com/earliest-games-procedural-generation/`.

So, A. R. P., Souza, A. C. C., Costa, L. M., Mantovani, R. G., e Souza, F. C. M. (2024). Design and evaluation of a procedurally generated dungeon game. In *Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames)*, pages 329–339. SBC.

Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., Nealen, A., e Togelius, J. (2018). Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270.

Togelius, J., Shaker, N., e Dormans, J. (2016). Grammars and l-systems with applications to vegetation and levels. In *Procedural Content Generation in Games*, pages 73–98. Springer.

Viana, B. M. e dos Santos, S. R. (2019). A survey of procedural dungeon generation. In *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 29–38. IEEE.

Werneck, M. e Clua, E. W. (2020). Generating procedural dungeons using machine learning methods. In *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 90–96. IEEE.

Zhou, M., Wang, Y., Hou, J., Zhang, S., Li, Y., Luo, C., Peng, J., e Zhang, Z. (2025). Scenex: Procedural controllable large-scale scene generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 10806–10814.