Meaningful content diversity through Non-Uniform Tile WFC

Rolf Piepenbrink[1] and Rafael Bidarra[2], *Senior Member, IEEE*

Fig. 1: A three-dimensional city skyline generated with nutWFC, highlighting the distribution and diversity of building types, as well as their variety of shapes, heights and attributes.

***Abstract*—Procedural Content Generation methods enable the creation of varied content algorithmically. One such method is Wave Function Collapse (WFC), a tile-based local constraint solver commonly applied to texture, map and level generation for grid-based content; it is able to create varied output from the same set of rules, usually derived from an input sample. However, a glaring limitation of WFC is that it only operates on tiles of the same shape and size. We propose Non-Uniform Tile Wave Function Collapse (nutWFC), an extension of WFC that supports multi-cellular tiles with varying shapes and sizes, so-called Non-Uniform Tiles (NUTs). Familiar examples of such tiles can be found in LEGO® and Tetris. The algorithm guarantees NUT shape and size preservation even under WFC's Overlapping Model in three dimensions. We show that nutWFC is a generalization of WFC that harmonizes strict NUT shape and size constraints with WFC's output diversity. We hypothesize that carefully designed non-uniform tilesets can significantly enhance the diversity of the algorithm's outputs while effectively safeguarding them against semantic corruption induced by uncontrolled combinations. We illustrate the expressive power and the output diversity of nutWFC with several results that explore the advantages of NUTs and would therefore not be feasible with standard WFC. This article considerably extends and improves our previous IEEE CoG 2025 paper [1], including a new 'Results and Discussion' section.**

***Index Terms*—Wave Function Collapse, non-uniform tiles, procedural content generation, constraint solving**

1 Rolf Piepenbrink started this work as part of his Master's thesis at Delft University of Technology, The Netherlands, and completed it as an independent researcher. (e-mail: rpiepenbrink@proton.me)

2 Rafael Bidarra is with Delft University of Technology, The Netherlands. (e-mail: R.Bidarra@tudelft.nl)

I. Introduction

WFC is a local constraint solver that has been a topic of strong research interest since its introduction in 2016 by Gumin [2]. The power of WFC lies with its ability to quickly generate diverse grid-based output from the same input: either i) a set of tiles and a set of adjacency constraints, specifying which tiles may be adjacent, or ii) a grid-based texture example from which the tile set and constraints are inferred.

Most approaches deploy WFC on rectangular or cuboid tiles of the same size, spanning the same number of cells in a single grid; the tiles must thus be uniform. However, much can be gained from tiles with varying shapes, spanning multiple cells, which we call Non-Uniform Tiles (NUTs). NUTs allow WFC to enter currently unexplored domains where tile variety, shape and size preservation is crucial. This can include brick-by-brick construction of LEGO® models, Tetris-inspired textures [3] and varied three-dimensional terrains. Unfortunately, standard WFC cannot achieve this, due to the following limitations:

1) WFC is unable to represent and handle NUTs and cannot, therefore, guarantee the preservation of a NUT's diverse shape and size.
2) even if this obstacle were overcome, WFC has no mechanism in place for representing and operating on NUT adjacency constraints either. Contrary to constraints between uniform tiles, NUT adjacency constraints are ambiguous: there can be multiple configurations that satisfy such constraints.
3) WFC can learn from a grid-based input only if it

assumes that *every tile* occupies a uniform grid pattern (e.g. 1×1 or 3×3); however, it cannot distinguish NUTs within the input (by their own non-uniform nature) which, in turn, prevents WFC from preserving NUTs' properties.

To overcome these issues and provide access to the creative freedom and expressiveness of such new domains, we present Non-Uniform Tile Wave Function Collapse (nutWFC), an extension of WFC capable of supporting NUT sets. Most notably, nutWFC can guarantee NUT shape and size preservation under WFC's Overlapping Model. This means combining the essential WFC capacity of learning adjacencies from an input example with the consistent use of only the tiles included in the given NUT set. As a reminder, under the Overlapping Model, standard WFC learns allowed tile adjacencies by scanning with a sliding window over the input grid.

The main contribution of this article is the nutWFC algorithm. It was recently developed as a part of our earlier work Expressive Wave Function Collapse [4], where it was shown to truly be a super-set of standard WFC. The foundation of the underlying structure of NUTs lies with uniquely identifying its single-cellular elements. In other words, by disabling this identifier uniqueness within a NUT, standard WFC emerges. To support the above claims, we illustrate the application of nutWFC to urban layout and architectural structure generation, based on dedicated NUT tilesets.

This article is an extended and updated version of the nutWFC paper published at the IEEE Conference on Games 2025 [1]. The article includes a totally new Section V, featuring (i) the city skyline experiment, a new and clear example of nutWFC's expressiveness (see Figure 1), (ii) a detailed discussion of the versatile features of its NUTs and patterns, together with (iii) an extensive analysis of the diversity of its output and of its performance. Finally, the article contains several new and improved figures and captions, and a thorough revision of the text with numerous wording and phrasing improvements, for clarity and conciseness.

II. Related Work

WFC's release by Gumin in 2016 attracted significant attention from researchers and artists alike [2]. Gumin's WFC repository currently features dozens of variations, including adaptations, optimizations, generalizations and implementations to name a few [5]–[9]. WFC can also be found in commercial games, such as Bad North [10], Townscaper [11] and Caves of Qud [12]. While WFC rests on the same foundations as Merrell's Model Synthesis published years prior [13], WFC had a larger impact, especially among game developers [8]. Nonetheless, Merrell's contributions remain an important source of inspiration.

Several researchers have addressed tiles spanning multiple cells for WFC. Stålberg's Bad North features modules that can span multiple cells in a grid [10]. Stålberg acknowledges the useful properties of such large modules, such as allowing for smooth transitions. To solve the issue of formulating the numerous tile configurations, he derives the modules' adjacency constraints based on their vertices at the edges of the module: if they match, adjacency is allowed.

Moreover, Newgas' Tessera [7] and Piepenbrink's Expressive Wave Function Collapse [4] tackle multi-cellular tiles by subdividing them into uniquely identified single-cellular components. By enforcing constraints between these elementary components, the tile's structure is maintained. Newgas performs this method on multi-cellular tiles, which he calls *big tiles*, through the addition of pre- and postprocessing. Big tiles consist of a connected arrangement of single-cellular components called *cubes*.

In order to determine the adjacency constraints between big tiles, Newgas opts for manually painting the big tiles' exposed faces: two big tiles are allowed to be adjacent if their colored edges match. However, specifying only these constraints is insufficient to rule out the overlap of non-rectangular big tiles. When the colors of two edges match, that does not necessarily prevent overlappings on all other big tile components – which must be collapsed as well to preserve its structure. Consequently, this method is prone to conflicts and, as the irregularity of the tiles increases, the likeliness of conflicts will also increase. Although a valid tiling may be found, it will be at the high cost of frequent conflict handling. Alternatively, one could choose to paint the edges so that overlap is reduced. However, determining this manually, as Tessera requires, can be an increasingly challenging task for complex big tile configurations.

While Newgas' approach is effective for the Simple Tiled Model, it largely lacks the capabilities for operating on the Overlapping Model because it does not learn big tile adjacency constraints from a grid-based input. Instead, Tessera requires manual user annotations to make them explicit. We can therefore conclude that his method is strongly compromised for use under the Overlapping Model.

Piepenbrink, in turn, generalizes Newgas' approach to multi-cellular tiles of any shape, where the adjacency constraints between the tiles are expressed in terms of the tile's single-cellular components. Rather than employing (manually) colored edges to infer constraints, Piepenbrink extrapolates adjacency constraints on multi-cellular tiles in two ways. The first method adheres to the same principles described in his earlier work with Bidarra [3]. Given an adjacency constraint between two multi-cellular tiles for a given direction, his algorithm computes the positions of two tiles such that (i) two single-cellular components are adjacent in the corresponding direction, and (ii) no intersection of single-cellular components of the two tiles is present. From such a position, the adjacency constraints on single-cellular components are obtained.

In its second method, Expressive Wave Function Collapse [4] features learning adjacency constraints from an input example model. Since this input does not explicitly define the composition of its multi-cellular tiles, the NUT set used in the input is also expected as a secondary input. This enables distinguishing and identifying in the input every single-cellular component, thereby allowing for inference of adjacency constraints. Through these two methods, multi-cellular tiles are not restricted to 'rectangles' (i.e. the shape of their grid-aligned bounding box), and may be an arbitrary composition of connected single-cellular components. For the Simple Tiled Model, Expressive Wave Function Collapse uti-

lizes both methods just described; for the Overlapping Model it uses the second method only.

An orthogonal problem to that approached here regards the generation of large patterns or structures spanning multiple grid cells in the output, but *always based on a uniform tileset*. Karth et al. [14], for example, proposed a combination of WFC with a a vector-quantizing variational autoencoder (VQ-VAE) to generate game maps based on one input map. Once trained, this approach is able to output a variety of maps, possibly containing several large structures. However, the WFC algorithm itself, which is applied in latent space to a grid of integer indexes, operates on a simple grid with a uniform tileset. Bateni et al. [15] propose a modified WFC to improve the resemblance between input and output, based on a context-sensitive heuristic for choosing the next tile to collapse. This heuristic is shown to perform better than that in Gumin's original algorithm. Again, they always use either uniform (1×1) tiles or overlapping (3×3) patterns on a grid.

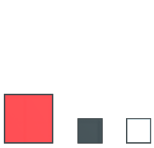
From a different perspective, Langendam and Bidarra's miWFC features stamp-like functionality [16]. Here, the designer is able to manually create a group of tiles, called a *template*, and later reuse it, by placing instances on the grid. One of the main advantages of this approach is that it offers the user control on the design process. This is akin to how NUTs can be perceived: a pattern of small tiles collapsed at once. Likewise, this notion extends to Alaka and Bidarra's HSWFC, following a hierarchical approach for the WFC tile set [17]. Here, one can separately specify, for each metatile, its tile (sub)set, adjacency constraints and weights. In some sense, different such metatiles can be seen as analogous to the multiple patterns accepted as input by nutWFC. Finally, both approaches share in our goal of better capturing the semantics of patterns and offering a tighter design control [18].

III. Non-Uniform Tile fundamentals

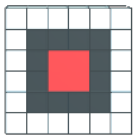
Plenty of domains require shape and size preservation for their diverse multi-cellular tiles; typical examples of these are LEGO® and Tetris. However, WFC is currently unable to cope with such heterogeneous tile sets. Consider the example shown in Figure 2 featuring a red square NUT, which shape and size must be preserved. With WFC, each single cell of the tile is considered equal, rendering it unable to preserve the tile's shape and size. With nutWFC, the red square NUT in the input remains intact. This property extends to arbitrarily shaped tiles.

Extending WFC to support NUTs, whose shapes and sizes must preserved, requires overcoming two challenges: i) storing NUT information at cell level and ii) expressing adjacency constraints between NUTs in a form usable for the WFC algorithm.

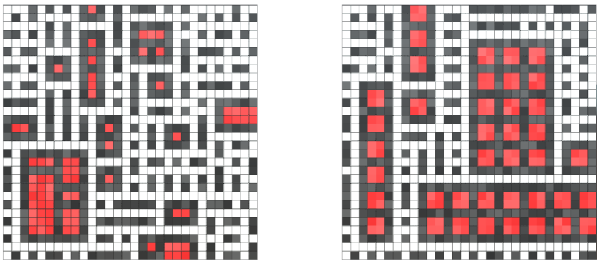
nutWFC solves the former challenge through splitting a tile into uniquely identified single-cellular units, which we call *atoms* (see Subsection III-A). To overcome the second challenge of NUT adjacency ambiguity (see Figure 3), we apply the aforementioned mechanism on the input grid, so that the adjacency of two NUTs can be expressed in terms of adjacency of their atoms (see Subsection III-B). As we will show, this can be done in the initialization phase of the



(a) The tileset used for the red square example.



(b) The input used for the red square example.



(c) Output generated by WFC (left), not preserving the shape and size of the red tile, and by nutWFC (right), preserving it. Color tone variations are added to reinforce each individual cell's character.

Fig. 2: Red square example, higlighting the contrast between WFC and nutWFC, arising from uniquely identifying the single cells of the red NUT.

Overlapping Model, enabling compatibility with WFC with minimal changes and marginal costs.

For the remainder of this section, we use the following notation:

- For a given vector $\mathbf{v}$, the $i\text{-}th$ element of the vector is referenced with $\mathbf{v}_i$.
- To reference properties of entities, we use dot notation. The extent of a given entity x is represented as Δ. For instance, to reference the $i\text{-}th$ element of x's extent, we write $x.\Delta_i$.
- Matrix cells will be referenced following array notation, i.e. the element at cell x, y of a matrix M is denoted as $M[x, y]$.
- Referring to grid cells is done similarly, where the cell at coordinate $\mathbf{c}$ is denoted as $G[\mathbf{c}]$.
- To represent a cardinal direction $\hat{d}$, we use unit vectors and say that $\hat{d} = \pm\hat{\mathbf{e}}_k$, where k corresponds to the cardinal direction's axis.

A. NUT preservation through tile atomization

The purpose of tile atomization is to split a NUT into a set of atoms. This allows for differentiation among those single-cellular units of the NUT, and opens the doors to WFC compatibility. Before we discuss NUT properties, we introduce a formal definition of NUTs:

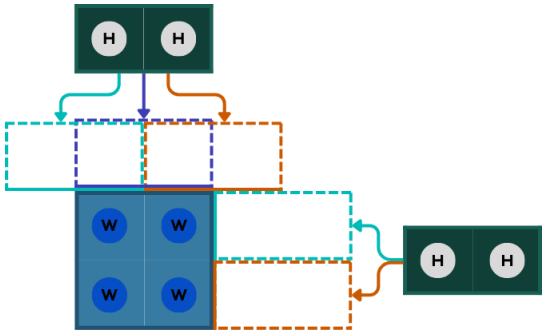


Fig. 3: Adjacency constraint ambiguity shown in 2D, indicating the possible positions satisfying the South-North and West-East constraints of the "H" tile with the "W" tile: three for South-North and two for West-East.

> ***Definition 1*** *(Non-Uniform Tile).* A *Non-Uniform Tile (NUT)* is a 3D tile that may span an arbitrary number of connected grid cells, and has an arbitrary shape and size that must be preserved. The shape is defined by a connected arrangement of uniquely identified single-cellular units called *atoms*, each with a relative position within the NUT. The size of the NUT, also referred to as *extent*, is the extent of the axis-aligned bounding box containing those atoms. Formally, a NUT n has an extent $n.\Delta$, and set of atoms $n.A = \{a_0, \ldots, a_m\}$.

From this definition, a NUT is an arrangement of atoms, whose relative positions within the NUT can be mapped to absolute coordinates within a 3D grid, based on four observations:

1) Cells are discrete and uniform within a grid.
2) Each cell contains at most one atom.
3) All atoms are single-cellular; they cannot overlap.
4) Each atom has a relative position within its NUT.

Since the atoms are single-cellular and cells are discrete, we can place the atoms according to their relative positions in a grid without any atoms overlapping. Each atom's coordinate within this grid is called the *atom coordinate* and is used to represent the atom's position. We denote the atom coordinate of an atom a as $a.c$. Since each cell can contain at most one atom, *the combination of an atom's associated NUT's identifier and atom coordinate is unique*. As a consequence, we can now distinguish between atoms within the same tile, which is crucial for NUT placement during the main loop of nutWFC. For ease of reference, we also assign a unique integer identifier to each atom. We denote the identifier of an atom a as $a.id$. Therefore, we can express a NUT in terms of the atoms within its extent, which brings us to the same domain as WFC's grid, with all its properties and advantages, including its handling of adjacency constraints.

Step one for NUT preservation is its atomization, yielding the building blocks that enable shape and size preservation. Per Definition 1, a NUT's atom arrangement, and thus its shape and size, must be preserved. We achieve this through specifying constraints between its atoms. In the grid of atoms we mentioned earlier, atoms can be neighbors of one another in a certain direction. We refer to that neighboring property as *atom adjacency*:

> ***Definition 2*** *(Atom adjacency).* Two atoms a_i and a_j are said to be *adjacent* in direction d if a_j is in a_i's neighboring cell in direction d. Atom adjacency between those atoms is denoted as $a_i \sim_d a_j$.

Whenever a NUT is placed (or collapsed) in the grid, its atoms must always occur in the same arrangement. Essentially, these can be expressed as adjacency constraints among the NUT's atoms, which we refer to as *inter-atom adjacency constraints*. These constraints will enforce that collapsing one of the atoms of a NUT requires collapsing all of its other atoms as well, thereby preserving shape and size. This follows from the observation that an atom a_i with an inter-atom adjacency constraint with an atom a_j in a direction d is not allowed to be adjacent to any other atoms in that direction. Thus, the atoms of a NUT are tightly bonded together. Since these relations are known beforehand, the constraints are determined in the initialization phase. These inter-atom adjacency constraints are fully compatible with WFC, since the constraints are expressed in terms of adjacency constraints between single-cellular units.

B. Adjacency between NUTs

With an individual NUT's properties preserved and compatibility with WFC achieved, we can turn into adjacency between NUTs. As illustrated in Figure 3, adjacency between NUTs is ambiguous and not directly compatible with WFC's tile adjacency. To overcome this, we have to find an extended definition for NUT adjacency. Because we expressed each NUT in terms of its differentiated atoms, we can use them to define NUT adjacency:

> ***Definition 3*** *(NUT adjacency).* Two NUTs n_i and n_j are said to be *adjacent* in a given direction d, if at least one atom of n_i is adjacent (see Definition 2) to at least one atom in n_j in direction d.

From this definition, the ambiguity becomes clear: there may be multiple configurations of two NUTs in which their atoms are adjacent. To express that two NUTs may be adjacent, we formulate NUT adjacency constraint in Definition 4.

> ***Definition 4*** *(NUT adjacency constraint).* A *NUT adjacency constraint* is a constraint stating that two NUTs may be adjacent along a given direction d, following Definition 3.

With this knowledge in mind, we can now describe how these constraints are learned from an input and used by the acrshortnutwfc algorithm.

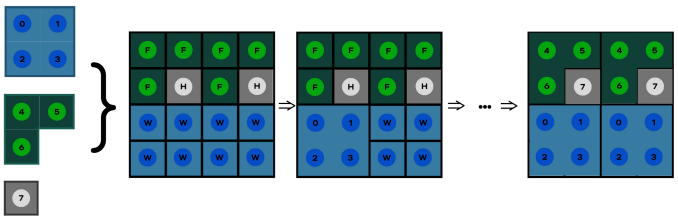


Fig. 4: Input grid atomization: given a set of atomized NUTs (left), we overlay and slide each NUT over the grid, to find a fitting NUT and associate its atom identifiers to the cells in the grid.

IV. Core nutWFC algorithm

A. Grid atomization: learning from an input

WFC traditionally learns the tile set and adjacency constraints from input examples, typically a pixel texture. During the learning phase, the input is scanned to derive both the tile set and the adjacency constraints.

However, when that input is already built up of NUTs, the NUT set is evidently prior to the input itself, rather than derived from it. Therefore, it is arguably a reasonable requirement to pass along, as input, both the NUT set and one (or more) grid example(s) using those NUTs. Typically, each example input involves a small subset of NUTs and is aimed at specifying how they are to be used. Because each NUT's atoms are uniquely identified, we are guaranteed to find a valid configuration of NUTs in each provided input such that the NUT placements cover the input without overlapping.

Certainly, standard WFC cannot cope with such a heterogeneous tileset anyway; but how can nutWFC learn from such an input? The answer to this is given by the notion of *grid atomization*, analogous to the mechanism behind tile atomization, described in Subsection III-A.

Input grid atomization essentially consists of determining a NUT's atom identifier for each of its cells. For this, the cells in the input grid G can be mapped to a matrix, G', in which each cell's value is the atom identifier of the NUT at that atom coordinate. For this, grid atomization follows a greedy sliding window approach, incrementally iterating over all cells of the input grid, starting at its zero-index, until all cells are assigned exactly one atom id. For each cell with coordinate $\mathbf{c}$ in grid G, we overlay a NUT n_i with extent $n_i.\Delta$ and atoms $n_i.A$ such that all atoms $a_j \in n_i.A$, with their respective atom coordinates $a_j.\mathbf{c}$ cover cells $\mathbf{c} + a_j.\mathbf{c}$. We say that a NUT *fits* in the input grid at a given coordinate $\mathbf{c}$, if for all atoms a_j of NUT n_i, the atom's identifier equals the value stored in grid G at coordinate $\mathbf{c} + a_j.\mathbf{c}$ and if the cells of G' at these coordinates are not yet mapped to an atom id. Figure 4 depicts a high-level illustration of this mapping. By starting from the $\mathbf{0}$ coordinate in G and selecting cells incrementally thereafter, we align the NUTs with this coordinate.

After each successful NUT fitting check on the input grid G, we map its atoms as well onto the atomized grid G'. For this, we store the identifier $a_i.id$ of an atom a_i at $\mathbf{c} + a_i.\mathbf{c}$.

Once the input has been atomized, the adjacency constraints between its atoms (and their respective NUTs) can be derived for the Overlapping Model, as described next.

B. Patterns for the Overlapping Model

The key for deriving and capturing nutWFC adjacency constraints between NUTs lies in the notion of a *pattern*, a small square portion of the atomized grid G'. A kernel w with extent $w.\Delta$ is used to obtain the patterns from the atomized grid G'. For this, the kernel slides cell-by-cell over the entire grid: at each position, one pattern is registered, and kept in set $P = \{p_0, \ldots, p_n\}$. From the definition of the Overlapping Model, an adjacency constraint between two patterns depends on how they overlap, hence the minimal size of the kernel is $\mathbf{2}$.

Pattern adjacency constraints are obtained by overlapping any two such patterns along a given direction $\hat{d} = \pm\hat{\mathbf{e}}_k$, and checking their compatibility, as shown in Algorithm 1. Let p_i and p_j be two patterns obtained as described above. Each pattern aggregates a square of atoms, with their identifiers. Consequently, each atom has a position within the pattern, which we refer to as a *pattern coordinate*. Evaluating whether two patterns are compatible (see line 6 of Algorithm 1), and their NUTs may therefore be adjacent for a given direction $\hat{d}$, involves the following procedure:

1) Position patterns p_j and p_i such that they fully overlap. That is, each pattern coordinate $\mathbf{c_i} \in p_i.C$ maps to a pattern coordinate $\mathbf{c_j} \in p_j.C$.
2) Translate p_j by $\hat{d}$; not all pattern coordinates overlap anymore.
3) For each pair of overlapping pattern coordinates $(\mathbf{c_i}, \mathbf{c'_j})$, check whether the value at $\mathbf{c_i}$ equals the value at $\mathbf{c'_j}$.
4) If all assertions pass, p_j and p_i are said to be *compatible* along direction $\hat{d}$.

One of the advantages of this use of NUTs in nutWFC is that a small kernel of size $\mathbf{2}$ can convey much more information than with standard WFC, due to NUT atomization: the patterns obtained through input processing contain atom identifiers rather than tiles. Since each atom belongs to a concrete NUT, collapsing a pattern on the output grid, and thereby the atoms in said pattern, will enforce that all other atoms of those NUTs (possibly outside of the pattern extent) must be collapsed as well. In other words, atomization facilitates maintaining the shape and semantics explicitly desired for each NUT.

In addition, patterns containing only atoms of the same NUT become redundant, since those atom relations are already enforced through the inter-atom adjacency constraints, maintained in the atom adjacency matrix. The larger a NUT is, the fewer patterns are required to represent it (relative to its size). While this sounds counter-intuitive, it makes sense: the purpose of a pattern is to contain information on how two (or more) NUTs may be adjacent in a given context. For a large NUT, only patterns at its perimeter contain information on how it touches others NUTs, all other internal patterns can be ignored.

Algorithm 1 Pattern Adjacency Inference

```
D = {d_0, ..., d_k}                          ▷ Set of directions
P = p_0, ..., p_n ▷ Set of patterns obtained form the input
for each p_i in P do
    for each p_j in P do
        for each d in D do
            if IsCompatible(p_i, p_j, d) then
                StorePatternAdjacency(p_i, p_j, d)
                StorePatternAdjacency(p_j, p_i, −d)
            end if
        end for
    end for
end for
```

Algorithm 2 nutWFC for the Overlapping Model

```
Initialize()
while not all cells are collapsed do
    Observe()
    Collapse()
    CollapseNUTsInPattern()
    Propagate()
end while
```

C. nutWFC for the Overlapping Model

The basic nutWFC algorithm uses the features and properties described above when extending the standard WFC algorithm, as shown in Algorithm 2.

During observation, the cell with the lowest entropy is selected and then collapsed. However, since the domain of the Overlapping Model is unionized — operating on both patterns and atoms — collapsing works slightly differently. At run-time, we must know which cells are collapsed to which atom *within* the pattern. Therefore, upon collapsing a cell into a pattern, we assign it the atom identifier at pattern coordinate $\mathbf{0}$. Subsequently, a new collapse wave propagates, where all other atoms belonging to that atom's NUT are also collapsed. However, a new challenge arises with this step: immediately collapsing cells into atoms, rather than according to patterns, results in a lack of pattern data. To overcome this, we precompute at initialization stage, the table of which patterns correspond to which atom identifiers. Then, at run-time we easily retrieve from this table, for each collapsed cell, which patterns correspond to their atom identifiers. In this way, we safely restore the correspondence between atom identifiers and patterns, and avoid including in the pattern adjacency matrix any patterns containing only atoms belonging to the same NUT.

D. Conflict handling

When a cell runs out of options, a conflict is encountered and WFC would fail, even though a solution could exist in the search space. By exploring this search space, the likelihood of finding a solution increases. Backtracking, for example, is guaranteed to return a solution should one exist, as it can traverse the whole search space [7], [16]. However, backtracking can only guarantee this at the cost of efficiency. For our current implementation, whose main focus was NUT set exploration rather than efficiency, we opted for a simpler conflict handling method employing save states. This method works by periodically creating save points that consist of the wave and a grid containing the collapsed atoms (nutWFC's Overlapping Model requires both). When a conflict is encountered, i.e. when a cell runs out of options, nutWFC is reverted to a previous save point to try again from there. The use of save points favors efficiency due to its lower time complexity, but cannot guarantee that a solution will always be found should it exist.

V. Results and Discussion

In this section, we showcase and discuss the results of the **city skyline experiment**. It features a city skyline consisting of a road network with scattered patches of grass, upon which a variety of buildings may stand. Its purpose is to illustrate how NUTs can help (i) strengthen creative freedom compared to standard WFC, and (ii) maintain the desired meaning for different components of the output. Moreover, we include a **performance study**, focused on the costs of conflict resolution in this experiment. Finally, we perform an **extensive diversity analysis**, for which over 400 city skyline outputs have been generated.

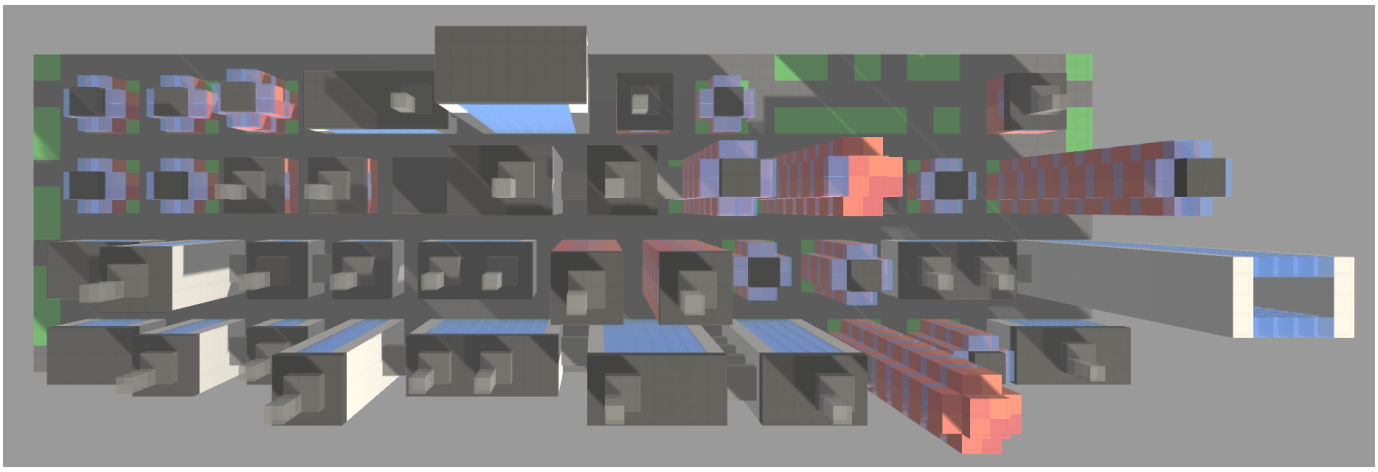
A. City skyline experiment

Some results of this experiment are shown in Figures 1 and 5. They clearly illustrate how nutWFC goes beyond standard WFC, effectively harmonizing output diversity with greater versatility. Several features in the aforementioned figures support this claim: the NUT set, the input patterns, and the characteristics of the output, which we now discuss in order.

The city skyline is built in a grid of size (w,h,d) $80 \times 25 \times 24$ using the NUT set and the inputs shown in Figure 6, from which a total of 480 patterns of size $\mathbf{2}$ were created at nutWFC initialization stage. **The NUT set** consists of NUTs of various shapes and sizes, ranging from simple cuboids to more elaborate shapes, like the red corner NUTs. With these NUTs, a road network can be created with scattered land patches upon which buildings of various shapes and sizes can be constructed. The same NUTs, such as the window panes or the red corners, can be used in different contexts without causing interference between the distinct building types. Even though the red square building and the red cross building share the same NUTs, the red square building cannot be partially comprised of sections of the red cross buildings and vice versa.

nutWFC features a modular approach utilizing **multiple inputs**, each representing a variety of allowed combinations, thus avoiding a combinatorial explosion of required input configurations. Moreover, each input, using only a few NUTs, typically focuses on conveying a clear meaning for a specific component of the output. The WFC patterns obtained from each input are matched during pre-processing according to their overlap, as described in Subsection IV-B. Due to the modularity of the inputs, buildings can be represented per section: (i) a base indicating how a building's NUTs may be placed and stacked on land surrounded by roads (e.g. inputs

(a) An elevated front view of the city skyline.



(b) A top view of the city skyline.

Fig. 5: The results of the city skyline experiment. Grid size (w,h,d): $80 \times 25 \times 24$.

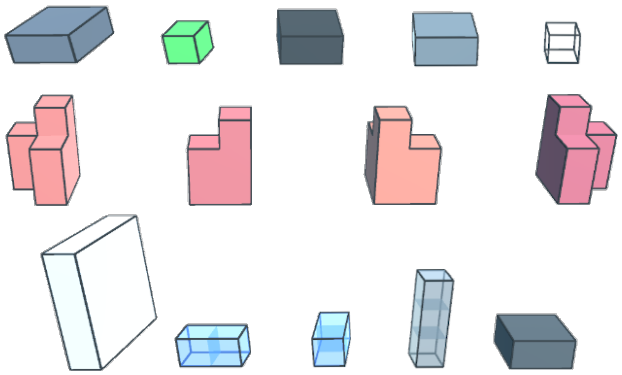
1) and 3) in Figure 6c), and (ii) a top that specifies how they may be topped with a roof (e.g. inputs 2) and 4) in Figure 6c). Combined with input 1) in Figure 6d, it follows that a roof may either be bare, or decorated with an antenna.

Altogether, these small modular inputs suffice for generating a **city skyline output** striking a balance between diversity and predefined structure. The road network, for example, is generated using only two inputs (see Figure 6b) and has a relatively high degree of freedom: roads may branch and spread, as long as the road NUTs are flush and remain intact. Figure 5b portrays long horizontal avenues on the bottom layer of the output, resembling a Manhattan-esque road network. Notice how the streets in the network are all composed of two-by-two road NUTs. Scattered patches of land complete the bottom layer, and are likely to be four cells deep, due to the depth of the building bases. This trend contributes to the typical appearance and structure of an urban landscape, with consistent blocks and straight avenues. However, this is but one permutation; the road network in Figure 1 features shorter streets and more patches of open land.

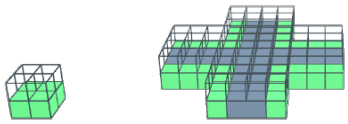
The buildings in Figure 5a are densely packed, yet varied. For instance, the red buildings are restricted to four-by-four width and depth, but their height may increase in multiples of three (excluding the roof), as the red corner NUTs are three atoms tall and must be stacked in order to rise. The two red building variations comprise the same NUTs, yet have a distinctly different appearance. Whereas the red square building is fairly straightforward, the red cross building is slim and has multiple variations in terms of roofing.

The white-blue buildings are designed differently and may rise in multiples of four, constrained by the height of the white wall NUT. They may expand in width in multiples of two, as its window panes are two atoms wide and may be self-adjacent along the x-axis (see Figure 6d). These properties promote the diversity seen in the white-blue buildings of Figures 1 and 5a, ranging from narrow and tall to wide and short. White-blue buildings, therefore, are not required to exactly resemble the input, which has a width of six.

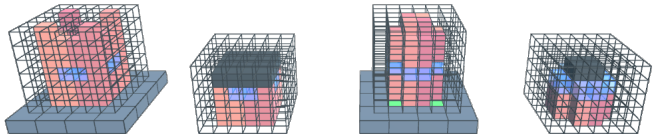
The buildings are then topped with a roof and possibly an antenna as well. If there is at least a two-by-two arrangement of roof NUTs, an antenna may be placed (see Figure 6d). The wide short building in the front center of Figure 5a,



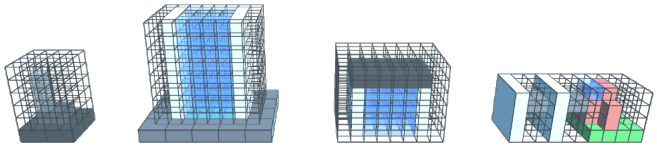
(a) The NUT set used in the city skyline experiment. From left to right. Top row: road, land, roof, skywalk, void. Middle row: corner NE, corner SE, corner SW, corner NW. Bottom row: white wall, window pane X, window pane Z, antenna, antenna base.



(b) The inputs used for generating the road network.



(c) The inputs used for generating the red buildings. From left to right: 1) red square building base 2) red square building top, 3) red cross building base and 4) red cross building top.



(d) The inputs used for generating the white-blue buildings. From left to right: 1) antenna on roof, 2) white-blue building base, 3) white-blue building top, and 4) skywalk

Fig. 6: The inputs used for the city skyline generation.

for example, has two antennas, whereas the left-most in the front has none. Yet, the white-blue building on the far right is roofless. This is due to the height of the grid, which perfectly lines up with the height of the building: one layer for the ground, 24 for the six stacked white walls, leaving no room for a roof since the grid is 25 cells tall. Therefore, there are numerous nuances one can tweak with nutWFC to steer the results in a certain direction, be it through the height of the NUTs, the composition of the input patterns, or the size of the grid.

Other interesting design intent can be easily captured in nutWFC as well. The skywalk NUT (see Figure 6a) illustrates this property in two ways: it can connect two white walls, or a white wall to the left of two red corners (see Figure 6d). The former configuration is relatively simple: it enforces that the skywalk may only be placed in the specific configuration shown in the aforementioned input. Figure 5a shows how a subset of white walls are connected. This can result in several skywalks (such as the two front tall buildings, with four), one skywalk at varying heights, or none. The latter skywalk configuration is more nuanced: it can be used to specify constraints related to common denominators, which can be used to enforce the position of certain NUTs, like the skywalk. Here, the red corners of the cross building may be adjacent to the skywalk at relative height two in that configuration. The white wall may feature a skywalk at relative height three. Since both buildings' bases must be placed on land and both buildings rise at different rates, this configuration only occurs at the common denominator of their heights. In other words: once every three stacks of white walls and once every four stacks of red corners. This explains the position of the skywalk seen between the white and red building in Figure 5a (in front, slightly to the right) and in Figure 1 (in front, slightly to the left).

Contrasting these features with WFC, one can but realize that WFC could not achieve similar result, by the same reasoning explained in Section III, Figure 2. For instance, roads would not be guaranteed to have the same constant width, resulting in a messy network. The buildings would be inconsistent as well, especially due to the white walls, which could randomly grow in depth and height. In short, WFC would fail to create a recognizable and relatively consistent city skyline as above.

B. *Performance analysis*

To gain insights regarding nutWFC's performance, we measured the duration of 583 runs of a city skyline generation. The execution times have been plotted in Figure 7. The distribution in the histogram clearly indicates that most outputs took around 290 seconds to create. This tall peak around 290 seconds is indicative of a permissive set of inputs: despite the constraints, nutWFC was able to find a valid tiling fairly consistently. However, in case of propagation conflicts, the execution time increases up to roughly 550 seconds, with a tail running up to approximately 1200 seconds.

We have identified two main potential directions for performance improvements: backtracking and propagation. The distribution of the results charted in Figure 7 provides valuable insights regarding the save point alternative to backtracking we employed (see Subsection IV-D). We hypothesize that might the search space have been more thoroughly explored while handling conflicts, neither the dip around 400 seconds nor the peak around 550 seconds would be so prominent. The long tail reaching outliers beyond 1200 seconds is interesting: conflicts were likely only detected towards the end of an attempt, and were resolved by returning to an early stage, effectively resetting the grid. Our findings therefore counter our hypothesis that the save point method implemented would be an acceptable backtracking approximation. Although these outliers account for less than 15% of the outputs, we recognize that there is plenty of room for performance improvement, and

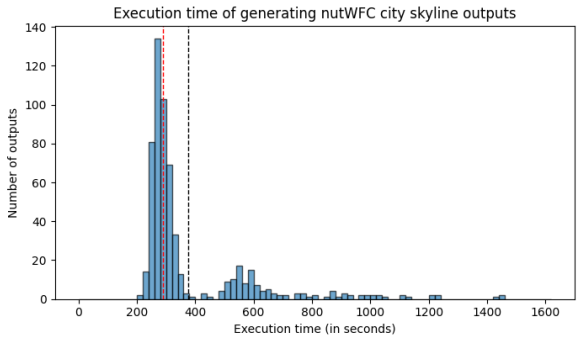


Fig. 7: Durations of 583 nutWFC city skyline outputs. Mean (dashed black line): 376.76 seconds. Median (dashed red line): 290.54 seconds.

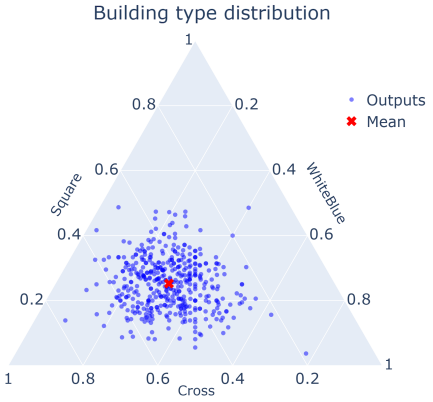


Fig. 8: Building type distribution per output city (n=437).

that it would be worth experimenting with more sophisticated backtracking methods.

We think that most overhead during a generation attempt is caused by the propagation wave: when a cell's wave has been updated, finding the union of the set of allowed neighbors for each cardinal direction, through a series of logical operations on arrays, proves to be time consuming due to the numerous patterns used for generation. The use of C# libraries optimized for logical operations on arrays or matrices could prove beneficial and these operations may also lend themselves for parallelization, as they are mostly independent. Both options are worth exploring. A more comprehensive discussion of performance can be found in Chapter 7 of Piepenbrink's earlier work [4].

C. Diversity analysis

We have analyzed 437 city skyline outputs to assess the output diversity of nutWFC's city skylines. For this analysis, each output can be conceptually subdivided into two components: the city layout, or bottom layer of the grid, and the buildings built upon land plots. The city layout comprises land plots covered by buildings ($\mu = 35.5\%$ $\sigma = 4.0\%$), land plots without buildings ($\mu = 12.9\%$) and roads ($\mu = 51.6\%$ $\sigma = 4.5\%$). We deem this variety distribution quite reasonable and aligned with our design intent: (i) plots are surrounded by roads, thus a high road percentage, and (ii) a plot should have roughly 75% chance to host a building – one of three, or none – and we observed 73.4% on average. Each city skyline hosts on average 36.7 buildings, with a standard deviation of 4.5, thus supporting a considerable layout variation. By analyzing the variety of these buildings, we can gather further insights into output diversity.

First, we analyze the building type distribution. Figure 8 provides insights into the extent to which each building type (red cross, red square and white-blue) is present in the output city. As reflected in the chart, diversity is characterized by a wide spreading of the building type distribution. The indicated mean is slightly off-center, leaning towards the red cross type. The building distribution across all outputs supports this observation: of the 16055 buildings placed, 7171 (44.7%) are red cross, 4807 (29.9%) are white-blue and 4077 (25.4%) are red square.

The red cross' dominance is expected. The reason lies in (i) the fact that the red cross building is not hollow – unlike the other two buildings – and (ii) pattern elimination. To illustrate, consider a four-by-four plot of land. The cells above are not yet collapsed. From the input patterns, patterns of size **2** are obtained, as explained in Subsection IV-B. For the skyline experiment, this results in four options for the plot of land: red square, red cross, white-blue or void (see Figure 6b and the building base patterns in Figures 6c and 6d). While the bases are different, the white-blue and red square buildings have a hollow two-by-two (x,z) core, filled with voids, which coincides with the pattern size. Since duplicate patterns are eliminated, the cell corresponding to the core only has two allowed patterns: void or red cross, both of which are weighted equally. As nutWFC employs the lowest entropy heuristic, this cell will be collapsed earlier; the other cells have a larger set of allowed patterns. Therefore, for a four-by-four plot of land, there is a 50% chance to host a red cross building, against 25% chance each for red square and white-blue, which is close to what we observe in the results. For larger plots of land, white-blue is the only allowed building, explaining its higher frequency compared to the red square building.

Second, we take a closer look at the variation among buildings. Each building has a flexible height, as a result of stacking floors. Figure 9 shows that buildings with very diverse heights are indeed present. However, red cross buildings' height seems atypical, especially when compared to the red square buildings' height. The main difference between these buildings placement conditions is that a skywalk between white-blue and red cross buildings may be placed (by pattern 4 in Figure 6d). Upon placing such a skywalk, the building at the other end must be at least as tall. For the white-blue red cross skywalk pattern, a skywalk can be placed at four or eight red cross floors. Across all outputs, 407 white-blue red cross skywalks could be realized, of which 380 were actually placed. However, the frequency deltas in the figure outnumber the skywalks. So, this does not fully account for building height trends. We hypothesize that (i) a skywalk's effects compound beyond its direct context, or (ii) a slight bias is present in our implementation of nutWFC randomness. The red square's nigh absence at three to five floors, with peaks at six and seven, and the red cross' peak at eight floors seem to give evidence

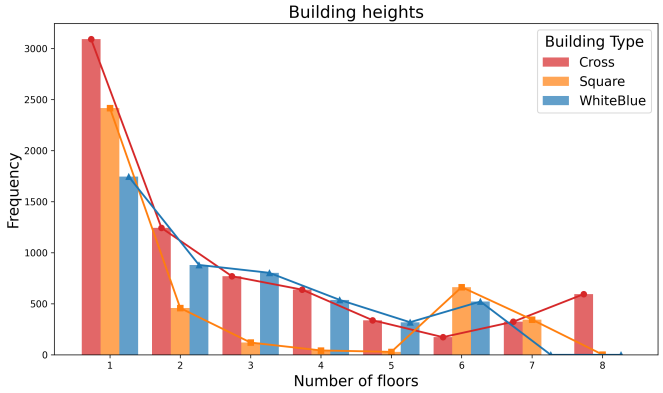


Fig. 9: Building height distribution, per building type. Grid height: 25. Red square and red cross: a floor is 3 atoms tall, max. 8 floors. White-blue: a floor is 4 atoms tall, max. 6 floors.

of this same effect. Most likely, there is a combination of (i) and (ii). We think more research is required to reach a decisive conclusion, as the white-blue buildings do not showcase either of these behaviors.

The distribution of the white-blue buildings' heights can be explained through the presence of skywalks. Upon collapsing cells into a skywalk, the building at the other end must be at least as tall, thus resulting in external factors nudging the creation of taller buildings. Of the 4807 white-blue buildings, 56% are adjacent pairs, that is both have same z-coordinate, separated by a road only (e.g. the front row of Fig 5a has six adjacent white-blue pairs).

The chart in Figure 10 sheds more light on the skywalks. When two white-blue buildings are adjacent, the maximum of skywalks that *can* be placed between them is the amount of floors of the lowest building: we call this the pair's (skywalk) *capacity*. The *realized* skywalks, in turn, is the number of skywalks that *are* actually placed between them. In this chart, we see there is a clear tendency to generate skywalks if possible, especially for lower capacities. In addition, the the white-blue buildings are likely to form chains of varying lengths ($\mu = 2.10$ $\sigma = 1.75$), with skywalks acting as connectors. Furthermore, the height variation between the buildings in the same chain is more likely to be similar, with a median height difference of 1 floor.

Contrary to the interplay of inputs affecting the heights and skywalks, the diversity in white-blue building widths is fairly straightforward, as illustrated in Figure 11. The data resembles a geometric distribution, as expected. In the inputs, we specify two options for the variable width of the white-blue buildings (shown in Figure 6d). Consider the event of placing a window pane while constructing a white-blue building. Next to pane, either (i) another pane is placed to increase the building's width, or (ii) a white wall is placed, acting as a terminal. Event (i) can occur in succession, akin to a series of coin tosses, until event (ii) happens. There also appears to be no correlation between the heights and widths of white-blue buildings as the heights distribution is fairly consistent across all widths. From these observations, we derive that white-blue buildings and skywalks are characterized by high output diversity in line with the design intent.

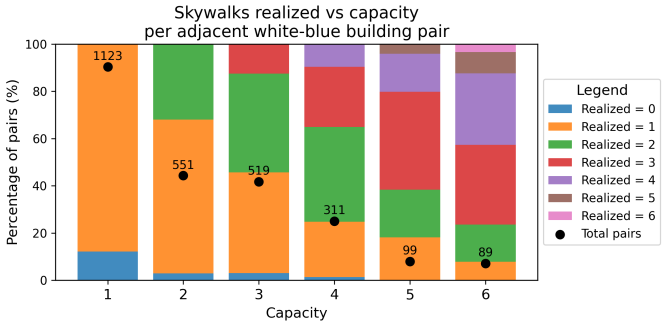


Fig. 10: Skywalks placed (realized) vs number of skywalks that can be placed (capacity) between a pair of adjacent white-blue buildings.

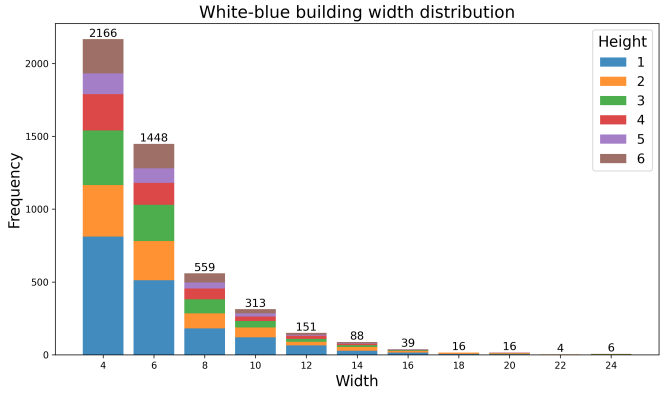


Fig. 11: White-blue building width distribution, decomposed into heights.

VI. Conclusion

Despite its popularity and versatility, the WFC algorithm is unable to work with heterogeneous tile sets, consisting of tiles of disparate shapes and sizes. We presented Non-Uniform Tile Wave Function Collapse (nutWFC), a major extension of WFC that introduces this possibility, through the notion of Non-Uniform Tile (NUT), a rigid agglomeration of distinct one-cellular atoms.

We described how nutWFC extends WFC to work under its Overlapping Model, under the condition of preserving NUT shape and size in all generated output. In addition, we showed how WFC can be seen as a particular case of nutWFC, in which all atoms of each NUT have the same identifier. This article considerably extends and updates our previous work, presented at IEEE CoG 2025 [1]. In particular, it includes a totally new Results and Discussion section, where we (i) thoroughly explore the expressiveness of a very simple yet versatile city skyline tileset, (ii) discuss the algorithm's performance, deducing potential improvements in nutWFC's propagation and backtracking approach, and (iii) analyze and discuss many diversity aspects of the generated city skyline output, providing evidence for nutWFC achieving quite meaningful content diversity.

Among nutWFC's most innovative and attractive features, we exemplified (i) the easier and more intuitive fine-tuning provided by individual input patterns, each of which helps clearly specify and preserve the intended semantics; (ii) the much richer and nuanced input enabled by nutWFC, which allows for a larger expressive power. The latter is especially promising for content types that can be naturally regarded as composed of distinct, interrelated part. We therefore believe nutWFC has the potential to impact new areas of content generation, in particular in a mixed-initiative context, empowering the incremental expression of a designer's intent. For this reason, it would be interesting to further investigate how such interactive facilities can best profit from the procedural core of nutWFC, developed for our current prototype implementation.

REFERENCES

[1] R. Piepenbrink and R. Bidarra, "Non-Uniform Tile Wave Function Collapse," in *Proceedings of IEEE Conference on Games 2025*, IEEE. Lisbon, Portugal: IEEE Press, 2025, pp. 1–8.

[2] M. Gumin, "Wave Function Collapse," https://github.com/mxgmn/WaveFunctionCollapse, 2016, accessed: 2024-08-15.

[3] R. Piepenbrink and R. Bidarra, "How much Tetris can Wave Function Collapse put up with?" Delft University of Technology, Delft, The Netherlands, Tech. Rep. CGV-24-1, May 2024. [Online]. Available: http://graphics.tudelft.nl/Publications-new/2024/PB24

[4] R. Piepenbrink, "Expressive Wave Function Collapse," Master's thesis, Delft University of Technology, August 2024. [Online]. Available: https://resolver.tudelft.nl/uuid:cc11b0d7-82b5-48e5-adc2-d5033a6ab661

[5] I. Karth and A. M. Smith, "WaveFunctionCollapse is constraint solving in the wild," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017, pp. 1–10.

[6] ——, "WaveFunctionCollapse: Content generation via constraint solving and machine learning," *IEEE Transactions on Games*, vol. 14, no. 3, pp. 364–376, 2021.

[7] A. Newgas, "Tessera: A practical system for extended WaveFunctionCollapse," in *Proceedings of the 16th International Conference on the Foundations of Digital Games*, 2021, pp. 1–7.

[8] ——, "Infinite modifying in blocks," https://www.boristhebrave.com/2021/11/08/infinite-modifying-in-blocks/, 2021, accessed: 2024-08-15.

[9] Marian42, "Wave Function Collapse - an algorithm for generating random structures," https://marian42.de/article/wfc/, 2020, accessed: 2024-08-16.

[10] O. Stålberg. (2018) EPC2018 - Wave Function Collapse in Bad North. Youtube. [Online]. Available: https://www.youtube.com/watch?v=0bcZb-SsnrA

[11] ——, "Townscaper," https://store.steampowered.com/app/1291340/Townscaper/, 2021, accessed: 2024-08-15.

[12] F. Games, "Caves of Qud," https://www.cavesofqud.com/, 2015, accessed: 2024-08-15.

[13] P. Merrell and D. Manocha, "Model synthesis: A general procedural modeling algorithm," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 6, pp. 715–728, 2010.

[14] I. Karth, B. Aytemiz, R. Mawhorter, and A. M. Smith, "Neurosymbolic map generation with VQ-VAE and WFC," in *Proceedings of the 16th International Conference on the Foundations of Digital Games*, 2021, pp. 1–8.

[15] B. Bateni, I. Karth, and A. Smith, "Better resemblance without bigger patterns: Making context-sensitive decisions in WFC," in *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 2023, pp. 1–11, accessed: 2024-08-16.

[16] T. S. Langendam and R. Bidarra, "miWFC - designer empowerment through mixed-initiative Wave Function Collapse," in *Proceedings of the 17th International Conference on the Foundations of Digital Games*, 2022, pp. 1–8. [Online]. Available: https://publications.graphics.tudelft.nl/papers/45

[17] S. Alaka and R. Bidarra, "Hierarchical Semantic Wave Function Collapse," in *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 2023, accessed: 2024-08-16. [Online]. Available: https://publications.graphics.tudelft.nl/papers/67

[18] ——, "Mixed-initiative generation of virtual worlds - a comparative study on the cognitive load of WFC and HSWFC," in *Proceedings of the 19th International Conference on the Foundations of Digital Games*, 2024, accessed: 2025-04-16. [Online]. Available: https://publications.graphics.tudelft.nl/papers/23

Rolf Piepenbrink received a Master's degree in computer science from Delft University of Technology, The Netherlands. He is an independent researcher working as a Software Engineer. His research interests include procedural content generation and serious game development.

Rafael Bidarra graduated in 1987 in electronics engineering at the University of Coimbra, Portugal, and received his Ph.D. in 1999 in computer science from Delft University of Technology, The Netherlands.

He is associate professor Game Technology at the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology. He leads the research lab on game technology at the Computer Graphics and Visualization Group. His current research interests include procedural and semantic modeling techniques for the specification and generation of both virtual worlds and gameplay; serious gaming; game adaptivity and interpretation mechanisms for in-game data. Rafael has numerous publications in international journals and conference proceedings, integrates the editorial board of several journals, and has served in many conference program committees.