CEIG - Spanish Computer Graphics Conference (2025)
O. Argudo and O. Iparraguirre (Editors)

Procedural Generation of 3D Maps with Wave Function Collapse: Optimization and Advanced Constraints

M. B. Villar López[1] and M. Chover[1]

[1]Universitat Jaume I, Institute of New Imaging Technologies (Castellón, Spain)

Abstract

The Wave Function Collapse algorithm is a widely used Procedural Content Generation technique for creating structured scenarios using local neighborhood constraints. This work presents an extension of the algorithm to generate three-dimensional scenarios, incorporating non-local constraints and key optimizations. The proposed improvements include assigning weights to tiles, layer-based generation, specific appearance constraints for unique or ranged tiles, and an automated neighbor creation and assignment method using connectivity rules. These modifications facilitates the generation of coherent and structured 3D environments, providing greater control and adaptability to the process. Finally, some optimizations are proposed and the approach's effectiveness is evaluated analyzing the impact of constraints on the algorithm's coherence, diversity, and runtime.

CCS Concepts

• ***Computing methodologies*** → ***Modeling and simulation; Computer graphics;*** • ***Applied computing*** → *Computer games;* • ***Theory of computation*** → *Constraint and logic programming;* • ***Software and its engineering*** → *Software performance;*

1. Introduction

Procedural Content Generation (PCG) has become a fundamental tool in video game development and virtual environments due to its ability to create diverse content quickly and efficiently. PCG techniques have been used in video games, virtual environments, textures, vegetation, buildings, terrain, architecture, and more [ZZH22]. They are highly useful for generating content rapidly as well as implementing replayability in video games so that each time the player starts a game the level is different [MC23]. In *Minecraft*, the second-best-selling game in the world after *Tetris* [Sta24], maps are generated using *Perlin Noise* [Per85] to create infinite maps [YD24]. In *dungeon* and *roguelike* games like *Hades [Sup18], The Binding of Isaac [McM11], or Enter the Gungeon [Dod16]*, procedural dungeon generation is an essential gameplay feature. However, PCG is also used for offline content creation, not during gameplay but as a tool for developers during game design, as seen in *Skyrim*'s map [Bet11]. There are multiple PCG techniques, but this article focuses on the Wave Function Collapse (WFC).

In this context, the WFC algorithm is relatively recent. Initially conceived for procedural texture synthesis from a sample image, it can also generate structured maps by propagating neighborhood constraints based on an initial collection of tiles and adjacency rules. Its main advantages are the artistic and creative freedom it offers users to design their own tiles, unlike other PCG techniques like *Perlin Noise*, and the simplicity of its basic concept. However, two major drawbacks are its high execution cost and lack of control over the final result, as the constraints are local rather than global [CHF20] [Sri25] [CKT*24] [Lux24].

The algorithm is generally used for two-dimensional maps. Extending this algorithm to three-dimensional environments poses significant challenges due to the exponential increase in computational complexity and the need to adapt neighborhood rules. This work proposes a three-dimensional adaptation of WFC by implementing a series of improvements.

First, a set of non-local constraints will be presented.

- **Weighted tiles** are used to assign probabilities to each tile, this way the user can decide the amount of a certain type of tile in the final result.
- **Generating layers of tiles** before the algorithm runs (e.g., a floor and a ceiling) allows more control of the final result without strange outcomes like holes on the ground.
- **Fixed tiles** are used to determine how a part of the map is before the algorithm executes. This way, the user can define a starting point and with WFC the rest is completed in a coherent way. It can also be more flexible without specifying the exact location of the tiles, allowing them to be placed randomly before the execution of the algorithm starts. For example, a map that contains from 1 to 10 tiles of a specific building.
- **Excluded tiles** are used to restrict more the final result, avoiding certain tiles from appearing together even if they could fit together.

DOI: 10.2312/ceig.20251107

Furthermore, a system for **automatically generating tile variants** from an original tile and **an automated neighbor assignment** using connectivity rules have been implemented. Additionally, optimizations are proposed and results are analyzed to improve the algorithm's runtime.

2. State of the art

The WFC algorithm was first introduced by **Maxim Gumin** in 2016 [Gum16] as a texture synthesis algorithm. The algorithm reads a sample texture and detects patterns, which it then uses to create new versions of that texture while maintaining the detected patterns through "overlapping" [Gri19]. Gumin also introduced a version for tiles ("Simple Tiled Version"), where the algorithm does not recognize patterns in a texture but instead uses a set of tiles provided by the user, along with rules specifying which tiles fit together. This turns it into a constraint propagation algorithm. Thus, unlike the previous version, this one requires not only a set of tiles but also adjacency rules that define which tiles can be placed next to each other.

Gumin's Wave Function Collapse (WFC) algorithm builds on the foundation laid by **Paul Merrell's Model Synthesis** [Mer07], published in 2007, which also focuses on procedural generation. While the two algorithms are conceptually similar, they differ in focus and implementation. The synthesis model is more oriented toward creating 3D models and architectural structures, whereas Gumin's WFC is more focused on procedural texture synthesis and 2D tile-based environments.

A key distinction between the two algorithms lies in how each one selects which cell to collapse. Merrell's model follows a sequential approach, processing cells in a fixed order, while WFC uses entropy-based selection (choosing the cell with the fewest possible tile options). This difference affects the final result, but WFC tends to struggle with large-scale maps. In contrast, Merrell's algorithm processes the map in smaller blocks, dividing the problem into smaller parts. Another notable difference is Gumin's use of **overlapping texture synthesis**, a technique not present in Merrell's approach. Merrell himself analyzed the distinctions between the two methods, highlighting how their different priorities shape their respective strengths and weaknesses. [Mer21].

Currently, one of the most prominent figures studying this algorithm is **Oskar Stålberg**, known for developing video games like Townscaper (2021) [Sta21] and Bad North (2018) [Sta18]. These works significantly contributed to the study of WFC and popularized it. Stålberg took the algorithm to another level combining it with *Marching Cubes* [WH87]. He also developed a technique to create grids with four-sided cells (suitable for WFC) that look irregular and organic [Sta20], and made several public demos exploring WFC and procedural texture synthesis [Sta17]. Another recent video game that uses this technique for map implementation is *Caves of Qud* [Fre24], a roguelike inspired by Dungeons & Dragons.

There are also some plugins and libraries, such as *Tessera* or *DeBroglie* [Bor20], both created by "Boris the Brave". His website explains many concepts of WFC, along with documentation and possible constraints that can be added, which inspired this research. Regarding the application of WFC to 3D grid maps and infinite generation, one of the most notable examples is **Marian Kleineberg**, who created an infinite city using this algorithm with over 100 different tiles. Marian laid the groundwork for the 3D adaptation of WFC, introducing concepts discussed in this paper, such as special tiles (solid/empty) and additional neighbor constraints [Kle19].

3. Wave Function Collapse Algorithm

The WFC is defined as "a constraint satisfaction solver that uses data-driven methods to search a generative space" [SCM19]. It is an algorithm that given an input (such as a collection of tiles) and a set of constraints (adjacency rules), can generate a coherent space. There are two variants: Overlapping WFC and Tile WFC.

- **Overlapping WFC:** Creates an image from a small sample by following the same patterns, often used to create procedural textures.
- **Tile WFC:** The algorithm studied in this article. It generates structured maps based on tiles and adjacency rules given by the user.

The algorithm main components are:

- **Tiles:** The fundamental pieces that conform a map or pattern. These can be images or 3D models. Tiles are placed next to each other to create the final result. They fit together based on neighborhood rules. Following these rules ensures that the final result is coherent.
- **Neighbors:** Local constraints that determine which tiles can be adjacent. Each tile has a set of allowed neighbors for each side, defining how they can be arranged. For example, a 2D tile of a horizontal path can only be adjacent to tiles with a path on their edges.

3.1. Algorithm behaviour

The possibility space is represented as a **wave** (hence the name Wave Function Collapse). This wave is a matrix where each cell contains all the possible tiles it can have. To select the cell to collapse, the algorithm looks for the one with the least entropy —that is, the fewest remaining options. The process is as follows:

1. **Initial State:** At the beginning, all positions have all options open. This means all cells have the maximum entropy, so a random cell is selected in the first iteration.
2. **Selection:** The cell with the lowest entropy is selected, and a tile is randomly chosen from the stack at that position. This removes all other tiles from that position, leaving only the selected tile.
3. **Constraint Propagation and Collapse:** Once a cell is collapsed, neighborhood constraints come into play. If the selected tile has restrictions on which tiles can surround it, these constraints affect the adjacent cells. The possibilities of the neighbors are reduced accordingly, eliminating tiles that do not respect the constraints.
4. **Cascade Effect:** This reduction in neighbor possibilities can, in turn, reduce the possibilities of other adjacent cells. This propagation continues until all cells have been visited.

5. **Handling inconsistencies:** This process continues until no further constraints can be propagated, and all cells are collapsed. If a cell runs out of possible tiles, the algorithm fails because there is no valid configuration where each cell has at least one possible tile. In this case, a backtracking algorithm can be applied to retrace steps, or the map can be regenerated, which is often the best option for non-infinite spaces.

4. WFC improvements

In this section, a method for automatic neighbors operations will be presented. In addition, a variety of global constraints will be proposed.

4.1. Automatic neighbors generation and assignation

To optimize the process and avoid the manual creation of unnecessary tiles, **tiles with symmetries can generate new tiles automatically from themselves**. Each one has attributes in its class that define these symmetries. For example, if there is a tile corresponding to a corner, the other three corners can be automatically generated without needing to import them manually. This is achieved by instantiating a new tile identical to the original, rotating it by the required angle, copying its neighbors to the corresponding side, and adding it to the list of all tiles. For instance, if it creates a new tile B by rotating tile A 90 degrees to the right, tile A's right face configuration now corresponds to the bottom face in tile B.

On the other hand, writing all adjacency rules manually is a tedious task. To address this problem, **a method has been developed to assign neighbors automatically** by simply defining each tile's edge type. For example, if a tile has a "grass" edge on its right, it will only fit with tiles that have a "grass" edge on their left.

This system significantly simplifies the definition of rules and facilitates the procedural generation of environments. However, its application in 3D environments has required certain additional modifications to guarantee the correct assignment of neighbors in all axes, explained in section 5.1.

4.2. Non-Local Additional Constraints

As mentioned earlier, one of the disadvantages of this algorithm is that it only features local constraints (adjacency rules for neighbors). This results in a global outcome that is too random and offers little control to the user. To address this, a series of non-local constraints have been proposed.

- **Weight Modification:** Weights and entropy provide greater control over the final result for the user. By implementing a weighted selection method, where tile selection is based on user-defined weights, the generated map better represents the probability of each tile's appearance. For example, if "grass" is given more weight than "path", there will be more fields than paths, which is more realistic. This is achieved by combining binary search and the cumulative weight method. (Figure 1).
- **Fixed Cells:** Certain cell values can be defined before the algorithm generation process begins. This allows parts of the map to be set manually and the algorithm to fill in the rest consistently. For example, the user can design certain paths first and let the algorithm fill in the surroundings, ensuring that the base structure of the map is consistent.
- **Layers:** Generating layers can be used to modify the final result both before and after build. Before generation, it is possible to establish base layers that serve to structure the terrain and avoid connectivity problems. For example, on the first floor, a layer can be generated completely occupied by underground ("solid"), ensuring that there are no holes or empty spaces. Similarly, on the top floor, a layer of air ("empty") can be generated, preventing the appearance of open mountains or floating structures. Borders can also be added around certain layers to create beaches, transitions between biomes, or other elements that improve the visual coherence of the map. After generation, elements that are not tiles (e.g., chests or keys) can be added based on layer constraints. These elements usually only appear on specific tiles (e.g., grass or ground tiles, but not water or walls). Inter-layer rules are required to handle these constraints. For example, if a water tile is placed, the algorithm must mark that a chest cannot appear on the layer above it in that cell.
- **Tiles with Maximums and Minimums:** Two variables can be used: "global maximum" and "global minimum." The first refers to the maximum number of a specific tile type that can appear on the map (e.g., at most 20 water tiles). The second defines a minimum number of tiles in specific positions or randomly. This concept allows the creation of tiles that must appear exactly once (e.g., a specific building) or a fixed number of times (Figure 2).
- **Excluded Tiles:** Sometimes, tiles that technically fit together according to adjacency rules might look visually unappealing. Therefore, an exclusion system is implemented for each tile on each side. For example, two walls should not be placed one behind the other. Instead, it would be better to leave a space with grass between them (Figure 3).

The combination of all these improvements provides developers with greater creative and design freedom, resulting in more controllable and visually coherent outcomes.

5. Implementation and 3D adaptation

The project has been implemented in Unity 6000.0.29f1 using C#. For testing, the hardware setup included an Intel i7-14700K processor (28 CPUs) at 3.4GHz, 32GB of RAM, and an NVIDIA GeForce RTX 4070 Ti SUPER graphics card. The implementation followed recommendations from W. Tristan [Tri22] and Marian's blog [Kle19]. Compared to the two-dimensional algorithm, the following modifications were necessary:

- Adapting the grid to incorporate a third dimension.
- Modifying neighborhood rules to include relationships between upper and lower tiles.
- Implementing an advanced system to configure connectivity for the six faces of each tile.

The models used were created in Blender (Figure 4). 14 objects were designed, themed around a natural landscape. The "grass" (base) tile was modeled with some depth to create higher terrain levels and allow edges to be placed around them. For the 3D implementation, a "solid" tile (for underground) and an "empty" tile

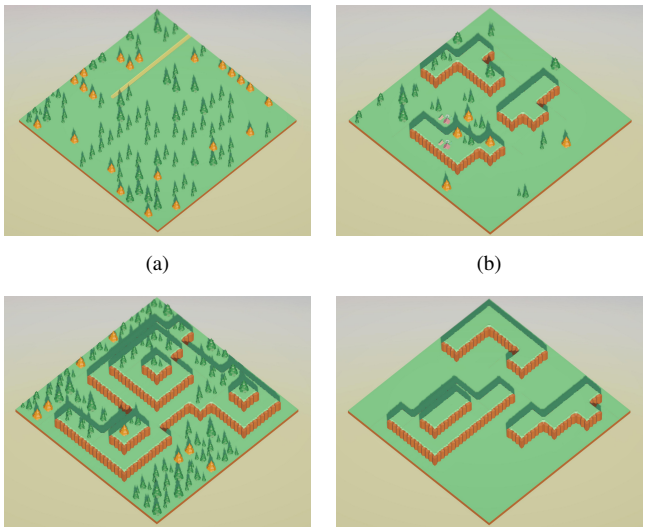

Figure 1: *Different approaches of the same map with different tile weights. (a) Low probability on mountains (b) Low probability on trees (c) More probability to mountains and trees (c) More probability to grass rather to trees*

(for air) were created, both without geometry, bringing the total to 16 tiles. These invisible tiles are essential because, in a 3D grid, every cell must be filled, even if it represents empty space or solid ground. For example, a "grass" tile can only have an "empty" tile above it and a "solid" tile below it, ensuring logical and coherent placement of visible tiles in the 3D structure. This approach also prevents inconsistencies like floating tiles or holes in the terrain. Therefore, when a cell is collapsed to an empty tile, all the above cells are collapsed to empty also, so it is not possible to create for example a cliff in a C shape.

5.1. Preprocessing using *sockets*

In the preprocess part, the tiles will be analyzed and a series of algorithms will be executed to prepare them for the WFC algorithm. First, it is important to have the essential components of the algorithm clear. The map grid is composed of **cells** that have tile options. If the cell is collapsed, it only has one tile option. **Tiles** are the models that can be instantiated in each cell. In 3D, **each tile has 6 faces**: up, right, left, down, front and back. Each face has neighbors (up neighbors, right neighbors...). (Figure 5);

- **First step: Automatic tiles generation**: This algorithm, explained in section 4.1, generates automatically all the tile variations so the user does not have to manually create and import all of them. The tiles that have symmetries in this case are the wall, corners, border, border corners, and paths. After this algorithm, the number of tiles goes from 16 to 44 tiles.
- **Second step: Automatic neighbors assignation**: The next step is to generate the connectivity rules between the different tiles. In 3D, this method becomes even more essential. However, more information is needed beyond just the edge type. To address this, structures called "*sockets*" were created. With this method, the user does not need to manually assign all neighbors on all 6 faces of all tiles. Instead, the user only needs to define the sockets before the WFC algorithm executes.

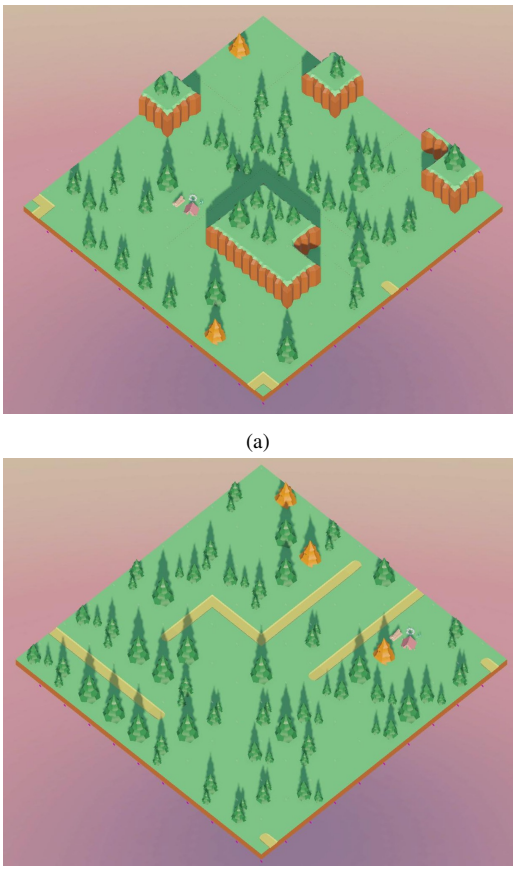

Figure 2: *Generating a map with fixed tiles: 1 campfire and between 1 and 4 orange trees*

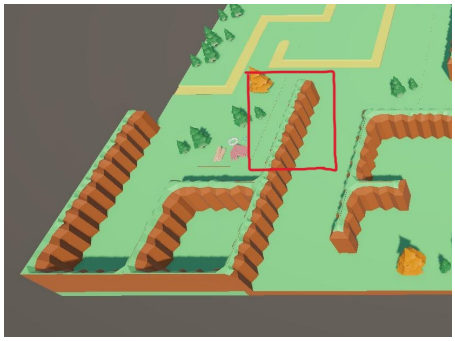

Figure 3: *Generation without excluded neighbors constraint*

Sockets are a structure of data that define the configuration of each face for every tile, and it is used to determine which faces can be adjacent between each other. This means that every

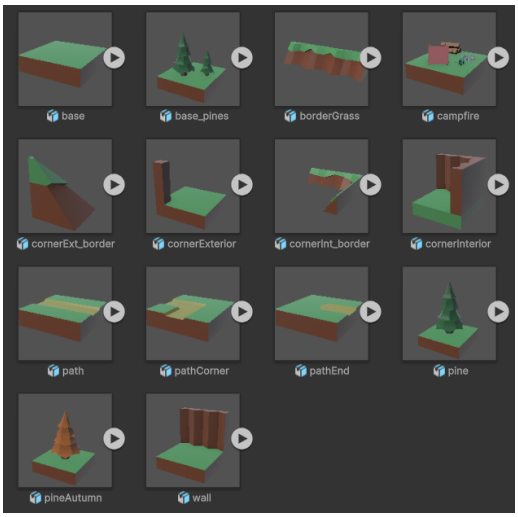

Figure 4: *The 14 models created.*

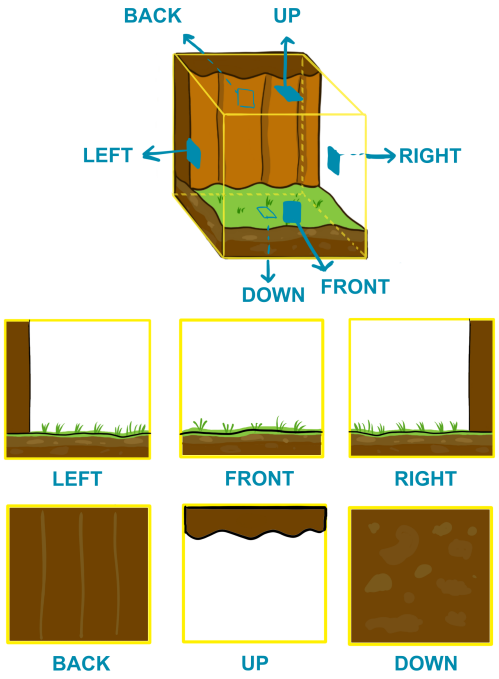

Figure 5: *The 6 faces of a wall tile.*

face of every tile has a socket, so **each tile has 6 sockets**. Each socket contains 2 main variables: the type of edge or "socket name" (grass, wall...) and the type of socket: horizontal or vertical. The up and down faces have vertical sockets, and the rest of the faces have horizontal sockets (Table 1).

These structures are used in the automatic neighbors assignation algorithm (second step) before the generation starts. All the faces of all the tiles are revised, and if two faces meet a certain socket condition, they can fit together and they are added as possible neighbors options in both tile's faces.

Table 1: *Wall tile sockets example.*

(a) *Horizontal*

Face	Name	Symmetry	Face type
LEFT	wall_lateral	No	Flipped
FRONT	grass	Yes	-
RIGHT	wall_lateral	No	Original
BACK	solid	Yes	-

(b) *Vertical*

Face	Name	Invariant rotation	Index
UP	wall_top	No	0
DOWN	solid	Yes	-

For two faces to be neighbors, the main condition is that **they must have the same socket name**. For example, a "grass" face can only join with another "grass" face. However, the orientation is also important, and that is why there are socket types. In addition to fulfill the same socket name condition, they must follow these rules:

- **Horizontal Faces**: For tiles to fit, they must meet one of two criteria:
  - Be symmetrical (e.g., the grass tile). This means that the orientation of the face does not matter.
  - If not symmetrical, one face must be the original face, and the other must be the flipped face (e.g., for walls). (Figure 6).
- **Vertical Faces**: For tiles to fit, they must meet one of two criteria:
  - Have invariant rotation (e.g., the grass tile).
  - Have the same rotation index. For instance, a corner tile may have four possible rotations, each with its index.

To sum up, each tile has a socket per face. The socket has a name and a type. If two faces have the same socket name and fulfill the previous orientation conditions, they can be neighbors. It is crucial to configure sockets correctly for the algorithm to work as intended. This is likely the most challenging part of the design.

After this step is completed, the sockets are no longer used and the WFC algorithm can start (Figure 7).

5.2. The Wave Function Collapse algorithm

Once neighbors have been created and assigned, and the layer constraints have been set, the grid space for the scenario is prepared. The grid consists of cells, each capable of holding multiple possible tiles but ultimately collapsing into one. Each cell tracks whether it has been collapsed and maintains an array of possible tiles that dynamically changes during execution.

The grid is implemented as a one-dimensional array for optimization purposes, with indices calculated as follows:

$$index = x + (z * dimensionsX) + (y * dimensionsX * dimensionsZ)$$

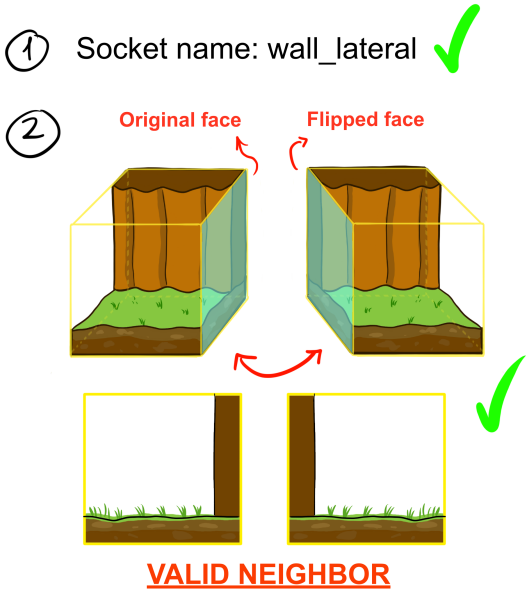

Figure 6: *Sockets configuration example.*

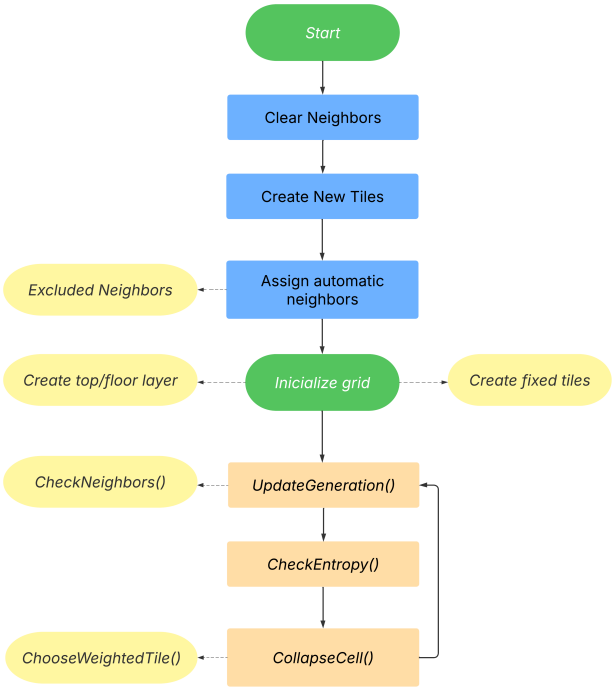

Figure 7: *Main program flow.*

5.2.1. Check neighbors

After all preparation steps are completed, the main loop function is called. On every new iteration (including the first), this function checks the entire map to update the possible options for each cell based on collapsed neighbors. If the cell is already collapsed, its state remains unchanged. If the cell is not collapsed, each neighboring cell is evaluated.

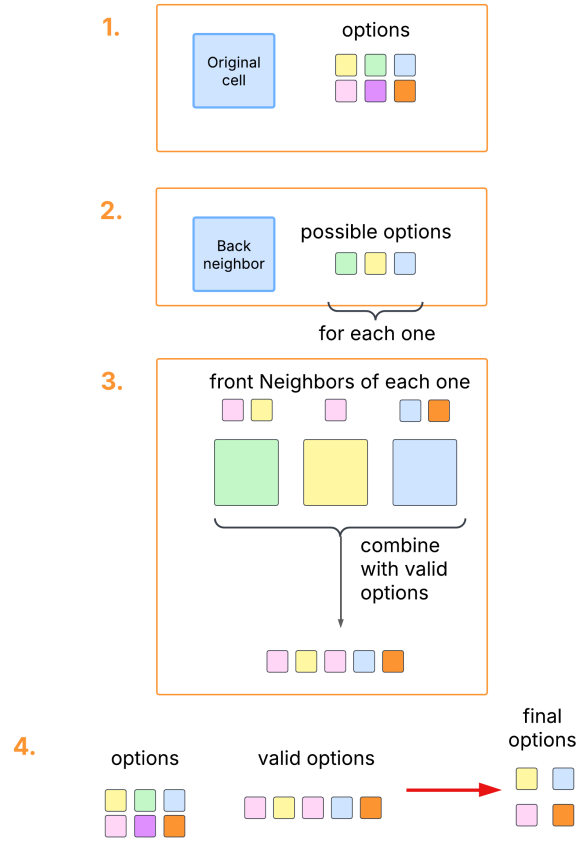

Figure 8: *Process of checking back neighbors in a cell.*

For example, this is the process to check the back neighbor of a cell (Figure 8).

1. **Create a list of possible options** for the cell: Initially, all existing tiles are included as options. Then, choose a neighbor (the back neighbor).
2. **Evaluate each possibility for the neighbor on that side**. Within each of those possible tiles, check its corresponding tiles on the side facing the current cell (their front neighbors). Combine compatible tiles into a list of valid options for the current cell.
3. **Eliminate incompatible options:** Check which elements of the original list are not in the valid options list and remove them.
4. **Repeat for all sides**.

When there are no more sides to check, the list of options becomes the new list of possibilities for that cell.

5.2.2. Calculate entropy and collapse the cell

Once all the cells are updated, the entropy is checked. The cells that have already been collapsed are not considered and the next cell to collapse is the one with the lowest entropy (i.e., the fewest

remaining options). After this, the function that will collapse the cell is called.

A tile is selected from the possible options of the chosen cell, taking into account its weight. If no valid tile exists, it is because there is an incompatibility and in that cell there is no possibility to fulfill the algorithm. In this case the entire map is regenerated which is usually faster if it is not an infinite world.

If successful, the selected tile becomes the only option for that cell. The tile is instantiated in the cell, and the main loop is called again. The loop ends when the total iterations matches the grid dimensions *(dimensionsX * dimensionsZ * dimensionsY)*.

6. Optimization and results

The WFC algorithm becomes more computationally expensive as the map size and the number of possible tiles increase. For large number of tiles and grids, it can become nearly impossible to complete. To address this, an optimization strategy was implemented: instead of propagating changes across the entire map in each iteration, **only the cells immediately adjacent to collapsed cells are updated**. This drastically reduces the number of cells that need to be checked on each iteration. Furthermore, implementing this optimization does not increase the number of incompatibilities.

On the other hand, beyond optimization, the algorithm uses an entropy-based selection. This means that in each iteration, a cell with the fewest tile options is chosen. However, using a **sequential order** instead of **entropy-based selection** significantly reduces execution time though it may affect the final result's appearance, as the WFC algorithm was not originally designed for sequential processing. This sequential approach is closer to its predecessor, the synthesis model by Paul Merrell in 2007. The reason why WFC employs entropy instead Merrell's sequential method is to reduce the possibility of incompatibilities and inconsistencies, as it always collapses the cell with the "safest" outcome (i.e., the one with the fewest possible tile options). This way, the propagation is more controlled. However, Merrell's sequential approach is more deterministic and structured, offering faster performance but potentially leading to more incompatibilities [Mer21].

6.1. Impact of number of grid cells

To test this case, the algorithm was executed with 44 tiles. It was tested with the optimization, without the optimization, in order generation and using the entropy. (Figure 9) All the restrictions were also implemented. Four different sized maps were generated (10x10x4, 20x20x4, 30x30x4, 40x40x4) and the average execution time was the following (Table 2).

Execution time increase is considerable high, probably exponential and with an order close to n^3. After this, the Relative Growth Rate was calculated between different intervals. It is demonstrated that the execution time increased significantly, likely at an exponential rate.

$$\text{Relative Growth Rate} = \frac{\text{Absolute Execution Time Increase}}{\text{Initial Time}}$$

Table 2: *Results of the average execution time in ms with different map sizes with and without optimization.*

(a) *With optimization*

Dimension	Order	Entropy
10x10x4	1248.00	1777.67
20x20x4	9992.67	11733.00
30x30x4	34210.67	66719.33
40x40x4	94607.00	177238.00

(b) *Without optimization*

Dimension	Order	Entropy
10x10x4	2841.00	3821.67
20x20x4	22504.67	46684.67
30x30x4	189602.33	287377.00
40x40x4	600539.00	905830.00

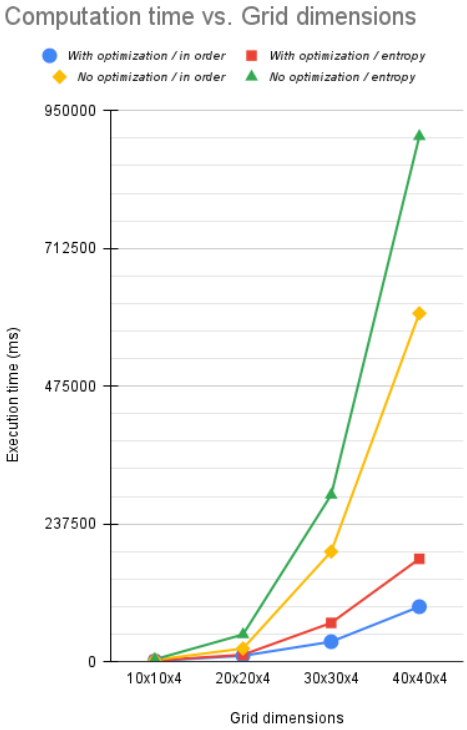

Figure 9: *Execution time vs. Grid dimensions graphic.*

With optimizations and in order, between 10x10x4 and 20x20x4 the absolute growth was 8744.67 ms, so the relative growth was about 700.7%. Without optimizations, this value is more than 1000%. A 40x40x4 map took over 15 minutes to complete, compared to approximately 1.5 minutes with optimizations. (Table 3).

This data suggests very rapid initial growth. Optimizations and in-order generation help mitigate this relative growth considerably. It is also noticeable that the growth slows down for larger maps.

As map size increases, the relative growth in time levels off, but remains significant. Without optimization, growth is brutally fast, especially at the first increases in map size.

Table 3: *Relative Growth Rate when the grid dimension increases.*

(a) *With optimization*

Dimension increase	Order	Entropy
10 → 20	700.7%	560%
20 → 30	242.4%	468.6%
30 → 40	176.5%	165.6%

(b) *Without optimization*

Dimension increase	Order	Entropy
10 → 20	692.1%	1121.6%
20 → 30	742.5%	515.6%
30 → 40	216.7%	215.2%

6.2. Impact of number of tile options

Using a 20x20x4 map, the impact of increasing the number of tiles was analyzed. (Figure 10) (Table 4).

Table 4: *Results of the average execution time in ms with different number of tiles with and without optimization.*

(a) *With optimization*

Tile options	Order	Entropy
44	9992.67	11733
27	5151.33	5652.67
19	4783.67	4923

(b) *Without optimization*

Tile options	Order	Entropy
44	22504.67	46684.67
27	10574.33	15847.67
19	7886.67	10813.67

The Relative Growth Rate was also calculated (Table 5). This indicates that entropy-based selection involves more calculations and greater complexity. Execution time scaled poorly as the number of tiles increased, especially for entropy-based selection. In conclusion, the number of tiles significantly affects execution time, with entropy-based selection being more computationally expensive.

6.3. Impact of restrictions

Lastly, the impact of additional constraints was evaluated using a 20x20x4 map with 44 tiles. Excluded neighbor constraints had minimal impact on execution time. However, adding the underground and air layers significantly reduced runtime because the algorithm processed fewer vertical layers of the map. The other constraints, like fixed tiles, are added before the generation so they do not affect directly to the runtime execution. (Figure 11).

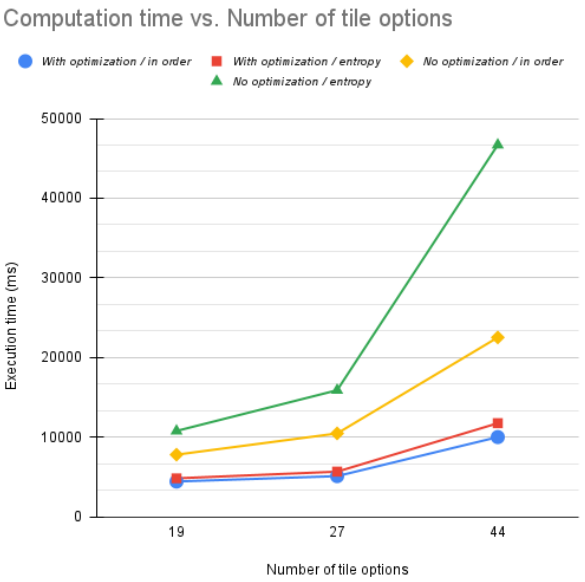

Figure 10: *Execution time vs. Number of tile options graphic.*

Table 5: *Relative Growth Rate when the number of tiles increases.*

(a) *With optimization*

Tile increase	Order	Entropy
19 → 27	7.69%	14.82%
27 → 43	93.98%	107.57%

(b) *Without optimization*

Tile increase	Order	Entropy
19 → 27	34.08%	46.55%
27 → 43	112.82%	194.58%

6.4. Summary

After the analysis, it is concluded that the number of cells in the map and the number of possible tiles are the two variables that affect performance the most. If we look at the two variables, increasing the map size has an exponential growth in time as we observed in the initial data. The runtime without optimization can grow by more than 1000%, and the relative growth remains high even on larger maps. This suggests a very high complexity, possibly cubic. Increasing the number of tiles also has a significant impact, but it is less pronounced in comparison. This suggests that complexity grows as a function of the number of tiles, but the relationship seems closer to linear or quadratic.

Therefore, the number of cells in the map scales faster and has a much larger impact on runtime, especially for larger maps. Increasing tiles, while significant, has a more manageable effect, especially compared to the time explosion caused by an increase in map size.

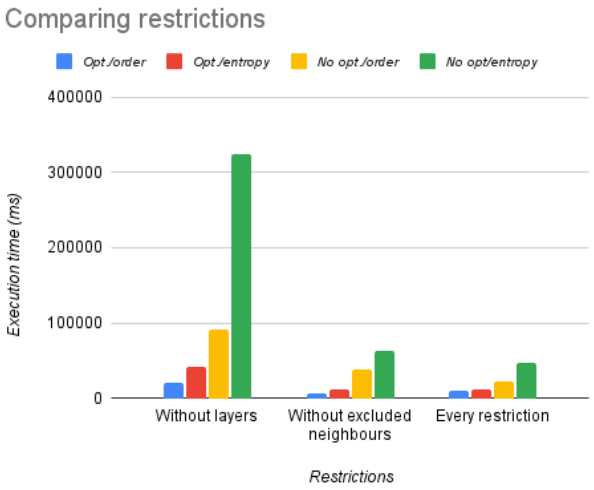

Figure 11: *Comparing different restrictions.*

7. Conclusions and future work

In conclusion, the WFC algorithm is a powerful tool for procedural map generation, especially when combined with non-local constraints. The results are coherent, beautiful and easy to apply to different art styles or themes (Figure 12). In this case a natural landscape is generated, but it could also be a castle or a city. However, its computational cost is extremely high, particularly in 3D, where it exhibits cubic exponential complexity. Thus, optimizations are necessary to make it practical for larger-scale applications. The constraints introduced in this study did not substantially increase computational complexity but did enhance the coherence of the final results. Selectively propagating constraints only to collapsed cells and their immediate neighbors significantly reduced execution costs.

Future work includes implementing parallelization by dividing the map into sections, each processed by separate threads. This could address the challenges posed by the number of cells. However, parallelization is complex because changes in one cell can affect the entire map. Another potential idea is implementing the algorithm on GPUs using compute shaders.

Acknowledgements This work has been developed within the framework of research project PID2023-149976OB-C21, funded by MCIN/AEI/10.13039/501100011033/FEDER, UE.

References

[Bet11] BETHESDA. *Skyrim*. https://elderscrolls.bethesda.net/es/skyrim10. 2011 1.

[Bor20] BORISTHEBRAVE. *Wave Function Collapse Explained*. https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/. 2020 2.

[CHF20] CHENG, D., HAN, H., and FEI, G. "Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm". *Entertainment Computing – ICEC 2020. Lecture Notes in Computer Science*. Vol. 12523. https://doi.org/10.1007/978-3-030-65736-9_3. Springer, Cham, 2020 1.

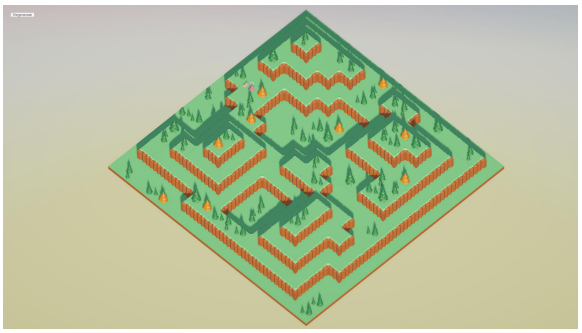

Figure 12: *Final result.*

[CKT*24] CHRISTIE, ROBERT, KITCHEN, BRIAN, TUMILOWICZ, WIKTOR, et al. "Procedurally Generating Large Synthetic Worlds: Chunked Hierarchical Wave Function Collapse". *2024 39th International Conference on Image and Vision Computing New Zealand (IVCNZ)*. 2024, 1–6. DOI: 10.1109/IVCNZ64857.2024.10794452 1.

[Dod16] DODGEROLL. *Enter the Gungeon*. https://enterthegungeon.com/. 2016 1.

[Fre24] FREEHOLDGAMES. *Caves of Qud*. https://www.cavesofqud.com/. 2024 2.

[Gri19] GRIDBUGS. *Procedural Generation with Wave Function Collapse*. https://www.gridbugs.org/wave-function-collapse/. 2019 2.

[Gum16] GUMIN, M. *Wave Function Collapse Algorithm*. https://github.com/mxgmn/WaveFunctionCollapse. 2016 2.

[Kle19] KLEINEBERG, MARIAN. *Infinite procedurally generated city with the Wave Function Collapse algorithm*. https://marian42.de/article/wfc/. 2019 2, 3.

[Lux24] LUX, M. "Procedural Content Generation - The Open Source Success Story of Wave Function Collapse". *ACM Digital Library*. 2024. URL: https://doi.org/10.1145/3708973.3708979 1.

[MC23] MARÍN-LORA, C. and CHOVER, M. "A Multi-agent Sudoku Through the Wave Function Collapse". *Malvone, V., Murano, A. (eds) Multi-Agent Systems. EUMAS 2023. Lecture Notes in Computer Science(), vol 14282. Springer, Cham* (2023). https://doi.org/10.1007/978-3-031-43264-4_24 1.

[McM11] MCMILLEN, EDMUND. *The Binding of Isaac*. https://store.steampowered.com/app/113200/The_Binding_of_Isaac/?l=spanish. 2011 1.

[Mer07] MERRELL, P. "Example-Based Model Synthesis". *Symposium on Interactive 3D Graphics (i3D)* (2007). https://paulmerrell.org/wp-content/uploads/2022/03/model_synthesis.pdf 2.

[Mer21] MERRELL, PAUL. "Comparing Model Synthesis and Wave Function Collapse". *Technical Report* (2021). https://paulmerrell.org/wp-content/uploads/2021/07/comparison.pdf 2, 7.

[Per85] PERLIN, K. "An image synthesizer". *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*. Vol. 19. 1985, 287–296 1.

[SCM19] SANDHU, ARUNPREET, CHEN, ZEYUAN, and MCCOY, JOSHUA. "Enhancing Wave Function Collapse with Design-level Constraints". *ACM Digital Library* (2019). https://dl.acm.org/doi/pdf/10.1145/3337722.3337752 2.

[Sri25] SRINIVAS JOSHI, A. "Reinforcement Learning-Enhanced Procedural Generation for Dynamic Narrative-Driven AR Experiences". *arXiv* (2025). URL: https://ui.adsabs.harvard.edu/link_gateway/2025arXiv250108552S/doi:10.48550/arXiv.2501.08552 1.

[Sta17] STALBERG, OSKAR. *Wave - by Oskar Stålberg*. https://oskarstalberg.com/game/wave/wave.html. 2017 2.

[Sta18] STALBERG, OSKAR. *Bad North*. https://www.badnorth.com/. 2018 2.

[Sta20] STALBERG, OSKAR. *Instant town building*. https://imgur.com/gallery/instant-town-building-i364LBr. 2020 2.

[Sta21] STALBERG, OSKAR. *Townscaper*. https://www.townscapergame.com/. 2021 2.

[Sta24] STATISTA. *Ranking de los videojuegos que más unidades vendieron a nivel mundial a fecha de mayo de 2024*. https://es.statista.com/estadisticas/635350/los-videojuegos-mas-vendidos-de-todos-los-tiempos-a-nivel-mundial-con-base-en-unidades-vendidas/. Accessed February 2025. 2024 1.

[Sup18] SUPERGIGANTGAMES. *Hades*. https://www.supergiantgames.com/games/hades/. 2018 1.

[Tri22] TRISTAN, WAUTHIER. *Using Wave Function Collapse algorithm for 2D and 3D level generation*. https://tristanwauthier.com/PDF/GW_2223_Tristan_Wauthier_EN_Paper.pdf. 2022 3.

[WH87] WILLIAM E., LORENSEN and HARVEY, E. CLINE. "Marching cubes: A high resolution 3D surface construction algorithm". *ACM SIGGRAPH Computer Graphics*. Vol. 21. 4. https://doi.org/10.1145/37402.37422. 1987 2.

[YD24] YILDIZ, D. and DEMIRCI, S. "Enhancing Wave Function Collapse Algorithm for Procedural Map Generation Problem". *ResearchGate* (2024). URL: http://dx.doi.org/10.28948/ngumuh.1361413 1.

[ZZH22] ZHANG, YUZHONG, ZHANG, GUIXUAN, and HUANG, XINYUAN. "A Survey of Procedural Content Generation for Games". *2022 International Conference on Culture-Oriented Science and Technology (CoST). IEEE.* (2022). https://doi.org/10.1109/CoST57098.2022.00046 1.