GENERATING TILE-BASED 3D VIRTUAL ENVIRONMENTS ON ARBITRARY CONVEX SURFACES USING WAVE FUNCTION COLLAPSE

Silviu STĂNCIOIU [1], Anca MORAR [2], Alin-Dragoș-Bogdan MOLDOVEANU [3], Florica MOLDOVEANU [4]

A common trend in the video games industry is making use of Procedural Content Generation methods to facilitate the efficient creation of game content such as game assets or levels. One such technique is the Wave Function Collapse (WFC) algorithm which proves to be useful for the creation of virtual 3D environments that satisfy certain pre-defined constraints. While some variants were proposed for the generation of environments on convex surfaces using the said algorithm, none of them extend into the third dimension. In this paper we introduce a novel extension of the algorithm that allows for the generation of 3D environments on top of any convex surface. We present the creation potential of the algorithm using a complex 3D virtual city environment generated using our technique and the impact that our modifications have on the performance of the algorithm.

Keywords: Wave Function Collapse, Procedural Content Generation, Constraint Solving, Game Content

1. Introduction

Procedural Content Generation (PCG) techniques are proved to be useful in the process of creating 3D virtual environments. These techniques can be used for assets that make up the scene, such as: textures, 3D models, sounds or character animations, but also for the scene layouts. In video games, the virtual environments must usually satisfy both visual and game-specific constraints (for example: reachability). One PCG algorithm is the Wave Function Collapse (WFC) algorithm, proposed by Maxim Gumin in 2016 [1]. The algorithm itself was initially used for image synthesis, extracting patterns and their adjacencies rules from input images and using them to produce new images. Although the name of the algorithm is loosely inspired by quantum physics, the author was inspired by the technique

[1] PhD Student, Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA of Bucharest, Romania, e-mail: silviu.stancioiu00@upb.ro
[2] Professor, Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA of Bucharest, Romania, e-mail: anca.morar@cs.pub.ro
[3] Professor, Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA of Bucharest, Romania, e-mail: alin.moldoveanu@cs.pub.ro
[4] Professor, Faculty of Automatic Control and Computer Science, National University of Science and Technology POLITEHNICA of Bucharest, Romania, e-mail: florica.moldoveanu@cs.pub.ro

proposed by Paul Merrell [2]. Since its inception, WFC was used for more than image synthesis, producing 3D models, 3D environments, video game levels and more.

Many video game levels are generated using WFC, either offline during production, or at runtime which allows them to constantly produce new game content for the user. *Caves of Qud* [3] is a 2D roguelike game in which a multi-layer WFC approach is used to generate portions of the levels. Oskar Stålberg used the algorithm in various games he developed. In *Bad North* [4] he used WFC to procedurally generate 3D islands. *Townscapper* [5], another game developed by the same person, uses a real-time variation of the algorithm to ensure the compatibility between the tiles being placed.

The WFC algorithm itself is a constraint satisfaction algorithm adapted for the generation of content inside regular grids of tiles with adjacency rules. Karth and Smith [6] examined WFC as an instance of constraint solving methods and provided an Answer Set Programming (ASP) implementation. Based on this, various implementations that extend the base capabilities of the algorithm appeared. *Sturgeon* [7] is one such implementation that can generate complete 2D game levels with reachability constraints in place, for various player movement methods (ex: platformers, mazes). The reachability constraints are defined through a custom high-level constraint solving API. The high-level API is implemented on top of various constraint solvers such as *clingo* [8] or *Z3* [9]. Another proposed solution for making the generated video game levels playable is using genetic algorithms [10] to optimize playability metrics given by the fitness functions. To improve the visual quality of the levels generated using WFC, rather than defining more complex constraints, various hierarchical versions of the algorithm were proposed [11, 12]. Another different tool implementing WFC is *miWFC* [13], which incorporates features such as controlled backtracking and manual editing of the generated output. To generate more organic game levels, Møller et al. [14] proposed running the Growing Grids [15] algorithm on an input image and subsequently applying WFC on the resulting grid. Besides the standard grid variants of the algorithm, different implementations use graph structures instead. Kim et al. [16] describe the changes required so that WFC can be applied on any graph structure and showcases it on various instances such as Sudoku games or Voronoi non-grids. *Tessera* [17] is a tool for generating game levels using WFC which implements a graph variant of the algorithm. This makes suitable for tile placement on “irregular 3d grids based on the surfaces of quad or triangle meshes”.

One limitation observed in most of the systems/ tools employing WFC for 3D environment generation is their exclusive dependence on flat and static grids. *Tessera* [17] allows for more flexibility in the sense that it allows for the generation of content on top of various surfaces given by meshes. However, the grids are not extended into the third dimension while maintaining the integrity of the tiles due to

the static nature of the grids. While a static grid approach may be desired in certain scenarios, it is worth noting that there are instances wherein the utilization of more complex dynamic grids derived from the subdivision of the original grid cells can yield more realistic outcomes. Subdividing the grid cells prior to the execution of the algorithm, while a viable option, imposes a limitation on the algorithm's capacity to determine the density of specific structures. In this paper, we present the modifications needed for adapting WFC to function properly on 3D grids generated based on convex surfaces composed of quads. We implemented all functionalities inside a Unity editor-level tool. With our modified WFC variant changes in place, the tool can create complex 3D virtual environments on various types of convex surfaces.

2. Generation of 3D environments

For the generation of 3D environments, the WFC algorithm is usually employed to place tiles within a grid while ensuring that adjacency constraints between neighboring tiles are satisfied. The tile distribution can be user-defined or learned from an input example. Alternative approaches to this problem include:

- Wang Tiles – Wang Tiles are tiles that dictate how they can connect to adjacent tiles. They get their name from a conjecture made in 1961 by Hao Wang. Given a set of tiles that can produce a valid tiling, a basic 3D content generation algorithm would consist of randomly placing valid Wang Tiles adjacent to one another.
- Greedy Tile Placement – By extending the concept of Wang Tiles through the addition of more complex adjacency rules, greedy algorithms can be used. These algorithms iteratively place random compatible tiles adjacent to existing ones in the grid until the environment is fully generated or a contradiction occurs. Tile selection can also be weighted by probabilities to control their distribution in the final environment.
- Cellular Automata [19] – Instead of using tiles characterized by adjacency constraints, each grid cell is assigned a random state. To update the state of each cell, all its neighbors are taken into account, then based on some predefined rules, the state may change or remain the same. This process is applied to each cell of the grid for a finite number of iterations. Once the grid reaches a desirable configuration, tiles corresponding to the final states are placed in the cells.

3. WFC Overview

WFC is a constraint-solving algorithm originally designed for image synthesis. It takes a bitmap made of pixels as input, and its goal is to create an output bitmap that respects the following conditions defined by Maxim Gumin [1]:

- "(C1) The output should contain only those NxN patterns of pixels that are present in the input."
- "(Weak C2) Distribution of NxN patterns in the input should be similar to the distribution of NxN patterns over a sufficiently large number of outputs. In other words, the probability to meet a particular pattern in the output should be close to the density of such patterns in the input. "

The output bitmap is initialized with pixels in an unobserved state, implying that each pixel of the bitmap can be assigned any value present in the input. The algorithm then starts an *Observe-Propagate* cycle which runs until either all pixels of the output image are set, or the algorithm reaches a contradiction. The *Observe-Propagate* cycle follows this sequence:

- *Observe* – Find the $\boldsymbol{N} \times \boldsymbol{N}$ pattern with the lowest entropy in the output bitmap and collapse it into a fixed state. The entropy function can be defined in multiple ways. The function used in the original implementation is the Shannon Entropy [18]. Alternatively, the entropy can be defined by the number of patterns that can be fixed in a section of the bitmap without breaking the adjacency rules.
- *Propagate* – For every neighboring pattern of an $\boldsymbol{N} \times \boldsymbol{N}$ pattern that has been collapsed or partially collapsed, update the list of input patterns into which it can be collapsed. As the count of input patterns into which a neighbor pattern can collapse decreases, the neighbor pattern becomes partially collapsed, and the change is recursively propagated to the other neighbors until the whole bitmap has been updated.

There are two models for the extraction of $\boldsymbol{N} \times \boldsymbol{N}$ patterns:

- The Simple Tiled Model – The tiles only present immediate adjacency constraints and are not parts of bigger patterns. For this model, the tiles and their constraints are usually defined manually.
- The Overlapping Tiled Model - The algorithm begins by extracting $\boldsymbol{N} \times \boldsymbol{N}$ patterns from the input image and determining their adjacencies. It also calculates their distribution within the input image to ensure a similar distribution in the output image.

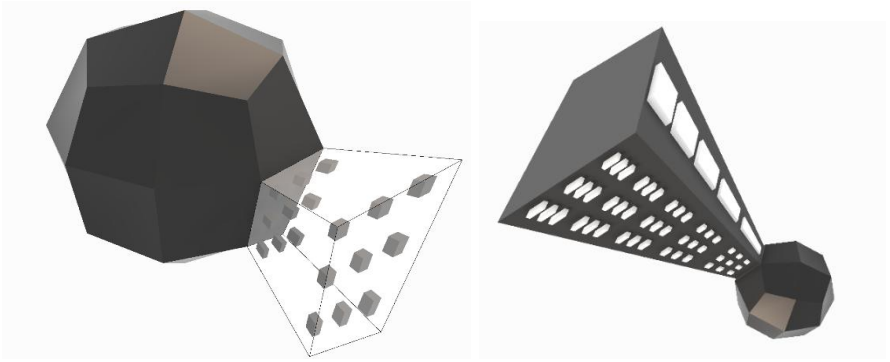
4. Our technique

We implement a form of WFC adapted for 3D tile placement. The terminology we use is different from the one used in the original description of the algorithm [1] to make it suitable for our specific use case. Instead of collapsing $\boldsymbol{N} \times \boldsymbol{N}$ patterns into an output bitmap as the original implementation does, we instead place tiles inside grid cells. The tiles are no longer pixels extracted from an input bitmap, or predefined 2D images, and instead, they are 3D models placed inside a bounding box for which adjacency rules are manually defined by a user.

4.1. Overview

To establish tile adjacencies, we adopted the same approach as *Tessera* [17]. Each tile is a cube, with 9 connectors on each of its faces. Each connector can be assigned a label, defined by an ID and a color. Valid label adjacencies are determined by a 2D Boolean array. By default, each label can exclusively connect with itself and the default label (transparent). To prevent the need for defining the same tile with different rotations, we implemented a method to augment the tiles that are already defined. Augmentations can be generated through rotations around the X, Y, and Z axes.

Our implementation takes a 3D model made of quads as input and generates 3D structures made of tiles on top of it. The quads of the mesh that form concave surfaces are filtered out, leaving only the quads of the convex surfaces. These quads are then used to create a 3D irregular grid with N layers. After the grid is created, our modified variant of WFC is used to place tiles inside the grid and generate 3D structures. Given the cubical nature of the tiles, each tile can be placed within a cell through trilinear interpolation. When extending this type of convex grid to multiple layers, its cells tend to expand, causing their top faces to become larger than the bottom ones. When placing tiles within such cells, this expansion will also have an impact on them, as shown in Fig. 1a.



a) b)

Fig. 1. a) Tile placed within a grid cell on top of a sphere. Because the surface is convex, the top face of the cell becomes larger as the cell extends upwards, leading to an increasing distortion of the tile. b) A tower with 5 layers. As the tower increases in height, its roof also becomes larger, resulting in an unrealistic-looking tower.

This in turn, limits the creative potential of the algorithm. As an example, consider a sphere on top of which a convex 3D grid is created. Suppose that this sphere is meant to have towers generated on top of it. As the towers increase in height, their roofs also linearly grow. While in some situations, such an outcome may be desired for cartoonish effects, it lacks realism. Additionally, the

enlargement of the roofs is solely controlled by the grid, leaving the person generating the environment with no creative influence over the aspect (Fig. 1b).

To address this issue, we introduce a new type of tile called a *shrink tile*. Shrink tiles allow the user to specify how much space within the cell the tile is allowed to occupy. By default, when shrink tiles are placed within a grid, they occupy all the space between the bottom quad of the cell and its projection onto the top quad, resulting in a straight appearance. The remaining space within the cell is filled with degenerate side cells (Fig. 2). These newly added cells are considered to degenerate, because their bottom faces are missing. If there is no remaining space within the pre-split cell, then no degenerate cells are added. This happens when the algorithm is run using meshes that form flat surfaces. The newly added degenerate side cells and their top neighbor cells are flagged accordingly so that only certain tiles can be placed inside them. The tiles that can be placed within these flagged cells must be manually specified by the user.

The amount of space filled by a shrink tile can be specified through a user-defined function that returns values between 0 and 1. A value of 1 represents the default behavior of the shrink tiles, which projects a bottom face onto the top face of the cell and fills the entirety of the space in between. A factor smaller than 1 reduces the size of the bottom face projection onto the top face. The ability to specify the behavior of shrink tiles can be useful for generating structures such as pyramids.

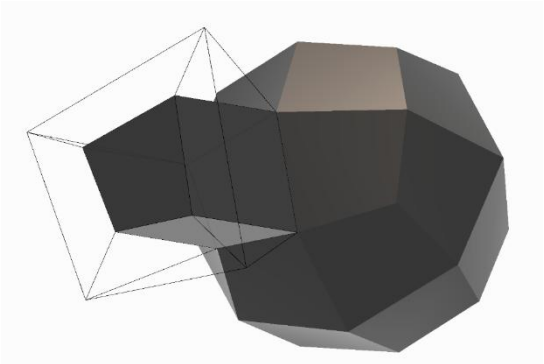


Fig. 2. A shrink tile placed within a cell, partially filling its volume. The remaining space is divided into four degenerate cells.

A class, identified by an integer ID, can be assigned to each shrink tile. The user-defined function takes as input the number of shrink tiles from the same class that were previously placed below the current tile. It returns a number between 0 and 1, which represents how much the bottom face of the shrink tile furthest below should be projected onto the top face of the current cell. The intuition behind this choice was that a class of shrink tiles would form the same structure as they are placed upward. This, in turn, simplifies the process of specifying custom functions for structures that linearly decrease in size.

In the case of grids with multiple layers, the placement of a shrink tile that breaks the cell affects its top neighbor cells, subsequently breaking the corresponding cells as well. Note that the addition of shrink tiles introduces a limitation to the ability of the algorithm to place tiles within cells. A new tile can only be placed within cells that are located above cells where tiles have already been placed. Otherwise, the structure of the grid may break due to the structural changes made by shrink tiles that could be placed below.

The addition of shrink tiles opens new possibilities for 3D environments generation using WFC.

4.2. WFC Changes

Our implementation follows the original C# code proposed by Maxim Gumin [1] with a few extra modifications to adapt for the nature of the grids. The original implementation assumes that the orientation of tiles is consistent across the entire grid. In other words, the directions of Left, Up, Right, and Bottom are the same for each cell in the 2D bitmap grid. Throughout the remainder of this paper, we will adopt the use of directional references such as North, East, South, West, Top, and Bottom in the context of 3D grids. This choice is made due to the absence of guaranteed absolute Up, Right, and Forward directions in such grids. Since our application can accept any 3D grid composed of quads as input, there is no guarantee that the quads will maintain uniform orientation throughout the mesh (Fig. 3). Rather than treating orientations as absolute, we instead treat them as relative to the orientations of neighboring cells. Let c_1 be a cell and c_2 be one of its neighbors. We introduce a value called the rotation number, which signifies the number of 90-degree clockwise rotations c_2 needs to perform for its orientation to align with that of c_1. A rotation number of 0 indicates that the cells share the same orientation.

Minimal changes are needed for the algorithm to function on single grid layers. The main additional constraint is ensuring that tiles in adjacent cells with non-zero rotation numbers remain compatible. To achieve this, the two primary multi-dimensional arrays, $propagator$ and $compatible$, are extended to account for all R possible tile rotations. For a given direction d, rotation number r, and tile t, $propagator[d][r][t]$ represents the list of tiles that can connect with t when the rotation between cells is r and the direction is d. This adjustment is needed since tiles no longer connect via opposite faces, requiring the $propagator$ to accommodate each of the R possibilities.

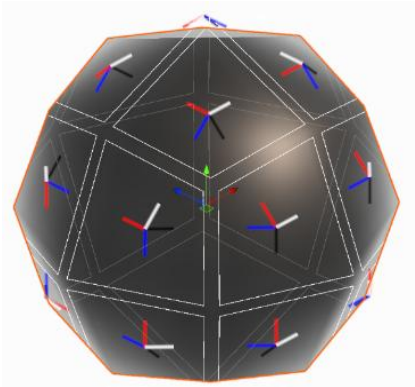


Fig. 3 Different orientations of the grid cells. The lines in the middle of the quads point toward their neighbors. Black, Blue, Red, and White lines indicate orientations towards the North, East, South, and West neighbors, respectively. If two cells share a South-North or an East-West edge, they have the same orientation; otherwise, their orientations differ. The middle-left cell shares a North-East edge with the middle right cell, resulting in different orientations.

The other main multi-dimensional array, $compatible$, is also extended, becoming a 4D array. For a given grid cell c and a tile t that can potentially be placed within a cell, $compatible[c][t]$ becomes a 2D array with $D \times R$ elements, where D is the maximum number of neighbors a grid cell can have. This 2D array represents the number of tiles that could potentially be placed near the current cell for each direction d and each cell rotation number r, so that the adjacencies of t and the neighboring tiles' adjacencies match. When any element of this 2D array, $compatible[c][t]$, becomes 0, it means that tile t can no longer be placed within cell c.

The only stage modified within the algorithm is the $Propagate$ stage where for each pair of neighbor cells, the rotation number is calculated and used as index for the extended arrays. In our context, we rotate directions within the horizontal plane, including North, East, South, and West. Cells below or above other cells share the same orientation, resulting in a rotation number of 0, requiring no adjustments.

4.2.1. Shrink Tiles

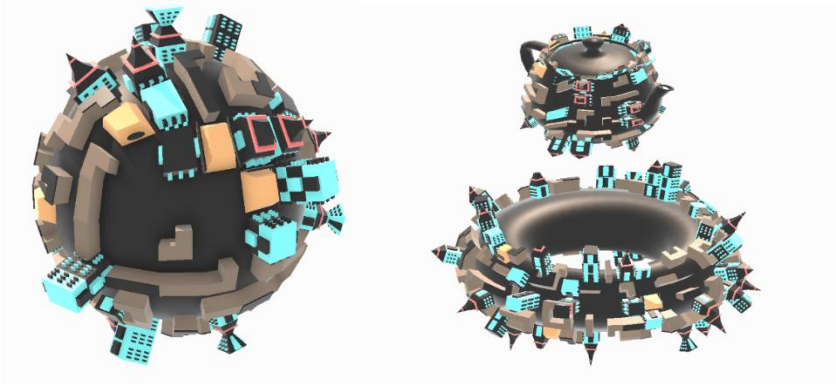
Wave Function Collapse cannot directly handle shrink tiles for several reasons. One reason is that when shrink tiles are placed, they may no longer be in contact with neighboring cells. Instead, they will be in contact with the degenerate cells automatically added upon placement. To address this issue, instead of directly using the shrink tiles, we employ multi-tile modules [17]. In our case, a multi-tile module comprises a central shrink tile and up to four degenerate side tiles. When collapsing such a multi-tile module into a cell, the shrink tile is positioned at the center of the cell, while the degenerate tiles are placed in the newly created

degenerate cells. When shrink tiles are placed within cells, they may not always need the addition of four padding degenerate side cells. Such an outcome can occur, particularly when the entire grid is generated based on a flat surface. Before initiating the WFC algorithm, every cell undergoes an evaluation against all potential shrink tiles or multi-tiles to check whether they can be placed within the given cell. Any incompatible tiles or multi-tiles are banned. For each cell, a shrink tile may potentially be placed directly or through a multi-tile, but not both ways.

4.2.2. Multiple Layers

Another problem that arises when using shrink tiles is that when they are placed, they may also subdivide the cells above them, thereby modifying the structure of the grid. This has two implications. Firstly, tiles must be placed in cells that are either on the first layer or above cells that already have tiles placed inside them. This is a direct consequence of the fact that shrink tiles alter the structure of the grid for the upward layers. Placing a shrink tile below an already placed tile will break the adjacencies on the next layers, resulting in invalid tile configurations or causing the WFC Algorithm to fail. Secondly, the *compatible* values of the newly created grid cells need to be calculated, and based on their values, changes must be propagated to their neighbors. To address this problem, we adopted a straightforward multi-layer approach. Since tiles must be positioned either above already placed tiles or on the first layer, utilizing multiple layers and sequentially generating one layer after another through WFC ensures compliance with this constraint. Once each layer is generated, the algorithm proceeds to generate the subsequent layer. To ensure proper connections between layers, each cell on the current layer is restricted from using tiles that do not connect with the top faces of the tiles placed below.

5. Creative potential



a) b)

Fig. 4 a) City generated on top of a spherical surface using a complex tileset containing elements such as blocks, pyramids and walls. b) The same tileset being used to generate cities on top of the convex surfaces of a teapot and a torus.

We evaluated our technique using a complex tileset designed for city creation (Fig. 4a), which includes blocks, towers, houses, pyramids, and walls. None of these tiles can float. For generating blocks, towers, and houses, two types of tiles represent their structural support. Shrink tiles without custom functions are used to avoid distortion when stacked. An intermediary shrink tile, with a custom function that decreases its size linearly, is used for roofs, giving them a sharp appearance. Only similar or structural tiles can be placed on top. Pyramid structures use a shrink tile with a similar size-reducing function, allowing for decorations in nearby degenerate cells. Pyramid decorators include spheres and straight lines. Walls consist of vertical tiles with Y-axis augmentations and corner tiles, with two special decorator tiles that resemble those used for pyramids. This tileset incorporates all key functionalities of our implementation: shrink tiles that maintain structural integrity, undistorted towers and blocks, size-reducing roofs and pyramids, placement in degenerate cells, non-ground tiles, tile rotations, and dynamic grid modifications.

6. Performance

6.1. Memory usage and complexity

The only memory usage changes stem from extending the $propagator$ and $compatible$ variables to account for all R possible rotations, where $R = 4$, as the grids are based on quads along mesh surfaces. When shrink tiles are used, memory usage increases due to the precomputation of all possible degenerate neighbor pairs, creating up to T^8 composite tiles. In the extreme case, where all tiles are shrink tiles and no compatibility constraints are defined for the tiles, the number of tiles, T', grows to T^9. The theoretical complexity of the WFC algorithm is $O(T^2 \cdot C^2)$, with T being the number of tiles and C the number of grid cells. In practice, it requires approximately $T^2 \cdot C$ iterations. Our modified version performs R additional iterations per step, but since $R = 4$ in our case, this does not affect the algorithm's overall time complexity.

6.2. Benchmarks

Performance testing was conducted on a system with an AMD Ryzen 5 5600H, 16GB RAM, NVIDIA RTX 3060 (laptop variant), and Windows 11. The tests used the previously described city tileset on a spherical surface, which highlights all the grid changes brought to the algorithm. Execution time was measured on sphere resolutions ranging from 1 to 16, as shown in Table 1. A key observation is that upper layers exhibit different execution times compared to lower layers, as the time differences are not uniform across layers. While each layer should theoretically have similar execution times, tiles placed on lower layers introduce constraints that can reduce the time for upper layers. However, if shrink

tiles are used in lower layers, they may cause structural changes that increase execution time for the upper layers.

Table 1

Execution times (in seconds) of the algorithm for 1, 2, 3, and 4 layers using the city tileset for spheres with various resolutions

Number of layers / Sphere resolution	1	2	3	4
1	0.036	0.012	0.012	0.014
2	0.025	0.049	0.071	0.093
3	0.073	0.337	0.218	0.233
4	0.118	0.204	0.290	0.392
5	0.183	0.528	0.699	0.661
6	0.496	0.618	0.589	1.028
7	0.556	0.761	1.022	1.325
8	0.446	0.985	1.638	2.185
9	0.609	1.470	1.971	2.732
10	1.155	1.735	2.381	3.251
11	1.103	1.974	3.057	4.555
12	1.621	2.606	3.877	5.380
13	1.879	3.162	5.192	6.124
14	1.880	3.485	5.213	7.594
15	1.932	4.267	6.567	9.402
16	2.497	4.957	7.672	11.660

7. Discussion

The tiles utilized in our algorithm extend beyond the capabilities of Wang Tiles, allowing for the definition of more complex adjacency rules for face tags. This enables the generation of more complex structures. However, defining such complex rules is challenging, making it more difficult to ensure that a given set of tiles can produce a valid tiling compared to Wang Tiles.

Our algorithm, along with other variants of WFC, offers significant advantages over greedy approaches for several reasons:

- Contradictions are more easily avoided as the algorithm prioritizes output patterns (grid cells, in this case) with the lowest entropy, making backtracking or restarting of the generation process more efficient.
- The distribution of local patterns is easier to control.
- When implementing WFC using constraint solvers, it becomes possible to define rules beyond local adjacency constraints, such as reachability.

Compared to cellular automata methods, our WFC variant offers the following key advantages:

- The patterns in the output are easier to control.
- The final output is guaranteed to be valid.
- The rules are easier to define and understand.

It is worth noting that procedural content generation systems based on cellular automata may adjust rules between iterations to achieve the desired results. In the case of WFC, this is analogous to manually placing tiles before the algorithm completes the generation process.

Cellular automata-based methods are particularly effective for generating 3D content with tiles, especially in creating natural-looking caves or other environments that involve large areas. As WFC is a local constraint-solving method, controlling larger patterns in the environment is harder compared to cellular automata methods, requiring modifications to the algorithm that our implementation does not currently support.

The modifications required to make controlling larger patterns easier when using WFC may include:

- Using a hierarchical version of WFC.
- Using cellular automata to define larger areas of the environment, and then applying WFC with different sets of tiles based on the areas.

Among WFC-based implementations for generating 3D virtual environments with tiles, our approach offers some key advantages. One major advantage is the ability to generate 3D structures on top of any convex surface defined by a 3D mesh (Fig. 4b). This extension to the third dimension (height) doesn't cause the tiles to get distorted as they would if a naïve approach was to be used. Another advantage is the ability to specify tiles that can be used as visual decorators for other tiles. Our algorithm has the potential to be used in tools that generate complex video game environments on top of surfaces that are not flat.

Some limitations compared to previous WFC variants include:

- The lack of a backtracking mechanism, which may prevent the algorithm from finding a solution. The implementations presented in [13, 16] address this problem by incorporating backtracking into the WFC algorithm. Solutions such as [7, 17] use constraint solvers that are guaranteed to find a solution if the constraints are not contradictory. [10] addresses this problem from a different perspective, by implementing WFC as a repair operator inside a genetic algorithm.
- The absence of reachability / path constraints, unlike the solutions proposed in [7, 10, 14, 16, 17], meaning generated environments are not guaranteed to be playable as game levels.
- Tile and adjacency constraints must be manually defined. Other implementations allow the designer to define levels and then automatically extract constraints from them [7, 10, 11].
- The implementation uses the Simple Tiled Model instead of the Overlapped Tiled Model [1, 11, 12]. In [7], a more flexible paradigm for defining pattern constraints is proposed. This paradigm allows custom pattern constraints, including column-based patterns.

- Only quad-based grids are supported. Implementations such as [16, 17] utilize graph-based WFC.
- Tiles cannot be manually placed inside the grid. This functionality is provided by many WFC implementations [7, 11, 12, 13, 14, 16, 17].

Because these limitations are common and can be addressed, our proposed changes can be integrated into the existing tools/algorithm variations without difficulties. One limitation specific to our implementation is the lack of support for concave surfaces. Another limitation is the reliance on multi-layer grids, preventing the generation of structures like trees or those requiring non-grid transformations (e.g., rotation and scaling).

8. Conclusions

This paper proposes modifications to the WFC algorithm to enable content generation over multiple layers on convex surfaces. The goal is to ensure generated structures are either undistorted or that any distortions can be controlled by the user. Our implementation successfully generates complex structures incorporating the proposed changes. The performance impact of these modifications is minimal, if tags are properly defined by the user. Future work will focus on adding functionalities for easier creation of playable 3D levels, including reachability constraints and integration with existing constraint solvers. We also aim to simplify the definition of tiles and adjacencies by implementing the overlapped version of WFC.

R E F E R E N C E S

[1] M. Gumin, Wave Function Collapse, https://github.com/mxgmn/WaveFunctionCollapse, 2016.
[2] P. Merrell, Model Synthesis. Ph.D. Dissertation, University of North Carolina at Chapel Hill, 2009
[3] B. Bucklew, Tile-Based Map Generation using Wave Function Collapse in 'Caves of Qud', GDC, 2019
[4] O. Stålberg, Wave Function Collapse in Bad North, Everything Procedural Conference, Breda University of Applied Sciences, 2018
[5] O. Stålberg, Organic Towns from Square Tiles, Indiecade Europe, 2019
[6] I. Karth, A. M. Smith, WaveFunctionCollapse is constraint solving in the wild, Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG '17). Association for Computing Machinery, New York, NY, USA, Article 68, 1–10. https://doi.org/10.1145/3102071.3110566, 2017
[7] S. Cooper, Sturgeon: tile-based procedural level generation via learned and designed constraints, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 18(1), 26-36. https://doi.org/10.1609/aiide.v18i1.21944, 2022
[8] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, TPLP, 19(1), 27–82, 2019
[9] L. de Moura, N. Bjørner, Z3: an efficient SMT solver, 2008 Tools and Algorithms for Construction and Analysis of Systems, 2008

[10] R. Bailly, G. Levieux, Genetic-WFC: Extending Wave Function Collapse with Genetic Search, IEEE Transactions on Games, vol. 15, no. 1, pp. 36-45, doi: 10.1109/TG.2022.3192930, 2023
[11] M. Beukman, B. Ingram, I. Liu, B Rosman, Hierarchical WaveFunction Collapse. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 19(1), 23-33. https://doi.org/10.1609/aiide.v19i1.27498, 2023
[12] S. Alaka, R. Bidarra, Hierarchical Semantic Wave Function Collapse. In Proceedings of the 18th International Conference on the Foundations of Digital Games (FDG '23). Association for Computing Machinery, New York, NY, USA, Article 68, 1–10. https://doi.org/10.1145/3582437.3587209, 2023
[13] T. S. L. Langendam, R. Bidarra, miWFC - Designer Empowerment through mixed-initiative Wave Function Collapse. In FDG '22: Proceedings of the 17th International Conference on the Foundations of Digital Games (FDG '22), September 5–8, 2022, Athens, Greece. ACM, New York, NY, USA 8 Pages. https://doi.org/10.1145/3555858.3563266, 2022
[14] T. N. Møller, J. Billeskov, G. Palamas, Expanding Wave Function Collapse with Growing Grids for Procedural Map Generation. In Proceedings of the 15th International Conference on the Foundations of Digital Games (FDG '20). Association for Computing Machinery, New York, NY, USA, Article 28, 1–4. https://doi.org/10.1145/3402942.3402987, 2020
[15] B. Fritzke, Growing Grid — a self-organizing network with constant neighborhood range and adaptation strength. Neural Process Lett 2, 9–13, doi: 10.1007/BF02332159, 1995
[16] H. Kim, S. Lee, H. Lee, T. Hahn, S. Kang, Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm, IEEE Conference on Games (CoG), London, UK, 2019, pp. 1-4, doi: 10.1109/CIG.2019.8848019, 2019
[17] A. Newgas, Tessera: A Practical System for Extended WaveFunctionCollapse. In Proceedings of the 16th International Conference on the Foundations of Digital Games (FDG '21), Association for Computing Machinery, New York, NY, USA, Article 56, 1–7. https://doi.org/10.1145/3472538.3472605, 2021
[18] C. E. Shannon, A mathematical theory of communication, in The Bell System Technical Journal, vol. 27, no. 3, pp. 379-423, doi: 10.1002/j.1538-7305.1948.tb01338.x, July 1948
[19] S. Wolfram, Statistical Mechanics of Cellular Automata, in Reviews of Modern Physics 55, pp. 601-644, doi: 10.1103/RevModPhys.55.601, July 1983