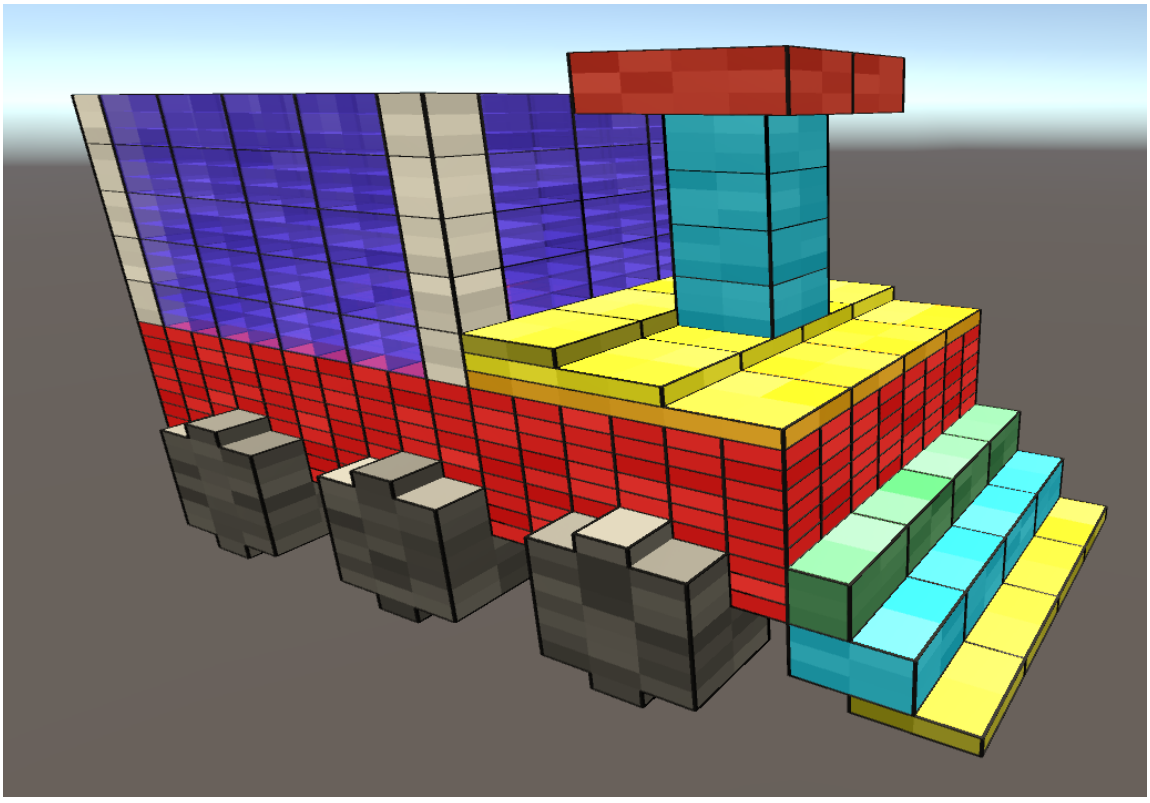
Expressive Wave Function Collapse

Rolf Piepenbrink

Master's Thesis



Delft University of Technology

TU Delft

Expressive Wave Function Collapse

by

Rolf Piepenbrink

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 30, 2024 at 3:00 PM.

Student number:	4689496	
Project duration:	October 9, 2023 – August 30, 2024	
Thesis committee:	Dr. ir. R. Bidarra,	TU Delft, supervisor
	Dr. A. Lukina	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Procedural Content Generation methods enable the creation of varied content algorithmically. Wave Function Collapse (WFC) is one such method. It is a tile-based local constraint solver commonly applied to world and map generation for grid-based content; it is able to create varied output from the same set of rules. While WFC is versatile, content generated with it is i) limited to one grid, ii) based on uniform tiles and iii) must use the same tile set and constraints throughout the grid. Due to these limitations, large classes of content are excluded, such as structured objects. Structured objects consist of an assembly of multiple components, each possibly based on a different tile set. We therefore propose Expressive Wave Function Collapse (XWFC), a major extension of WFC that enables solving and combining multiple grids with different Non-Uniform Tile (NUT) sets. Additionally, we can guarantee NUT shape and size preservation even under WFC's Overlapping Model. With these generalizations, new domains are within reach for structured objects based on NUT sets, such as Tetris or LEGO.

Preface

Dear reader, before you lies the master's thesis "Expressive Wave Function Collapse", the result of nearly a year's worth of effort and commitment. It was also a year's worth of playing with LEGO and Tetris, a strange idea. In summer 2023, my supervisor Rafa and I were sparring and, at one point, we were hypothesising how to procedurally generate simple LEGO cars — a goal that sounded too ambitious. But, what if? It was that simple question that sparked this research. It was also that question that resulted in me demonstrating part of my work at the FDG 2024 conference in Worcester.

On a more personal note, the past year was a year of challenge, change and growth. It was a year of challenge, for the daunting task of working on a thesis presented itself. However, as my research progressed, the direction I was heading became clear. It was a year of change, for I spent three months in New York City to collaborate with researchers at the Game Innovation Lab, New York University. During the early stages of the sparring sessions with Rafa, he asked me if I wanted to go to New York and collaborate with researchers there, even though I did not even have a concrete research topic yet. However, my research topic ended up aligning perfectly. And so, I went.

My time in New York was a life-changing experience, it was one of the greatest decisions I ever made. During the three months I stayed there (March 4 - May 28 2024) I got to experience things unlike anything I ever experienced: true self-reliance, independence, navigating in an unknown and bustling environment far-away and finally thriving. I felt a stranger, yet strangely at home. These experiences changed me for the better.

It was a year of growth, for I have overcome the challenges that arose, both academically and personally, and I can finally say that I am proud of what I have achieved.

My thanks extend to people who aided me throughout the year, each in their own helpful ways. Without these people, I would not stand where I stand now. Rafa, I am incredibly grateful for having you as my supervisor, mentor and (city) guide. You always asked the right questions and knew how to spark my creativity. I will remember especially fondly the week we spent (cycling) in New York and our time at the conference. I want to thank Hedwig, Eric, Benk and Anne for their unconditional support and also for visiting me in New York, that was an amazing time. I want to thank Lili for her unconditional love and for listening to someone ramble about LEGO and Tetris. My dear friends, among whom Dylan, Daan, Mark, Timon, Martijn and Angela, thank you for helping me and for being the great people you are. Finally, I want to thank the people at the Game Innovation Lab, Julian, M, Roman, Yuhe, Sam, Tim, Zehua, Matt, Graham and others, for having me and for making me a part of your lab community.

Rolf Piepenbrink
Delft, August 2024

Contents

1

Introduction

The sheer prevalence of Procedural Content Generation (PCG) is reflected in the numerous games and tools that have adopted its various automated algorithmic content generation techniques. It is for instance used for creating expansive and diverse worlds (MineCraft[1], Caves of Qud[2] and No Man's Sky[3]), randomized level play-throughs (Dead Cells[4] and Hades[5]) and generated architectural designs (CityEngine[6], TownScaper [28] and methods discussed by Kelly & McCabe [11]). Wave Function Collapse (WFC) is one such highly versatile PCG technique.

WFC is a local constraint solver that has been a topic of great academic interest since its introduction by Gumin in 2016 [7]. 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 adjacent or ii) a grid-based texture example from which the tile set and constraints are inferred. In recent years, several authors have embraced WFC and have improved upon it in from different perspectives. This ranges from grid generalizations [29, 12], additional constraint support [20, 25], mixed-initiative solving [1, 13], algorithm optimizations [19, 21] and tile set modifications and generalization [21, 1, 20, 28, 29] to name a few. We are particularly interested in the latter.

Most approaches employ WFC operating on rectangular or cuboid tiles of the same size, spanning the same number of cells in a 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 can allow WFC to enter currently unexplored domains. These include any domains where tile shape and size preservation upon placement is crucial, be it brick-by-brick construction of LEGO models or the creation of Tetris-like textures for instance [24]. By harnessing such domains, we extend WFC's capabilities and thereby provide access to the creative freedom and expressiveness of these domains.

Moreover, the possibility of applying WFC on multiple grids sparked our idea of generating structured objects, such as vehicles or architecture, that consist of an assembly of components. Each component may be based on a different tile set and adjacency constraints. Here, each component has a geometric relation with other components, determining their relative positions; components

[1] Minecraft, Mojang Studios, 2011. https://www.minecraft.net/en-us

[2] Caves of Qud, Freehold Games, 2015 (early access). https://www.cavesofqud.com/

[3] No Man's Sky, Hello Games, 2016.https://www.nomanssky.com/

[4] Dead Cells, Motion Twin, 2018. https://deadcells.com/

[5] Hades, Supergiant Games, 2020. https://www.supergiantgames.com/games/hades/

[6] CityEngine, Esri, 2023 (stable release). https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview

thus depend on each other. Together with the expressiveness NUTs provide, we believe that the generation of more complex objects is within reach.

However, WFC currently cannot achieve this. First, WFC cannot represent and handle NUTs. Therefore, it cannot guarantee the preservation of a NUT's shape and size. Second, even if this obstacle were overcome, no mechanism is in place for operating on NUT adjacency constraints either. Contrary to constraints between uniform tiles, NUT adjacency constraints are ambiguous; there could be multiple configurations that satisfy the constraints. Finally, while WFC can learn from a grid-based input texture, it cannot distinguish NUTs within the input and can therefore not learn how to preserve the NUTs' properties. We therefore hypothesise that the solution to these problems lies with distinguishing the individual parts of NUTs:

Hypothesis 1. Distinguishing individual parts of a NUT will enable compatibility between WFC and NUTs.

Supporting structured objects is also not straightforward. Structured objects require free-form shapes, otherwise their components would be limited to be rectangular or cuboids. Although this problem can be solved through the inclusion of empty tiles to represent void space or air [15], maintaining the component dependencies is non-trivial. Due to these dependencies, the results of one components can affect how its dependent components should be generated and positioned. Moreover, the surface of a solved component can be jagged, which is troublesome for proper component attachment. Therefore, we pose that if we can pass additional information to the components while generating the structured object, we expect to be able to assemble it. These suspicions are captured in our second hypothesis:

Hypothesis 2. A structured object can be generated and assembled through sequentially solving components and updating their relative positions and grid shape upon solving a component, while respecting their geometric relations.

To overcome these issues, we introduce Expressive Wave Function Collapse (XWFC), a major extension of WFC capable of solving and combining multiple grids with different NUT sets. Additionally, we can guarantee NUT shape and size preservation even under WFC's Overlapping Model. In our work, we show how XWFC overcomes three challenges: i) extend WFC with 3D NUTs, ii) perform a structured object decomposition into solvable components and iii) support component assembly. This generalization of WFC enables the creation of structured objects, which we illustrate in the Tetris and LEGO domains with acceptable performance.

1.1. Research questions

Our research and results will be directed to answering the following main research question, along with three other research questions based on the challenges mentioned earlier.

> **Main Research Question:** *How can Wave Function Collapse generate structured objects consisting of 3D Non-Uniform Tiles?*
>
> **Research Questions:**
>
> 1. How can WFC support NUTs while preserving their shapes and sizes?
> 2. How can WFC generate the different components of a structured object?
> 3. How can the structured object components be assembled while maintaining their geometric relations?

1.2. Methodology

The phases of this research work are threefold: i) designing and implementing WFC on NUTs, ii) designing and implementing structured object decomposition and assembly and iii) verification of our solution's capabilities.

During the first phase, we focus on the synthesis of NUTs and WFC. This involves finding a formulation of NUTs that will enable us to breach the gap between WFC's current limitations and a NUT's requirements. This will be accompanied with designing a visualization of the process for ease of iteration and empirical evaluation. Initially, a `Python` implementation will be created as a proof of concept. Once the algorithm has matured, we implement it in `C#` and migrate to Unity[7] for its expansive visualization suite. This will enable us to view how our solution operates more granularly, while also providing the means for proper communication of our findings.

During the second phase, we work on designing and implementing structured object decomposition and assembly. Part of this requires formulating components such that we can solve them with XWFC. However, this phase does not depend on the previous phase, it is extended by it. At the end of this phase, we construct a set of examples that illustrate the versatility of our solution.

The final phase is characterised by designing experiments and obtaining results. We identify the strengths and weaknesses of our approach and implementation through qualitative experimentation. This further highlights the convenience of proper visualization.

[7] Unity Engine, Unity Technologies, 2005. `https://unity.com/`

2

Related Work

WFC's release by Gumin in 2016 attracted significant attention from researchers and artists alike [7]. Gumin's WFC repository currently features dozens of variations, including adaptations, optimizations, generalizations and implementations to name a few [9, 10, 20, 19, 15]. WFC can also be found in commercial products, such as the games Bad North [27], Townscaper [**Townscaper**] and Caves of Qud [6]. While WFC rests on the same foundations as Merrell's Model Synthesis published years prior [16], WFC incited a greater impact. Nonetheless, Merrell's insights remain a source of inspiration [19].

2.1. WFC algorithm

The goal of WFC is to fill or *solve* a finite grid with tiles according to the adjacency constraints between pairs of tiles, where each cell is assigned exactly one tile. WFC thus operates on a tile set and a set of tile adjacency constraints, which specify which tiles may be adjacent in a given cardinal direction. Rather than providing these inputs directly, they can also be derived from from a grid-based example. There are thus multiple ways of specifying the input. The tiles and the constraints can be learned based on which tiles are directly adjacent in the example. With this information, a grid is solved in three phases: *Initialization*, *Observation* and *Propagation*, as shown in Algorithm 1.

During the Initialization phase, WFC's environment is set up. The tiles and constraints are learned, the adjacency matrix representing the adjacency constraints is initialized and the grid containing the remaining tile choices per cell, also called the *Wave*, is prepared. Since WFC employs an elimination process, all cells in the grid are initially unassigned and may contain any tile. This grid state is referred to as *superposition*.

In the Observation phase, a cell is selected according to some heuristic, usually *lowest entropy*. This prioritizes cells with the fewest remaining options, that is the number of tiles in this case. That cell is then assigned a tile chosen at random from its set of remaining options, after which the cell is considered *collapsed*. Due to the adjacency constraints, the collapsed cell instigates elimination of disallowed tiles from its direct neighbors. This is handled in the Propagation phase.

During the Propagation phase, the restrictions imposed by the instigating cell are propagated to its direct neighbors. All tiles that may not be adjacent to the instigating cell are removed from the neighbor's set of allowed tiles. This process is repeated for the changed neighbor, causing a wave-like effect. This continues until no further changes occur. At that point, the observation phase is entered again and this cycle is repeated until all cells are collapsed.

WFC's Simple Tiled Model and Overlapping Model

There are two modes of WFC, which both share the same core: the Simple Tiled Model and the Overlapping Model. The Simple Tiled Model works on individual tiles, with adjacency constraints expressed in terms of allowed pairs of tiles for a given direction. Its strengths lie with finding varied arrangements that need not be explicitly defined. It can derive the tile set and adjacency constraints from a grid-based example. The Overlapping Model also learns from such an example, though it scans the input by sliding a fixed-size window to obtain local patterns of tiles present in the grid. Once all patterns are obtained, the constraints are expressed in terms of overlapping patterns. This can be seen as positioning one pattern on top of each another then offsetting one pattern by one cell in the given direction. If the patterns' overlapping regions completely match, they may be adjacent. While the Overlapping Model is computationally more expensive, it can better capture relations that are difficult to express solely with tile adjacency constraints.

Algorithm 1 Wave Function Collapse

```
1: Initialize()                          ▷ Learn and store the tile set and adjacency constraints
2: while not all cells are collapsed do
3:     Observe()
4:     Propagate()
5: end while
```

2.2. WFC on multi-cellular tiles

Since we are interested in making modifications to WFC's utilization of its tile set, a closer look is taken at other authors' research in this field.

There are three main works that have encountered similar challenges concerning tiles spanning multiple cells. Stålberg's Bad North features modules that can span multiple cells in a grid [27, 30]. Stålberg acknowledges the useful properties of such large modules, such as allowing for smooth transitions. He also points out that formulating all the different possible configurations of such modules, which are many, is incredibly time consuming. To solve this, he derives the modules' adjacency constraints based on their vertices at the edges of the module. If they match, adjacency is allowed. In a similar light, Newgas takes inspiration from Stålberg. Newgas approaches tiles spanning multiple cells, which he calls *big tiles*, by splitting a big tile into single-cellular tiles and maintaining the big tile's structure with adjacency constraints between those smaller tiles [20]. Newgas' answer to the problem is effective with little to no overhead. In his work, he explains that big tile adjacency constraints are inferred with a paint-based system that revolves around coloring a big tile's edges. If the edges of two big tiles have compatible color patterns, they may be adjacent. This shares the convenient properties with Stålberg's vertex-based approach.

From a different perspective, Langendam and Bidarra's miWFC features stamp-like functionality [13]. Here, the user is able to manually create and place a group of tiles, called a *template*, in a 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 considered: a group of small tiles placed at once. The templates are also user-defined, which could be an interesting approach for creating NUTs.

Another way to possibly retain a multi-cellular tile's shape and size could be through different heuristics. Bateni *et al.* show how input resemblance can be obtained without requiring larger patterns [2]. They achieve this through the introduction of different heuristics, such as ones that take a larger context into account. This means that they can capture more context-dependent patterns without causing a significant performance overhead.

From these findings we conclude that the idea of splitting a multi-cellular tile into individual uniquely identified single-cellular tiles likely extends well to NUTs. We therefore take inspiration from this work and apply it in a different context. However, a gap remains: neither of these approaches

explicitly discusses the possibility of formulating adjacency constraints between tiles of arbitrary shape and size; adjacency based on vertex or color pattern compatibility does not guarantee the absence of overlap. A different method for formulating the adjacency constraint between NUTs is thus required. One such way could be through heuristics by which the structural integrity of, and relations between, NUTs could be maintained. Less research is done on the Overlapping Model; to the best of our knowledge, the idea of multi-cellular tiles with consistent shapes and sizes remains unexplored in a WFC or generative context.

2.3. WFC and grid compartmentalization

Another facet of our research concerns combining multiple components, each with different tile sets and adjacency constraints. Alaka and Bidarra's HSWFC offers paint-like features for selecting regions in a grid that have to adhere to a subset of the global tile set and adjacency constraints [1]. This idea of compartmentalizing certain regions has similarities with solving individual components. The selected regions could be considered to be their own grid, with their own constraints and tile sets. In HSWFC, the solved regions seamlessly integrate with the remainder of the grid. Inspired by these authors' work, Beukman *et al.* propose another hierarchical approach to using patterns [3]. These patterns may vary in size and span multiple cells. Moreover, Merrell's Model Synthesis discusses modifying in blocks [17]. In essence, he splits a grid into a series of blocks, where each block is solved in order and the results of one serve as the input for the next. Nie *et al.*'s N-WFC [21] adopts this same principle and combines it with restrictions on the tile set to prevent backtracking. However, rather than solving the blocks in order, they apply it in a diagonal manner. Solving across blocks with different boundaries and shapes and using their output as input for solving other blocks is directly relevant for answering Research Question 2, but has not been approached so far.

2.4. Structured objects and feature-based modelling

Structured object decomposition and assembly based on components solved with WFC takes inspiration from feature-based modelling, as we can formulate structured object decomposition and assembly as a simplification thereof. Feature-based modelling concerns modelling a product based on its regions of interest in a part model, called *features* [26, 34]. Entities such as walls or columns are classified as *component features* and specific shapes, such as slots and holes, can be denoted as *form features* [26, 5]. This directly correlates to our structured objects, which consist of a series of components that have to be assembled while taking their shapes into account. Feature assembly is based on a set of feature dependencies, which can be represented as a *feature dependency graph* [4]. In our case, this translates to connectivity relations, such as placing one components directly on top of another component such that their faces touch. They can be referred to as *attachments* [33]. Given an initial feature dependency graph, the dependencies must be maintained throughout the assembly. Bidarra and Bronsvoort solve this by re-evaluating those dependencies iteratively [4]. This flexible method likely extends well to the dynamic nature of our components — their final shapes and sizes are initially unknown — and repositioning the components upon changes may thus prove to be effective.

3

XWFC: Wave Function Collapse on Non-Uniform Tiles

There are plenty domains where multi-cellular tiles with shape and size preservation are required, such as the LEGO and Tetris domains. However, WFC is currently unable to achieve this. An example of this is shown in Figure 3.1, where the bricks' shapes and sizes are not preserved. In this chapter we show how XWFC can support multi-cellular tiles whose shapes and sizes must preserved, which we call Non-Uniform Tiles (NUTs), for both the Simple Tiled Model and the Overlapping Model in three dimensions. To achieve this, two challenges must be overcome: i) storing NUT information on cell level and ii) expressing the adjacency constraints between NUTs such that WFC is compatible with it.

We show how XWFC conquers the former challenge through splitting a tile into uniquely identified single-cellular tiles, which we call *atoms*, in a process called *tile atomization*. The second challenge concerns the ambiguity of adjacency constraints between NUTs, caused by a NUT's freedom of shape and size. Multiple relative NUT positions could satisfy the same adjacency constraint, as is exemplified in Figure 3.2. For the Simple Tiled Model, this problem is solved through a combination of a sliding window sliding window approach and masking. For the Overlapping Model this is solved by performing the tile atomization mechanism on the grid-based input samples WFC learns from. Both of these approaches can be achieved through the addition of a couple of pre-computation steps in the initialization phase of WFC and nigh seamlessly integrates with standard WFC. We also explain how XWFC resolves conflicts without resetting to the initialization phase.

For the remainder of this chapter, we use the following notation. For a given vector $\mathbf{v}$, the $i-th$ 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 $i-th$ element of x's extent, we write $x.\Delta_i$. Layers in a grid G are referred to as follows: the $i-th$ layer along an axis k is written as $G.L_i^k$. Matrix cells will be referenced following array notation, i.e. the element at cell x, y of a matrix M is written as $M[x, y]$. Referring to grid cells is done similarly, where the cell at coordinate $\mathbf{c}$ is accessed by writing $G[\mathbf{c}]$. 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.

3.1. NUT preservation through tile atomization

The purpose of tile atomization is to split a NUT into a set of atoms, hence its name. This allows for differentiating between the different single-cellular parts of the NUT and opens the doors to WFC compatibility. To formulate a solution for NUT compatibility, we require a formal definition of NUTs

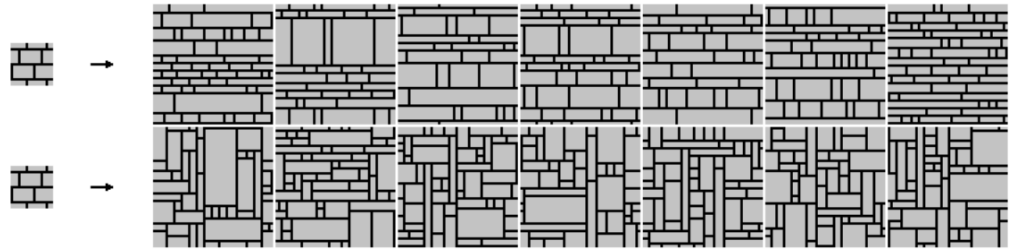


Figure 3.1: An example of WFC's behavior on brick patterns, unable to preserve the bricks' shapes and sizes. This image is obtained from Gumin's WFC repository [7].

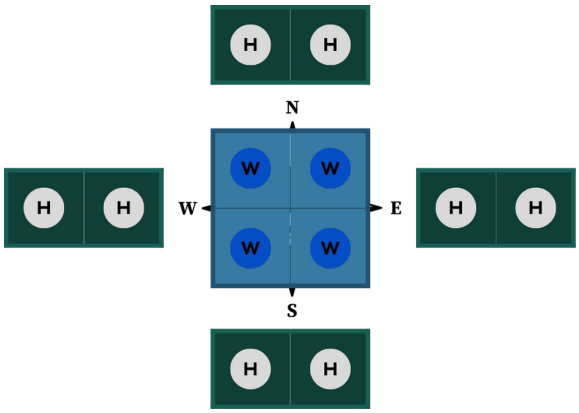


Figure 3.2: NUT adjacency constraint ambiguity shown in 2D. The green "H" tile can be adjacent to the blue "W" tile in multiple ways per direction (i.e. N(orth), E(ast), S(outh) and W(est).

and their properties, as is formally described in Definition 1.

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

In Definition 1, we state that a NUT is said to be an arrangement atoms. Moreover, these atoms' relative positions within the NUT can be mapped to absolute coordinates within a 3D grid thanks to four observations.

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

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 concretely. We denote the atom coordinate of an atom a as $a.c$. Since each cell can contain at most one atom, the combination of its 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 XWFC. For ease of reference — which will useful for the implementation discussed in Chapter 5 — we also assign a unique integer identifier to each atom. We denote the identifier of an atom a as $a.id$. We can thereby express a NUT in terms of the same domain as WFC's grid, gaining access to all of WFC's properties, including adjacency constraints. Coincidentally, the size of said grid containing a NUT's atoms equals its AABB's extent. A NUT on which the atomization process has been applied is referred to as an *atomized NUT*.

This is step one to NUT preservation. Atomization yields 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*, as more formally defined in Definition 2. The atoms forming an atom adjacency in the atom grid must occur in the same arrangement upon placing their associated NUT. Essentially, these are adjacency constraints between atoms all belonging to the same NUT, which we refer to as *inter-atom adjacency constraint*. These constraints will enforce that placing one of the atoms of a NUT requires placing all of its other atoms as well, thereby preserving shape and size. This is fully compatible with WFC, since the constraints are expressed in terms of adjacency constraints between single-cellular units.

> **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 $a_i \sim_d a_j$.

3.2. Adjacency of NUTs

With an individual NUT's properties preserved and compatibility with WFC achieved, a closer is taken at adjacency of NUTs. As mentioned, NUT adjacency is ambiguous and not directly compatible with WFC. In order solve this, we must clearly formulate what this means, starting with understanding what NUT adjacency entails. We expressed each NUT in terms of atoms and we can use these atoms to define NUT adjacency, as is formalized in Definition 3.

> **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. NUT adjacency between those NUTs is denoted as $n_i \sim_d n_j$.

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. The next step is to express these NUT adjacency constraints such that WFC is compatible with it.

> **Definition 4** (NUT adjacency constraint)**.** A *NUT adjacency constraint* is a constraint stating two NUTs along with a direction in which the NUTs may be adjacent, following Definition 3.

3.3. Convex NUTs

Besides adjacency constraints within a NUT, constraints between two NUTs are also possible, as defined in Definition 4. There may be multiple ways of positioning NUTs n_j relative to n_i such that they are adjacent, which we refer to as *configurations*, as illustrated in 3.3. In this section we explain how we disambiguate NUT adjacency constraints and achieve compatibility with WFC.

The main idea is to express a NUT adjacency constraint as a set of atom adjacency constraints. For this, two types the types of NUTs need to be supported: *convex* and *concave* NUTs. An example of

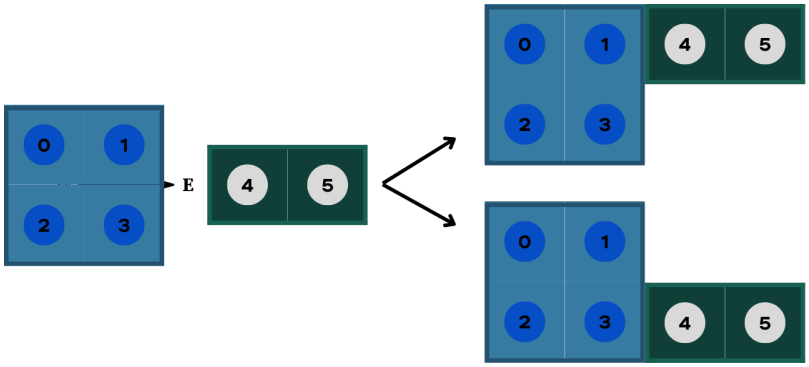


Figure 3.3: Illustration of two configurations satisfying an Eastern NUT adjacency between two atomized NUTs

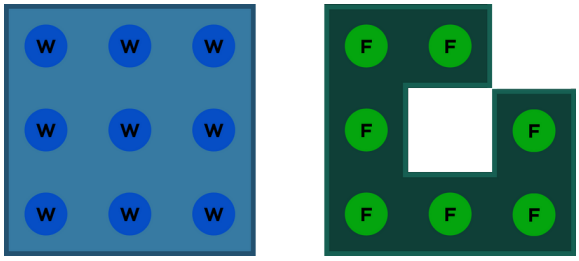


Figure 3.4: Convex (left) and concave (right) tiles

each is shown in Figure 3.4. A *convex* NUT has no holes (genus-0) and no jagged borders. NUTs with holes (genus larger than zero) or with jagged surfaces are called *concave*. Since convex NUTs are simpler by design, we first formulate a general approach that works on convex NUTs, based on the sliding window technique. After that, we refine it to support concave NUTs as well through the addition of void masks.

The approach for finding atom adjacency constraints that express a NUT adjacency constraint between convex NUTs n_i with extent $n_i.\Delta$ and n_j with extent $n_j.\Delta$ in direction $\hat{d} = \pm\hat{\mathbf{e}}_k$ rests on two observations.

1. Only atoms at the borders of a NUT can be adjacent to another NUT's atoms. Concretely, referring to the aforementioned NUT adjacency constraint, the atoms of NUT n_i with no inter-atom adjacency constraints in direction $\hat{d}$ can only be adjacent to atoms of NUT n_j with no inter-atom adjacency constraints in direction $-\hat{d}$ and vice versa. We call atoms with no inter-atom adjacency constraint in a certain direction $\hat{d}$ *d-border atoms*.
2. Due to the absence of holes and jagged borders for convex tiles, d-border atoms of a NUT n_i have the same k-atom coordinate c_k (corresponding to k used in $\hat{d} = \pm\hat{\mathbf{e}}_k$), which is either 0, or equal to n_i's maximum possible atom coordinate for dimension k, i.e. the coordinate value is 0 if $\hat{d} < \mathbf{0}$ and $n_i.\Delta_k - 1$ otherwise.

From these observations we can conclude that, given a NUT adjacency constraint between NUTs n_i and n_j and these NUTs' atoms placed in their respective grids (see Section 3.1), the atoms of interest for n_i are all in the same layer orthogonal to direction $\hat{d}$. The same applies to the atoms of n_j. We denote this layer as: $n_i.L^k_{c_k}$.

After obtaining the NUTs' layers of the convex tiles for the given adjacency constraints, all configurations can be found by linearly translating these two layers following a sliding window approach, where a window W will be slid along a base B. Consider a NUT adjacency constraint between two NUTs n_i and n_j in direction d with respective k-atom coordinate values $n_i.c_k$ and $n_j.c_k$, as obtained

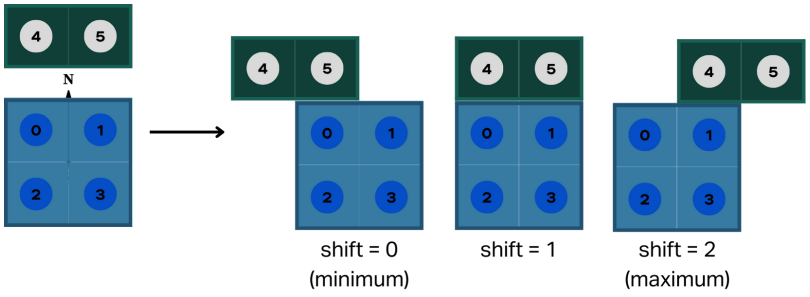


Figure 3.5: For a Northern NUT adjacency constraint between two NUTs (left), the valid configurations obtained through shifting are illustrated.

through Observation 2. Here, the window W is $W = n_j.L^k_{n_j.c_k}$ and the base B is $B = n_i.L^k_{n_i.c_k}$, with extents $W.\Delta$ and $B.\Delta$ respectively.

Now, we can consider all possible permutations of aligning the window with the base, for which an example is illustrated in Figure 3.5. We do this by linearly sliding or *shifting* the window cell-by-cell along the base from one extreme to the other. These extremes are: i) the minimum position, that is the position wherein the window's cell with the maximum coordinate is adjacent to the base's cell with minimum coordinate (see shift 0 in Figure 3.5) and ii) The maximum position, that is the opposite of the minimum position (see shift 2 in Figure 3.5). The number of shifts that need to be performed depends on the extents of the base and window. The number of shifts per dimension equals the sum of the extents of the base B and window W minus one: $shifts = B.\Delta + W.\Delta - \mathbf{1}$. Note the offset by one, as without it, corner configurations would be included, which cannot yield adjacency.

Each of these configurations thus satisfy the NUT adjacency constraint between NUTs n_i and n_j for a direction $\hat{d}$, since these satisfy Definition 3. If we find and store all atom adjacency constraints that are present in these configurations, we obtain the set of atom adjacency constraints that together represent NUT adjacency constraint unambiguously. These are formatted such that an atom adjacency constraint $(n_i, n_j, \hat{d})$ reads as n_j may be adjacent to n_i in direction $\hat{d}$. The atom adjacency constraints in aforementioned example are listed in order: $(0, 5, N)$,$(0, 4, N)$,$(1, 5, N)$ and $(1, 4, N)$. For all pairs of adjacent atoms that satisfy NUT adjacency, an atom adjacency constraint is stored in an adjacency matrix M for the corresponding directions, i.e. $M_{\hat{d}}[p, q] = true \wedge M_{-\hat{d}}[q, p] = true$. Note that there are two assignments, since one implies the other: $a_p \sim_{\hat{d}} a_q \Leftrightarrow a_q \sim_{-\hat{d}} a_p$. With this, the basic procedure of finding the atom adjacency constraints between two convex tiles is complete. In order to achieve the same for concave tiles, we utilize void masks, as explained next.

3.4. Concave NUTs and void masking

Concave NUTs are more challenging than convex ones due to the presence of jagged borders or holes. The sliding window approach cannot capture such deviations. Figure 3.6 shows how the current approach would fail to find one of the valid configurations. Obtaining that configuration requires translating the window against the constraint's direction. If we were able to find the void regions present in the Non-Uniform Tile (NUT)s, this translation could be found. For this reasons, XWFC employs *void masks* as defined in Definition 5.

Definition 5 (Void mask)**.** A *void mask* is a mask of a grid that contains the number of empty cells, referred to as *voids*, of a grid G for a given direction $\hat{d} = \pm\hat{\mathbf{e}}_k$. A void mask is formalized as $V(G, d)$ where G is the grid and $\hat{d}$ is the direction for which the voids are counted.

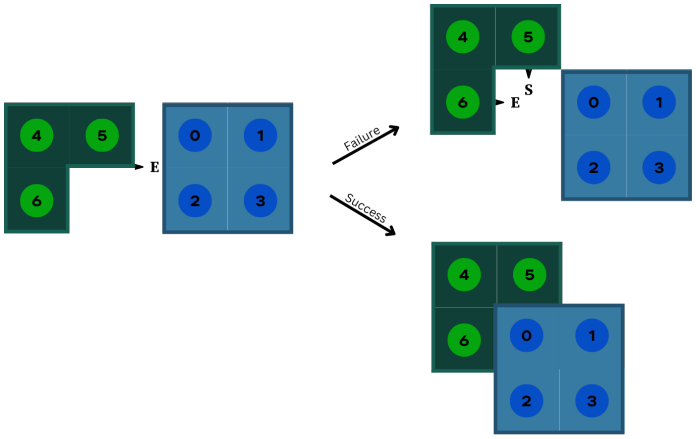


Figure 3.6: For an Eastern NUT adjacency constraint between a concave (green corner; base) and a convex NUT (blue square; window), The position of the window tile (see top right), obtained as described in Section 3.3, misses the Southern and Eastern atom adjacency constraints. The expected configuration (see bottom right) does not miss these.

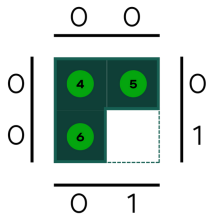


Figure 3.7: Example of void mask calculation for a concave NUT. The black lines and numbers denote the void masks, where the numbers denote the number of voids counted.

The void mask's extent equals that of the projection of the original grid G onto the plane orthogonal to $\hat{d}$. A resulting set of void masks is shown in Figure 3.7. To obtain a void mask for a direction $\hat{d} = \pm\hat{\mathbf{e}}_k$, we start counting from the outer layers of a grid G relative to that direction, that is (in order) layers $L^k_0, \ldots, L^k_{G.\Delta_k-1}$ if $\hat{d} < \mathbf{0}$ and layers $L^k_{G.\Delta_k-1}, \ldots, L^k_0$ otherwise. The void masks can be pre-computed are passed along with the sliding process described in Section 3.3.

Since a NUT adjacency constraint requires the atoms of two different NUTs to be adjacent (see Definition 3), every valid configuration should result in some overlap when the tiles are projected onto the dimension corresponding to $\hat{d} = \pm\hat{\mathbf{e}}_k$. The same principle of overlap can also be applied to void masks, as illustrated in Figure 3.8. Consider a NUT adjacency constraint between NUTs n_i and n_j in direction $\hat{d}$. The void mask of NUT n_i, $V(n_i, \hat{d})$, and the void mask of NUT n_j, $V(n_j, -\hat{d})$, are overlaid corresponding to the overlap found in the projection just described. As the overlapping sections of the void masks depend on the shift applied to the window n_j, the overlaid void masks are a subset of the actual void masks, referred to as $V(n_i, \hat{d})'$ and $V(n_j, -\hat{d})'$ respectively. The overlapping mask elements are then summed element wise and the minimum sum dictates the offset required for the two NUTs to be adjacent without their atoms overlapping.

A consequence of offsetting in direction $-\hat{d}$ is that the atoms can be adjacent in directions other than $\hat{d}$. Therefore, for each d-border atom (see Observation 1), the atom pair checking process should be done in every direction. Figure 3.8 shows an example of overlaying two void masks. In that figure, note how in the bottom configuration on the right, the Southern and Eastern atom adjacency constraints are detected, as a result of translating the window Westward.

It is also important to mention that the minimum sum of the overlaid void masks may not be larger than or equal to the k-extent of the base B, that is $B.\Delta_k$. This would result in an invalid configuration,

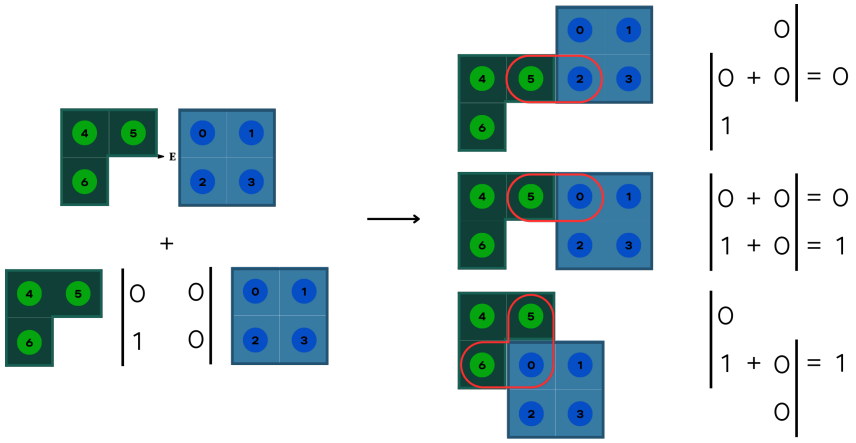


Figure 3.8: An example of overlaying the void masks of two NUTs. On the left, an Eastern NUT adjacency constraint between a concave (green corner) and a convex (blue square) NUT and the NUTs' void masks are shown. On the right, the different valid configurations along the with the overlaid void mask subsets. The red selections on the right indicate the detected atom adjacency constraints.

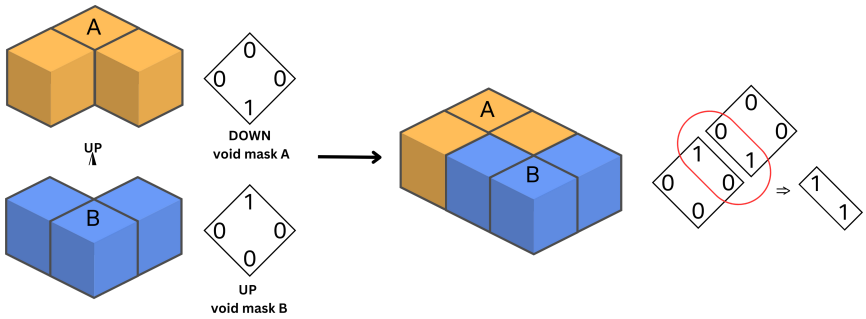


Figure 3.9: An Up NUT adjacency constraint between two NUTs. Despite the configuration leading to adjacent atoms, none of these atoms are adjacent in the up-direction, thus the configuration is invalid.

even though atoms could be adjacent, but none of them are adjacent in direction $\hat{d}$. More formally, consider an NUT adjacency constraint between NUTs n_i and n_j in direction $\hat{d} = \pm\hat{\mathbf{e}}_k$. Let $V(n_i, \hat{d})'$ and $V(n_j, -\hat{d})'$ be their respective overlaid void mask sections. Then, it must be that $min(V(n_i, \hat{d})' + V(n_j, -\hat{d})') < n_i.\Delta_k$ is true in order for the configuration to be valid. This is very important for operating in three dimensions, of which an example is shown in Figure 3.9.

3.5. Grid atomization: learning from 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 obtain the tile set and adjacency constraints.

A NUT's atoms can look identical in an input, yet each atom must be treated as unique. This information is not captured in the input, thus WFC cannot comply with NUT's requirements and therefore cannot inherently support NUTs. Input learning would succeed if the different cells composing a NUT were uniquely identified. This is where the idea of *grid atomization* comes into play, based on the mechanism behind tile atomization (see Section 3.1).

The constraints between NUTs can be learned from the input if the NUT set, that is a tile set consisting of NUTs, were passed along as well. We argue that this is a reasonable requirement, because the input is generated based on a NUT set already, thus it is already known. Passing this

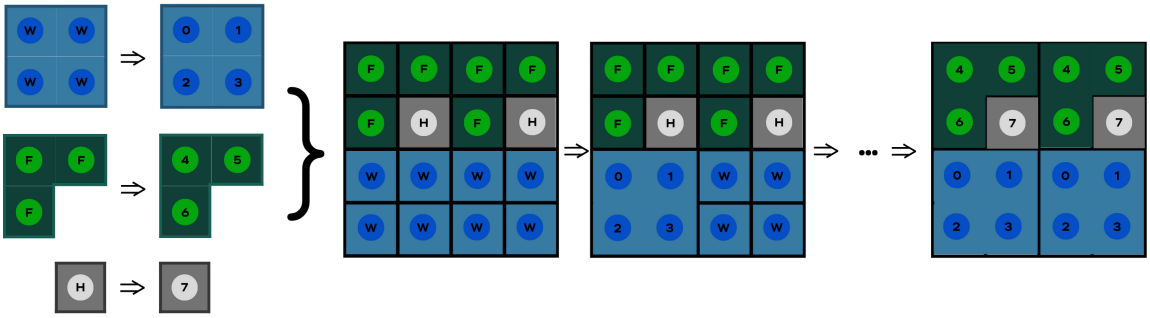


Figure 3.10: Grid atomization. Given a set of atomized NUTs (left), overlay each NUT to find a fitting tile and assign atom identifiers to the associated cells in the grid.

set can be as simple as providing a series of small input grids (similar to how NUTs are treated in tile atomization).

In order for XWFC to learn from input, we first perform tile atomization on all NUTs in the NUT set. After that, the input grid is atomized. We can therefore create a unique input-identifier mapping stored as a grid, where a cell's input value maps to the atom identifier with an atom coordinate equal to the cell's coordinate. We call this process *grid atomization*. Concretely, grid atomization follows a sliding window approach, of which a high-level illustration is shown in Figure 3.10. We iterate over all cells in the input grid incrementally, starting at its zero-index. Then, 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 $\mathbf{c} + a_j.\mathbf{c}$. In other words, let $G.\Delta$ represent grid G's extent and $a.id$ be the id of atom a, then $\forall a_j \in n_i.A \Rightarrow a_j.id = G[\mathbf{c} + a_j.\mathbf{c}] \wedge \mathbf{0} \leq \mathbf{c} + a_j.\mathbf{c} < G.\Delta$. By starting from the $\mathbf{0}$ coordinate in G and selecting cells incrementally thereafter, we align the NUTs with this coordinate. It allows us to map the atom coordinates to those of G.

Performing the NUT fitting in the G is but one step, we have to assign the atoms as well. If a NUT fits, we can store the identifier $a_i.id$ and atom coordinate $a_i.c$ of an atom a_i at $\mathbf{c} + a_i.\mathbf{c}$ in a grid G', which has the same extent as G. G' is then the atomized version of G. Once the input has been atomized, the adjacency constraints between atoms can be derived, both for WFC's Simple Tiled Model as well as the Overlapping Model, as explained next.

3.6. XWFC Simple Tiled Model synthesis

Similar to WFC, XWFC can infer tile adjacency constraints from an input. After obtaining the atomized grid as described in Section 3.5, the atom adjacency constraints can be derived following the standard procedure used in WFC. For each cell in the grid, the atom assigned to the cell may be adjacent to the atoms assigned to the neighboring cells in the cardinal directions.

We have thereby formulated a solution to the NUT problem through a couple of added steps in the initialization phase of WFC. After initialization, the main loop WFC can be entered, but on atom level; the observation, collapse and propagation phases are similar to standard WFC. During the observation phase, the cell with lowest entropy is selected and collapsed. It is randomly assigned an atom from the allowed set of atoms for said cell. Next, collapsing a cell imposes constraints on the neighbors, according to the atom adjacency constraints derived earlier. These are propagated, after which the observation phase is entered again. This cycle is repeated until all cells are

collapsed, similar to WFC's algorithm shown in Algorithm 1.

Here `Initialize()` depends on the mode Simple Tiled Model is run with, i.e. learn from NUT adjacency constraints (see Algorithm 2) or from an input (see Algorithm 3).

Algorithm 2 Initialization on NUT adjacency constraints

$N = \{n_0, \ldots, n_n\}$ ▷ NUT set, where each NUT n_i has extent $n_i.\Delta$
$A = \{adj_0, \ldots, adj_m\}$ ▷ Set of adjacency constraints
$D = \{d_0, \ldots, d_k\}$ ▷ Set of direction unit vectors
G_Δ ▷ Output grid extent
InitializeAdjacencyMatrix(N, D)
IntializeWaveAndAtomGrid(G_Δ)
for n_i in N **do**
    Atomize(n_i)
    for d in D **do**
        CalculateAndStoreVoidMask(n_i, d) ▷ Void mask of n_i for d is accessed through $V(n_i, d)$
    end for
end for
for each adj_i in A **do**
    $n_0, n_1, d = adj_i$ ▷ n_0 is the base, n_1 the window
    shifts = CalcShifts(n_0, n_1, d)
    SetWindowToMinimumPosition() ▷ See Section 3.3
    for s where $s \leq$ shifts **do**
        $p =$ FindRelativePosition(n_0, n_1, s) ▷ Position of n_0's origin relative to n_1's origin
        min = OverlaidVoidMasksMinSum($V(n_0, d)$, $V(n_1, -d)$, s)
        if min $\geq n_0.\Delta$ **then**
            continue
        end if
        Translate(p, min, $-d$) ▷ Offset p against d to reach configuration
        for valent atom in n_0 **do**
            FindAndStoreAtomAdjacencyConstraints()
        end for
    end for
end for

3.6.1. XWFC Overlapping Model patterns

The atomized grid can be used for the Overlapping Model as well. A kernel w with extent $w.\Delta$ is used to obtain the patterns from the input, after which the adjacency constraints between those patterns are derived. Concretely, the kernel is a window that slides over the entire input grid cell-by-cell. Each position of the kernel captures a small section of the input grid, this section is called a *pattern*. The patterns are stored in set P, where $P = \{p_0, \ldots, p_n\}$. Since the patterns' compatibility depend on overlapping them, the minimal size of the kernel is $\mathbf{2}$.

Algorithm 4 shows the procedure for finding pattern adjacency constraints. Pattern adjacency constraints are obtained by overlapping two region for a given direction $\hat{d} = \pm\hat{\mathbf{e}}_k$. Let p_i and p_j be two patterns obtained by sliding a kernel w with extent $w.\Delta$ over an input grid as described earlier. Each of these patterns thus have a set of values stored in grid-like fashion and thus have a coordinate within the pattern ranging from $\mathbf{0}$ to $w.\Delta - \mathbf{1}$, which we call the *pattern coordinate*. We write a patterns set of pattern coordinates as $p.C$. Then, a pattern adjacency constraint is derived as follows:

Algorithm 3 Initialization on Input

```
N = {n_0, ..., n_n}                          ▷ Where each atom has an input representation
G                                            ▷ Input grid
G' = AtomizeGrid(G, N)                       ▷ See Section 3.5
D = {d_0, ..., d_k}                          ▷ Set of directions
Δ                                            ▷ Output grid extent
InitializeAdjacencyMatrix(N, D)
IntializeWaveAndAtomGrid(Δ)
for each cell coordinate c' in G' do
    for each d_i in D where d_i > 0 do
        StoreAtomAdjacency(G_c', G_c'+d_i, d_i)
        StoreAtomAdjacency(G_c'+d_i, G_c', −d_i)
    end for
end for
```

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

One of the advantages of XWFC's use of NUTs is that a small kernel of size $\mathbf{2}$ can represent much more information than with standard WFC. This is thanks to the properties of NUTs. The patterns obtained through input processing contain atoms rather than tiles. Since each atom belongs to a NUT, placing a pattern, and thereby the atoms in said pattern, will enforce that all other atoms (possibly outside of the kernel) must be collapsed as well.

Algorithm 4 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
```

Additionally, patterns only containing atoms of the same tile become redundant, since those atom relations are already enforced through inter-atom adjacency constraints. The larger the NUTs are, the fewer patterns are required to represent them relative to their size. While it sounds counter-intuitive, it makes sense. The purpose of a pattern is to contain information on how two or more tiles (or NUTs) may be present in a given context. For large tiles, only patterns at the perimeter of the tiles contain information of how two NUTs may be adjacent, thereby obsoleting patterns containing only atoms belonging to the same NUT.

Consequently, the procedure for the Overlapping Model is adjusted to facilitate these changes. The

observation and collapse phases are still the same. The propagation phase is adjusted to exploit NUT properties.

During observation, the cell with the fewest remaining patterns is selected and then collapsed. However, since the domains of Simple Tiled Model and Overlapping Model are unionized and thus operate on atoms, collapsing works slightly differently. During run-time, we must know which cells are collapsed to which atom *within* the pattern. To achieve this, we say that the cell should be assigned the atom identifier of the $\mathbf{0}$ cell of the pattern. While other coordinates could work, choosing the zero coordinate prevents the need for additional calculations. By keeping this pattern atom coordinate consistent, pattern adjacency constraints are still satisfied, while also operating in the atom domain. Henceforth, this pattern atom coordinate is assumed to be $\mathbf{0}$.

Upon collapsing a cell, a second collapse wave follows, where all other atoms belonging to the same NUT as the chosen atom are collapsed. This is done in the same fashion as with Simple Tiled Model. A new problem arises with this step: immediately collapsing into atoms, rather than patterns, results in loss of pattern data. Deciding which pattern fits within a given context is possible; it would be a matter of checking the surroundings of the collapsed cell and fitting patterns accordingly (similar to how pattern adjacency constraints were derived). However, unlike during constraint inference of the input, the states of the cells are not necessarily final while solving a XWFC problem; they may still be in superposition for instance. This means that finding all patterns that fit given a context is relative expensive and inefficient, especially when considering that this operation would have to be performed frequently. An important observation here is that the set of patterns that may be placed adjacent to a NUT's atoms is deterministic and is not tied to a specific context. So, they can be pre-computed for each tile. We can thus make use of memorization to store this information, which is encapsulated in a Tile Pattern Adjacency Mask.

3.6.2. Tile Pattern Adjacency Mask

This inference step is what enables the exploitation of the properties of NUTs in the Overlapping Model. During the pre-processing phase, all NUTs are atomized and all patterns are defined. With this information, a Tile Pattern Adjacency Mask (TPAM) can be derived. A TPAM is a mask that indicates which patterns may be adjacent to the atoms of the NUT. By encasing the NUT in a miniature XWFC Overlapping Model problem, we can reach a state where we pre-compute the allowed patterns of uncollapsed neighbors of all NUT's atoms. The XWFC problem requires a grid, a tile set (a singular NUT in this case) and a set of patterns (obtained in the initialization phase of XWFC). The grid is initially in superposition and its extent is set to the extent of the NUT in question plus $\mathbf{2}$. This ensures that the NUT's atoms can be placed such that there are neighboring cells for all of its atoms. Note that this grid extension suffices, regardless of kernel size. If the kernel size is larger than the grid, any cells in the pattern that would be out of bounds can be ignored. Those would not impose any further restriction, as the out-of-bounds neighbors are in superposition. For this approach, checking the direct neighbors of each atom suffices.

Concretely, given a NUT n_i with extent $n_i.\Delta$, we create an augmented grid G' in superposition with extent $G'.\Delta = +\mathbf{2})$. The NUT's atoms are placed in the center of the grid, where an atom a at atom coordinate $a.\mathbf{c}$ corresponds to G''s coordinate $a.\mathbf{c} + \mathbf{1})$. After that, for each atom's uncollapsed neighbors (i.e. in directions where the atom has no inter-atom adjacency constraints), perform the pattern fitting procedure with neighbors in superposition. This procedure slightly deviates from the standard pattern compatibility described in Algorithm 4.

Consider a cell at coordinate $\mathbf{c}$ in grid G' that is occupied by an atom a of a NUT n_i, let P be the set of patterns obtained from the input, let the pattern atom coordinate $pac = \mathbf{0}$ and let M be the pattern adjacency matrix, containing a 2D matrix for each direction specifying which patterns may be adjacent. The matrix index of a pattern p is written as $index(p)$. Then, for each direction $\hat{d} > \mathbf{0}$, the set of patterns that are allowed for the neighboring cell at coordinate $\mathbf{c} + \hat{d}$ depends on the set of patterns P' containing the patterns where a is at the pac. For each pattern p'_i in P', the allowed

pattern indices correspond to the indices of the $true$ elements in a row of adjacency matrix M, $M[\hat{d}][index(p'_i)]$, and are stored in P'_i. We then obtain this set for each pattern p'_i in P' and take the union of the resulting sets. Those denote the indices of the patterns allowed for the neighboring cell. If $\hat{d} < \mathbf{0}$, the set of patterns allowed for the neighbor at $\mathbf{c} + \hat{d}$ is simply the set of patterns where a is at pattern coordinate $pac - \hat{d}$.

Note that this includes patterns that should not be allowed, as the process does not take a cell's full context into account. The neighbors are thus said to be *under-constrained*. However, this is no problem: propagation will eliminate any disallowed patterns by propagating the patterns adjacency constraints determined in an earlier step in the initialization. An additional consideration is that starting with all neighbors in superposition also contributes to under-constraining in a larger context. However, propagation will resolve this issue while solving the main XWFC problem for the same reason. The main advantage of this approach is that it requires fewer duplicate computations during run-time. The neighbor's resulting states obtained after propagation can be stored in a mask, which can be overlaid upon collapsing a pattern.

3.7. NUT and pattern elimination at a grid's edges

While the inter-atom adjacency constraints ensure that a NUT's atoms are properly laid out on the grid, another mechanism is required to enforce this at the edges of the grid. Upon collapsing a cell into an atom, other cells within the grid corresponding to the other atoms are collapsed as well. However, it could be the case that one of the atoms would be outside of the grid, which would violate a NUT's properties. This can only be the case at the edges of the grid in this context. Therefore, during the initialization phase of XWFC, a NUT (for the Simple Tiled Model) and pattern (for the Overlapping Model) elimination step is performed. The NUTs or patterns that should be eliminated are those that are not allowed. Thus, if we define what it means for a NUT or pattern to be allowed, we can determine which are not. Definition 6 formulates this for a given position in a grid for one of a NUT's atoms. It is required to be specific here, due to the different ways a Non-Uniform Tile (NUT) can be laid on a grid. With this definition, elimination can be explained for the Simple Tiled Model and Overlapping Model.

Definition 6 (Allowed NUT)**.** A NUT n with extent $n.\Delta$ and atoms $n.A$ is said to be allowed for an atom $n.a_i \in n.A$ with atom coordinate $a_i.\mathbf{c}$ on a grid G with extent $G.\Delta$ at grid coordinate $\mathbf{c}$ if the following criteria are met:

1. (NUT within bounds) $\mathbf{c} - a_i.\mathbf{c} \geq \mathbf{0} \wedge \mathbf{c} - a_i.\mathbf{c} + n.\Delta \leq G.\Delta$
2. (free) If requirement 1 holds, assert that for each n's atom coordinates $a_j.\mathbf{c} \in n.A$ grid cell $G[\mathbf{c} + a_j.\mathbf{c}]$ may be collapsed into atom a_j.

For the Simple Tiled Model eliminating NUTs that are not allowed from the waves of the cells located at the edges of the grid wave suffices. Due to the inter-atom adjacency constraints, propagation will ensure that atoms belonging to disallowed NUTs are eliminated.

For the Overlapping Model, elimination is more involved, due to the discrepancy between patterns and atoms. The waves operate on patterns, but NUT elimination is based on atoms. Since patterns consist of a group of atoms, elimination is still possible. Similar to the elimination for the Simple Tiled Model, only the cells at the edges of the grid need to be considered. For the Overlapping Model, pattern elimination has six steps:

1. Given a cell at coordinate $\mathbf{c}$, consider it to correspond to the pattern atom coordinate relative to the pattern p, which has extent $p.\Delta$.
2. For each pattern cell coordinate $p.\mathbf{c}$ where $\mathbf{0} \leq p.\mathbf{c} < p.\Delta$, assert that the NUT associated with the atom a in the pattern at $p.\mathbf{c}$ is allowed for atom a and cell $\mathbf{c} + p.\mathbf{c}$ in the grid.
3. Disallow the pattern if at least on of its atoms results in a disallowed NUT in the step prior and exclude the disallowed pattern from the cell's wave.

4. Edge case: consider an atom resulting in a disallowed NUT and the grid positions of all other atoms associated to the same NUT. If all of the cells at those positions were already collapsed, the NUT would not be placed at all. The properties of a NUT that will no be placed cannot be violated. This means that the pattern should thus still be allowed.
5. If a cell's wave was changed, add it to the propagation queue.
6. Perform propagation once all cells at the grid's edges have been processed.

Once completed, XWFC initialization is done. Before synthesizing the different facets of our solution, we also propose a way to handle conflicts that could arise due to contradicting constraints.

3.8. Conflict handling

Unlike WFC, XWFC has conflict handling. XWFC periodically creates save points that consist of the wave and, only for the Overlapping Model, a grid containing the collapsed atoms. Note that both are required for the Overlapping Model, as is relies on both patterns and atoms. The Simple Tiled Model only relies on the wave containing atoms. When a conflict is encountered, i.e. when a cell runs out of options, XWFC reverts to a previous save point and tries again. While there are more sophisticated ways of conflict handling, such as backtracking, the trade-off between quality and efficiency should be considered. The use of save points favors efficiency and has a lower time complexity, but cannot guarantee that a solution is always found should one exist.

Algorithm 5 XWFC Overlapping Model

```
1: Initialize()
2: while not all cells are collapsed do
3:     Observe()
4:     Collapse()
5:     CollapseNUTsInPattern()
6:     ApplyNUTMasks()
7:     Propagate()
8: end while
```

3.9. XWFC Overlapping Model synthesis

Once the TPAMs are obtained, the procedure for the Overlapping Model is as shown in Algorithm 5. Observe the grid and choose an uncollapsed cell with lowest entropy, i.e. the cell with lowest number of allowed patterns. Collapse the cell into a pattern. Assign the atom identifier at the pattern atom coordinate of the pattern to the cell in a grid containing the atom values. Collapse the cells corresponding to the other atoms belonging to the same NUT into their respective atom identifiers. Reference the Tile Pattern Adjacency Mask and update the placed tiles' uncollapsed neighbors accordingly. Propagate and go back to the observation phase until all cells are collapsed. With this, we have addressed the first challenge of our research: extending WFC with NUTs. We have shown that NUTs have useful properties that can be exploited, and that much can be achieved through several pre-computations. XWFC is truly a superset of WFC.

4

Structured objects and XWFC

This chapter highlights the second phase of this thesis and discusses how we can generate structured objects, consisting of an assembly of multiple components.

4.1. Structured object representation and decomposition

In order to formulate the representation and decomposition of a structured object, we need define what a structured object is exactly, which is done in Definition 7.

> **Definition 7** (Structured object)**.** A structured object is an object that consists of an assembly of multiple components, each component possibly consisting of a different tile set and constraints.

Generating structured objects (see Definition 7) with WFC requires formulating the problem such that WFC is compatible with it (see RQ 2). Structured objects are generally not supported by WFC, since WFC only concerns itself with objects consisting of one grid only. However, if it can be applied on one grid, it surely can be applied to multiple. If those resulting solutions could be combined, we can represent structured objects. To achieve this, we start with structured object decomposition to obtain the set of components the structured object consists of. Each components has a size, or extent, and a position in space and therefore an origin. They also have their own NUT set and constraints. We can thus *solve* a component with XWFC, meaning that we run XWFC on its grid with its NUT set and constraints.

Per Definition 7, a structured object is an *assembly* of components. This means that the components have dependencies that together define the structure of the structured object. Thus, if such an object were to be decomposed into individual components, these dependencies need to be maintained during assembly in order to retain the object's structure. Since these components represent one object, the components' dependencies can be expressed in terms of *attachments* [33], as formalized in Definition 8.

Now that a structured object can be expressed in terms of components and attachments, we can formulate structured object decomposition. Structured object decomposition is a method of splitting a structured object into a set of attached components (see Definition 8) $C = \{C_0, \ldots, C_n\}$. Each component has a set of properties: a NUT set, an origin, an extent and an input from which the constraints are learned. The input depends on XWFC model: it can be a set of NUT adjacency constraints (for Simple Tiled Model only) or a set of grid-based input examples. Additionally, each component can be represented as an AABB, with the component's origin its minimum value and origin plus extent its maximum value. The main reasons for choosing this representation is

that i) it allows for spatial positioning and ii) its cuboid shape directly corresponds to a grid with an extent equal to the component's extent, which nicely integrates with XWFC. To capture attachments between components we formulate an *attachment tree* as described in Definition 9. This decomposition into a set of components along with the attachment tree captures the structure of the structured object.

Definition 8 (Component attachment)**.** Let C_i and C_j be components with respective origins $C_i.\mathbf{O}$ and $C_j.\mathbf{O}$ and extents $C_i.\Delta$ and $C_j.\Delta$. C_i and C_j are ***attached*** if two properties are satisfied:

1. (component intersection) Two components intersect if either of these cases hold:
   (a) $C_i.\mathbf{O} + C_i.\Delta \geq C_j.\mathbf{O} \wedge C_i.\mathbf{O} \leq C_j.\mathbf{O}$
   (b) $C_j.\mathbf{O} + C_j.\Delta \geq C_i.\mathbf{O} \wedge C_j.\mathbf{O} \leq C_i.\mathbf{O}$
2. (no corner adjacency) Two components are said not to be corner adjacent if the minimum coordinate (i.e. origin) of one does not match the maximum coordinate of the other (i.e. origin plus extent). Formally, consider two vectors $\mathbf{u}$ and $\mathbf{v}$ and let n be the number of dimensions, then $NoCornerAdjacency = \sum_{k=1}^{n} \delta(\mathbf{u}_k, \mathbf{v}_k)$, where $\delta(\mathbf{u}_k, \mathbf{v}_k) = \begin{cases} 1 & \text{if } \mathbf{u}_k = \mathbf{v}_k \\ 0 & \text{if } \mathbf{u}_k \neq \mathbf{v}_k \end{cases}$
   Compute $noCornerAdjacency$ for:
   (a) $\mathbf{u} = C_i.\mathbf{O} + C_i.\Delta$ and $\mathbf{v} = C_j.\mathbf{O}$
   (b) $\mathbf{u} = C_j.\mathbf{O} + C_j.\Delta$ and $v = C_i.\mathbf{O}$
   If $noCornerAdjacency \leq 1$ for both cases, then C_i and C_j are not corner adjacent

.

Definition 9 (Attachment tree)**.** An attachment tree is a tree that represents the attachments between components. To construct an attachment tree $T_A = \{V, E\}$, where V is the set of vertices and E the set of edges, let C be the set of components. Let $v_i \in V$ map to a component $C_i \in C$. Then, for each attachment between a component C_i and C_j, there must be an $e_{ij} \in E$.

Before moving on, we need to point out that the shape of a solved component should not be limited to cuboids, even though the AABB that represents it is cuboid. We must address this, as a structured object can comprise non-cuboid components. For this reason, the component's NUT set may include the empty tile. Due to the possible inclusion of the empty tile in a component C_i's NUT set, a solved component C_i' is not necessarily completely filled with solid tiles. Therefore the shape of C_i' may differ from C_i, but C_i' can never exceed C_i's grid's extent. C_i's extent is thus the maximum bound. With this in mind, structured object assembly can be defined.

4.2. Structured object assembly

Structured object assemble is the process of solving each component and positioning them such that the structured object's structure, i.e. the set of attachments, is preserved. Solving a component is done by employing XWFC based on a component's NUT set and constraints, where its grid representation's extent corresponds to the grid used in XWFC. Structure preservation is achieved with the aid of the attachment tree (see Section 4.1).

If a solved component's shape and size differ from its unsolved counterpart, attachments could be violated and components that depend on the solved component, called *dependent components*, thus need to be repositioned. The attachment tree is used to find these dependent components. Each of those components is translated such that the attachments are preserved again. Since the components depend on each other and can thus not be solved all at once, we take an iterative approach and therefore need to define a component solving order. In order to assemble a structured object we therefore need to be able to i) define the solving order, ii) find the set of dependent

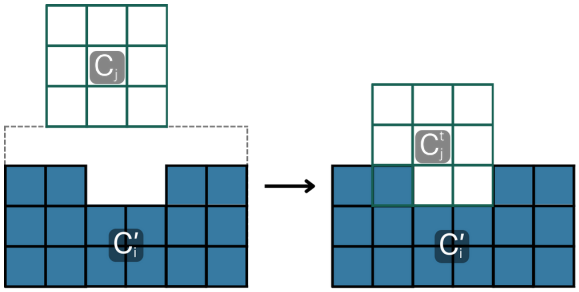


Figure 4.1: Positioning of C_j relative to C'_i to obtain C^t_j for assembly

components and iii) formulate what translations need to be applied.

The solving order is derived from the attachment tree. The attachment tree T_A is traversed in Depth-First Search (DFS) order to obtain component sequence $S = [C_i, \ldots, C_j]$, containing all components, i.e. $\forall i(0 \leq i \leq n) \Rightarrow C_i \in S \rightleftharpoons C_i \in C$. Note that the root of T_A can be any component, what matters is that there is a clear and acyclic order of dependence.

The dependent components can be derived from the attachment tree as well. Let $T_A = \{V, E\}$ be the attachment tree with a set of vertices and edges constructed as described in Definition 9. Let V' be the set of vertices connected to vertex v_i corresponding to C_i. Each $v_j \in V'$ maps to a component C_j affected by C_i.

The translation needed to attain attachment depends on the result of the solved component, as illustrated in Figure 4.1. In this figure, the top layer of C'_i's grid's is empty and its unsolved dependent component C_j has to rest on top of C'_i. Then, C_j must be translated such that its bottom face follows the contours of C'_i top side in order for the two to be attached. By translating C_j we obtain C^t_j. Jagged surfaces further complicate this endeavor; the solved components are not required to yield flat surfaces, due to empty cells. The translation can determined based on the direction two components are attached and the difference of their relative positions, as explained in the next sections.

4.3. Attachment direction of components

Given a solved component C'_i and an unsolved component C_j, the attachment direction corresponds to the dimension in which C'_i and C_j are attached (see Definition 8, Property 2). We do not consider corner adjacency in the construction of the structured object decomposition, since corners do not result attachment. Do note that the two component's AABBs may intersect, this could be the case when a component is translated due to assembly. In order to limit the induced complexity of intersecting components, we restrict the translation dimension to one. All that remains is to infer the sign of the direction. The direction is derived from the type of connection of two components (see Definition 8, Property 1).

The connecting direction $\hat{d}$ of two attached components is derived from Definition 8. For the attachment direction, let direction $\hat{d} = s\hat{e}_k$, where the dimension k is the k for which $\delta(\mathbf{u}_k, \mathbf{v}_k) = 1$ in Property 2 of Definition 8. If there is no such k, then the components completely intersect and a default dimension k can be chosen.

The sign can then be computed: sign $s = \begin{cases} -1 & \text{if } C_i.\mathbf{O} + C_i.\Delta \geq C_j.\mathbf{O} \wedge C_i.\mathbf{O} \leq C_j.\mathbf{O} \\ 1 & \text{if } C_j.\mathbf{O} + C_j.\Delta \geq C_i.\mathbf{O} \wedge C_j.\mathbf{O} \leq C_i.\mathbf{O} \end{cases}$

The region of attachment of two attached components is also required, which specifies the region of the touching, or intersecting, AABBs of two components and is defined in Definition 10. With this information, we can express the relation between two components' AABBs with R and $\hat{d}$.

Definition 10 (Region of attachment)**.** The region of attachment R, with origin $R.\mathbf{O}$ and extent $R.\Delta$, for two connected components C_i and C_j with origins $C_i.\mathbf{O}$, $C_j.\mathbf{O}$ and extents $C_i.\Delta$, $C_j.\Delta$ respectively is defined as follows. Let n be the number of dimensions, then:

1. $R.\mathbf{O} : \forall k(0 \leq k < n) \Rightarrow R.\mathbf{O}_k = max(C_i.\mathbf{O}_k, C_j.\mathbf{O}_k)$
2. $R.\Delta : \forall k(0 \leq k < n) \Rightarrow R.\Delta_k = min(C_i.\Delta_k, C_j.\Delta_k)$

4.4. Component translation

To assemble two components, they must be attached. A component C_j must thus follow the contours of a solved component C'_i. A translation $\mathbf{t}$ to C_j in $-\hat{d}$ is applied to achieve this yielding a translated unsolved component C^t_j. Translation $\mathbf{t}$ is calculated through the use of the C'_i's void mask $V(C'_i, \hat{d})$ for direction $\hat{d}$ (see Definition 5) and the region of attachment R of C'_i and C_j (see Definition 10). The coordinates of R are essentially a subset of the coordinates of $V(C'_i, \hat{d})$; they are both derived based on the same direction and the same component. Since the region of attachment is formulated as $R = \{R.\mathbf{O}, R.\Delta\}$, the index of R's origin relative to C'_i's origin $C_i.\mathbf{O}$ is $R.\mathbf{O}' = R.\mathbf{O} - C_i.\mathbf{O}'$. $R.\mathbf{O}'$ can then be used to derive the void mask associated with R: $V(R, d)$, where $V(R, \hat{d}) \subseteq V(C'_i, d)$. Let $V.\mathbf{O} = \mathbf{0}$ be the origin and $V.\Delta$ be the extent of $V(C'_i, \hat{d})$, then its origin and the extent are $V.\mathbf{O}' = (R.\mathbf{O}_p, R.\mathbf{O}_q)$ and $V.\Delta' = min(V.\Delta, (R.\Delta_p, R.\Delta_q))$ where $p \neq k \wedge q \neq k$.

With void mask $V(R, \hat{d})$, the translation can be computed, depending on the geometric relations between the involved components. If a component is required to be fully attached, translation $\mathbf{t}$ is based on the maximum difference, that is $\mathbf{t} = -\hat{d} \cdot max(V(R, \hat{d}))$. This fully offsets one component relative to the other, maximizing the surface of attachment and thereby maximum attachment for the given direction. However, other options are relevant and supported as well, such as the minimum difference and the median difference. The minimum difference can be applied to minimally attach components: translation $\mathbf{t} = -\hat{d} \cdot min(V(R, \hat{d}))$. This could for instance be useful when positioning a component on a hollowed components. The median difference can be used to be more resilient against outliers, resulting in $\mathbf{t} = -\hat{d} \cdot median(V(R, \hat{d}))$. It could aid with preventing over- or undershooting component translation. Note that the goal here is to reach attachment in the desired direction; functions that cannot guarantee this are excluded. Basing the translation on the average difference, for instance, could result in translating a component between two other outliers and thereby result in no attachment at all. This would violate the geometric properties between components.

After translating a component, its attached components must also be translated by $\mathbf{t}$. We thereby achieve a chained component-wise assembly procedure that propagates the changes of one component's position to its dependent components. This highlights the importance of the acyclic nature of trees, otherwise the translation would never result in a new stable state of translated components.

There is another case of translation that ought to be considered: translating solved components. Upon solving a component, it could be the case the initial region of attachment is not maintained. For this reason, we add an additional translation step, as shown in Figure 4.2, that mirrors that of translating solved components. Given an initial region of attachment between a solved component C'_i and an unsolved component C_j and then the solved component C'_j, use C'_j's void mask to offset it to reach the desired connection again. Note that, for this, the minimum value in C'_j's void mask $min(V(C'_j, \hat{d}))$ is used, as choosing any other method could result in overlapping two solved components, which is not allowed. As with unsolved component translation, the translation is propagated to C'_j's attached components. Thanks to solving the components in DFS order, translating other solved components is not required.

Translating a component comes with consequences. For instance, the inclusion of the empty tile in a component's tile set could result in a scenario where attachment is impossible for the region of

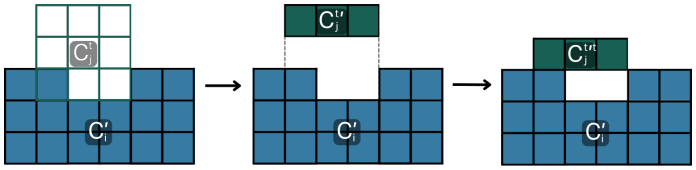


Figure 4.2: Solved component translation. Translating component C_j (left; continuation of Figure 4.1), it is solved (center) and then translated again to attach to component C'_i.

overlap regardless of translation in direction $-\hat{d}$. More concretely, consider the region of attachment R of two components C_i and C_j, where C_j depends on C_i, and their attachment direction $\hat{d} = \pm\hat{\mathbf{e}}_k$. Then, if the void mask $V(R, \hat{d})$ of R for that direction has a minimum value equals to C_i's extent $C_i.\Delta$ in dimension k, i.e. $min(V(R, \hat{d})) = C_i.\Delta_k$, no attachment is possible in direction $\hat{d}$, regardless of the translation in that that direction. Since the void mask is based on the layers of the component's grid's subset based on R, all of those layers are empty.

This would add numerous layers of complexity to component assembly. First and foremost, if such empty layers exist, the extent of the component changes. This can directly affect the relative position of its connected components, potentially requiring translations in multiple directions and changing the origins and extent of those components before solving them. Another example could be a solved component C' with two connected unsolved components C_i and C_j, with extents $C.\Delta$, $C_i.\Delta$ and $C_j.\Delta$ respectively, in direction $\hat{d}$. Suppose that $C_i.\mathbf{O} = (x, y, z) \wedge C_j.\mathbf{O} = (x + C_i.\Delta_x, y, z)$. Now suppose that C''s x-extent is smaller than its initial extent. Simply translating in x would not always yield a desirable solution, since the relative position of C_i and C_j would have to be changed. Likewise, changing the extents of C_i and/or C_j to mimic the extent change of C' could have serious consequences. Changing the attached components' grids could result in altered ratios or it could result in a grid too small to contain a set of its NUTs (if NUTs are used). Or worse, the connected components might become disconnected. This particular problem requires more research and we consider it future work. Instead, we tackle this issue by translating the component without considering these edge cases. If there is a risk of non-attachment, the solved component could be regenerated instead; it would fail to meet the constraints imposed on it.

4.5. Component seeding

Due to component translation, it is possible that a section of component C'_i will intersect with a section of component C^t_j, as shown in Figure 4.1. That section should not be overwritten when C^t_j is solved. Yet, we cannot simply copy the values C'_i to C^t_j, since both components may have different adjacency constraints and tile sets. Instead, we solve this issue by changing the shape of C^t_j's grid to ensure attachment, essentially blocking all cells occupied by C'_i, as shown in Figure 4.3. We call that process *component seeding*.

The process of component seeding has three steps:

1. Find the cells of C'_i that intersect with C_j.
2. Map the coordinates of those cells to coordinates in C_j.
3. For each cell, block the cell at the corresponding coordinate in C_j.

Step 1 obtains the set of cells, containing the cells whose coordinate $\mathbf{c}$ plus the origin of C'_i, $C'_i.\mathbf{O}$ is within the bounds of C^t_j with origin $C^t_j.\mathbf{O}$ and extent $C^t_j.\Delta$. By finding the region of attachment of these two components, as described in Definition 10, we obtain the region containing the cells of C'_i that affect C^t_j. We refer to this region as R with origin $R.\mathbf{O}$ and extent $R.\Delta$.

Step 2 and step 3 are responsible for actually changing the shape of the grid. The coordinates of R are mapped to those of C^t_j such that a coordinate $\mathbf{c}$ in R maps to coordinate $c_j = c + R.\mathbf{O} - C_j.\mathbf{O}$.

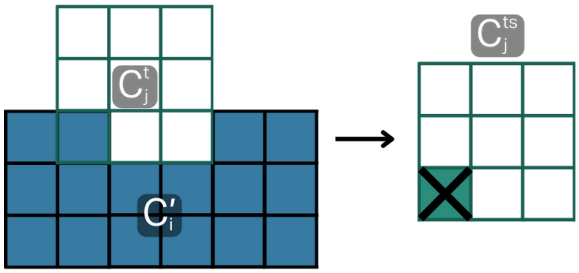


Figure 4.3: Seeding C_j^t based on C_i' to obtain C_j^{ts}. The bottom left cell of C_j^t intersects with C_i'. In C_j^{ts}, the crossed region represents the blocked cell.

For each cell in R that is collapsed a non-empty tile, we block its corresponding cell in C_j^t. With that, C_j^t is considered to be seeded (as shown on the right in Figure 4.3).

4.6. Tile and pattern elimination on seeded components

While assembly and NUTs are not co-dependent, if they are used together, then the elimination process (see Section 3.7) must be adjusted slightly. For tile and pattern elimination , the set of cells at the edges of the grid is required. Due to component seeding, a component's grid changes shape and thus its edges change compared to an unseeded grid; the requirements for completeness are still the same as in Definition 6. Since we know which cells are blocked by component seeding, we know which cells are at the edges of the grid: the unoccupied cells adjacent to the blocked cells joined with the unoccupied cells at the edges of the unseeded grid. This way, we capture the contours of the grid and can proceed with the elimination on the updated set of edge cells.

4.7. Structured objects and XWFC synthesis

After performing the previous initialisation steps, the main loop of XWFC can be entered. The procedure remains the same, except that the blocked cells are now introduced. To integrate this idea with XWFC, blocked cells are considered to be occupied. With this approach, we show that we can decompose a structured object into a set of components and maintain the component's geometric relations by iterative solving and positioning each component. Note that XWFC is not necessarily required for components, it can also be run with WFC.

Algorithm 6 XWFC on components

$C = \{C_0, \ldots, C_n\}$; T_A = AttachmentTree(C); S = DFS(T_C)
C_{prev} ▷ The penultimate solved component
for S_i **in** S **do** ▷ Where S_i contains a NUT set, origin, extent and input
    G = SeedComponent(Grid($S_i.\Delta$)) ▷ Where $S_i.\Delta$ is the associate component's extent
    if MODE is STM **then**
        SimpleTiledModel(S_i, G_i') ▷ See Algorithm 1, 2, 3
    else
        OverlappingModel(S_i, G_i') ▷ See Algorithm 5
    end if
    if $C_{prev}$is assigned **then**
        TranslateSolved(S_i', C_{prev}) ▷ Edge case: component S_i''s region of attachment is void
    end if
    $C_{prev} = S_i'$
    TranslateUnsolvedConnectedComponents()
end for
Assemble(S)

5

Implementation

The approach explained in the Chapters 3 and 4 has been implemented in C#, visualized in Unity [32] and can be found here: [22]. This chapter highlights the practical implications of NUTs, learning from input and discusses component assembly in greater detail. It also touches upon optimization with Single Instruction, Multiple Data (SIMD) and showcases how visualization is done. A Python implementation of a prototype XWFC Simple Tiled Model on NUT adjacency constraints is also available here: [23].

5.1. Data structures

The data structures underlying the implementation are important to understand for the subsequent sections in this chapter. In this section, we showcase the data structures for the most prominent parts of XWFC: the grids, NUTs, atom mapping and adjacency constraints. We also applied a few optimizations using Single Instruction, Multiple Data (SIMD).

First, XWFC utilizes several grids: `wave` of units — be it atoms, NUTs or patterns — and an `atom grid`. The `wave` is a 4D matrix, where each cell in the 3D grid consists of a 1D array of booleans, one boolean per unit. It is used to represent the state of each cell, i.e. which units are allowed per cell. The `atom grid` is a 3D matrix filled with integers, since integers are used to identify the atoms.

Second, one of XWFC's main contributions is supporting NUTs with a simple yet elegant solution that has a couple of underlying nuances. As described in Definition 1, a NUT consists of a set of atoms. In the back-end, this is represented as an atom mask consisting of a 3D matrix of booleans, where `true` denotes an occupied cell and `false` denotes void. This way, we can express the shape and size of the NUT akin to approach explained in 3.1. The index in this matrix corresponds to the atom coordinate, the combination of `tileId` and `atomCoordinate` is unique, yet atom identifiers are still desirable, as that is compatible with indexing.

Each NUT in the NUT set is assigned a unique tile identifier, that is their associated index in the set represented as a list. These NUTs' atoms are assigned unique identifiers as well following the same procedure, ranging from 0 to the total number of atoms (i.e. the count of `true` values in all NUT atom masks). This is akin to using registers, where each NUT is assigned a range of atom identifiers for its atoms.

Third, as our approach requires frequent access to this atom mapping data, it is pre-computed, which is more efficient than computing it on the fly. We store the mapping in an `AtomMapping` structured as a `Bidirectional Dictionary` for mapping `(tileId, atomCoordinate)` $\leftrightarrow$ `atomId`,

where `tileId` and `atomId` are `int`s and the `atomCoordinate` is a `Vector3Int`[1]. This information is especially important for inferring atom adjacency constraints from NUT adjacency constraints and atomizing input.

Fourth, an adjacency constraints are represented as a tuple $(u_i,\ u_j,\ \hat{d})$. It is read as: unit u_j is adjacent to unit u_i in direction `d`, where u_i and u_j are `integer` unit identifiers and $\hat{d}$ is a 3D unit vector such that $d = \pm\hat{e}_k$. These constraints are stored in an `AdjacencyMatrix`, consisting of a grid of size NxN for each direction in the set of directions, where N is the total number of units used. This shows that mapping identifiers directly to indices, at least under the hood, prevents the need for an identifier-to-index mapping and therefore removes a layer of complexity.

Finally, SIMD provides the means to greatly reduce the run-time of XWFC through vectorization. This is especially relevant during propagation as it heavily relies on abundantly performing element-wise comparisons of arrays, such as rows in the adjacency matrix. For this reason, dotnet's `Vector<T>` struct is used [18]. The adjacency matrix' rows are vectorized during pre-computation and referenced during run-time. Likewise, each cell in the `Wave` matrix is also vectorized. Rather than comparing a single value each time, sixteen are compared instead (limited by the capacity of a single Vector instance), thus significantly speeding up the process.

5.2. Void masking

With the background knowledge established, a closer look can be at taken more intricate operations, such adjacency inference of concave tiles (see Section 3.4). Inference requires the use of void masks. These are computed as described in Algorithm 7. In this algorithm, the generality of void masks becomes more evident. As long as there is at least one termination condition for counting voids, this approach works. We specify the direction from which the voids are counted and continue doing so until a cell containing a non-empty atom is encountered. Thanks to its generality, it can be directly used in component assembly as well.

Optimizations are possible, but not very impactful. In the worst case, each cell in the grid would have to be considered, thus still yielding a polynomial complexity. One could for instance consider to stop counting as soon as all values in V^{-}_{done} and V^{+}_{done} are true. However, void mask calculation is not performed frequently. It occurs in the initialization phase for NUTs and once per component. Its effective cost is thus low.

Likewise, counting voids halts from a given direction for a given cell column as soon as a non-void cell is encountered. Grids with holes, i.e. genus larger than 0, or with overlap/overhang relative to the given direction are not accounted for. Even though the computation method can be easily extended to achieve this, it was not part of our requirements.

5.3. NUT adjacency constraint atomization

In Section 3.4, we formulate a general approach for connecting two arbitrarily shaped grids with arbitrary sizes by translating one relative to the other. Using said approach, NUT adjacency constraints can be formulated as a set of atom adjacency constraint. Deriving atom adjacency constraints from tile adjacency constraints use the atomized NUT and the its void masks.

Note that, upon finding a configuration, all atoms in the base still must be considered as possible candidates for atom adjacency constraints. Even though the NUT adjacency constraint is in direction $\hat{d}$, it may require constraints in other directions.

This method could be enhanced by a more generalized approach and is be considered future work (see Section 8.2). The current approach only catches the first region of voids per direction, even though a tile could have multiple such regions. This is the case for tiles whose projections in direction $\hat{d}$ would result in self-overlap in the given direction. A useful observation is that only atoms

[1] Vector3Int

Algorithm 7 Directional Void Masks Computation

function CalculateVoidMasks
  $D \leftarrow$ nDimensions
  $N \leftarrow \{n_0, \ldots, n_n\}$ ▷ NUT set, where a NUT n_i has extent $n_i.\Delta$ and atoms A
  $V \leftarrow \{d_0 : \{n_0 : \{\}, \ldots, n_n : \{\}\}, \ldots, d_k : \{n_0 : \{\}, \ldots, n_n : \{\}\}\}$ ▷ $k = 2 * D$; two directions per dimension
  for each $n_i \in N$ **do**
    for $d < D$ **do**
      x', y' = ComplementaryDimensions(d) ▷ The complementary dimensions, i.e. the void mask's axes x' and y' compared to the global axes. d = y -> x' = z, y' = x. d = x -> x' = z, y' = y. d = z -> x' = x, y' = y.
      $\Delta \leftarrow n_i.\Delta$
      $V^+, V^- \leftarrow$ InitializeMatrices$(\Delta_{x'}, \Delta_{y'}, 0)$
      $V^+_{\text{done}}, V^-_{\text{done}} \leftarrow$ InitializeMatrices$(\Delta_{x'}, \Delta_{y'}, \text{false})$
      $G \leftarrow$ AtomGrid$(n_i.A)$ ▷ See Section 3.1
      for each cell $c = (x, y, z)$, where $\mathbf{O} \leq c < \Delta$ **do** ▷ in incrementing order
        $c^- \leftarrow (x, y, z)$
        if not $V^-_{\text{done}}[c_{y'}, c_{x'}]$ **then**
          if $G[x, y, z]$ is void **then**
            $V^-[c_{y'}, c_{x'}] \leftarrow V^-[c_{y'}, c_{x'}] + 1$
          else
            $V^-_{\text{done}}[c_{y'}, c_{x'}] \leftarrow$ true
          end if
        end if
        $c^+ \leftarrow \Delta - \mathbf{1} - c$
        if not $V^+_{\text{done}}[c_{y'}, c_{x'}]$ **then**
          if $G[x, y, z]$ is void **then**
            $V^+[c_{y'}, c_{x'}] \leftarrow V^+[c_{y'}, c_{x'}] + 1$
          else
            $V^+_{\text{done}}[c_{y'}, c_{x'}] \leftarrow$ true
          end if
        end if
      end for
    end for
    $direction \leftarrow (0, 0, 0)$
    $direction_d \leftarrow 1$
    $V[direction][n_i] \leftarrow V^-$
    $V[-direction][n_i] \leftarrow V^+$
  end for
end function

Algorithm 8 NutAdjacencyAtomization

function NutAdjacencyAtomization(d, n_i, n_j, V)
  Expresses the given NUT adjacency constraint between NUTs n_i and n_j as a set of atom adjacency constraints for direction d using the void masks V. All entities' extents are accessed through their Δ property
  $k \leftarrow k$ where $d_k \neq 0$ ▷ Get the dimension from the direction
  sign $\leftarrow d_i$
  $B, S \leftarrow n_i, n_j$ ▷ Slider will be shifted while base remains in place
  $\mathbf{s} \leftarrow V(n_i, d).\Delta + V(n_j, -d).\Delta - \mathbf{1}$ ▷ Number of shifts required to cover all configurations that could yield adjacency constraints
  x', y' = ComplementaryDimensions(d) ▷ See Algorithm 7
  for $\delta_y < s_{y'}$ **do**
    $S_y^-, S_y^+ \leftarrow$ SliderRange$(\delta_y, S.\Delta_y, B.\Delta_y)$ ▷ See Algorithm 9
    $B_y^-, B_y^+ \leftarrow$ BaseRange$(\delta_y, S.\Delta_y, B.\Delta_y)$ ▷ See Algorithm 10
    for $\delta_x < s_{x'}$ **do**
      $S_x^-, S_x^+ \leftarrow$ SliderRange$(\delta_x, S.\Delta_x), B.\Delta_x)$
      $B_x^-, B_x^+ \leftarrow$ BaseRange$(\delta_x, S.\Delta_x, B.\Delta_x)$
      $V(S, d)' \leftarrow V(S, d)[S_y^- : S_y^+, S_x^- : S_x^+]$
      $V(B, -d)' \leftarrow V(B, -d)[B_y^- : B_y^+, B_x^- : B_x^+]$
      $V_{sum} = V(S, d)' + V(B, -d)'$
      canAttach$= false$
      for $v \in V_{sum}$ **do**
        if $v \neq B.\Delta_k \wedge v \neq S.\Delta_k$ **then**
          canAttach$= true$
          break
        end if
      end for
      $\epsilon \leftarrow \min(V_{sum})$
      if ¬canAttach **then**
        continue ▷ No adjacency, base and slider are both only void in this configuration
      end if
      $O' \leftarrow \mathbf{1} - d$ ▷ Complementing vector used as offset for skipping corner adjacency
      $P_S.\Delta = -S.\Delta + O'$ ▷ Position of slider in relative to base in the minimum configuration
      if sign > 0 **then**
        $P_{S,min}.\Delta \leftarrow P_{S,min}/\Delta + d \cdot (B.\Delta + S.\Delta)$ ▷ If the sign is positive, the first overlap configuration is at the positive side of the base
      end if
      $P_{S,y'}.\Delta \leftarrow P_{S,y'}.\Delta + \delta_y$
      $P_{S,x'}.\Delta \leftarrow P_{S,x'}.\Delta + \delta_x$
      $P_{S,k}.\Delta \leftarrow P_{S,k}.\Delta + \epsilon \cdot -$sign ▷ Offset the slider against d
      for each base's atom a with atom coordinate c **do**
        $a_{B,\text{id}} \leftarrow$ AtomMapping$[(B.\text{id}, c)]$ ▷ Where B.id corresponds to the base's associated NUT's identifier.
        for each $d' \in D$ **do** ▷ Adjacency is not limited to direction d
          $P_{B,\text{rel}} \leftarrow c + d'$ ▷ Position of the neighbor
          $P_{S,a_b} \leftarrow P_{B,\text{rel}} - P_{S,\Delta}$ ▷ Base atom coordinate mapped to that in the slider
          if $P_{S,a_b} \in A_S$ **then** ▷ If the mapped coordinate is associated with an atom in the slider
            $a_{S,\text{id}} \leftarrow$ AtomMapping$[(S.\text{id}, P_{S,a_b})]$
            $M[d]_[d', a_B.\text{id}, a_S.\text{id}] \leftarrow$ true
            $M[-d]_[a_S.\text{id}, d', a_B.\text{id}] \leftarrow$ true
          end if
        end for
      end for
    end for
  end for
end function

Algorithm 9 Range Calculation for Slider

function SliderRange($\delta, S.\Delta, B.\Delta$)
Calculates the index range of the section of the slider's void mask that overlaps with the base's void mask given a shift δ in 1D. $S.\Delta$ and $B.\Delta$ are the slider's and base's extents.
$S^- = \max(0, S.\Delta - 1 - \delta)$ ▷ Two cases: $\delta < B.\Delta \rightarrow S.\Delta - 1 - \delta$. Or $\delta \geq B.\Delta \rightarrow S.\Delta - 1 - \delta$.
$S^+ = \min(S.\Delta - 1, S.\Delta - 1 + B.\Delta - 1 - \delta) + 1$ ▷ Two cases: $\delta < B.\Delta \rightarrow S.\Delta - 1$. Or $\delta \geq B.\Delta \rightarrow S.\Delta - 1 - (\delta - B.\Delta)$
return S^-, S^+
end function

Algorithm 10 Range Calculation for Base

function BaseRange(δ, $S.\Delta$, $B.\Delta$)
Calculates the index range of the section of the base's void mask that overlaps with the slider's void mask given a shift δ in 1D. $S.\Delta$ and $B.\Delta$ are the slider's and base's extents.
B^- = max(0, δ - $S.\Delta$ + 1) ▷ δ_{max} = $B.\Delta$ + $S.\Delta$ - 1. For linear translation with δ = 0 ... δ_{max}, $B.\Delta$ = δ - $S.\Delta$ + 1. Cannot exceed base's bounds
$B^- = \max(0, \delta - S.\Delta + 1)$
B^+ = min(δ, $B.\Delta$ - 1) + 1 ▷ Linear translation along base. Cannot exceed base's bounds
return B^-, B^+
end function

without inter-atom adjacency constraints in a given direction can be adjacent to an atom of another tile without inter-atom adjacency constraints for the opposing direction. Then, the configurations could be inferred from those atoms' coordinates.

5.4. Input representation through grid atomization

XWFC takes a string matrix as an input. In the definition of a NUT, we can augment the atom mask to contain string values instead of booleans, where the empty string would represent `false` and any other string represents `true`. A NUT's atoms do not necessarily have to be represented by the same string. This is akin to how a tile could consist of an arrangement of different single cellular textures. Consequently, different NUTs can have atom representations in common; it is the NUT's atom configuration that should be unique.

It could be the case multiple solutions exist for atomizing an input grid, or that a conflict could be encountered during inference. However, since the matrix is known to be derived from a valid arrangement of NUT's, at least one solution must exist. If there are multiple, they could each serve as a separate input matrix from which constraints could be derived. resulting in a grid with a higher information density as well as a higher level of ambiguity.

We favor a lower density with no ambiguity over high density and high ambiguity. Choosing the latter would introduce much complexity and diverge from our main research focus. Therefore, we acknowledge this limitation, but decide to reduce our scope to operate on tiles whose atoms can only be compatible with a single NUT. In other words, each tile must have its own unique value that represents it in the grid. This decision allows the use of LEGO or Tetris tile sets or any tile set whose tiles are uniquely identified through their value.

5.5. Component seeding

In Section 4.5 we discuss component seeding. We mention that we block cells that belong to the intersection of the solved and dependent components. The blocked cell in the dependent

component in neither empty nor solid, it is a change of the grid's shape. It should therefore not map to any atom that exists for in the NUT set. The atoms range from 0 until n, where n is the number of atoms in the set. Since XWFC utilizes both a wave and an atom grid, we also use a default value to fill the atom grid to represent uncollapsed cells, which is -1. Therefore, we require another value to represent the blocked cells, for which we use -2. A cell is considered collapsed if its value is not equal to the default value, any collapsed cell cannot be collapsed again. So, blocking a cell can be achieved through assigning the cell the value -2, as that has the properties we require for changing the grid's shape.

5.6. Visualization

The visualization consists of three components: an Animator, XWFC, a Component Manager (CM) and a Graphical User Interface (GUI).

The Animator is the entity responsible for calling XWFC and the CM, and visualizing the results. It thus also serves as the inferface between XWFC and the CM to enable granular visualization possibilities and maintaining the modularity of the software. The Animator is thus at the core of the visualization and regulates all communication. Therefore, we have chosen to employ the singleton pattern for the Animator, such that all other entities interact with the same Animator. The XWFC instance is responsible for solving a component or grid. It has been designed such that it is completely decoupled from the other entities. It can be run independently and its results can obtained through exposed variables. The CM's task is to manage and prepare the components and perform structured object assembly.

The visualization loop:

1. Initialize the components.
2. Ask assembler to select and prepare a component C_i to solve.
3. Clear the canvas.
4. Initialize XWFC with component C_i.
5. Enter the Animator's update loop.
6. Call XWFC to collapse when given the command.
7. Observe and draw the changes.
8. Once component C_i is solved, inform the assembler, save the output and return to step 2.
9. Once all components are solved, assemble and visualize them when given the command.

The animator manages an instance of XWFC and observes state changes through function calls. Likewise, XWFC and the CM are not co-dependent. What matters for the CM is the component relations and requires a grid-based output, thereby making it compatible with other WFC implementations for example. Since XWFC and the assembler are decoupled, the Animator serves as a communication layer between those entities.

5.6.1. Animation

There are two steps involved in XWFC's animation. First, a mapping of the back-end to the front-end is created. This includes mapping atom to a visual element as well as detecting state changes in output grid of XWFC. Second, the animation itself is granular to allow for more control while running XWFC and component assembly.

Atoms are represented as integers by XWFC, though for visualization, these should be 3D single unit tiles. For this reason, we also specify a NUT's color and atom asset. These could be interchanged with actual textures if needed. In our implementation, this atom asset is set to be same for all atoms, where NUTs are differentiated by color.

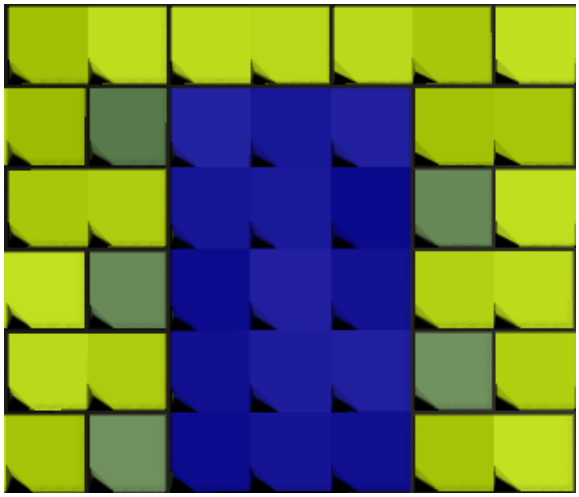


Figure 5.1: A magnified sample of an output grid, highlighting the color fluctuations and the edges.

In order for the animator to know what must be drawn, it observes the state of the XWFC output grid. When the animator observes a change in the output grid, it draws said change onto a 3D canvas and keeps a reference of it in a separate grid, which is important for applying the changes occurring in XWFC. That grid facilitates differentiating between old and new states. When a change is observed in a cell, that atom associated with that cell is removed (if any) and a new instantiated atom takes its place. This approach is robust and can deal with state reversion caused by conflict handling.

Due to the uniformity of a NUT's colors, it may be difficult to distinguish between instance of NUT's if two their colors are the same, e.g. when two instances of the same NUT are adjacent. To add contrast between placed NUTs and emphasize their shape and size, we perform two operations: color fluctuations and dynamically added edges. The color fluctuations are achieved by introducing a small random color deviation to the NUT's color upon drawing its corresponding atoms. A NUT's edges can be added to the drawn units dynamically to add a wire frame overlay as it were. Here, for each face at the associated directions, edges are drawn if there are no inter-atom adjacency constraint in said direction. This ensures that the drawn NUTs are clearly distinguishable. Dynamically adding edges is independent of the unit used to represent an atom. Figure 5.1 illustrates these properties.

Granular control is achieved by designing the animator and XWFC such that the animator may iteratively call XWFC to perform a collapse and update the canvas accordingly. The animator can observe the changes in XWFC's output grid and compare it to what is currently drawn on the canvas. Through the introduction of a step size and a delay between steps, it is possible to control the speed and intensity at which the grid is solved and visualized.

5.6.2. GUI

The GUI consists of a drawn 3D canvas and a menu. The collapsed NUTs are visualized on the canvas and the menu facilitates interaction. There are four tabs in the menu: Grid, Adjacency View, Tile Set and Config.

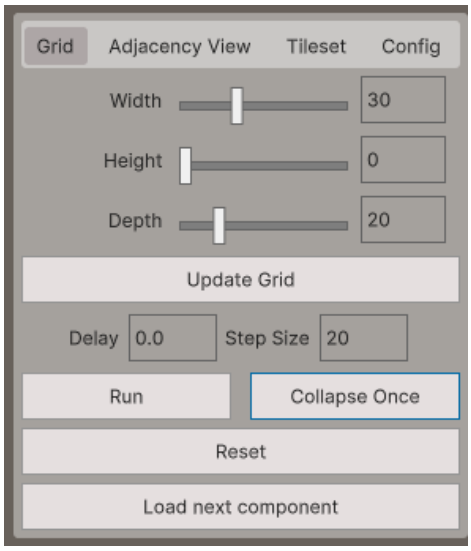
The 'Grid' tab hosts the grid configuration (see Figure 5.2a). Here, the grid's extent can be set and the visualization parameters can be specified. For instance, the step size and the delay denote the number of cells to collapse per update and the time between each update. The tab also provides the means of running, pausing and resetting the solving process. Once this process is complete, the user can continue to the next component. If there is no other component, the user is provided

the option to start and visualize the resulting assembly of components.

The ‘Adjacency View’ tab displays the tile adjacency constraints (see Figure 5.2b). This is especially helpful for the XWFC implementation relying on NUT adjacency constraints. It allows the user to tweak the constraints from within the editor; the user can thus play with the constraints as desired, e.g. by toggling them. The constraints can be specified per tile and per direction. To ease this process, we support inverting selections and make sure that the constraint’s symmetric nature is maintained. Said tab is not relevant if the constraints are learned from an input.

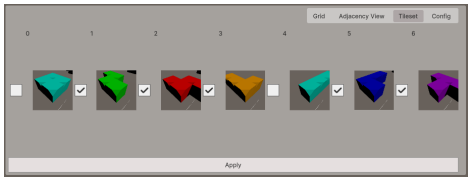
The ‘Tile Set’ tab enables the selection of tiles during run time (see Figure 5.2c). It shows a preview of each tile along with their identifier. Since the tile identifiers are determined in the back-end, and these are used to communicate or represent the tiles in the front-end, we have added an overview of the NUTs in the visualization.

Finally, the ‘Config’ tab (see Figure 5.2d). This tab allows for saving a certain configuration, that is the constraints and tile set selection. These configurations can later be selected to immediately load those constraints and tile set.

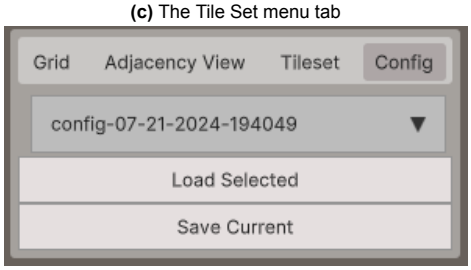


(a) The Grid menu tab. The sliders control the grid’s extent. The delay is in seconds and the step is the number of collapses per individual cycle

(b) The Adjacency View menu tab



(c) The Tile Set menu tab



(d) The Config menu tab

Figure 5.2: Menu tabs overview.

6

Experiment Design

In this chapter we design and discuss a series of experiments for each of XWFC's facets to assess the quality of XWFC's ability to generate structured objects consisting of tiles from the Tetris and LEGO domains. The experiment results will be discussed in Chapter 7. Specific details of the experiments' parameters can be found in Appendix A.

The experiments follow a bottom-up order. First, we focus on XWFC's ability to solve individual components with varying NUT sets and adjacency constraints. For this, we experiment with both the Simple Tiled Model and the Overlapping Model in 3D. The purpose of these experiments is to assess Hypothesis 1. Second, we use the results of the Overlapping Model experiments and assess how XWFC can handle a simple stacking problem to generate a structured object. Finally, we focus on component assembly and inspect how it performs for a variety of component structures for more complex structured object component relations. These experiments assess Hypothesis 2. In order to design the experiments, we need the Tetris-inspired and LEGO-inspired tile sets (which WFC would struggle with) used for these experiments, as we explain next.

6.1. Tetris and LEGO Non-Uniform Tile sets

The Tetris-inspired NUT set consists of a set of tetracube NUTs that mimic each of Tetris' "Tetriminoes" [31], as shown in Figure 6.1. Each NUT has a different and relatively complex shape and consists of exactly four units. To further increase its variety, we magnified all NUTs by a factor two, as shown in Figure 6.2. We refer to the tetracubes by their names, as shown in the aforementioned figures. To differentiate between the small and large tetracubes, we add the `2` suffix.

By supporting both Tetris-inspired and LEGO-inspired NUT sets, we show that we gain a considerable degree of expressiveness, thanks to the generality of XWFC approach to solving with NUTs.

For the Simple Tiled Model in particular, many different configurations exist in which the tetracubes can be adjacent, piecing the NUTs together in a grid is certainly non-trivial without leaving gaps. Just like in the classic Tetris game, leaving gaps where no Tetrimino (or tetracube in our case) fits is undesired.

The LEGO domain is known for its versatility and support for nearly endless creativity. The LEGO-inspired NUT set used for the experiments consists of a subset of the vast array of bricks of different shapes and sizes (see Figure 6.3). It also contains as a set of *compound bricks*, which are NUTs consisting of a combination of of other bricks, such as a door or window. The names of the brick NUTs follow a `width,height,depth` naming scheme, with a `b` prefix to signify brick, `p` for plate and `w` for window/translucent. Compound bricks have names that represent their meaning, such as "door". The set also includes "void" NUTs to represent empty space. The LEGO NUT set will

Figure 6.1: A NUT set of tetracubes, inspired by Tetris' Tetriminoes[31], each consisting of four units. The names of the tiles are from left to right: `L`, `T`, `J`, `I`, `Z`, `S` and `O`. Color fluctuations are applied to distinguish between individual atoms, signifying the difference in sizes compared to the NUTs' large counterparts.

Figure 6.2: A NUT set of magnified tetracubes (See Figure 6.1 for their base versions), inspired by Tetris' Tetriminoes[31]. Each NUT comprises sixteen units. The names of the NUTs are from left to right: `L2`, `T2`, `J2`, `I2`, `Z2`, `S2` and `O2`. Color fluctuations are applied to distinguish between individual atoms, signifying the difference in sizes compared to the NUTs' smaller counterparts

allow us to showcase the versatility of NUTs. Being able to create NUTs derived from other NUTs should enable a larger degree of expressiveness. The challenges of the LEGO NUT set relate to how bricks are placed with intent and purpose, for instance for enforcing certain patterns or combinations of NUT arrangements. With the domains in mind, the experimental setup can be formulated.

6.2. Simple Tiled Model on tetracubes

There are two qualities we want to assess. First and foremost, XWFC should be able to find valid configurations of NUTs, while taking their shape and size into account, and express them as a set of constraints. Second, XWFC should display its ability to express various NUT adjacency constraints and find a solution should the constraints allow it.

We design two experiments to assess these qualities. **Experiment 1** is designed to be a variation of the graph coloring problem [8]. We have selected a NUT set of five NUTs: `I`, `L`, `O`, `S` and `T`. We set their NUT adjacency constraints such that no NUT may be self-adjacent in any direction. With those constraints, we aim to enforce XWFC to find patterns that were not specifically defined. The set is large and diverse enough to be solvable and sufficiently small and restrictive to make the solution non-trivial. See Figure A.1 for graphic details.

Experiment 2 assesses to what extent XWFC is capable of handling NUTs of vastly different sizes. The NUT set for this experiment contains these NUTs: `J`, `L`, `S`, `T` and `Z` as well as large NUTs: `O2`, `S2`, `T2` and `Z2`. The NUT adjacency constraints are very permissive, where all NUTs are allowed to be adjacent. This isolates the ability to solve with vastly different shapes and sizes. See Figure A.2 for graphic details.

6.3. Overlapping Model house assembly

In this section, we design a series of experiments that assess to which extent XWFC can create various outputs with varying adjacency constraints and NUT sets. The experiments are themed after building a house comprising five components: first floor, second floor, balcony, roof and chimney. Each experiment yields one such component. In addition, each experiment will operate on LEGO-inspired NUT sets whose NUTs' shapes and sizes must be preserved, even when used in patterns, as is the case for the Overlapping Model). See the Figures in Appendix A.2 for graphic details per experiment.

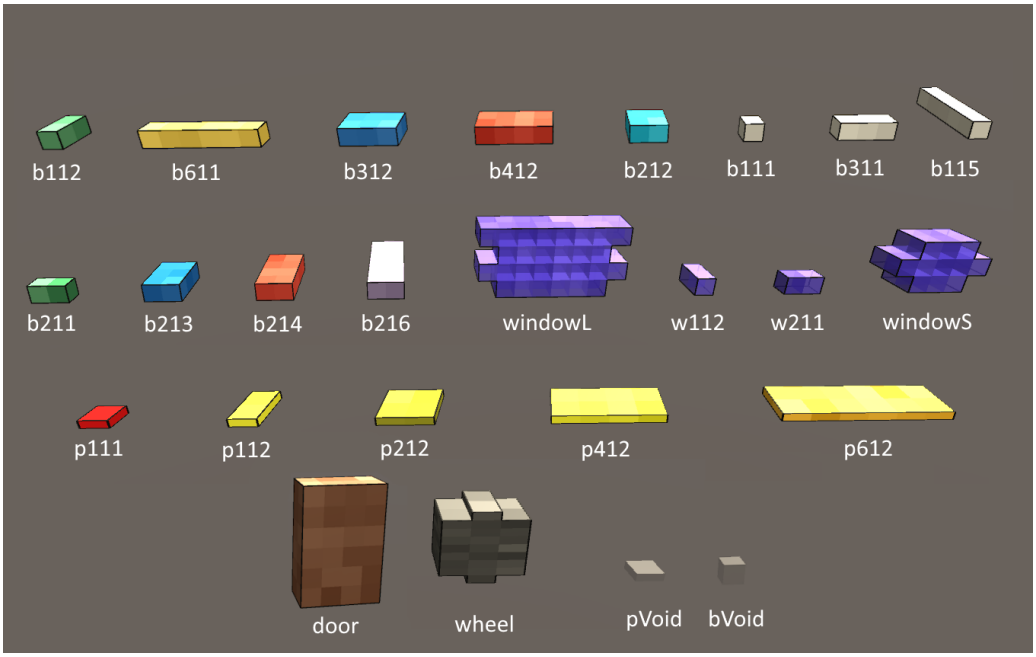


Figure 6.3: The LEGO-inspired NUT set (annotated).

Experiment 3 assesses to what extent XWFC is able to replicate simple patterns and to which extent it maintains a simple NUT's properties. We choose a running bond brick pattern using NUTs for the former and the `door` NUT for the latter. XWFC is expected to preserve the size of the door, unlike WFC. The results will serve as the first floor of a house.

After that, **Experiment 4** inspects whether XWFC is able to synthesise larger patterns that are not strictly defined in the input while handling large NUTs with jagged shapes (the `windowL` NUT). The results will represent the second floor of a house.

Then, **Experiment 5** differs from the previous experiments. It features a more loosely defined compound NUT, namely a balcony. We have created a balcony with a specific configuration of individual NUTs. Here we assess to what extent the small `2x2x2` patterns can capture such specific configurations of NUTs. Furthermore, we add a stacked bond brick configuration comprising `b412` NUTs. Such a pattern would be prone to forming bricks of arbitrary width if run with WFC on `2x2x2` patterns. This balcony will be the third layer.

Next, **Experiment 6** challenges XWFC with the inclusion of `void` NUTs, which is expected to affect the overall shape of the result. Since the output will serve as the roof, the overall roof shape should be preserved. The input is fairly simple. It is a small portion of a roof, forming a triangular prism shape. Finally, **Experiment 7** is a case of component seeding in a small grid that yields a chimney with a running brick pattern consisting of NUTs, to be placed on the roof.

6.4. Component assembly: house experiment

The experiments in Section 6.3 will be used to formulate a simple component assembly problem, captured in **Experiment 8** as part of assessing Hypothesis 2. Here, the results are stacked on top of each other according to their geometric relations. This involves a relation where the first floor has a second floor on top, on which the balcony layer resides. On top of the balcony is the roof and the chimney rests on top of the roof. All layers but the roof are simple. The roof poses an additional challenge where the results of the solved component have to be taken into account to ensure attachment of components, i.e. translation of a solved component. Likewise, the chimney

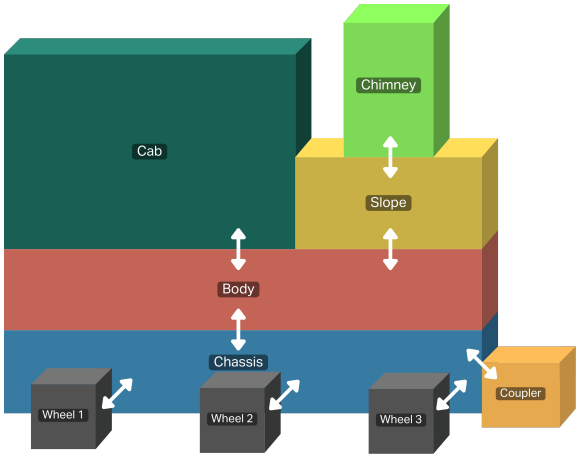


Figure 6.4: Initial relations between components. The bidirectional arrows denote the geometric relations.

component has to be on top of the roof and follow the roof's contours.

6.5. More complex relations: steam train experiment

In this section, we take a closer look at more complex component assembly. Component assembly of the house was fairly straightforward, with little interference between components. To truly challenge Hypothesis 2, we design **Experiment 9**. This experiment focuses on XWFC's ability to assemble components with relatively complex geometric relations. These relations are characterized by how a solved component can change the regions of attachment, including intersections of the components' AABBs, which must be updated accordingly. It could thus be the case that solving a component and translating others yield new or different regions of attachment. See the Figures in Appendix A.3 for graphic details per component.

We consider a steam train decomposed into twelve components: six wheels (three on both sides, symmetric in z-axis), a chassis, a body, a cab, a chimney, a slope and a coupler. The initial relations are shown in Figure 6.4. The NUT sets includes both bricks and plates (bricks are thrice the height of plates in LEGO [14]) and is kept relatively simple. The wheels are simply the `wheel` NUT. There are three on each side, symmetrically aligned in the z-axis. The chassis is assigned a bounding box larger than the result. This is to ensure that the body does not initially interact with the wheels, but only once the wheels and chassis are solved. The cab, chimney and coupler have simple geometric relations.

7

Results and Discussion

In this chapter, we present and discuss the results of the conducted experiments described in Chapter 6. The results of these experiments will aid with answering our main research question: “How can Wave Function Collapse generate structured objects consisting of 3D Non-Uniform Tiles?”.

7.1. Simple Tiled Model Tetris

In general, XWFC’s Simple Tiled Model works mostly as expected, based on the conducted experiments described in Section 6.1. The results of Experiment 1 show that XWFC is able to find an arrangement of NUTs that adheres to the imposed constraints, displayed in a grid of size 20,1,20. No NUT is self-adjacent. The result also shows how XWFC finds diverse patterns that were not strictly provided in the input. XWFC encountered several conflicts while solving the grid, but was able to handle those aptly. The results also show how the NUTs can induce biases; the green `S` NUT frequently occurs along with the orange `L` NUT in the same configuration. Unfortunately, the result also expose a limitation: the shapes and sizes of NUTs at the borders of the grid are not preserved. The tile elimination step described in Section 3.7 should work in theory, which it does for the Overlapping Model implementation, it does not seem to function properly for the Simple Tiled Model implementation.

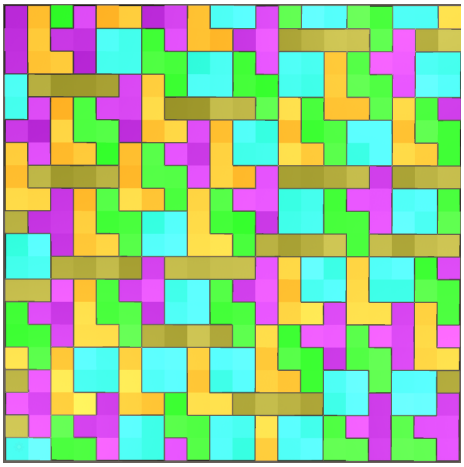
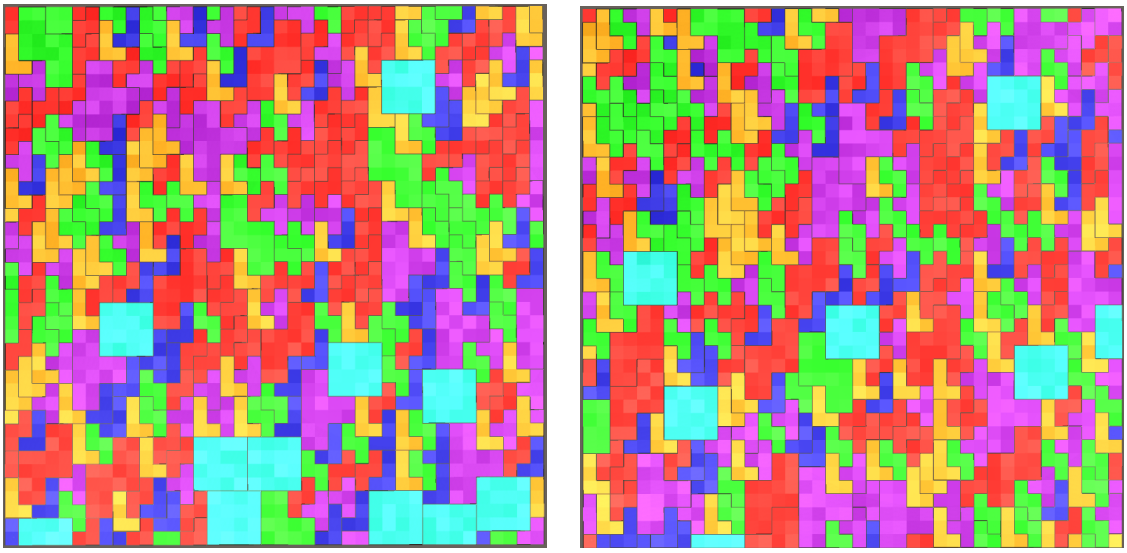


Figure 7.1: Results of Experiment 1 in a grid of size 20x1x20: no NUT is self-adjacent



(a) Large and small NUTs coexist.

(b) Occasional NUT property violation within the grid (see top left).

Figure 7.2: Two results obtained by running Experiment 2, with grid size 40x1x40.

Experiment 2 yields expected results as well, as illustrated in Figures 7.2a and 7.2b. They clearly show how the small and larger NUTs coexist; the NUT adjacency constraint are properly represented as a set of atom adjacency constraints, which are correctly propagated. Similar to the results of Experiment 1, the NUTs have biases, as can be seen from the green `S` NUT. This NUT is frequently stacked in similar configuration, likely due to how it is a seamless fit for itself.

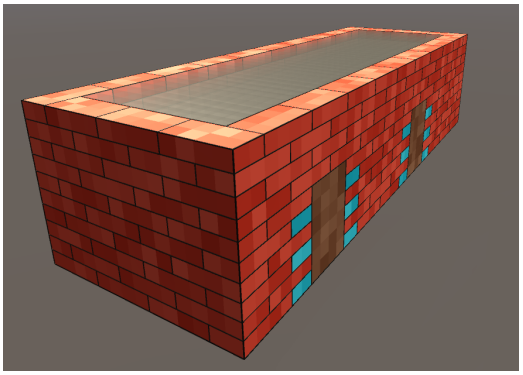
However, in certain cases, a NUT's shape and size are not preserved, as can be seen in the top-left corner of Figure 7.2b. Here the `J` NUT is truncated prematurely. The is most likely the cause of a bug in our implementation related to saving and restoring save points. Upon reaching a conflict, XWFC reverts to an earlier state. We suspect that there is an issue where the Wave (i.e. the grid containing the remaining allowed atoms per cell) is not saved correctly and thus not restored correctly. If this were in another part of the program, we would expect these problems to arise more consistently.

7.2. Overlapping Model: house assembly experiment

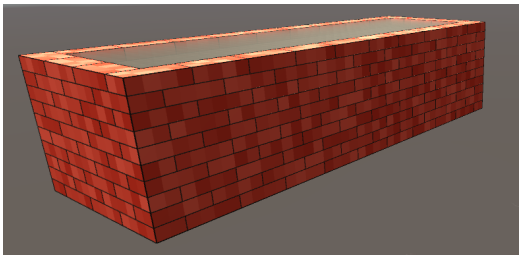
In this section, we display and discuss the results of house component experiments described in Section 6.3 followed by the house assembly experiment mentioned in Section 6.4. An overview of the LEGO NUT set, along with the names of NUTs can be found in Figure 6.3.

The first component is the first floor, consisting of the running brick pattern and a door pattern, as mentioned in Experiment 3. Figure 7.3 shows how XWFC applies the brick patterns from the input, while also preserving the size of the door. Standard WFC would not be capable of achieving this consistently. Furthermore, all NUTs properties are preserved correctly. In this instance, we successfully performed the Overlapping Model, operating on patterns, while also operating in the atom domain.

Next, the second floor component is obtained by running Experiment 4, the results of which are visualized in Figure 7.4. This result shows how XWFC does not merely copy recurring patterns, but is able to combine them to create new ones. XWFC thus still provides variety. The chains of cyan `b212` NUTs were not present in the input, for example. By choosing a pattern with odd-sized bricks (`b312`) and an even-sized grid, XWFC is encouraged to find a tiling that requires the small patterns, while preserving the NUTs' shapes. In addition, XWFC's Overlapping Model is not limited

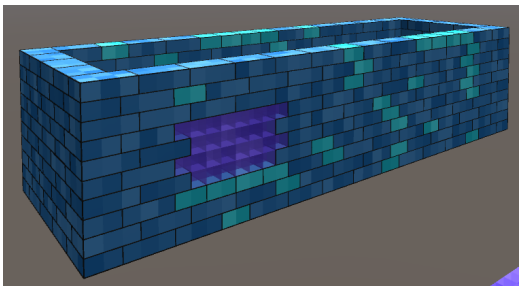


(a) Front view

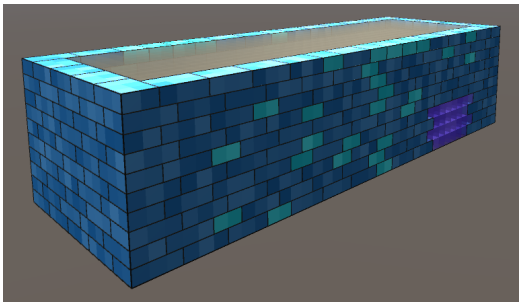


(b) Rear view

Figure 7.3: Two views of the first floor component in a 40x10x14 grid.



(a) Front view



(b) Rear view

Figure 7.4: Two views of the second floor component in a 40x10x14 grid

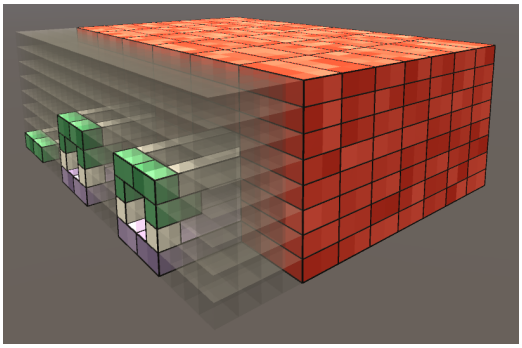
to cuboid bricks; it can support arbitrarily sized ones, as is clear from the `windowL` NUT.

The third component, that is the balcony, is the result of running Experiment 5. In Figure 7.5, three balconies are part of the output, two of which are complete. This illustrates how much information is stored in relatively small patterns. Moreover, the stack brick pattern is also correctly maintained. Yet, XWFC is not perfect. The balcony on the left in Figure 7.5 is not completely enclosed in walls, neither is it complete itself. Technically, the output adheres to all constraints present in the input, there are no constraints that enforce each of the balcony's NUTs to only be present if all of them are present. This can be circumvented by formulating the input differently.

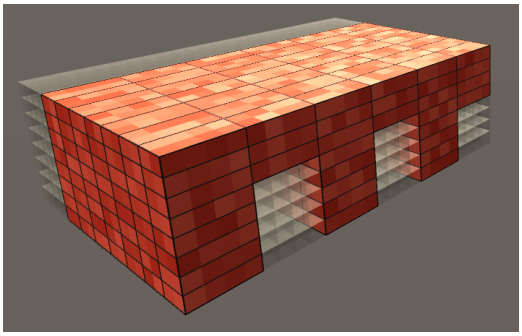
The next component is the roof, obtained from Experiment 6. With a more prominent, and disrupting, inclusion of the `void` NUT. While the pattern is simple, the output, as shown in Figure 7.6, is not directly compatible with the balcony component it should rest on top of. This important to note for the component assembly experiment (Experiment 8). Regardless, the simple input pattern is expanded as expected and yields satisfactory results. Note that the size of the grid is important here, the ratio of width and height should be such that the staircase pattern reaches tall and wide enough.

The final component is the chimney, where a simple case of component seeding is tested, as described in Experiment 7. The chimney should follow the roof contours, as illustrated in Figure 7.7. Additionally, that figure shows how any patterns that would result in partial NUT placement are eliminated, particularly the cells at the updated border of the grid and the running bond brick pattern is maintained.

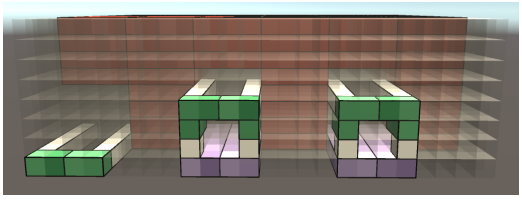
With all the components presented, the house structured object can be assembled in Experiment



(a) Angled front view



(b) Rear view



(c) Front view

Figure 7.5: Three views of the balcony component in a 24x8x18 grid

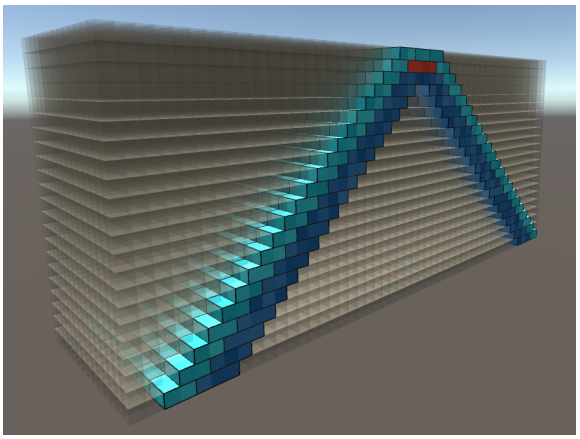
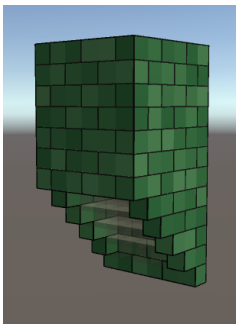
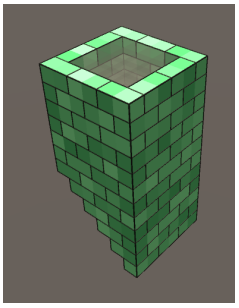


Figure 7.6: Roof component in a grid of size 50x22x8, obtained from Experiment 6.



(a) Angled view



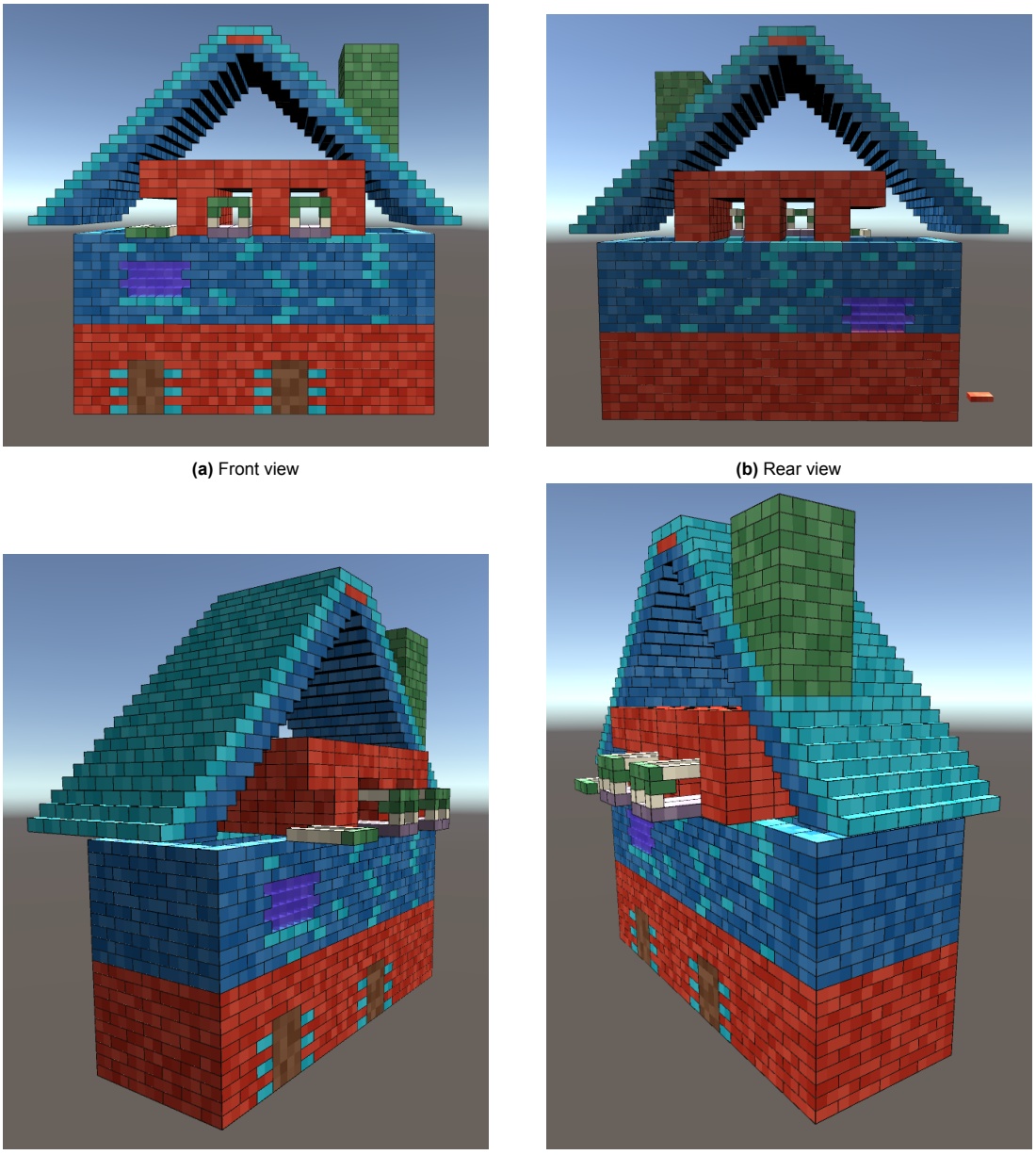
(b) Front top view

Figure 7.7: Chimney component in a grid of size 6x12x6, obtained from Experiment 7.

8. Figure 7.8 shows how the different components are assembled. While stacking the first, second and third components are fairly straightforward, the roof and chimney are more involved. Due to the presence of `void` NUTs (which are discarded during assembly) in the region of attachment between the roof and balcony, the roof is correctly translated after generating it with XWFC. This translation is also properly propagated to the chimney component. Together, they form one single structured object consisting of multiple components with vastly differing NUT sets and constraints while operating on 3D patterns. This thereby provides evidence in favor of Hypothesis 2.

Component	Initialization	Generation	Total per component
First floor	28.31	1.28	29.59
Second floor	69.58	12.11	81.69
Balcony	8.09	3.17	11.26
Roof	22.64	42.55	65.19
Chimney	0.09	0.01	0.10
TOTAL	**128.71**	**59.12**	**187.83**

Table 7.1: Performance of house generation. Time taken per component (in seconds).



(a) Front view

(b) Rear view

(c) Angled left view

(d) Angled right view

Figure 7.8: Multiple views of the assembled house

Performance

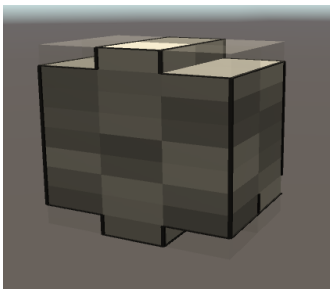
The time taken for generating each components is shown in Table 7.1. The initialization includes the pre-computation steps of XWFC. Especially incomplete pattern elimination was relatively expensive. However, once initialized, the generation process yielded acceptable performance, only severely influenced by the grid size, as was the case for the roof.

7.3. Overlapping Model: steam train assembly experiment

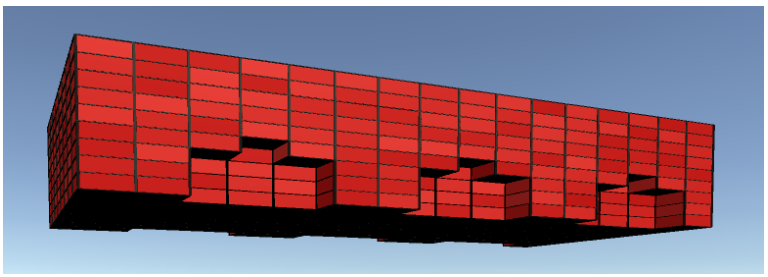
The house experiment (Experiment 8) requires relatively simple component assembly. A more complicated arrangement is described in Experiment 9, where a steam train is to be generated.

The results of running Experiment 9 are shown in Figures 7.9 and 7.10. The chassis and the wheels are solved first. The base component is dependent on the chassis and should rest on top of it. Since the chassis ends up less tall than initially, due to the layer of voids (see Figure 7.9c), the chassis has to be translated down. This in turn causes the base to overlap with the solved wheel components. As is shown in Figure 7.9b, the wheels correctly cause the base's grid's shape to be adjusted. The regions of attachment, including intersection of components, are thus correctly updated during run-time. Together with the other components of the train illustrated in Figure 7.9, the steam train is assembled composed of both bricks and plates. Figure 7.10 displays different views of the structured object. Specifically Figure 7.10b shows how the different components are fit together at the most challenging regions of the train.

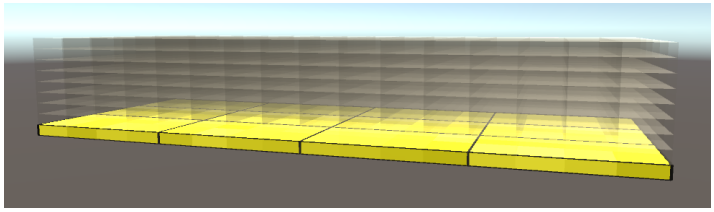
We can therefore state that our Hypotheses are confirmed. The house components show that XWFC is capable of generating diverse single-grid output similar to the input while preserving the NUTs' properties (Hypothesis 1). Together, the house example and the steam train example confirm Hypothesis 2. From the steam train example we can infer that that XWFC is indeed capable of solving grids sequentially, where the results of one component can affect another, whereas the house example shows that translating solved components functions as expected. Additionally, the components' affected may reach beyond those initially captured in the regions of attachment, which is processed accordingly. This is for instance the case for the train's wheels, chassis and base components' interaction; the base adapted to the wheels' contours. Furthermore, upon solv-



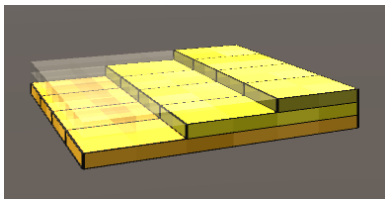
(a) Wheel x component, all 6 wheels are identical.



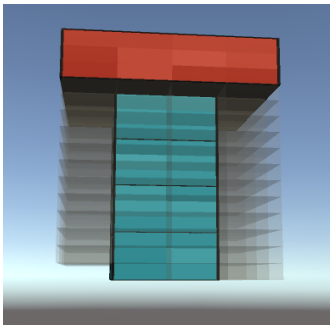
(b) Base component. Side bottom view.



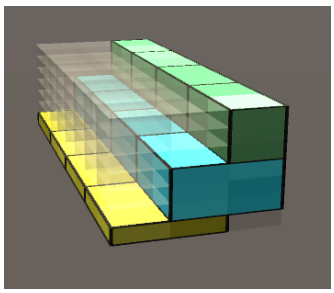
(c) Chassis, multiple layers of `void` NUTs on top.



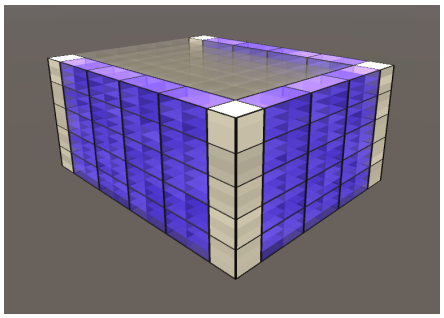
(d) Slope component.



(e) Chimney component. Note the cavity on the bottom left caused by seeding.

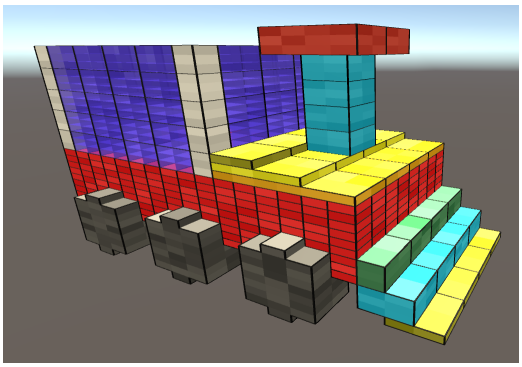


(f) Coupler component.

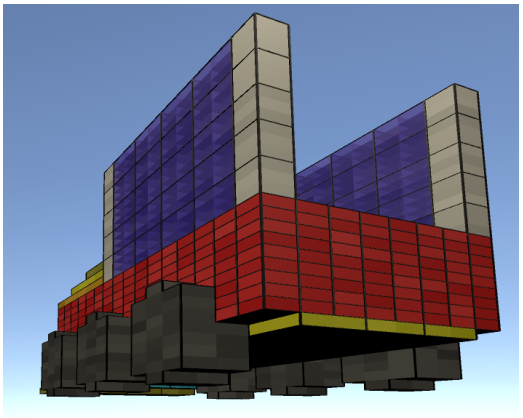


(g) Cab component.

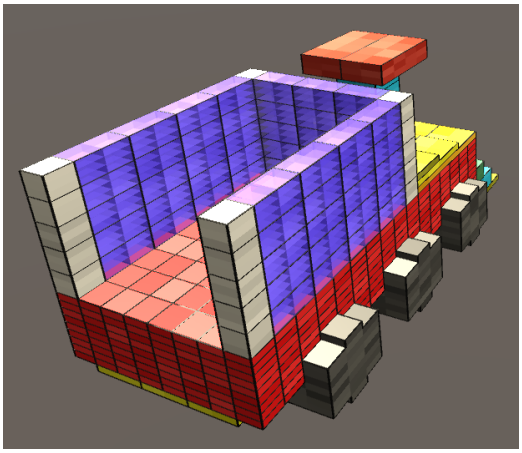
Figure 7.9: Steam train components.



(a) Front view.



(b) Low rear view. Note how the components are attached at the wheels



(c) Top rear view.

Figure 7.10: The assembled steam train.

ing a component, it is translated correctly and adapts to the previously solved components; the house's solved roof had to be translated post solving for example. And finally, our examples are fully in 3D with acceptable performance, thereby reinforcing our belief that XWFC is scalable to 3D, albeit highly dependent on the number of patterns and the grid size. Assembly is not expensive, as the total time taken for the steam train indicates.

Performance

Generation of the steam train component was fast, as shown in Table 7.2. The components neither required many patterns, nor were their grids large; these were factors identified in the house experiment results (see Section 7.2). Scaling height by three to accommodate both plates and bricks also did not have a considerable impact on performance in this experiment. Since the grid sizes and number of patterns per component were nigh trivial, we conclude that assembly of components also did not impose severe performance costs for generating the steam train.

Component	Initialization	Generation	Total per Component
Wheels (x 6)	0.36	0.06	0.42
Chassis	0.13	0.01	0.14
Base	0.03	0.36	0.39
Cab	0.69	0.15	0.84
Chimney	0.18	0.02	0.20
Coupler	0.06	0.01	0.07
TOTAL	**1.45**	**0.61**	**2.06**

Table 7.2: Performance of steam train generation. Time taken per component (in seconds).

7.4. Discussion

With the results in mind, we can formulate an answer to our main research question:

How can WFC generate structured objects consisting of 3D NUTs?

We address each of the individual research questions, which together formulate our answer to our research question.

1. *How can WFC support NUTs while preserving their shapes and sizes?*

First, The results from all experiments provide ample evidence for XWFC's ability to support NUTs while preserving their shapes and sizes in various scenarios (Research Question 1). The experiments in 7.1 show how XWFC is able to find solutions based on NUT adjacency constraints for non-trivially shaped NUTs (of the Tetris-inspired NUT set), while respecting the NUTs' properties, with the exception at the borders. This extends to NUTs of varying sizes as well. Our approach for performing tile atomization and constraint inference through void masking is thus effective for the Simple Tiled Model: XWFC is able to learn and represent the NUT adjacency constraints as a set of atom adjacency constraints. However, our results also suggest that we must reconsider how the NUTs can be preserved at the border of the grid. The pattern elimination for the Overlapping Model is an extension of the NUT elimination mechanism used for the Simple Tiled Model (see Section 3.7). We thus surmise that this imperfection is likely caused by an oversight in our implementation.

Furthermore, our approach appears to be effective for the Overlapping Model as well. The results of the individual components in Sections 7.2 and 7.3 show how XWFC is able to learn from atomized input. The synthesis of the atom and pattern domains proves to be an effective solution for preserving a NUT's shape and size with acceptable performance. Moreover, the results in Section 7.2 show that XWFC is able to match patterns from multiple inputs, thereby contributing to a vast array of 2x2x2 patterns. Our solution is thus able to create varied output from small individual patterns, while keeping all NUTs intact.

2. *How can WFC generate the different components of a structured object?*

Second, the resulting components in Sections 7.2 give sufficient grounds for claiming that XWFC is able to generate different components effectively (Research Question 2). By considering each component to be within an individual grid, with its own NUT set and adjacency constraints or pattern inputs, we argue that the individuality of each component is maintained. Thus, it allows for decomposing a structured object into individual components and the attachment tree is used effective. This solution extends to 3D with acceptable performance, thereby reaching a wider scope of applicable domains.

3. *How can the structured object's components be assembled while maintaining their geometric relations?*

Finally, our results confirm that assembling the components is performed adequately (Research Question 3); the assembled components resemble the structured object they compose. We there-

fore believe that our approach of decomposing a structured object into a set of components with their relative positions is effective. From these components and their positions, XWFC is able to express their geometric relations and determine the sequential solving order of components through an attachment tree (see Section 4.1). Upon solving a component, its effects are properly propagated to its dependent components. Thus, applying void masking for determining the required repositioning of components works as intended. The house and steam train examples show that a component's grid's shape dynamically adapts to the other components its AABB intersects with, as could be the result of component translation. For instance, the train's body follows the contours of the wheels and chassis, despite not initially being attached to the wheels. While the approach with the component tree and regions of attachment works, each component cannot depend on multiple components. This slightly limits the degree of expressiveness. Despite this, we can create structured object in ways that go beyond what WFC is capable of.

8

Conclusion and Future Work

In this chapter, we present our conclusions. We also highlight several topics we consider future work.

8.1. Conclusion

In this thesis, we have presented Expressive Wave Function Collapse (XWFC) as our approach to answer our main research question:

> *How can Wave Function Collapse generate structured objects consisting of 3D Non-Uniform Tiles?*.

XWFC addresses each of the research questions as follows:

1. *How can WFC support NUTs while preserving their shapes and sizes?*

In Chapter 3 we have shown how both the Simple Tiled Model and Overlapping Model can operate on NUTs with tile atomization at its core (Section 3.1). For the Simple Tiled Model we explain how this generalized approach utilizes void masking and sliding windows to formulate NUT adjacency constraints as atom adjacency constraints. Overlapping Model compatibility is achieved through sample grid atomization in conjunction with the NUT set and Tile Pattern Adjacency Masks. We show how the NUTs' properties enable this for the smallest non-trivial pattern size in 3D, being 2x2x2. Both of these models make use of an initialization step to eliminate any incomplete tiles and pattern, for which the inter-atom adjacency constraints are used (Section 3.7). Our qualitative results confirm the effectiveness of our approach and show that our solution operates well in both two and three dimensions (Chapter 7).

2. *How can WFC generate the different components of a structured object?*

The different components of a structured object can be generated through structured object decomposition (Section 4.1), where each components is considered its own XWFC problem with its NUT set and a set of samples or NUT adjacency constraints. A structured object is represented as a set of components, where the object's structure is captured in an attachment tree derived from the object's components' dependencies.

3. *How can the structured object components be assembled while maintaining their geometric relations?*

In Chapter 4 we have shown how the components constituting a structured object can be solved according to their dependency order, where the dependencies are expressed as geometric relations. Upon solving a component according to its associated NUT set and constraints with XWFC, its

results are used as input for its depending components. These are processed through component seeding, which enables a component to follow the contours of the solved component it depends on. We also explain how XWFC adapts to such changes, following a generalized tile and pattern elimination process. By re-evaluating the dependencies iteratively and translating the components' positions accordingly, we flexibly create a structured object, thereby confirming Hypothesis 2. Our results support this claim, as is illustrated by the generated house and steam train in Chapter 7, where the assembly of components induces negligible performance costs.

Our research described in this thesis offers the following contributions.

- The algorithm for XWFC's WFC on NUTs for both the Simple Tiled Model and Overlapping Model in three dimensions (Chapter 3).
- The algorithm for XWFC's structured object components decomposition, capturing the components' dependencies. (Section 4.1)
- The algorithm for XWFC's structured object assembly based on the decomposition. (Sections 4.2 upto and including 4.6)
- The synthesis of WFC on NUTs and structured object decomposition and assembly (Section 4.7).
- Open-source code for the visualization of XWFC in Unity written in C# [22].
- An open-source prototype of XWFC on single components for the Simple Tiled Model written in Python [23].

In conclusion, we have presented XWFC, a major extension of WFC capable of generating structured object consisting of 3D NUTs. It is able to i) support NUTs while preserving their shapes and sizes, ii) generate the different components of a structured object and iii) assemble the structured object's components while maintaining their geometric relations.

8.2. Future work

XWFC offers several possibilities for future work. WFC on NUTs could be improved upon with generalized valid NUT adjacency constraints configuration searching, more sophisticated conflict handling and NUT rotation. For structured object decomposition and assembly, additional geometric constraints, component tile set compatibility, and more robust component relations and assembly can be considered.

Generalized valid NUT configuration searching

For the Simple Tiled Model, expressing a NUT adjacency constraint as a set of atom adjacency constraints relies on formulating the different NUT configurations (see Section 3.3). While powerful, the current configuration searching method based on shifting NUT adjacency constraint fails in certain cases, such as NUTs with holes. Instead, a relatively simple and more generalized approach can be based on the observations described in Section 3.3. An important observation is that a configuration can only be valid if at least one pair of atoms exists such that the two atoms in the pair belong to a different NUT are adjacent in the same direction as the one in the NUT adjacency constraint. One can thus find the valid configurations based on that set of atom pairs. The NUTs can then be positioned based on the positions of those atoms, for which the atom adjacency constraints can be used. In order for the configuration to indeed be valid, the positioned NUTs must not overlap. This approach is more general and would eliminate the need for void masking.

Conflict handling

XWFC employs fairly simple conflict handling in the form of save points, with the assumption that backtracking would be too computationally expensive. Preservation of a NUT's shape and size can make it difficult to find a tiling of a grid and could result in conflicts relatively quickly. XWFC

could thus benefit from more sophisticated backtracking. A better alternative would be running local backtracking balanced against global restarts, as suggested by Karth and Smith [9].

NUT rotation

XWFC currently does not perform Non-Uniform Tile (NUT) rotations. Each rotated version of a NUT would have to be represented as an individual tile in the algorithm regardless in order to be able to distinguish between them. Rotations can thus currently be achieved only through adding the rotated versions of the NUT. However, it would add some convenience if XWFC were able to do this automatically.

Component constraints

An interesting extension of XWFC would be the introduction of more constraints between components. This is based on the observation that structured objects commonly allow for constraints other than component attachments. A few of these are listed below.

- *Symmetry*. Many structured object have symmetric properties. For example, an animal's left and right side are symmetric. The same can apply to cars or architecture. It also beneficially impacts performance, a fully symmetric object would require half the work.
- *Relative sizes*. The final sizes of components can deviate from their grids. Thus, their final size cannot be enforced solely through the size of that grid, making it difficult to maintain relative sizes. Size constraints can regulate this. For instance, the limbs of a human body could be considered separate components where bodily proportions must be respected. The sizes of the solved components must be according to a certain ratio.
- *Contour continuation*. Contour continuation is the idea of a component perfectly following the resulting shape of the component it depends on. Take a simple example such a light house consisting of a number of vertically stacked components. A lighthouse's walls are typically round. It could be desirable that the shapes of solved components constituting the lighthouse are consistent. However, solving components is not guaranteed to achieve that consistency. To ensure consistency between the components' final shapes, it could be useful to build according to the contours of a previous result.

Structured object component assembly

Currently, we only allow components to depend on at most one other component, due to the numerous challenges multi-dependency of components pose for dynamically sized and shaped components. If a component could depend on multiple components, this could potentially yield more accurate representations of the structured object in question. However, solving for those additional constraints is non-trivial and would require more research. Similarly, component size change is not factored in, even though this is very much a likely scenario. It is currently unclear as to how components depending on such a changed component should adapt to this change, as described in Section 4.4.

In a different context, attachment is currently expressed in terms extremes: either minimum or maximum area of attachment. Much could be gained from a more granular form of expressing attachment, for instance in terms percentages or otherwise. This would offer more control over components connections.

Glossary

NUT adjacency constraint Non-Uniform Tile adjacency constraint. iii, 2, 8–13, 15, 20, 26–29, 33, 35, 39, 45, 47, 48

NUT set Non-Uniform Tile set. i, iii, 2, 13–15, 20, 21, 25, 26, 28, 31, 34–37, 39, 41, 45, 47, 53–55, 57–61

AABB axis-aligned bounding box. 8, 9, 20–22, 37, 46

adjacency constraint . 34, 45

atom adjacency constraint . 9–14, 27, 39, 45, 47, 48

atom coordinate . 8, 14, 17, 18

atomized grid . 14, 15

CM Component Manager. 31

DFS Depth-First Search. 22, 23

GUI Graphical User Interface. iii, 31, 32

inter-atom adjacency constraint . 9, 10, 16–18, 30, 32, 47

NUT Non-Uniform Tile. i, iii, 1–3, 5–21, 24–41, 43, 45, 47–50, 53–55, 57–61

Overlapping Model . i, iii, iv, 2, 5–7, 14–19, 34, 35, 38–45, 47, 48, 55–61

pattern atom coordinate . 17–19

PCG Procedural Content Generation. i, 1

SIMD Single Instruction, Multiple Data. 26, 27

Simple Tiled Model . iii, iv, 5, 7, 14–20, 26, 33–35, 38, 45, 47, 48, 53, 54

structured object . i, iii, 1–3, 6, 20–22, 25, 31, 34, 40, 41, 45–49

tile atomization . iii, 7, 8, 13, 14

TPAM Tile Pattern Adjacency Mask. iii, 17, 19, 47

WFC Wave Function Collapse. i, iii, 1–9, 13–16, 19, 20, 25, 31, 34, 36, 38, 39, 45–48

XWFC Expressive Wave Function Collapse. i, iii, 2, 3, 7, 9, 11, 14–21, 25–27, 30–49

References

[1] Shaad Alaka and Rafael Bidarra. “Hierarchical Semantic Wave Function Collapse”. In: *Proceedings of the 18th International Conference on the Foundations of Digital Games*. Accessed: 2024-08-16. 2023, pp. 1–10.

[2] Bahar Bateni, Isaac Karth, and Adam 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*. Accessed: 2024-08-16. 2023, pp. 1–11.

[3] Michael Beukman et al. “Hierarchical WaveFunction Collapse”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 19. 1. 2023, pp. 23–33.

[4] Rafael Bidarra and Willem F Bronsvoort. “Semantic feature modelling”. In: *Computer-Aided Design* 32.3 (2000), pp. 201–225.

[5] Willem F Bronsvoort and Frederik W Jansen. “Feature modelling and conversion—key concepts to concurrent engineering”. In: *Computers in industry* 21.1 (1993), pp. 61–86.

[6] Freehold Games. *Caves of Qud*. `https://www.cavesofqud.com/`. Accessed: 2024-08-15. 2015.

[7] Maxim Gumin. *Wave Function Collapse*. `https://github.com/mxgmn/WaveFunctionCollapse`. Accessed: 2024-08-15. 2016.

[8] Tommy R Jensen and Bjarne Toft. *Graph coloring problems*. John Wiley & Sons, 2011.

[9] Isaac Karth and Adam 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.

[10] Isaac Karth and Adam M Smith. “Wavefunctioncollapse: Content generation via constraint solving and machine learning”. In: *IEEE Transactions on Games* 14.3 (2021), pp. 364–376.

[11] George Kelly and Hugh McCabe. “A survey of procedural techniques for city generation”. In: *The ITB Journal* 7.2 (2006), p. 5.

[12] Hwanhee Kim et al. “Automatic generation of game content using a graph-based wave function collapse algorithm”. In: *2019 IEEE conference on games (CoG)*. Accessed: 2024-08-16. IEEE. 2019, pp. 1–4.

[13] Thijmen Stefanus Leendert Langendam and Rafael 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.

[14] LEGO Group. *Comparing LEGO Bricks, Plates, and DUPLO Bricks*. Accessed: 2024-08-14. 2024. url: `https://www.lego.com/en-us/service/help/brick_facts/comparing-lego-bricks-plates-and-duplo-bricks-kA009000001dbm5CAA`.

[15] Marian42. *Wave Function Collapse - An Algorithm for Generating Random Structures*. `https://marian42.de/article/wfc/`. Accessed: 2024-08-16. 2020.

[16] Paul Merrell and Dinesh Manocha. “Model synthesis: A general procedural modeling algorithm”. In: *IEEE transactions on visualization and computer graphics* 17.6 (2010), pp. 715–728.

[17] Paul C Merrell. “Model Synthesis”. Available at: `https://paulmerrell.org/model-synthesis/`. Accessed: 2024-08-17. PhD thesis. University of North Carolina, 2009.

[18] Microsoft Docs. *Vector<T> Class (System.Numerics)*. https://learn.microsoft.com/en-us/dotnet/api/system.numerics.vector-1. Accessed: 2024-08-11. 2024.

[19] Adam Newgas. *Infinite Modifying in Blocks*. https://www.boristhebrave.com/2021/11/08/infinite-modifying-in-blocks/. Accessed: 2024-08-15. 2021.

[20] Adam Newgas. "Tessera: A practical system for extended WaveFunctionCollapse". In: *Proceedings of the 16th International Conference on the Foundations of Digital Games*. Accessed: 2024-07-23. 2021, pp. 1–7.

[21] Yuhe Nie et al. "Nested Wave Function Collapse Enables Large-Scale Content Generation". In: *IEEE Transactions on Games* (2024). Accessed: 2024-08-16.

[22] Rolf Piepenbrink. *XWFC Animator*. https://github.com/Rolf1501/XWFCAnimator. Accessed: 2024-07-23. 2024.

[23] Rolf Piepenbrink. *XWFC Python*. https://github.com/Rolf1501/XWFC-Python. Accessed: 2024-07-23. 2024.

[24] Rolf Piepenbrink and Rafael Bidarra. *How much Tetris can Wave Function Collapse put up with?* Accessed: 2024-08-22. Demo paper. 2024. url: https://graphics.tudelft.nl/Publications-new/2024/PB24/FDG24.Rolf.demo.Final.pdf.

[25] Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. "Enhancing wave function collapse with design-level constraints". In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*. Accessed: 2024-08-16. 2019, pp. 1–9.

[26] Emilio M Sanfilippo and Stefano Borgo. "What are features? An ontology-based review of the literature". In: *Computer-Aided Design* 80 (2016), pp. 9–18.

[27] Oskar Stalberg. *EPC2018 - Oskar Stalberg - Wave Function Collapse in Bad North*. Accessed: 2024-07-23. Youtube. June 11, 2018. url: https://www.youtube.com/watch?v=0bcZb-SsnrA.

[28] Oskar Stålberg. *Townscaper*. https://store.steampowered.com/app/1291340/Townscaper/. Accessed: 2024-08-15. 2021.

[29] Oskar Stålberg. *Tweet*. https://x.com/OskSta/status/784847588893814785. Accessed: 2024-08-15. 2016.

[30] Stålberg, Oskar. *EPC2018 - Oskar Stalberg - Wave Function Collapse in Bad North*. Accessed: 2024-08-17. 2018. url: https://youtu.be/0bcZb-SsnrA.

[31] Tetris Holding, LLC. *Tetrimino: The Tetris Game Piece*. https://tetris.com/about-us. Accessed: 2024-08-11. 2024.

[32] Unity Technologies. *Unity Game Engine*. https://unity.com/. Accessed: 2024-07-23. 2005.

[33] Winfried Van Holland and Willem F Bronsvoort. "Assembly features in modeling and planning". In: *Robotics and computer-integrated manufacturing* 16.4 (2000), pp. 277–294.

[34] PR Wilson and MJ Pratt. "A taxonomy of features for solid modeling". In: *Geometric modeling for CAD applications* (1988), pp. 125–136.

A

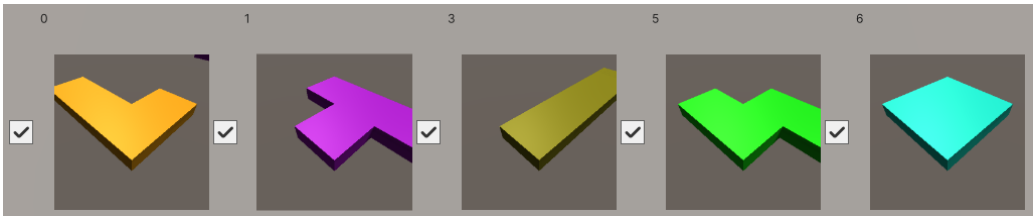
Experiment Parameters

This Appendix contains the parameters used for running the various experiments described in Chapter 6. Grid sizes are denoted as: width x height x depth.

A.1. Simple Tiled Model experiment parameters

This section contains the parameters used for the Simple Tiled Model experiments. The directions mentioned here are: North (0,0,1), East (1,0,0), South (0,0,-1), West (-1,0,0), Up (0,1,0), Down(0,-1,0).

- Experiment 1: no self-adjacent NUTs (see Figure A.1).
- Experiment 2: varying NUT sizes (see Figure A.2).



(a) The NUT set. From left to right: `L`, `T`, `I`, `S`, `O`.

(b) The adjacency constraints; they are identical in each direction.

Figure A.1: The NUT set and adjacency constraints for Experiment 1: no self-adjacent NUTs. Grid size 20x1x20.

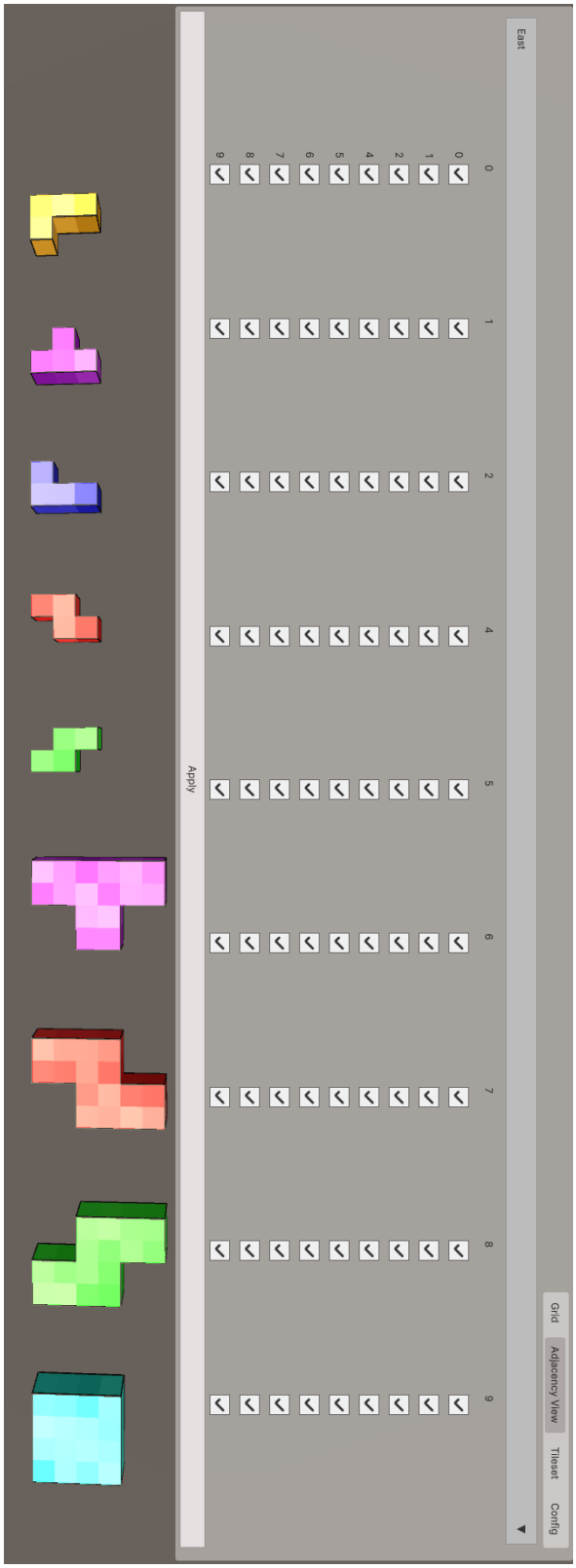


Figure A.2: NUT set and constraints for Experiment 2. The order of the NUTs at the bottom correspond to the identifiers in the adjacency matrix on the top. Names and identifiers from left to right: (`L`, 0), (`T`, 1), (`J`, 2), (`Z`, 4), (`S`, 5), (`T1`, 6), (`Z1`, 7), (`S1`, 8), (`O1`, 9). The adjacency constraints are identical in each direction. Grid size: 40x1x40.

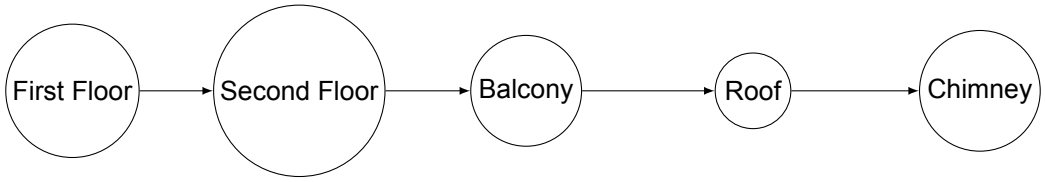


Figure A.3: Initial component tree for a house; the nodes are the components, the direction edges indicate a geometric relation, where the child depends on the parent.

A.2. Overlapping Model house experiment parameters

This section contains the house generation parameters used for the experiments described in Sections 6.3 and 6.4. See Figure 6.3 for a complete overview of the NUTs in the LEGO-inspired NUT set. Table A.1 show the parameters used for each component's grid extent. These are used to determine the components' relative positions, shown in Table A.2. Figure A.3 shows the tree representation of the components' geometric relations.

- Experiment 3: first floor (see Figure A.4).
- Experiment 4: second floor (see Figure A.5).
- Experiment 5: balcony (see Figure A.6).
- Experiment 6: roof (see Figure A.7).
- Experiment 7: chimney (see Figure A.8).

Component	Width (units)	Height (units)	Depth (units)
First floor	40	10	14
Second floor	40	10	14
Balcony	24	8	18
Roof	50	22	14
Chimney	6	12	6

Table A.1: Extents of the house components.

A.3. Overlapping Model steam train experiment parameters

This section contains the steam train parameter used for the experiments described in Section 6.5. See Figure 6.3 for a complete overview of the NUTs in the LEGO-inspired NUT set. Below is a list of the components referencing their associated parameters. Table A.3 show the parameters used for each component's grid extent. These are used to determine the components' relative positions, shown in Table A.4. Figure A.9 shows the tree representation of the components' geometric relations.

- Wheel: Figure A.10. The same parameters are used for all six wheels, only their initial positions differ.
- Chassis: Figure A.11.
- Body: Figure A.12.
- Slope and Chimney: Figure A.13.
- Cab: Figure A.14.
- Coupler: Figure A.15.

Component	**Position** (x, y, z)	**Description**
First floor	(0,0,0)	Root
Second floor	position(first floor) + (0, height(first floor), 0)	Relative to first floor in direction (0,1,0)
Balcony	position(second floor) + (0.5(width(second floor) - width(balcony)), height(second floor), -4)	Relative to second floor in direction (0,1,0), centered horizontally. Inset in z-axis to accommodate balcony.
Roof	position(balcony) + (-0.5(width(roof) - width(balcony)), height(balcony), 4)	Relative to balcony in direction (0,1,0), centered horizontally, restoring alignment with first and second floor in z-axis.
Chimney	position(roof) + (0.7width(roof), height(roof), 0)	Relative to roof in direction (0,1,0), on the right (x+) side of the roof's apex.

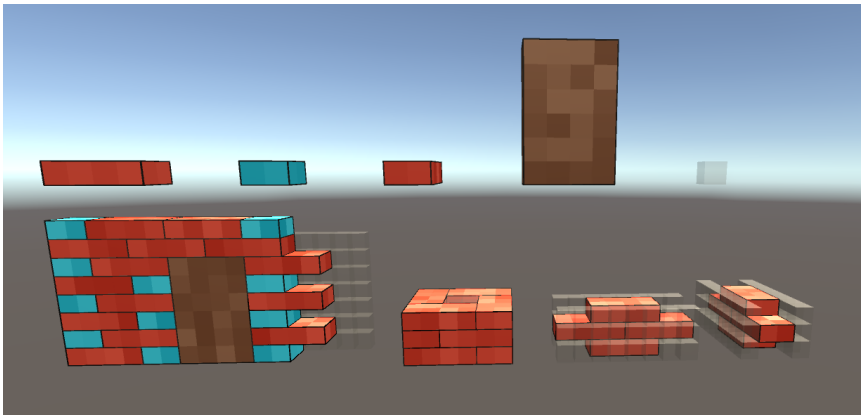
Table A.2: House component positions.

Component	**Width (units)**	**Height (units)**	**Depth (units)**
Wheel 1-6	3	9	2
Chassis	16	9	6
Body	16	9	8
Cab	10	15	8
Chimney	6	3	8
Coupler	3	7	8

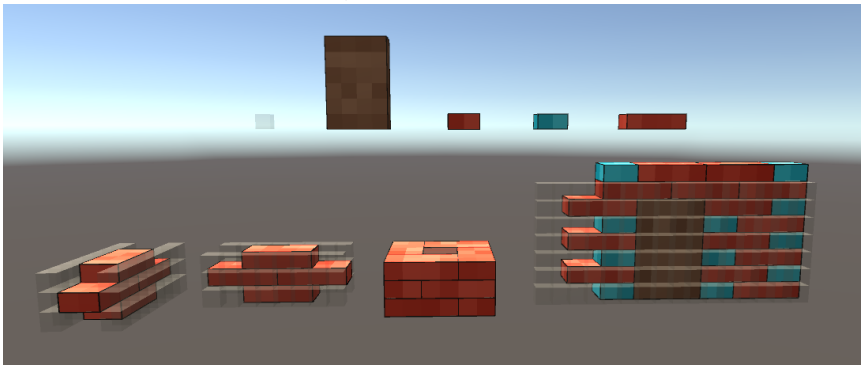
Table A.3: Train component extents.

Component	**Position** (x, y, z)	**Description**
Chassis	(0, 0, 0)	Root
Wheel n ($0 < n \leq 3$)	position(chassis) + (2 + 5(n-1), -0.5(height(Wheel n)), -2)	Position varies with n, relative to Chassis in direction (0,0,-1)
Wheel n, ($3 < n \leq 6$)	position(chassis) + (2 + 5(m-4),-0.5(height(Wheel n)), depth(chassis))	Position varies with n, relative to Chassis in direction (0,0,1)
Body	position(chassis) + (0, height(chassis), -1)	Relative to Chassis in direction (0,1,0)
Cab	position(body) + (0, height(body), 0)	Relative to Body in direction (0,1,0)
Chimney	position(body) + (width(body) - width(chimney), 0, 0)	Relative to Body in direction (0,1,0)
Coupler	position(chassis) + (width(chassis), 3 - height(coupler), 0)	Relative to Chassis in direction (1,0,0)

Table A.4: Component positions and their calculations. Refers to the width, height and depth columns in Table A.3

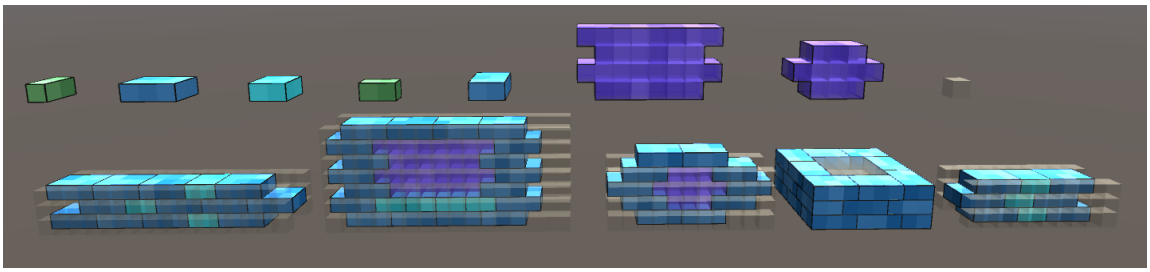


(a) The NUT set (top row) from left to right: `b112`, `b312`, `b212`, `b211`, `b213`, `windowL`, `windowS`, `bVoid`.

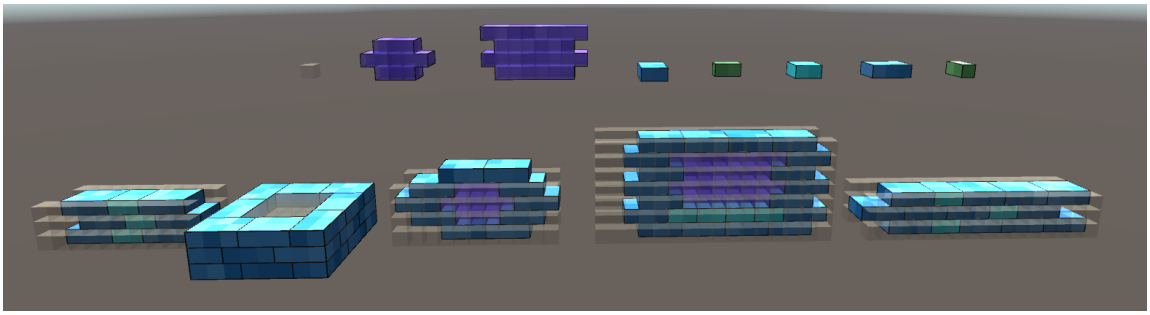


(b) Rear view of Figure A.4a.

Figure A.4: The NUT set (top row) and input patterns (bottom row) for Experiment 3: second floor. Grid size: 40x10x14.

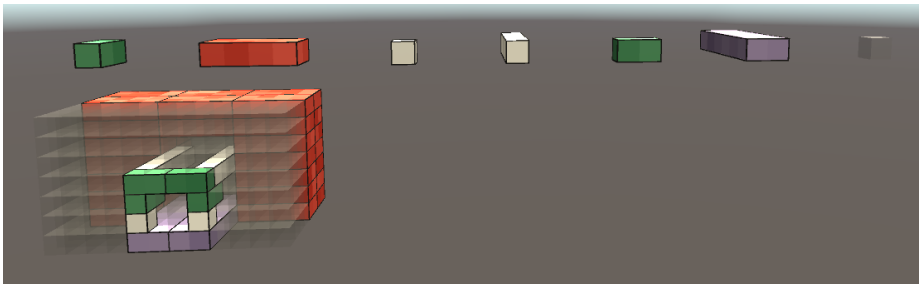


(a) The NUT set (top row) from left to right: `b112`, `b312`, `b212`, `b211`, `b213`, `windowL`, `windowS`, `bVoid`.



(b) Rear view of Figure A.5a.

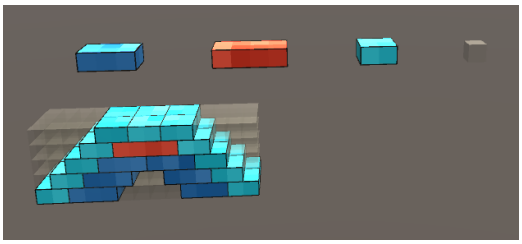
Figure A.5: The NUT set (top row) and input patterns (bottom row) for Experiment 4: second floor. Grid size: 40x10x14.



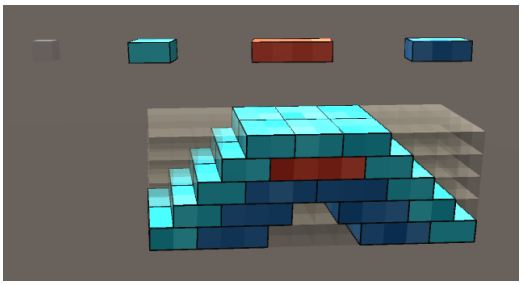
(a) The NUT set (top row) from left to right: `b112`, `b412`, `b111`, `b115`, `b211`, `b216`, `bVoid`.

(b) Rear view of Figure A.6a.

Figure A.6: The NUT set (top row) and input patterns (bottom row) for Experiment 5: balcony. Grid size: 24x18x8.

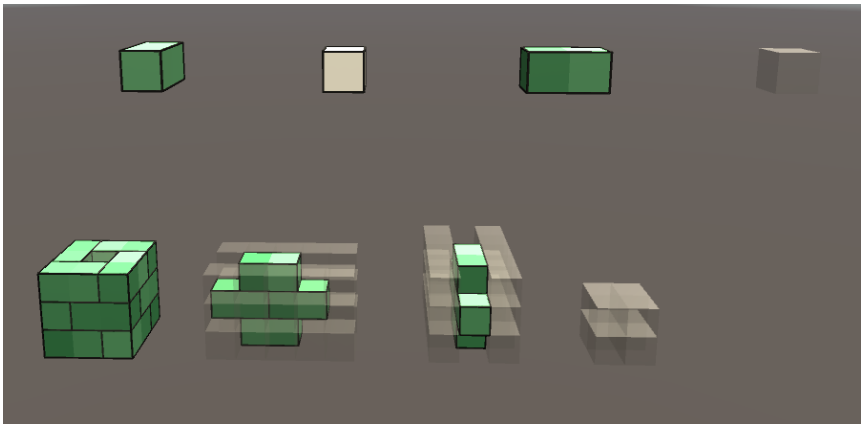


(a) The NUT set (top row) from left to right: `b312`, `b412`, `b212`, `bVoid`.

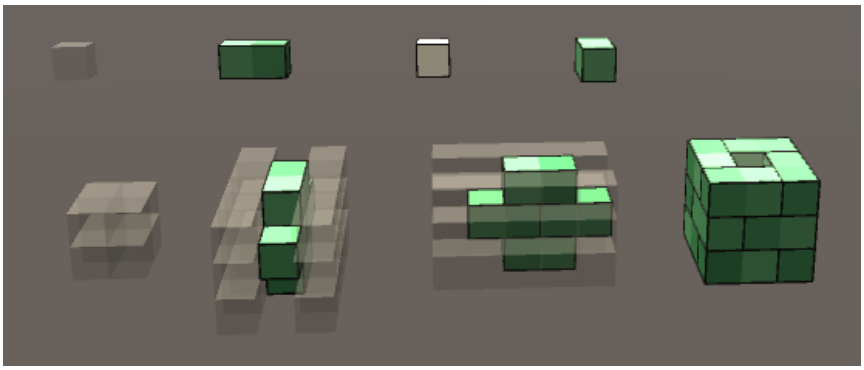


(b) Rear view of Figure A.7a.

Figure A.7: The NUT set (top row) and input patterns (bottom row) for Experiment 6: roof. Grid size: 50x22x14.



(a) The NUT set (top row) from left to right: `b112`, `111`, `b211`, `bVoid`.



(b) Rear view of Figure A.7a.

Figure A.8: The NUT set (top row) and input patterns (bottom row) for Experiment 7: chimney. Grid size: 6x12x6.

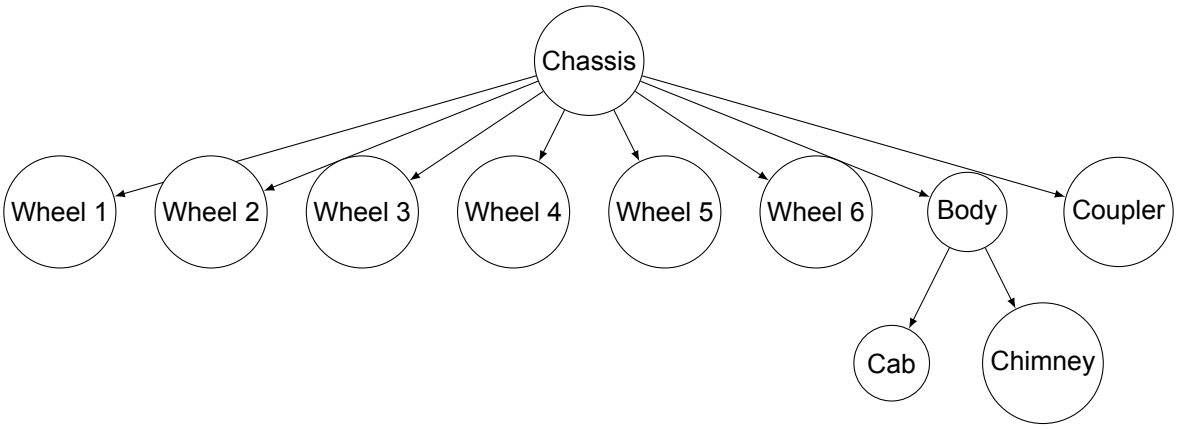
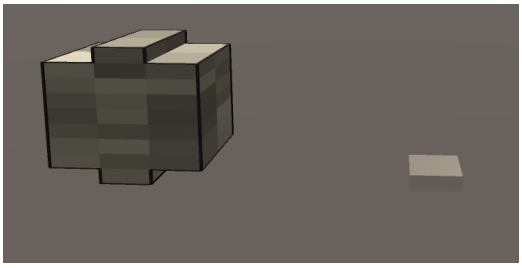
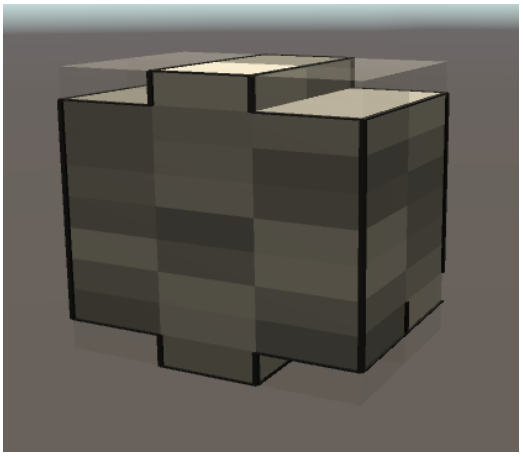


Figure A.9: Initial component tree for the steam train; the nodes are the components, the direction edges indicate a geometric relation, where the child depends on the parent.

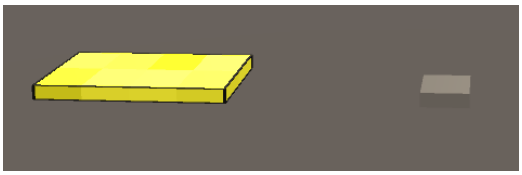


(a) The NUT set, from left to right: `wheel`, `pVoid`.

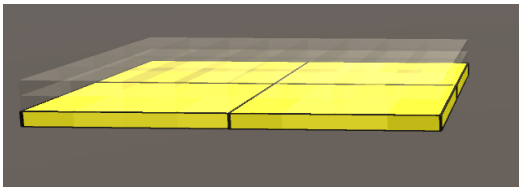


(b) The input patterns.

Figure A.10: The wheel NUT set (Figure A.10a) and input patterns (Figure A.10b). Grid size: 3x9x2.

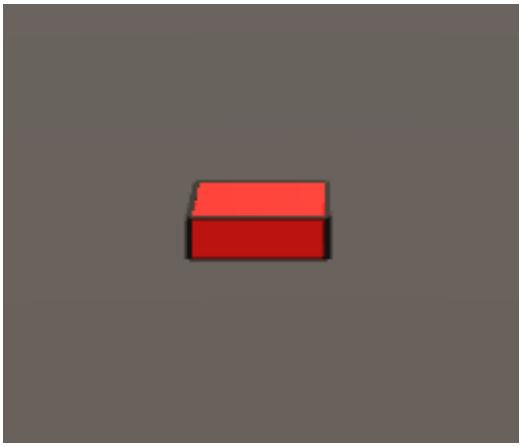


(a) The NUT set from left to right: `p412`, `pVoid`.

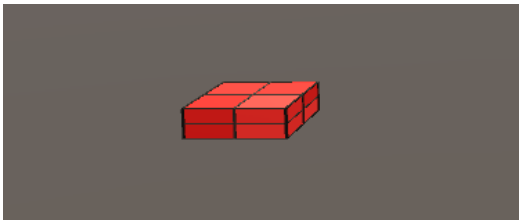


(b) The input patterns.

Figure A.11: The chassis NUT set (Figure A.11a) and input patterns (Figure A.11a). Grid size: 16x9x6



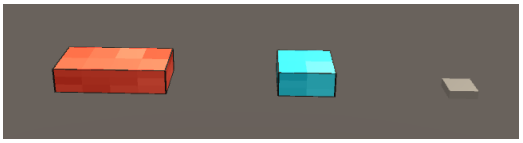
(a) The NUT set: `p111`.



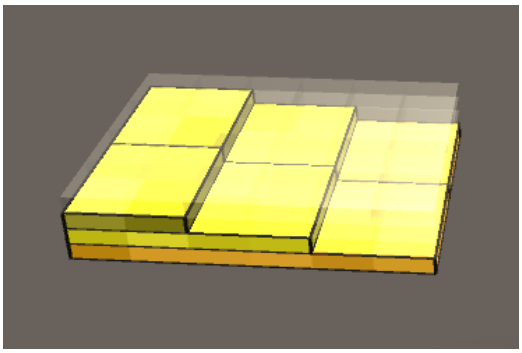
(b) The input patterns.

Figure A.12: The body NUT set (Figure A.12a) and input patterns (Figure A.12b). Grid size: 16x9x6

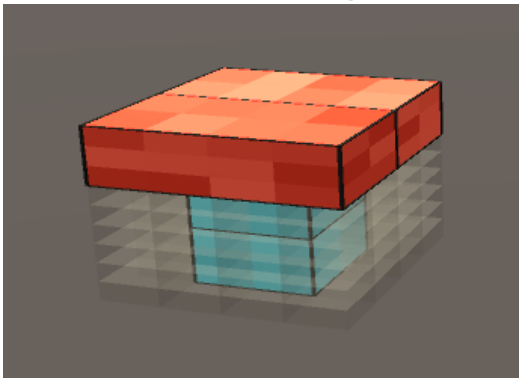
(a) The slope NUT set: `p111`.



(b) The chimney NUT set: `p111`.

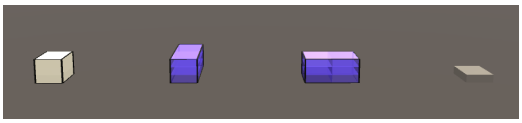


(c) The slope input patterns.

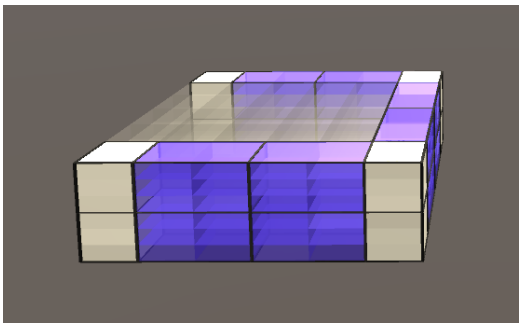


(d) The chimney input patterns.

Figure A.13: The slope (grid size: 6x3x8) and chimney (grid size: 4x15x4) NUT sets (Figures A.13a and A.13b) and input patterns (Figure A.13c and A.13d).

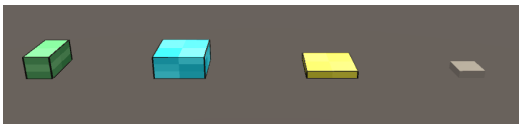


(a) The NUT set, from left to right: `b111`, `w112`, `w211`, `pVoid`.

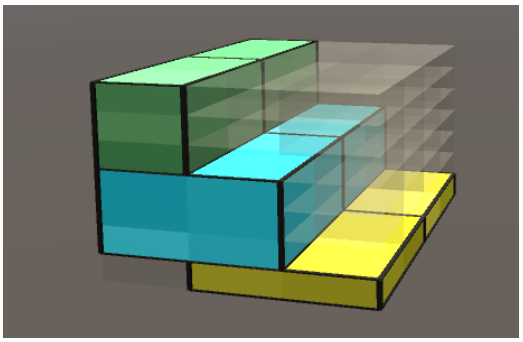


(b) The input patterns.

Figure A.14: The cab NUT set (Figure A.14a) and input patterns (Figure A.14b). Grid size: 10x15x8.



(a) The NUT set, from left to right: `b112`, `b212`, `p212`, `pVoid`.



(b) The input patterns.

Figure A.15: The coupler NUT set (Figure A.15a) and input patterns (Figure A.15b). Grid size: 3x7x8