MASTER THESIS

Jakub Mifek

Procedural Generation of Minecraft Villages utilizing the Wave Function Collapse Algorithm

Department of Software and Computer Science Education

Supervisor of the master thesis:	Mgr. Vojtěch Černý
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature, and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, 6th of January 2022 . . . .

Author’s signature

I would like to thank all my loved ones for their support throughout my life towards this point in time. Special thanks belong to my supervisor, Vojtěch Černý, for his guidance, support and time spent on our consultations. Last, but not least, I would like to thank all those who shared their knowledge and wisdom with me during my life.

Title: Procedural Generation of Minecraft Villages utilizing the Wave Function Collapse Algorithm

Author: Bc. Jakub Mifek

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Vojtěch Černý, Department of Software and Computer Science Education

Abstract: Maxim Gumin's Wave Function Collapse (WFC) algorithm is an example-driven image generation algorithm emerging from the craft of procedural content generation. The intended use of the algorithm is to generate new images in the style of given examples by ensuring local similarity. Our work aims to generalize the original work to make the algorithm applicable in other domains. Furthermore, we aim to apply it in a more difficult task of village generation in the 3D sandbox video game Minecraft. We will create a generic WFC library and a Minecraft mod, which will allow for structure generation using WFC. We hope that our WFC library will be beneficial to anyone exploring WFC and its applications in the Kotlin language and that our Minecraft showcase reveals some of the benefits and limits of the algorithm in complex problems.

Keywords: procedural content generation, Wave Function Collapse, village generation, Minecraft

Název práce: Procedurální generování vesnic ve hře Minecraft pomocí algoritmu Wave Function Collapse

Autor: Bc. Jakub Mifek

Katedra: Katedra softwaru a výuky informatiky

Vedoucí: Mgr. Vojtěch Černý, Katedra softwaru a výuky informatiky

Abstrakt: Wave Function Collapse algoritmus Maxima Gumina generuje nové obrázky na základě poskytnutých ilustrací. Zamýšleným použitím algoritmu je generování obrázků ve stylu ilustrací tak, aby byla dodržena lokální podobnost ve výstupním obrázku se vstupním. Naše práce má za cíl generalizování původní myšlenky algoritmu pro použití v dalších doménách. Dále máme za cíl použití algoritmu na problém generování vesnic ve 3D sandboxové hře Minecraft. Výstupem práce bude generická WFC knihovna a Minecraft mod, který umožní generování struktur pomocí WFC. Doufáme, že naše WFC knihovna pomůže komukoliv s použitím WFC v rámci jakéhokoliv projektu v jazyce Kotlin a že náš Minecraft mod pomůže nastínit výhody a limity použití WFC algoritmu na komplexních úlohách.

Klíčová slova: procedural content generation, Wave Function Collapse, generování vesnic, Minecraft

Contents

Introduction

In September 2016 Maxim Gumin came up with a new way of procedural content generation (PCG) which he called Wave Function Collapse (WFC) (1). WFC got its name from similarities it shares with the quantum theory where each particle is in a superposition until collapsed by observation to a concrete state (2). Gumin's WFC analyzes an input image, extracts patterns and treats every pixel of the output image as a superposition of all applicable patterns from the input image. The algorithm cycles through stages of observation and propagation. In the observation phase, the algorithm picks an output pixel and collapses it to a random applicable pattern. In the propagation stage, the algorithm bans non-applicable patterns in the output image based on the new information from the last observation. We describe the algorithm further in chapter 1.2.

The idea of the WFC algorithm can be generalized to higher dimensions. We can produce 3D structures or even generate animations using well-defined input patterns with an additional dimension representing time. However, there is almost no research done on what are the limits and drawbacks of using the WFC algorithm.

With this idea in mind, we decided to explore the usability, benefits, and drawbacks of the WFC algorithm in 3D structure generation. We chose Minecraft (3) as the target platform.

Minecraft is a wide-spread sandbox survival game with over 131 million monthly active users (4). There is no goal to the game other than survival, building and optionally defeating the End boss (5), which does not, however, end the game. The player is free to continue playing as long as they find interest in the game. Minecraft uniqueness comes from the fact that it was designed to not have any goal. The End boss was added to the game in later releases and the game prospered even before it. To keep the game's community engaged, Mojang is adding new types of blocks, auto-generated structures and many other features to their Minecraft releases (6).

The building of structures is a core mechanic of Minecraft. Minecraft allows players to harvest blocks of materials from the world and place them wherever they like. This, and Minecraft's support for mods, allows for easy integration with WFC since we can simply assign each block of different material an integer ID and work with those as if they were colour codes.

Goals

We would like to explore whether the usage of 3D voxel structure generation using WFC would yield benefits over the simple template matching that is currently used by Mojang (7). To summarize, our goals consist of:

- creation of a WFC library capable of Minecraft structure generation
- creation of a village generating mod for Minecraft that emphasizes the diversity of generated structures
- evaluation of our results and their implication on WFC's potential
- comparison with existing structure generation in Minecraft, namely Mojang's template matching

Structure

This document is divided into the following chapters. Chapter 1 will be dedicated to the original idea of WFC and its implementation as well as Mojang's implementation of village generation. In chapter 2, we will go through our requirements, analyze how WFC works, and how are we going to improve upon it as well as how are we going to tackle the village generation in Minecraft. In chapter 3 we are going to design and implement a generic WFC library in Kotlin. The following chapter 4 will contain details about middleware Village Generation Library (VGL). We will describe our implementation of the Minecraft mod in chapter 5. In chapter 6 we will discuss details about our command implementations and their results. The final chapter 7 will conclude our work and offer ideas for future work.

1 Background

1.1 Procedural Content Generation

Procedural Content Generation (PCG) is the algorithmic creation of game content with limited or indirect user input (8). The most common usage of PCG today is in video games. It is used to generate a wide variety of content, using many different techniques (9). By its definition, PCG can be combined with handcrafted material. This approach is called mixed-initiative (8). In WFC we take human crafted templates and generate new content using these templates.

1.2 Wave Function Collapse

In this section, we go into further details of the inner workings of the WFC algorithm.

1.2.1 Core idea

Wave Function Collapse (WFC) is an algorithm designed by Maxim Gumin in 2016 (1). The algorithm was created to generate bitmaps that would be locally similar to an input bitmap. An example can be seen in Figure 1-1. This similarity is achieved by:

1. Ensuring that every local window of pixels in the output bitmap can be found somewhere in the input bitmap.
2. The distribution of the different local windows in a sufficient number of output bitmaps is similar to the distribution of different local windows in the input bitmap.

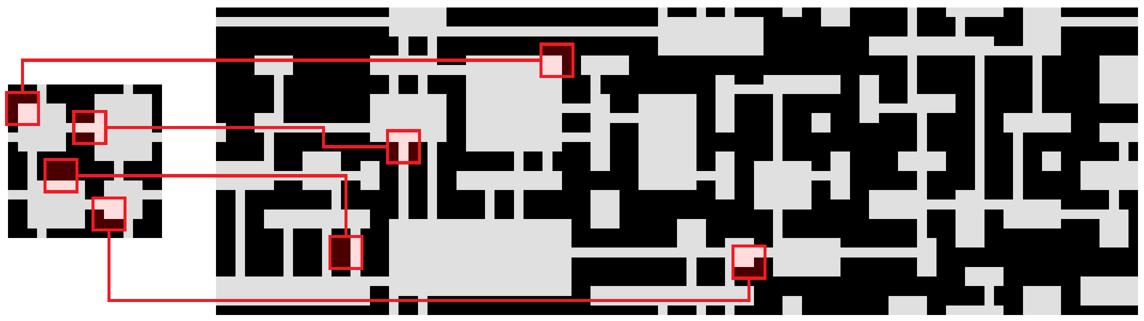
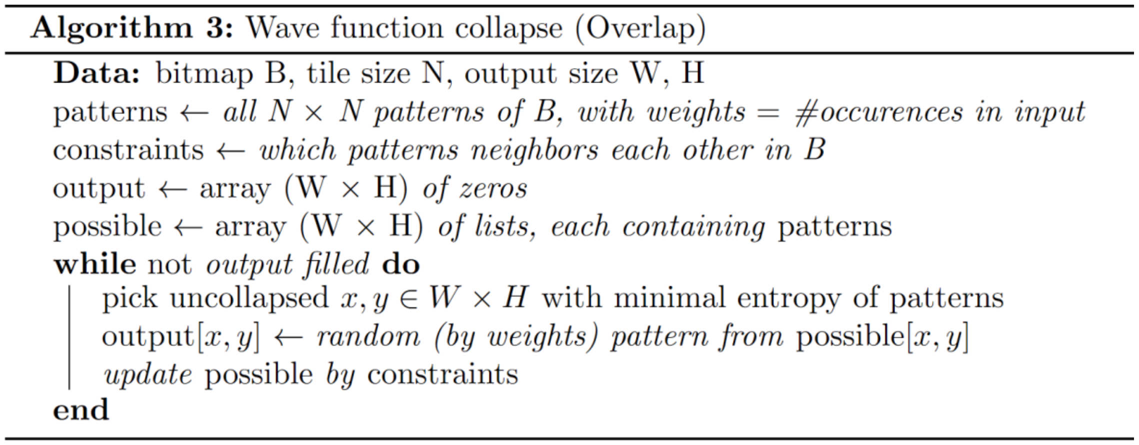


Figure 1-1: Example of WFC input (on the left) and output (on the right) created by Maxim Gumin (1) with highlighted local similarities

In the output bitmap, each pixel is initialized as a non-collapsed position (superposition) of all applicable patterns from the input bitmap. By cycling observation and propagation phases, the number of applicable superpositions is reduced until it is collapsed into a single pattern. Pseudocode for the algorithm can be seen in Figure 1-2.

During the observation phase, the algorithm selects a non-collapsed pixel using the lowest entropy heuristic (regarding the number of applicable patterns in the pixel's superposition). The selected pixel's superposition is collapsed to a single state of a random pattern using weighted probabilities according to the distribution of patterns in the input bitmap. This new observation is then stored for propagation.

In the propagation phase, the algorithm processes gained information from the observation phase. The algorithm bans patterns in pixels neighbouring the observed one so that the remaining neighbouring patterns match the one that was observed. The banning of patterns in pixels introduces new information into the system which is repeatedly propagated to the neighbouring pixels' neighbours. This process repeats until the system stabilizes. The stabilization is guaranteed, as we are only removing possible patterns of pixels, which are finite.



```
Algorithm 3: Wave function collapse (Overlap)
Data: bitmap B, tile size N, output size W, H
patterns ← all $N \times N$ patterns of B, with weights = #occurences in input
constraints ← which patterns neighbors each other in B
output ← array (W × H) of zeros
possible ← array (W × H) of lists, each containing patterns
while not output filled do
    pick uncollapsed $x, y \in W \times H$ with minimal entropy of patterns
    output[$x, y$] ← random (by weights) pattern from possible[$x, y$]
    update possible by constraints
end
```

Figure 1-2: Overlapping WFC pseudocode by Mgr. Vojtěch Černý (10)

This process ensures arc consistency (11) in the resulting bitmap, however, that is not enough to ensure a successful result. The algorithm can ban all patterns in a superposition due to a series of contradictory observations that can lead to failure. Some measures can be taken, such as backtracking or restarting the algorithm with a different seed, however, both measures merely decrease the probability of a failure and cannot guarantee a successful run for a finite number of tries. The probability of a failure is dependent on the size of the output bitmap as well as the number of patterns used and their compatibility. Issac Karth and Adam M. Smith tested several heuristics for WFC and the algorithm's robustness (12) and concluded that for the test examples of Maxim Gumin's repository for 48×48 output bitmaps, it is nearly impossible to yield a conflict state thanks to the used heuristics and robustness of the input patterns.

The core idea of WFC has been by several authors generalized to higher dimensions. Although, it has been mentioned by the authors that performance can become an issue (1). We have found only two repositories for 3D WFC generation:

- Maxim Gumin's C# implementation of voxel model (13)
- edwinRNDR's Kotlin implementation of voxel WFC (see Figure 1-3) (14)

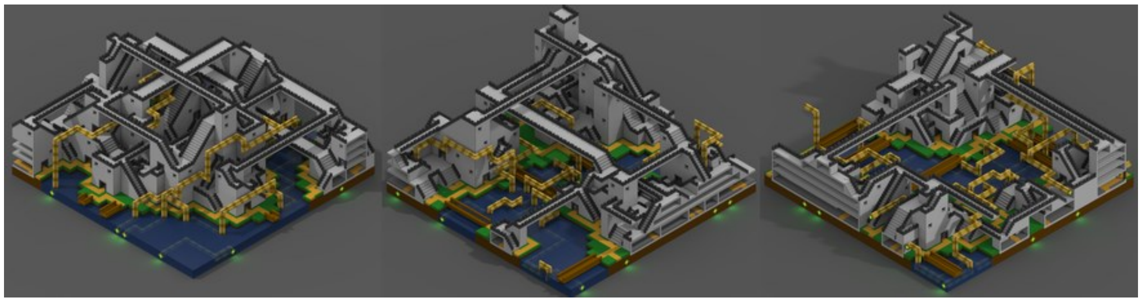


Figure 1-3: multidimensional WFC example generated using the Tiled model (1)

1.2.2 Models

There are two main approaches to pattern extraction and usage. Maxim Gumin calls them Tiled and Overlapping models. Inputs for the Tiled model are usually user-defined as it requires tileable patterns. The user defines which two patterns in 2D space are tileable and can therefore be put next to each other. Weights for these patterns are user-defined as well. For an example, please refer to Figure 1-4.

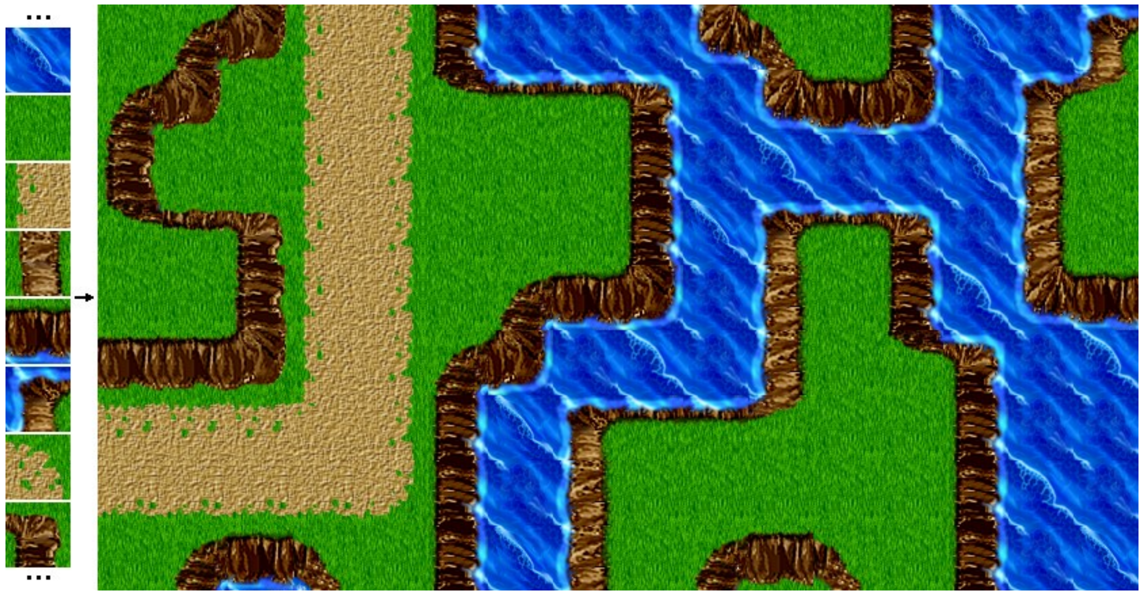


Figure 1-4: Maxim Gumin's example of the Tiled model generation (1)

Input for the Overlapping model can be for example an image. The model extracts all local $N \times N$ windows which are used as patterns. We count how many times any distinct pattern appears in the input and treat that count as the pattern's weight. Two overlapping patterns can neighbour each other from opposite sides if and only if their $(N-1) \times N$ overlap on these sides is equal. An example of such a generation can be seen in Figure 1-5.

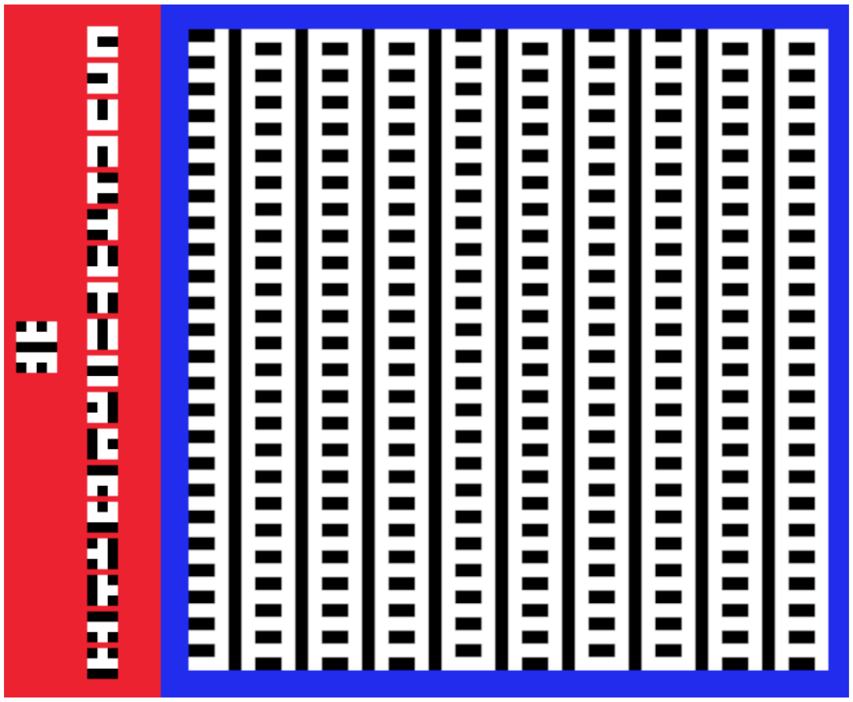


Figure 1-5: Input image (on the left), extracted patterns (second column) and resulting generated image (on the right) – this image was generated by our library using the fabric_small example from Maxim Gumin's repository (1).

There is also an option to extend the Overlapping model so that we have $N \times N$ patterns with $N \times M$ and $M \times N$ overlaps where $M \ll N$. However, this model is not much used throughout any of the projects we explored with exception of Maxim Gumin's repository (13) where patterns are again predefined by the developer.

1.3 Minecraft

Minecraft is an open-world sandbox survival game made up of many different blocks which players can pick up and place down remaking the world to their liking (3). To make the game more entertaining for players, new procedurally generated structures are added within new releases of the game. One of the more significant structures is a village (see Figure 1-6).

1.3.1 Minecraft Villages

Villages serve as an early boost for players' equipment as well as a home for villagers who can become an invaluable source of many items for the players via villager trading (15).

However, even though villages can play a major role in a player's progress through the game, Mojang's generation of villages is quite simple. A random point in the world is picked and if it meets predefined requirements, a village is spawned in that location by placing numerous houses using predefined blueprints (7) and connecting them with paths (15).

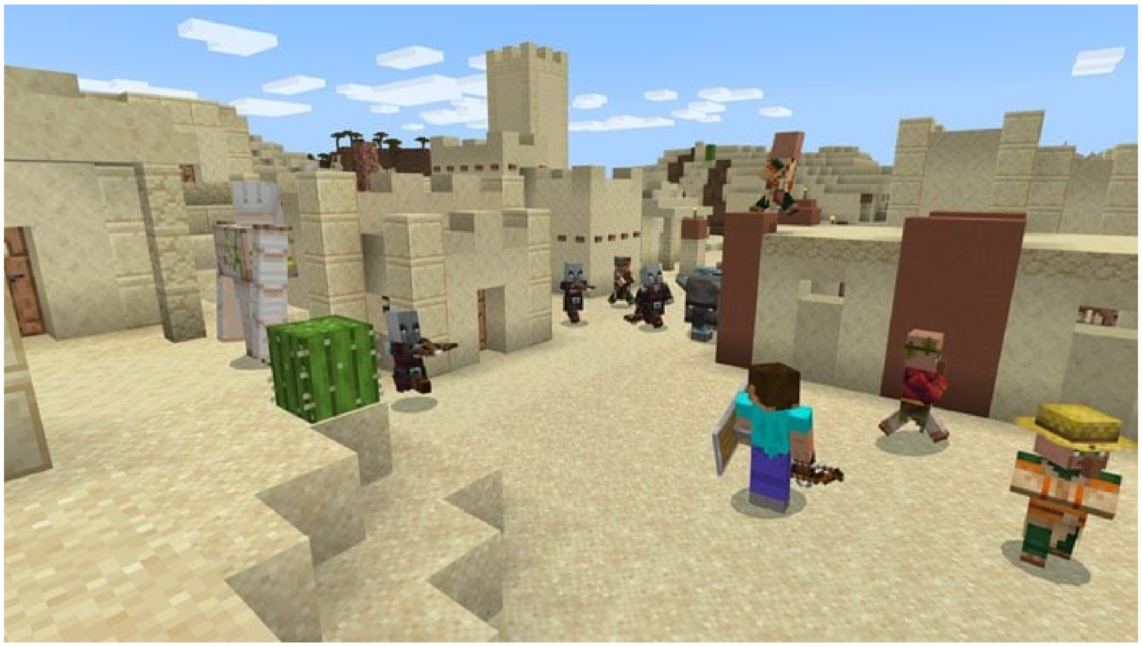


Figure 1-6: Minecraft village screenshot from the official website (3)

The template generation has some variations for different biomes that can be found within the Minecraft world, however, there are no modifications done to the blueprints upon generation. This process yields several types of houses in a village and ensures slight variations in house designs across biomes. However, if a player finds two villages in the same biome or another village of the same biome somewhere else in the world, houses of the same type are 1:1 copies.

Mods are trying to improve the village mechanics but most of the popular mods focus on villagers (16) or just simply add new blueprints (17). Moreover, competitions, such as Generative Design in Minecraft (GDMC) (18) are organized each year to improve village generation and make villages feel more realistic and interesting. Unfortunately, there is not much information on what algorithms competitors of GDMC use since they are free to use whatever algorithm they please. However, from the reports of the first year of the competition, we can see that the most used approach is generating structures using parametrized grammar (19).

1.4 Related work

As mentioned in section 1.1, there are already two existing repositories with 3D voxel model implementation. Unfortunately, neither of those serve our purpose well. Maxim Gumin's (13) is implemented in C# and would be hard to port to Java. On the other hand, edwinRNDR's Kotlin implementation (14) can be compiled to Java, but it is designed strictly for OPENRNDR models and extension of the repository for our needs would require a major rewrite.

For those reasons, we decided to design our own Kotlin based library that generalizes the algorithm. We wanted to create a library that would be easily portable not only to Minecraft but voxel models in general and other possible problem domains.

We found no projects that would attempt to generate villages using WFC.

2 Analysis

In this chapter, we analyse and summarise what we aim to achieve in this work.

2.1 WFC requirements

As mentioned in section 1.3, there are WFC libraries, however, none of them serves our purpose well. In this section, we describe what are our requirements for a WFC library.

We need the library to be portable to Minecraft, but we do not want to create a Minecraft-only version of the WFC library. Rather, we would like the library to work strictly with integers and accept mappings between integers and other objects. In other words, the library should be easy to integrate with bitmaps (mapping from/to colours), voxels, and Minecraft (block IDs) alike with minimum effort.

While the WFC model (see subsection 0) can change, the core WFC algorithm (see subsection 1.2.1) remains the same. For that reason, we would like to achieve an abstraction where models are strictly separated from the WFC algorithm and serve as data providers for the algorithm.

Furthermore, since the generation of higher-dimensional data is expected to run longer, we would like to have optional real-time feedback from the algorithm itself. Preferably in form of optional events to which we could subscribe (e.g.: failed, succeeded or superposition-collapsed events).

We have also considered the inclusion of backtracking into the algorithm, however, we do not deem it necessary at this point, since previous work promises great results on small outputs (12). Despite that, we keep the idea of backtracking in mind for simple future integration.

To summarize, the library should:

- be abstracted from the actual usage
- be easily extendable
- support real-time feedback

2.2 Minecraft mod requirements

First, we decided what version of Minecraft to develop our mod for. We settled on Minecraft 1.12.2 since that is the version GDMC (18) used at the time. We use Forge (20) as the Minecraft API framework because of its preferability by mod creators according to several sources (21).

We would like to be able to generate villages. For that purpose, we need to be able to generate houses and other structures. Furthermore, we would like to connect the houses with paths to aesthetically tie the village together.

We use Minecraft 1.12.2 definition of a house "… they (houses) are enclosed with a roof and wooden door." (22).

Because our main aim is to study WFC usability for village generation itself, we simplified other aspects of the problem. We do not require terrain selection and adjustment – instead, we use "flat world" without any terrain complexity. Furthermore, we do not require lighting of the village nor decoration of houses.

We would like the mod to provide the user with an interface for saving templates of custom structures and further generation of these structures. To be precise, we would like to be able to:

- save custom structures as template files
- rebuild a structure from a saved template file
- generate a structure based on saved template file using WFC
- generate a village based on several saved template files using WFC

2.3 Project structure

We divided the whole project into 3 subprojects:

1. WFC library
2. Village library
3. Minecraft mod

We made the WFC library standalone, because, as mentioned in section 2.1, one of our requirements is that the library is generic and usable for more than just Minecraft structure generation. Furthermore, we decided to write the library using an experimental version of a multiplatform Kotlin library which can be compiled to both Java Virtual Machine (JVM) binary and JavaScript (JS). As a result, our project does not contribute only to the Kotlin developer community but to Java and JavaScript developers as well.

We decided to split the Minecraft mod project described in section 2.2 into two subprojects – generic village library implemented in Kotlin, and Minecraft 1.12.2 Forge mod implemented in Java. That is because Forge and Minecraft are written in Java and rely on a Gradle version which does not support Kotlin yet. Another benefit is that it will require minimal work to port the mod to higher versions of Minecraft if desired.

3 WFC Library

This chapter explains our design decisions and implementation details of the standalone WFC library.

3.1 Project structure

The core of the library is the class `WfcAlgorithm` and its extended subclasses. This class contains the main logic of the WFC algorithm and its methods such as `observe` or `propagate`. This class is similar to the original implementation of Maxim Gumin (1).

The algorithm needs to understand the topology of the output data it is working with because it needs to propagate information between neighbouring superpositions. This information can change depending on the output space we work with. For that reason, we changed the original implementation so that all neighbouring operations are contained within the `Topology` class (and its subclasses) which is passed to the `WfcAlgorithm` class upon initialization.

Furthermore, we wanted to grant developers full control over the selection of observed pixels. We achieve this by introducing the `SelectionHeuristic` class. Its default subclass is the `LowestEntropyHeuristic` which is used in the original implementation (1).

`WfcAlgorithm` class can be either instantiated by the developer or built by a `Model` class. The purpose of `Model` classes is to extract patterns from input data as well as provide the developer with the possibility of adding additional constraints to the algorithm based on the input. Such constraints could be e.g. “Make a bottom row of the output bitmap contain only pixels found in the bottom row of the input bitmap.” or “The centre pixel of the output bitmap must be red.”. In the original implementation of Maxim Gumin (1), the `Model` class extended the `WfcAlgorithm` class. We follow the composition over inheritance rule. Therefore, in our implementation, the `Model` class stands alone, builds, and controls the `WfcAlgorithm` class, as well as translates data to and from the WFC algorithm.

For our project, we use overlapping models which allow simple pattern extraction from input data without any need for manual weights setting as described in subsection 0. A simplified class diagram for the `OverlappingCartesian2DModel` can be seen in Figure 3-1.

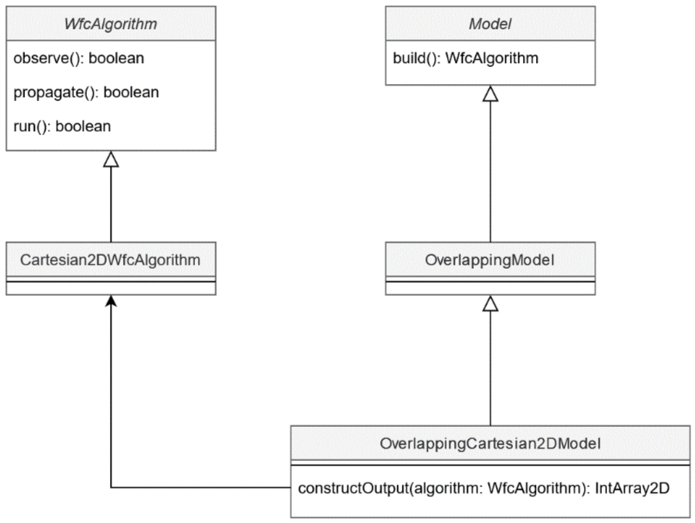


Figure 3-1: A simplified class diagram of the WFC library for Overlapping Cartesian 2D Model

The `Model` class can have many settings that the developer might want to control such as whether the input patterns can be rotated or flipped and in which axes. For that reason, we introduced `Options` data classes for each subclass of the `Model` class. `Options` data classes encapsulate all the respective model's settings with their default values.

In addition, we created a higher layer of abstraction which we call `Adapters`. `Adapters` are to separate developers from the inner workings of the WFC algorithm. They provide a simplified interface accepting raw data such as bitmaps or Minecraft templates and return generated results (refer to section 2.1 for further information). `Adapters` internally take care of data mapping, `Model`, and `WfcAlgorithm` instantiation. The developer can control the generation in a limited manner using "Adapter Options".

3.2 Implementation

In this section, we describe our implementation of the inner mechanics of the WFC library.

3.2.1 WFC Algorithms

As mentioned in section 3.1, the core class of the WFC library is the `WfcAlgorithm`. The class diagram for `WfcAlgorithm` can be seen in Figure 3-2. This class accepts topology, selection heuristic, weights, and a propagator as its constructor parameters. Topology encapsulates methods for the selection of neighbouring superpositions so that we do not have to burden the core algorithm with multidimensionality. The selection heuristic is by default set to the lowest entropy heuristic which selects a random superposition with the lowest number of remaining acceptable patterns, which is the only heuristic used in the original implementation. Weights contain pattern weights and ensure that generated output has approximately the same ratio of respective patterns as the input. Propagator holds information about what patterns support what patterns from which directions.

Upon initialization, a `Boolean` array of size $OutputPositions \times Patterns$ is created. The algorithm counts how many compatible neighbouring patterns (supports) each pattern in each output position has. If the number of supports for any pattern in any position drops to 0, the pattern is banned from the position.

When the `run` method (see Figure 3-2) is called, the algorithm initializes the selection heuristic, clears all changes made in previous runs, calls the `beforeStart` event, calls `warmup` method and then cycles until all superpositions are collapsed (or until it reaches the provided number of cycles) while calling the `step` method. The `step` method tries to observe a new pattern and if successful, propagates newly observed information to the system.

The `observe` method selects a non-collapsed position (superposition) using the selection heuristic. In the selected superposition a random non-banned pattern is selected using pattern weights. All other patterns are banned from the superposition.

In the `propagate` method we go through all the banned superposition-pattern pairs and decrease the number of supports from their neighbours resulting from the ban. If any of the support removals result in any number of supports dropping to 0, we ban the respective pattern in the neighbouring superposition. This ban is placed on a stack for further propagation.

The frequent reason for a failure, which we encountered while generating simple 2D bitmaps, was a possibility of selection of a pattern observed on an edge of the input bitmap in the middle of the output bitmap. Since the pattern was observed on an edge of the input image, it might not have any neighbouring patterns from the direction of the edge. Lack of neighbouring patterns causes an invalid state in the respective neighbour superpositions. This issue proved especially problematic when we did not treat the input bitmap as a periodic one (opposite edges in periodic inputs are treated as connected) and tried to generate larger bitmaps ($> 48 \times 48$). To tackle the problem, we created the `warmup` method. The method bans all patterns which lack neighbouring patterns in a direction for each superposition that has any neighbouring superposition in the same direction. By calling the method, we ensure that any "edge patterns" observed in the input bitmap are used only at the same edge in the output bitmap. This is one of our innovations that were not present in the original implementation.

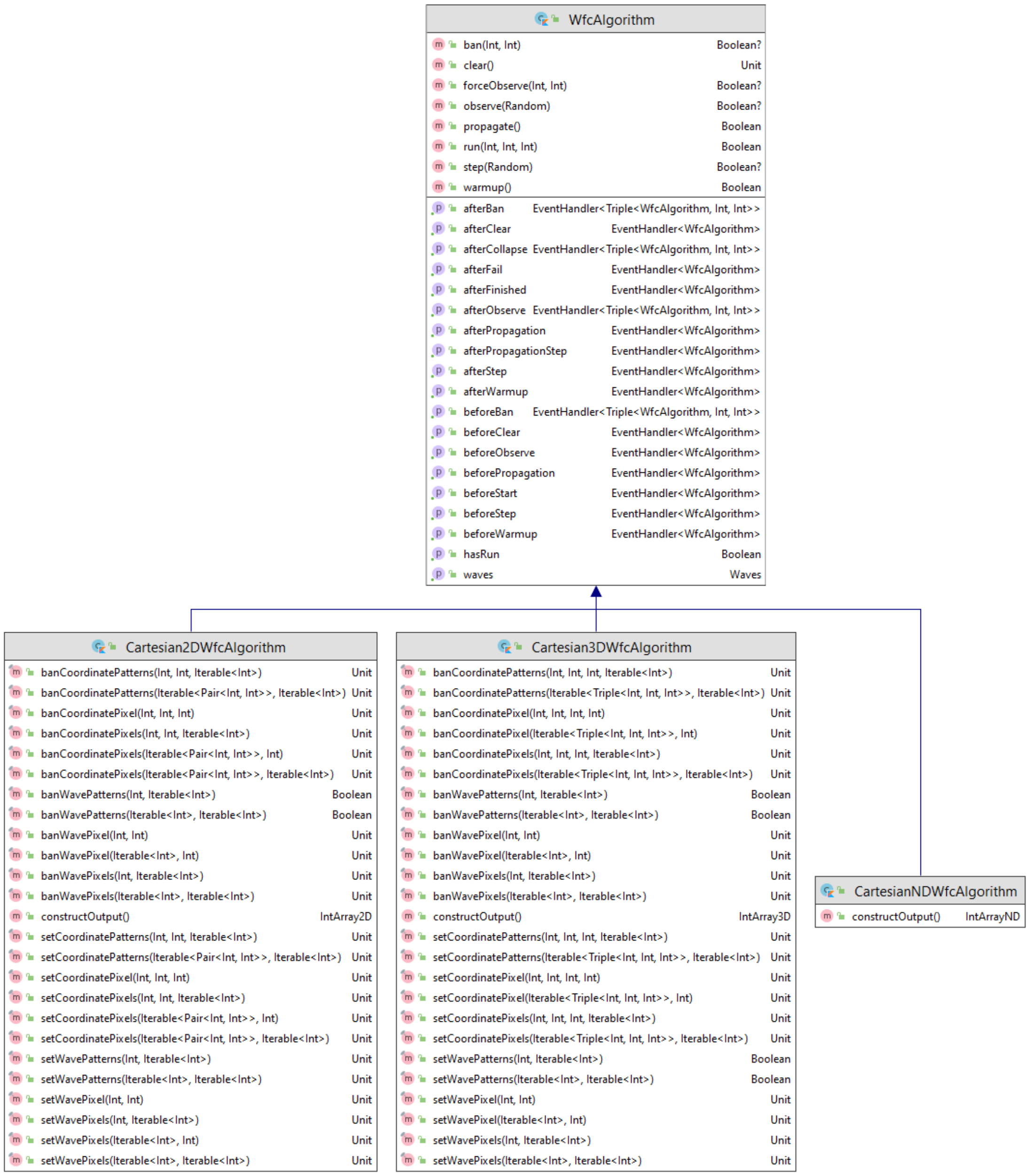

WfcAlgorithm
ban(Int, Int) Boolean?
clear() Unit
forceObserve(Int, Int) Boolean?
observe(Random) Boolean?
propagate() Boolean
run(Int, Int, Int) Boolean
step(Random) Boolean?
warmup() Boolean
afterBan EventHandler<Triple<WfcAlgorithm, Int, Int>>
afterClear EventHandler<WfcAlgorithm>
afterCollapse EventHandler<Triple<WfcAlgorithm, Int, Int>>
afterFail EventHandler<WfcAlgorithm>
afterFinished EventHandler<WfcAlgorithm>
afterObserve EventHandler<Triple<WfcAlgorithm, Int, Int>>
afterPropagation EventHandler<WfcAlgorithm>
afterPropagationStep EventHandler<WfcAlgorithm>
afterStep EventHandler<WfcAlgorithm>
afterWarmup EventHandler<WfcAlgorithm>
beforeBan EventHandler<Triple<WfcAlgorithm, Int, Int>>
beforeClear EventHandler<WfcAlgorithm>
beforeObserve EventHandler<WfcAlgorithm>
beforePropagation EventHandler<WfcAlgorithm>
beforeStart EventHandler<WfcAlgorithm>
beforeStep EventHandler<WfcAlgorithm>
beforeWarmup EventHandler<WfcAlgorithm>
hasRun Boolean
waves Waves
Cartesian2DWfcAlgorithm
banCoordinatePatterns(Int, Int, Iterable<Int>) Unit
banCoordinatePatterns(Iterable<Pair<Int, Int>>, Iterable<Int>) Unit
banCoordinatePixel(Int, Int, Int) Unit
banCoordinatePixels(Int, Int, Iterable<Int>) Unit
banCoordinatePixels(Iterable<Pair<Int, Int>>, Int) Unit
banCoordinatePixels(Iterable<Pair<Int, Int>>, Iterable<Int>) Unit
banWavePatterns(Int, Iterable<Int>) Boolean
banWavePatterns(Iterable<Int>, Iterable<Int>) Boolean
banWavePixel(Int, Int) Unit
banWavePixel(Iterable<Int>, Int) Unit
banWavePixels(Int, Iterable<Int>) Unit
banWavePixels(Iterable<Int>, Iterable<Int>) Unit
constructOutput() IntArray2D
setCoordinatePatterns(Int, Int, Iterable<Int>) Unit
setCoordinatePatterns(Iterable<Pair<Int, Int>>, Iterable<Int>) Unit
setCoordinatePixel(Int, Int, Int) Unit
setCoordinatePixels(Int, Int, Iterable<Int>) Unit
setCoordinatePixels(Iterable<Pair<Int, Int>>, Int) Unit
setCoordinatePixels(Iterable<Pair<Int, Int>>, Iterable<Int>) Unit
setWavePatterns(Int, Iterable<Int>) Unit
setWavePatterns(Iterable<Int>, Iterable<Int>) Unit
setWavePixel(Int, Int) Unit
setWavePixels(Int, Iterable<Int>) Unit
setWavePixels(Iterable<Int>, Int) Unit
setWavePixels(Iterable<Int>, Iterable<Int>) Unit
Cartesian3DWfcAlgorithm
banCoordinatePatterns(Int, Int, Int, Iterable<Int>) Unit
banCoordinatePatterns(Iterable<Triple<Int, Int, Int>>, Iterable<Int>) Unit
banCoordinatePixel(Int, Int, Int, Int) Unit
banCoordinatePixel(Iterable<Triple<Int, Int, Int>>, Int) Unit
banCoordinatePixels(Int, Int, Int, Iterable<Int>) Unit
banCoordinatePixels(Iterable<Triple<Int, Int, Int>>, Iterable<Int>) Unit
banWavePatterns(Int, Iterable<Int>) Boolean
banWavePatterns(Iterable<Int>, Iterable<Int>) Boolean
banWavePixel(Int, Int) Unit
banWavePixel(Iterable<Int>, Int) Unit
banWavePixels(Int, Iterable<Int>) Unit
banWavePixels(Iterable<Int>, Iterable<Int>) Unit
constructOutput() IntArray3D
setCoordinatePatterns(Int, Int, Int, Iterable<Int>) Unit
setCoordinatePatterns(Iterable<Triple<Int, Int, Int>>, Iterable<Int>) Unit
setCoordinatePixel(Int, Int, Int, Int) Unit
setCoordinatePixel(Iterable<Triple<Int, Int, Int>>, Int) Unit
setCoordinatePixels(Int, Int, Int, Iterable<Int>) Unit
setCoordinatePixels(Iterable<Triple<Int, Int, Int>>, Iterable<Int>) Unit
setWavePatterns(Int, Iterable<Int>) Boolean
setWavePatterns(Iterable<Int>, Iterable<Int>) Boolean
setWavePixel(Int, Int) Unit
setWavePixel(Iterable<Int>, Int) Unit
setWavePixels(Int, Iterable<Int>) Unit
setWavePixels(Iterable<Int>, Iterable<Int>) Unit
CartesianNDWfcAlgorithm
constructOutput() IntArrayND

Figure 3-2: WfcAlgorithm class diagram

3.2.2 Models

The purpose of models in our project is to provide an abstraction layer that lifts provided data to generic integer form for the `WfcAlgorithm` class. The `Model` class diagram can be seen in Figure 3-3. The `Model` class accepts user-defined settings and inputs. It creates and stores maps between user data and integer inputs for the algorithm. After the generation process of the algorithm is finished, the model class can translate the integer output to the user-expected output.

3.2.2.1 Overlapping Cartesian 2D Model

`OverlappingCartesian2DModel` serves as a factory class for `Cartesian2DWfcAlgorithm` class, described in subsection 3.2.1. Upon instantiation of the `OverlappingCartesian2DModel` class, the instance takes provided 2D integer array and extracts all patterns contained inside the array and computes their weights. The class also defines topology (`Cartesian2DTopology`) for the `WfcAlgorithm` class. The model class provides an interface for banning patterns in positions in the output array and a method for building the output 2D integer array from the `WfcAlgorithm` instance.

3.2.2.2 Overlapping Image Model

`OverlappingImageModel` class derives from `OverlappingCartesian2DModel` and expects provided 2D integer array to consist of encoded ARGB colors in `[0..255,0..255,0..255,0..255]` format. Its `constructAveragedOutput` method takes this information into account and splits the integer into the 4 bytes before computing the averaged output value (averaged colour of the pixel-based on patterns in its superposition).

3.2.2.3 Overlapping Cartesian 3D Model

`OverlappingCartesian3DModel` is a factory for the `Cartesian3DWfcAlgorithm` class, described in subsection 3.2.1. It differs from `OverlappingCartesian2DModel` in having extended options and interface for the extra dimension and in constructing `Cartesian3DTopology`.

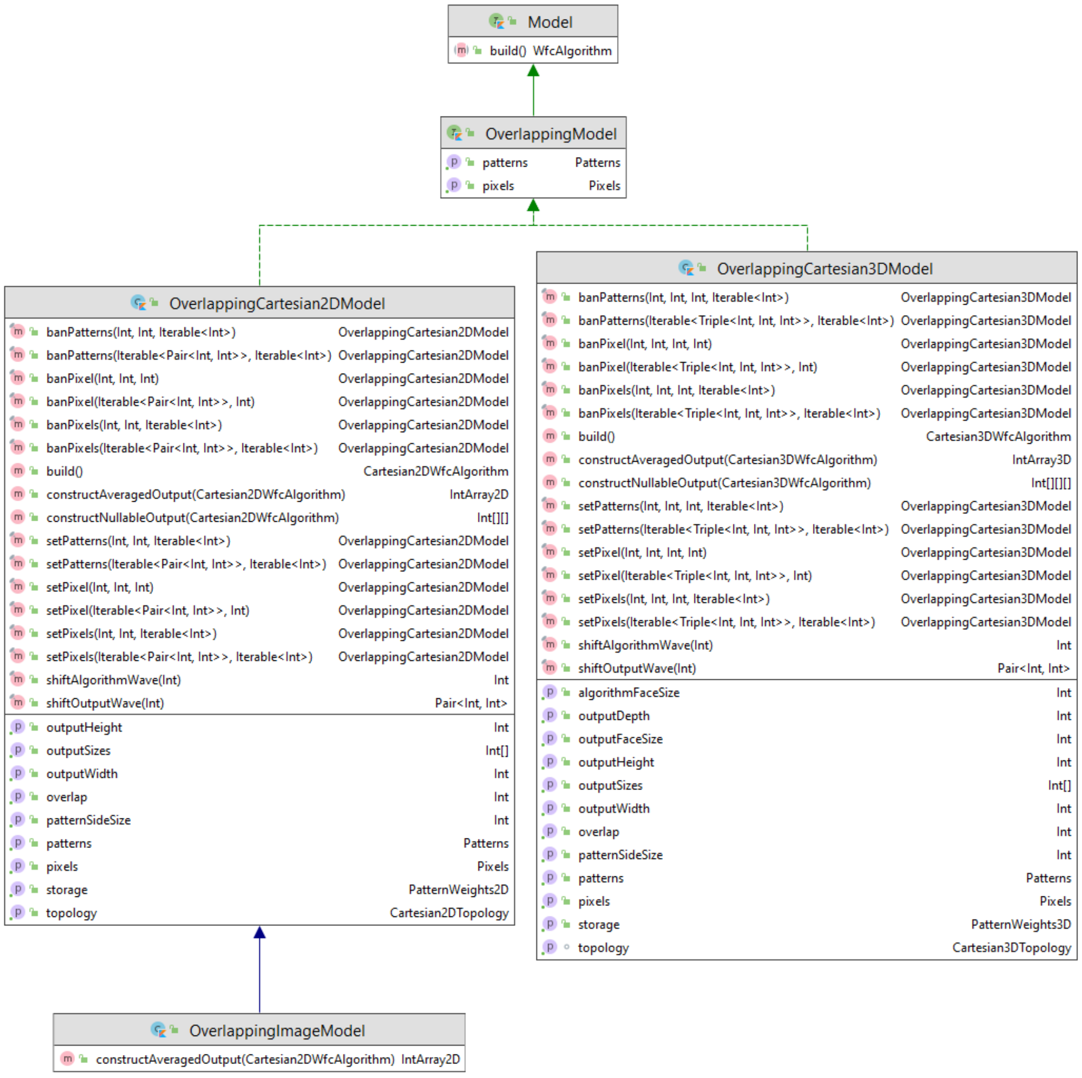

Model
build() WfcAlgorithm
OverlappingModel
patterns Patterns
pixels Pixels
OverlappingCartesian2DModel
banPatterns(Int, Int, Iterable<Int>) OverlappingCartesian2DModel
banPatterns(Iterable<Pair<Int, Int>>, Iterable<Int>) OverlappingCartesian2DModel
banPixel(Int, Int, Int) OverlappingCartesian2DModel
banPixel(Iterable<Pair<Int, Int>>, Int) OverlappingCartesian2DModel
banPixels(Int, Int, Iterable<Int>) OverlappingCartesian2DModel
banPixels(Iterable<Pair<Int, Int>>, Iterable<Int>) OverlappingCartesian2DModel
build() Cartesian2DWfcAlgorithm
constructAveragedOutput(Cartesian2DWfcAlgorithm) IntArray2D
constructNullableOutput(Cartesian2DWfcAlgorithm) Int[][]
setPatterns(Int, Int, Iterable<Int>) OverlappingCartesian2DModel
setPatterns(Iterable<Pair<Int, Int>>, Iterable<Int>) OverlappingCartesian2DModel
setPixel(Int, Int, Int) OverlappingCartesian2DModel
setPixel(Iterable<Pair<Int, Int>>, Int) OverlappingCartesian2DModel
setPixels(Int, Int, Iterable<Int>) OverlappingCartesian2DModel
setPixels(Iterable<Pair<Int, Int>>, Iterable<Int>) OverlappingCartesian2DModel
shiftAlgorithmWave(Int) Int
shiftOutputWave(Int) Pair<Int, Int>
outputHeight Int
outputSizes Int[]
outputWidth Int
overlap Int
patternSideSize Int
patterns Patterns
pixels Pixels
storage PatternWeights2D
topology Cartesian2DTopology
OverlappingCartesian3DModel
banPatterns(Int, Int, Int, Iterable<Int>) OverlappingCartesian3DModel
banPatterns(Iterable<Triple<Int, Int, Int>>, Iterable<Int>) OverlappingCartesian3DModel
banPixel(Int, Int, Int, Int) OverlappingCartesian3DModel
banPixel(Iterable<Triple<Int, Int, Int>>, Int) OverlappingCartesian3DModel
banPixels(Int, Int, Int, Iterable<Int>) OverlappingCartesian3DModel
banPixels(Iterable<Triple<Int, Int, Int>>, Iterable<Int>) OverlappingCartesian3DModel
build() Cartesian3DWfcAlgorithm
constructAveragedOutput(Cartesian3DWfcAlgorithm) IntArray3D
constructNullableOutput(Cartesian3DWfcAlgorithm) Int[][][]
setPatterns(Int, Int, Int, Iterable<Int>) OverlappingCartesian3DModel
setPatterns(Iterable<Triple<Int, Int, Int>>, Iterable<Int>) OverlappingCartesian3DModel
setPixel(Int, Int, Int, Int) OverlappingCartesian3DModel
setPixel(Iterable<Triple<Int, Int, Int>>, Int) OverlappingCartesian3DModel
setPixels(Int, Int, Int, Iterable<Int>) OverlappingCartesian3DModel
setPixels(Iterable<Triple<Int, Int, Int>>, Iterable<Int>) OverlappingCartesian3DModel
shiftAlgorithmWave(Int) Int
shiftOutputWave(Int) Pair<Int, Int>
algorithmFaceSize Int
outputDepth Int
outputFaceSize Int
outputHeight Int
outputSizes Int[]
outputWidth Int
overlap Int
patternSideSize Int
patterns Patterns
pixels Pixels
storage PatternWeights3D
topology Cartesian3DTopology
OverlappingImageModel
constructAveragedOutput(Cartesian2DWfcAlgorithm) IntArray2D

Figure 3-3: Model class diagram

3.2.3 Adapters

In case of the WFC algorithm failing, we often want to run the algorithm in a loop until we get a successful run or until we exceed some user-defined limit such as time or number of tries. For this purpose, we defined adapters as tied-to-use classes which expect input in a fixed format and serve as a mapping to the more generic format of `Models` and/or `Algorithms` as well as an enclosure that can be used as an easy-to-use façade for the whole generation process.

3.2.4 Supporting data structures

To ease manipulation with data and due to lack of pre-implemented options, we defined several data structures such as `IntArray2D`, `IntArray3D`, `IntArrayND`, `Dequeue`, `Quadruple` and `EventHandler`.

Most of these structures serve for data storage and manipulation. `EventHandler` is used in the WFC algorithm implementations to provide a dynamic interface for easier debugging and the possibility to animate outputs.

3.3 Results

In this section, we summarize our findings and try to evaluate the library from a functional and performative perspective.

3.3.1 Usability

We successfully created a generic library capable of replicating the original repository's example images as well as voxel data. For generated examples, refer to Figure 3-4. The library defines a clear distinction between the WFC algorithm and model and their purpose. We also provide additional interfaces and options that can modify the run of the algorithm. Each part of the library is extensible and easily replaceable if needed.

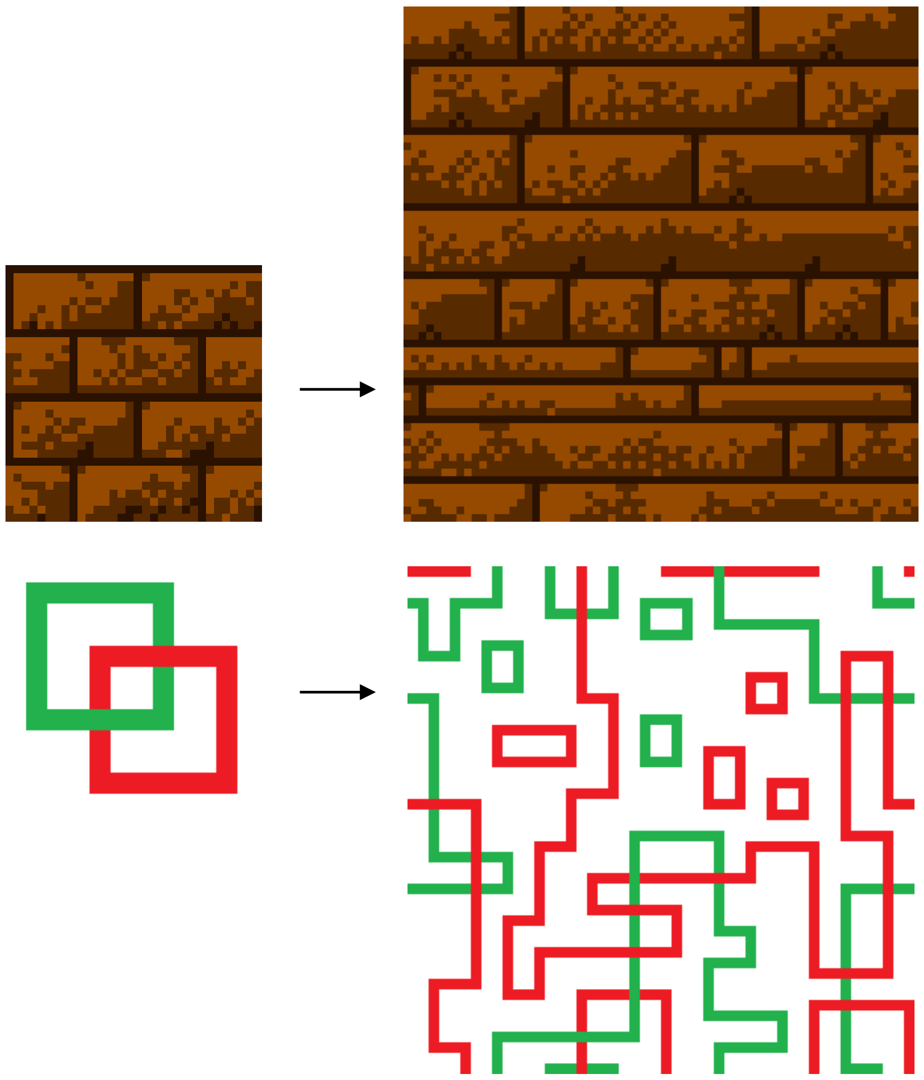


Figure 3-4: Sources (on the left) and generated results using our library (on the right). The top image sample is called Bricks and the bottom image is called Link. Both source images are from the original repository (1)

More examples with animations of the generation process can be found on YouTube (23).

3.3.2 Performance

We have run several tests comparing the performance of our library with the performance of the original implementation. We run the tests on two samples from the original repository – Flowers and Skylines. We choose these two samples because of their higher complexity in comparison with most of the other samples. We also modified the original repository to not output any images so that we could compare algorithm runtime only.

We run the algorithm for each sample 1000 times with the following settings:

- Flowers (48x48 pixels):
    - Maximum of 10 repeats if any fail occurs
    - Overlap 2
    - No rotations nor flips of input patterns
    - No grounding
    - No periodicity
    - Random seed
- Skylines (48x48 pixels):
    - Maximum of 10 repeats if any fail occurs
    - Overlap 2
    - No rotations nor flips of input patterns
    - Grounded, roofed (our library distinguishes, original does not)
    - No periodicity
    - Random seed

For the flowers example, our implementation needs 577% (rounded to 0 decimal places) of the original time per output (61ms for original implementation, 352ms for ours; both rounded). There were no complete generation failures in both libraries. There were some fails that both libraries overcame by repeating the generation process. The number of failed attempts to generate the output was in both libraries very similar (93 for original implementation, 101 for ours).

For the skylines example, our implementation needs 1000% (rounded to 0 decimal places) of the original time per output (89ms for original implementation, 890ms for ours; both rounded). There were no complete generation failures in both libraries. There were some fails that both libraries overcame by repeating the generation process. The number of failed attempts to generate the output was in both libraries in very low numbers (4 for the original implementation and 17 for ours).

We have profiled our code to tackle these performance issues and found a few issues that could be addressed in the future. We found that the algorithm spends the vast majority of the runtime within the propagation method. We use event handlers to propagate the information of banned patterns within superposition to heuristics. According to our findings, invocation of event handlers takes a nontrivial amount of time. Furthermore, there was some time spent on the creation of sequences within the topology class that is used during the traversal of the superposition's neighbourhood.

In addition, we believe that a nontrivial time overhead comes from the generalization of the library and separation of responsibilities of the algorithm into multiple classes which introduce additional function calls necessary during the propagation phase.

Even with the 1000% decrease in time efficiency for the generation process, we should be able to generate reasonably large outputs within seconds.

3.4 Conclusions

We have successfully replicated the functionality of the original repository with structural changes to the library. These changes came with a price of time needed to generate outputs. However, even with the deceleration of the process, the library yields results in a short time.

Our library introduces new features to the generation process e.g.:

- Possibility to add additional conditions on the result in form of banning given patterns in a given position before the algorithm starts
- Possibility to subscribe to events raised within the generation process
- Possibility to define a mapping between user's data format and integers which allows for reusing already existing models and algorithms – removing the need to implement a dedicated algorithm and/or model for the given problem
- Possibility to add own heuristics, topologies, and models separately – without the need to alter the rest of the code

4 Village Library

This chapter explains our design decisions and implementation details of the standalone Village library.

4.1 Project structure

Village library is a middleware between WFC library and Minecraft mod. Its purpose is to lift as much logic as possible from the Minecraft mod. Using this design, we can easily create mods for different Minecraft versions without the need for any changes. Another benefit is the ease of portability to servers.

The library uses `OverlappingCartesian3DModel`, described in subsection 3.2.2.3, and defines WFC adapters for Minecraft generation.

To achieve more pleasing results, we use the `afterCollapse` event from `WfcAlgorithm` class (refer to subsection 3.2.1) and offer to stream collapsed blocks to a given queue. This allows for a gradual placement of blocks as they become ready rather than spawning the whole structure at once.

4.2 Implementation

In this section, we describe our design choices, data structures, WFC extensions and commands that we define in VGL.

4.2.1 Design choices

The library aims to generate structures within Minecraft world. Minecraft is a 3D sandbox game where the whole world is made up of cubes (blocks) placed in cartesian space. There are many different blocks. Each block has its name, ID, and properties. However, ID, block properties and position in the world are everything we need for serialization and deserialization.

WFC replicates input patterns. Because of that, we need to be able to (de)serialize Minecraft structures. Since there is a finite number of block types and a finite number of properties and property values, we can assign each different block an integer value and represent any Minecraft structure as a 3D integer array that can be passed to `OverlappingCartesian3DModel`, described in subsection 3.2.2.3. Both integer array and block mapping can be easily (de)serialized.

4.2.2 Supporting data structures

The purpose of this library is to lift as much logic from the mod as possible. For that reason, we need a way to pass information between the two projects. We define several supporting data structures that help us with that task:

- The `Area` data class stores x, y and z coordinates as well as width, height and depth.
- The `Blocks` enum defines all known block IDs.
- The `Block` class contains information about the block's ID and its properties in form of a `HashMap`. It also provides an interface for (de)serialization (from)to a text format.
- The `PlacedBlock` class extends the `Block` class with information about the block's position coordinates.
- The `TemplateHolder` singleton class serves as a repository of structure templates and handles the (de)serialization (from)to files.
- The library also defines the `IBlockStream` interface, which we describe in subsection 4.2.2, that is later used in commands for passing already collapsed blocks back to the mod.

4.2.3 WFC

Since we can serialize blocks into integers, as mentioned in subsection 4.2.1, we can reuse `OverlappingCartesian3DModel` for replication of any Minecraft structure.

We define the `MinecraftWfcAdapter` singleton class which serves for replication of Minecraft templates. It accepts a 3D `Block` array and generation preferences and returns the generated 3D `Block` array or streams the array during the generation process, depending on the passed preferences.

In addition, we define `MinecraftVillageWfcAdapter` singleton class which uses the `Cartesian2DWfcAlgorithm` and produces a parcel plan of the given width and height. The adapter defines its patterns, which we created by hand. We also empirically hand-picked their weights. The algorithm adjusts the weights before the generation process begins so that we increase the probability of getting the desired number of houses in the given area. The adapter requires a HashMap of all templates that it is to use and their preferred dimensions and assigns them to generated parcels (see Figure 4-1).

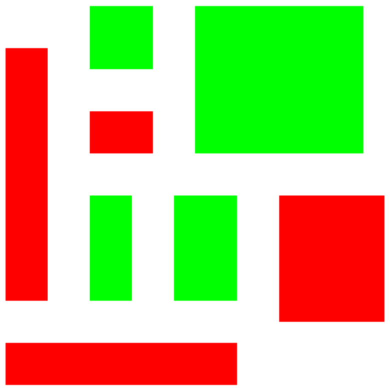


Figure 4-1: Example of generated parcels in a grid with 8 desired houses of two types (signified by different colours)

4.2.4 Commands

We define 5 commands that can be used by the mod – Save Template, Load Template, Replicate, Generate and Generate House. We did not implement Generate Village command in this library because of Kotlin's lack of options to run concurrent code. There is an option to use Java support for concurrency, however, by doing that we would tie the library strictly to JVM. Nevertheless, Generate Village command in Minecraft mod, which we will describe in subsection 5.2.2.3, reuses Generate House command, which takes care of most of the work.

4.2.4.1 Save template

Save Template command serves as an interface to `TemplateHolder` class. It accepts a 3D array of `Block` objects and an optional name of the template to be saved.

4.2.4.2 Load template

Load Template command accepts a name of a template to be loaded and a `StreamOptions` instance, which is used for streaming the template into the Minecraft mod.

4.2.4.3 Generate

Generate command tries to replicate any template it is given. It accepts a name of a template to replicate, an `Area` instance in which the output should be generated, and `MinecraftWfcAdapterOptions` specifying preferences of the generation process. The command does not do any preprocessing or postprocessing.

4.2.4.4 Replicate

The Replicate command tries to replicate any template it is given. The inputs remain the same as in Generate command, described in subsection 4.2.4.3. The command does not do any preprocessing or postprocessing. However, it does assume that the template it is given contains a whole structure.

All peripheral planes of a generated structure are set to resemble the peripheral planes of the input structure to ensure that the generated structure is contained within given boundaries. In other words, only the blocks that are on the left side of the input template will be used to generate the left side of the output template, and so on for all the sides.

The input template may contain many air blocks. Generally, having air blocks in the template is not an issue for WFC, because the algorithm tries to preserve the ratio of input patterns in the output structure. However, if the output space is limited to a relatively small size, the propagation phase in the WFC algorithm, described in subsection 1.2.1, can force the output structure to fill the whole space with air blocks and not generate any structure at all.

To counter this issue, we add a condition to the generation process. We assume that the structure to be replicated lies in the middle of the input template and therefore, we can fix the middle bottom block from the input template in the output space. Hence, if there is, for example, a floor block in the middle of the template, we will force the generator to use that floor block in the middle bottom of the output structure as well. This way, we eliminate the possibility of generating an empty output for the price of variation in the middle of a generated structure.

4.2.4.5 Generate House

Generate House command is the most complex of all our commands. It has the same inputs as Generate command, described in subsection 4.2.4.3. The command preprocesses the given template and postprocesses the generated output.

In the preprocessing phase, the command removes all doors from the template. That is because Door blocks have many properties which can lead to many different integer representations and force the Overlapping Model (see subsection 3.2.2) to create multiple patterns that differ only in small changes of door properties (e.g.: the orientation of the doors). This issue might impact the speed and success rate of the generation process as well as variations of the output.

Once the preprocessing is done, the generation process is identical to the Replicate command, described in subsection 4.2.4.4, with exception of streaming, which is forcibly disabled. We cannot stream the output of the algorithm since it has to go through postprocessing first.

In the postprocessing phase, we go through the generated structure and detect all rooms and passages between them. All rooms must be accessible. To achieve that, if we find an isolated room, we create a passage into one of its neighbouring rooms. We repeat the process until every room is accessible from all the others.

Once all rooms are connected and accessible, we continue with door placement. We detect all passages between rooms and place doors in them.

When the postprocessing is finished, we stream the results to the Minecraft mod.

5 Minecraft Village Mod

In this chapter, we explain our Minecraft mods' responsibilities and functions.

5.1 Project structure

Since most of the hard work is done in Village Generation Library (VGL), described in chapter 4, Minecraft Village Mod's primary goal is to provide an interpretation between VGL's commands and the Minecraft interface.

To speed up the generation process, we utilize a thread pool of size $\max(2, cores - 4)$. Each command creates its task, which is executed by a free thread from the thread pool. We use a thread-safe linked dequeue to avoid race conditions, which serves as a block stream, described in subsection 4.2.2.

Each block received for placement is deserialized from VGL's block definition into Minecraft's definition. This translation middleware was necessary because Minecraft does not provide Kotlin libraries, and therefore, communication with our pure Kotlin library was impossible. We made sure to use the same block IDs as Minecraft does for easier developer comprehension and software translation.

5.2 Implementation

This section describes the implementation details of the mod and its commands.

5.2.1 Entry point

Each Minecraft mod creates its entry point to the application, which is called upon initialization of the application and handles the registration of all resources needed by the mod.

Upon server start, our mod registers all its commands to the application instance and initializes a `BlockStream` instance used for placement in the world of Minecraft by the commands. The `BlockStream` instance is then ticked on every Minecraft world tick (24). The number of blocks placed per tick is customizable via the mod's configuration accessible in the game.

5.2.2 Commands

This subsection contains general information about available commands and their inner workings. For detailed information, please refer to the User Documentation or Programmer Documentation (see chapter A). Almost all the commands simply call their VGL counterparts, described in subsection 4.2.4, with no particular logic involved. There are 3 exceptions to this rule.

5.2.2.1 List Command

For easier work, we implemented additional command called List Command, which lists all available structure templates saved using Save Command, described in subsection 4.2.4.1, to the user's console.

5.2.2.2 Replicate Many Command

Replicate Many Command is designed to replicate a given structure multiple times in a 2D grid. It is a predecessor to Generate Village Command (described in subsection 5.2.2.3) and makes use of Replicate Command (see subsection 4.2.4.4), which is simply executed multiple times. There is no preprocessing or postprocessing involved.

5.2.2.3 Generate Village Command

Generate Village Command could not have been placed into VGL, because multiplatform Kotlin has no support for parallel execution of tasks and generation of the whole village one house after another would take an unnecessarily long time. However, the command uses Generate House Command, described in subsection 4.2.4.5, which takes care of most of the generation logic.

The command accepts a position in the Minecraft world and dimensions of the desired village. Then, the `MinecraftVillageAdapter` (see subsection 4.2.3) is used to divide the area into parcels and assign templates to them. Multiple layouts are generated and we pick the one with the most reasonable number of houses inside. We empirically set the optimal size of a house to be 18×18 blocks. Therefore, the optimal number of houses equals to $\max\left(1, \frac{width \times depth}{18^2}\right)$.

After the parcels are prepared, we submit a Generate House Command instance for each one to the executor thread pool (see section 5.1). Once all the houses are generated, we postprocess the generated result and add paths and grass to it for aesthetic effect. Paths are placed using the A* algorithm between nearby doors. The grass is placed in a random manner outside of houses. An example of the finished village can be seen in Figure 5-1.

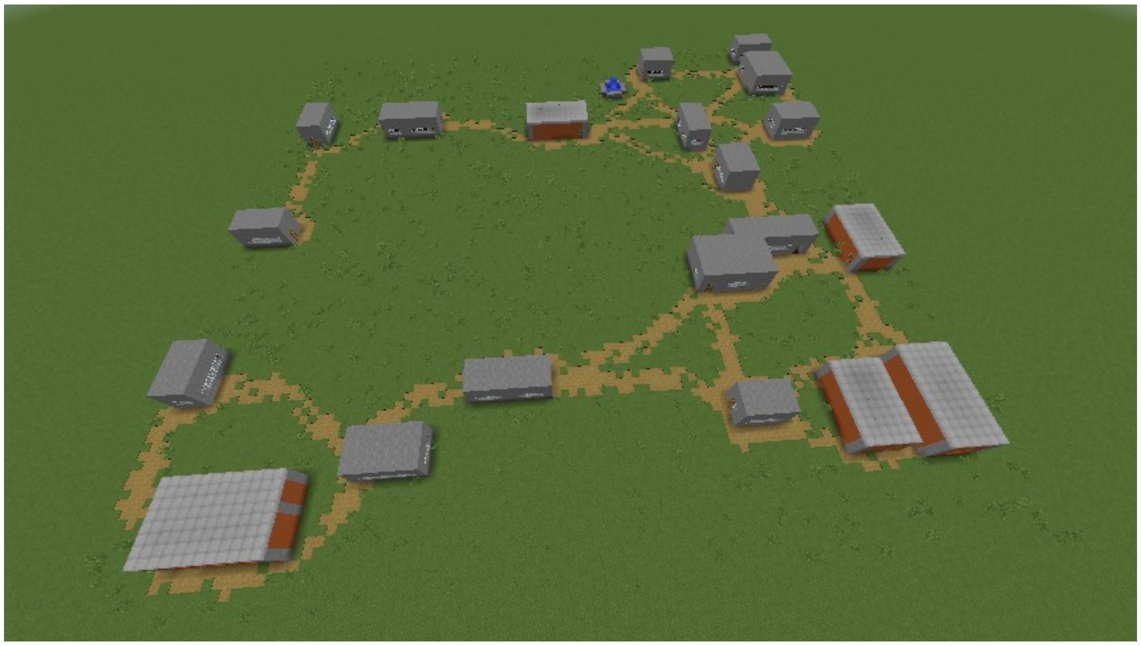


Figure 5-1: Generated Minecraft Village

6 Discussion

In this chapter, we describe difficulties that we faced during the process and how we tackled them.

6.1 Development process

In this section, we will go through the steps we took and why we took them. We will describe what experiments we did and how they turned out.

6.1.1 Project structure

First, we experimented with the setup of all three libraries. We wanted to use Kotlin for the majority of the implementation for its simplicity and ability to compile to multiple languages. We also wanted to use Minecraft Forge for Minecraft version 1.12.2 which was used at the time by GDMC (18). However, Minecraft Forge 1.12.2 uses Java version 8 which does not have support for Kotlin. We overcame this issue by packing necessary compiled functions and classes from Kotlin into our JVM package of the VGL library. Further information on the VGL library can be found in chapter 4.

6.1.2 First generation

When we confirmed that this project structure is compilable, we continued with the development of the WFC Library, described in chapter 3. Once that project was finished, we switched our focus back to the VGL and Minecraft mod projects. In the beginning, we experimented with the Save command and Generate command (see subsection 4.2.4). We soon realized that the simple application of WFC will not meet our needs.

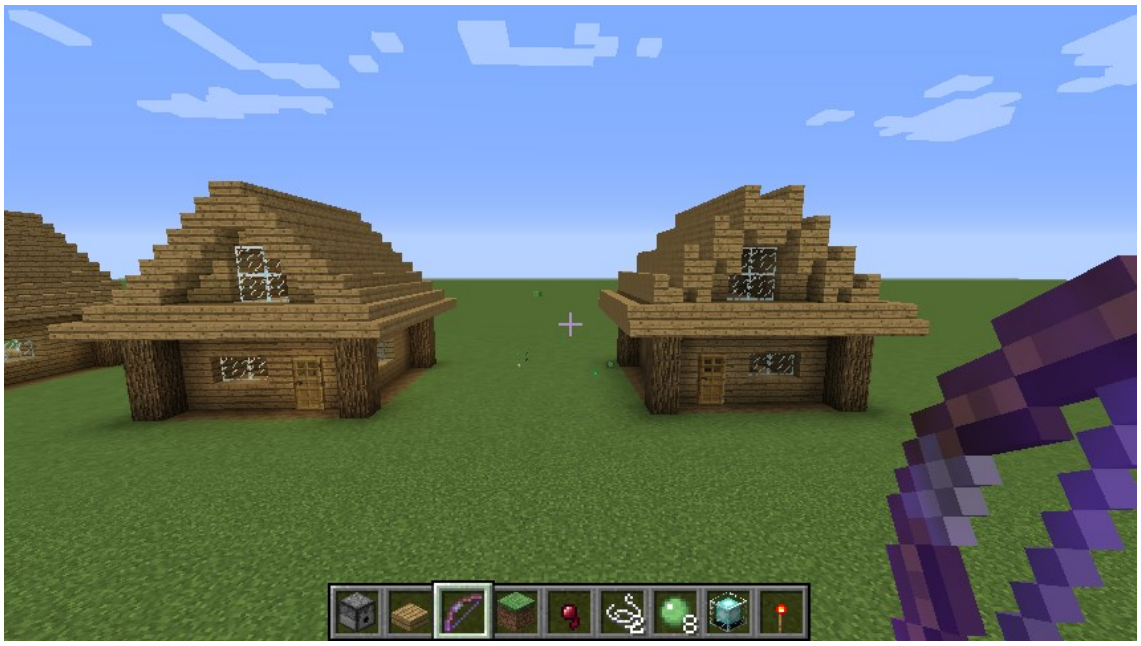


Figure 6-1: A finished replication of the house on the left which resulted in a reflected house with a wrong orientation of stair blocks representing a roof

One of our main targets, as described in the Introduction chapter, was to generate more diverse results than a simple copy of a source template. To achieve higher diversity in its results, WFC Models often do not consider only the original patterns found in the input template, but also their rotations and reflections. That, however, is not feasible with Minecraft by itself as certain blocks' properties can contain orientation of the block and by, for example, mirroring the block pattern, we can yield undesired results. An example of such a failed result can be seen in Figure 6-1.

If one would like to use blocks with orientation, their orientation would have to be changed within the preparation phase of the WFC Model (for more information about WFC models, refer to subsection 3.2.2) during pattern rotation operations. That would require creating a custom Minecraft WFC Model within the VGL project. Since this is the first proof of concept of usage of WFC on this scale, we decided to solve this issue by restricting our structures to use only blocks that do not have orientation properties. One exception is doors that are postprocessed into the house in Generate House Command, as described in subsection 4.2.4.5.

6.1.3 Diversity

Another issue we encountered was that when trying to replicate smaller templates with high details, the WFC algorithm extracts many unique patterns with a low amount of overlap. This low diversity in pattern neighbours results in 1:1 copies or mirrored copies of the original template instead of diverse results (see Figure 6-1).

From this observation, we concluded that there are 3 approaches we might take:

1. We make the houses simpler to allow the algorithm higher variability in pattern neighbours (see Figure 6-3)
2. We make the houses bigger so that details are not so dominant, and the algorithm has more free space to work with
3. We split the generation process into multiple phases – e.g.: floor plan generation, wall and ceiling generation, decoration generation (see Figure 6-2)

We discarded the second option, because, as mentioned in subsection 3.3.2, the performance of the WFC algorithm is dependent on the number of patterns in the input template and the size of the output structure. Therefore, pursuing option 2 would most likely result in very high time and memory demands.

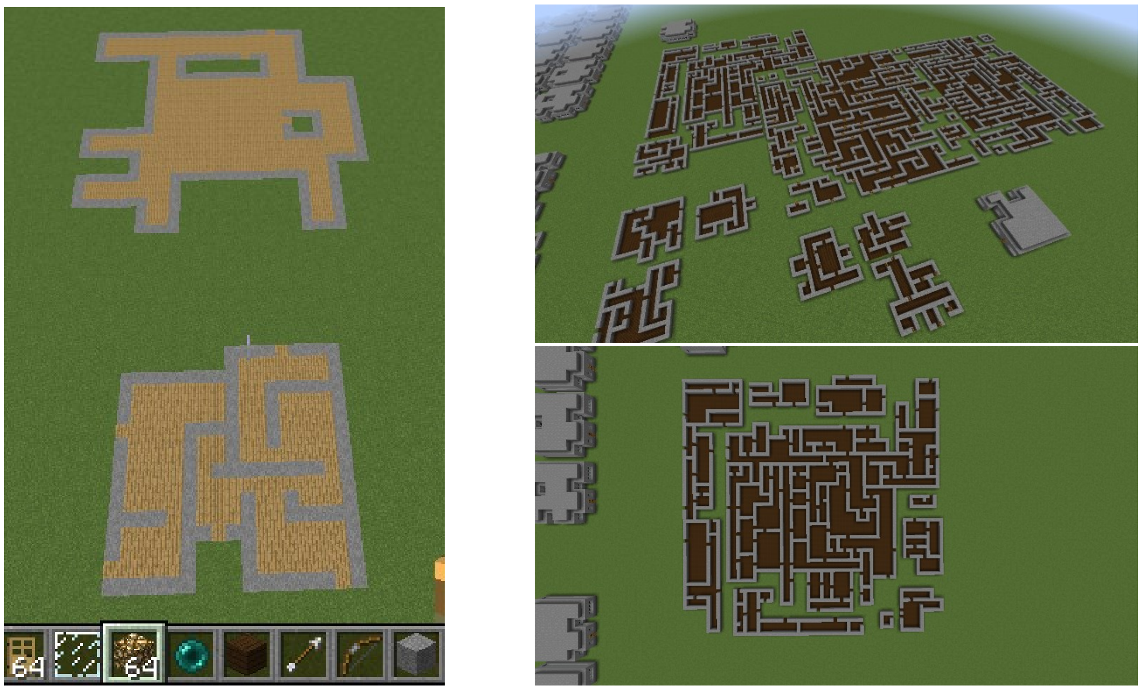


Figure 6-2: Examples of generated floor plans

At first, we decided to pursue option 3 and try the generation of floor plans, which you can see in Figure 6-2. We dynamically extracted a floor plan from a given structure template and used `OverlappingCartesian2DModel`, described in subsection 3.2.2.1, for its replication. The floor plan generation yielded much higher variance in house structure, and with the postprocessing, we could ensure that all generated rooms in a house are accessible.

However, the high variance of the floor plan generation caused issues. Because the floor plan 2D patterns allowed for many neighbours, the algorithm often reached a contradiction when the result was applied to 3D generation. We found out that there are patterns in the 3D template which have compatible floor patterns but are not compatible overall.

This issue could be solved by compressing the higher dimension into the floor plan (for example by representing each pixel in the floor plan with a combination of all three IDs of the vertical blocks in the 3D structure template at the same position). However, that approach gets us back to similar complexity of the original generation. For that reason, we decided to switch to option 1 for its simplicity. An example of such a template can be seen in Figure 6-3.

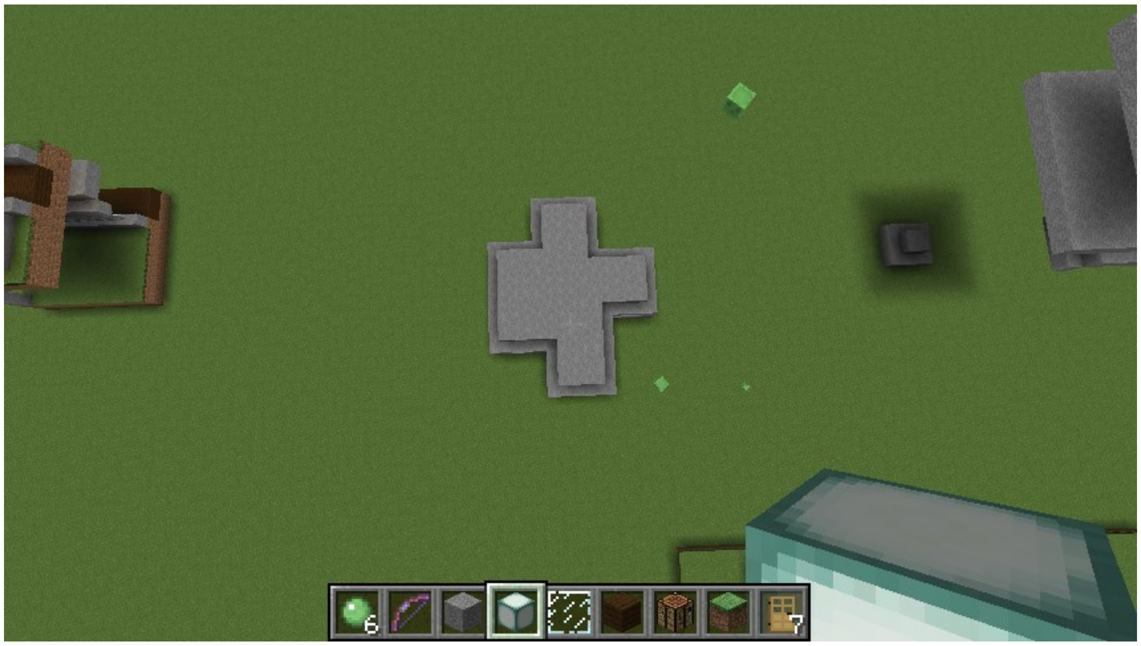


Figure 6-3: An example of a simple template - the house contains no inside walls

Option 1 strips the house to the bare minimum and leaves only the floor, walls, ceiling and doors. Since there are no decorations in the build, the patterns have a higher probability to be compatible as neighbours, and thus the algorithm has higher freedom in generation of new structures, which you can see in Figure 6-4.

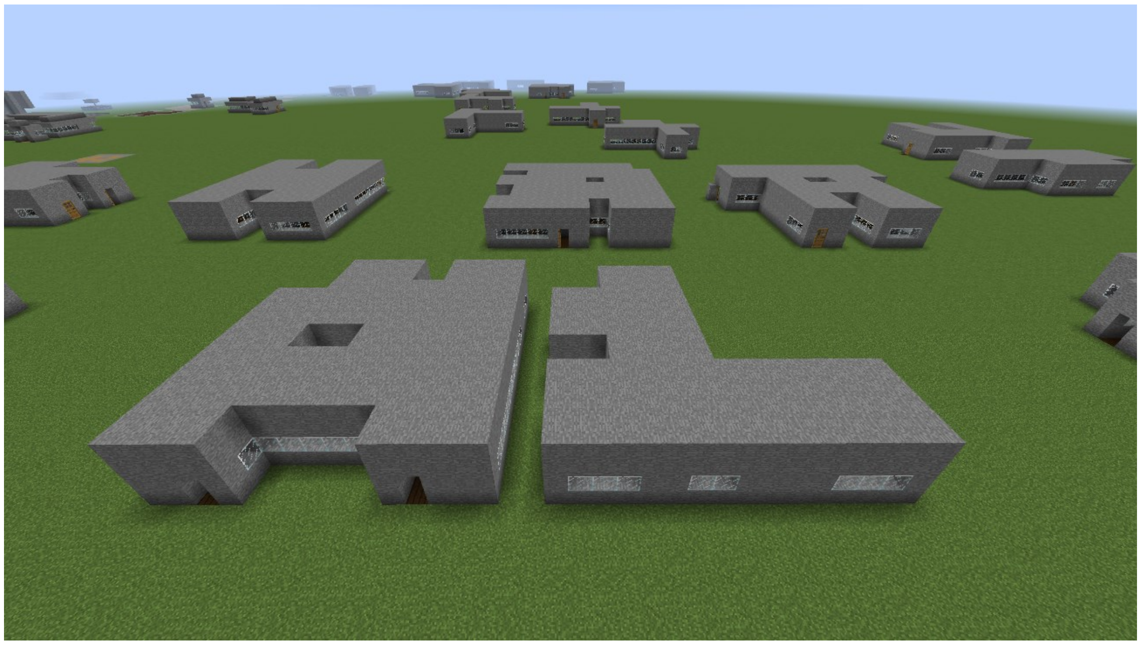


Figure 6-4: Examples of houses generated from a simple template and their variance in size and shape

After the implementation of house postprocessing, described in subsection 4.2.4.5, which ensures that all rooms are accessible and that doors are filled in all passages, we began experimenting with other structures and tried to find limits in the generation process. We discovered that WFC tends to simplify the output and if the input template consists of multiple different components, the algorithm often omits some of these components completely, as can be observed in Figure 6-5.

6.1.4 Village Generation

Finally, when we were satisfied with several template outputs, we explored village generation. The implementation was straightforward and makes use of two WFC generation phases. First, we divide the given area into parcels using our special instance of WFC and assign each parcel to one of the selected templates, as described in subsection 4.2.3. Next, we run generation separately for each of these parcels using Generate House Command, described in subsection 4.2.4.5. When all structures are generated, we execute the postprocessing phase in which we locate doors of each house and connect nearby doors using the A* pathfinding algorithm (see subsection 5.2.2.3).

6.2 Results

6.2.1 House generation showcase

In this subsection, we will present 8 of our template designs and the results we achieved. Please refer to figures from Figure 6-5 to Figure 6-12.

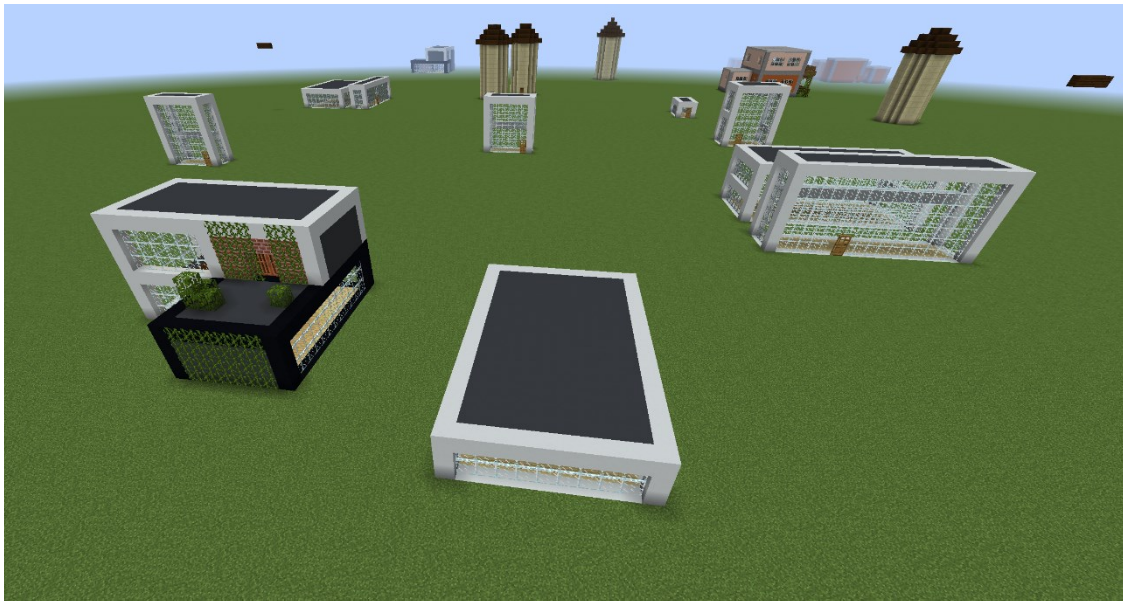


Figure 6-5: Examples of "moderna2" template (the more complicated building on the left) and generated results (the simplified white buildings on the right and in the background)

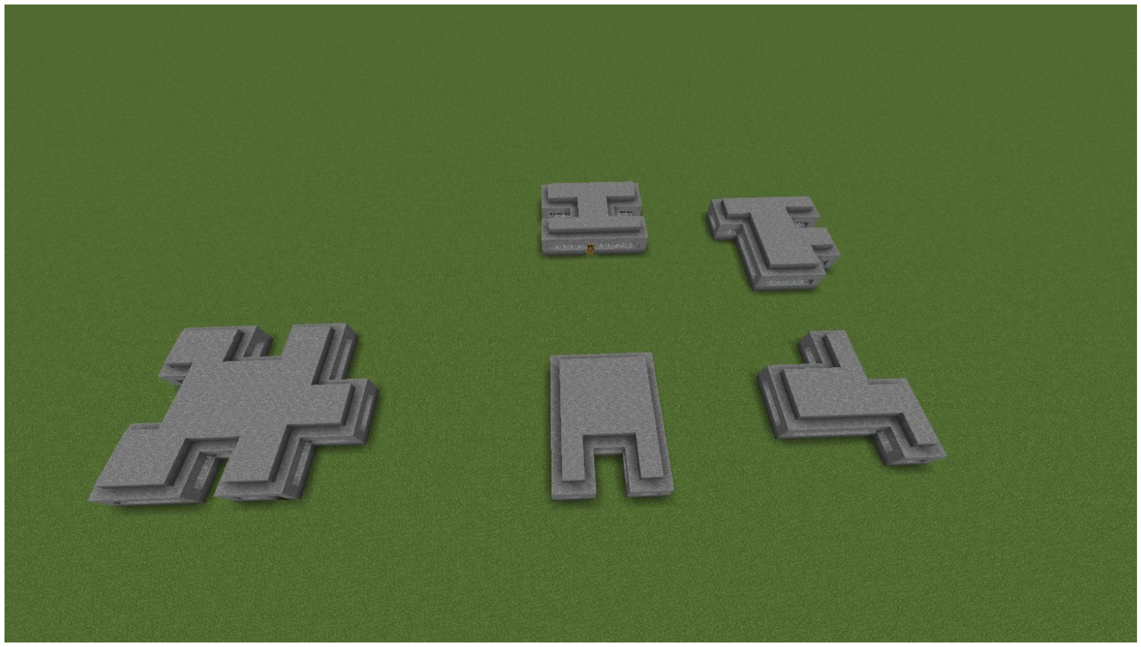


Figure 6-6: An example of 4 generated structures (on the right) from the "bunker" template (on the left)

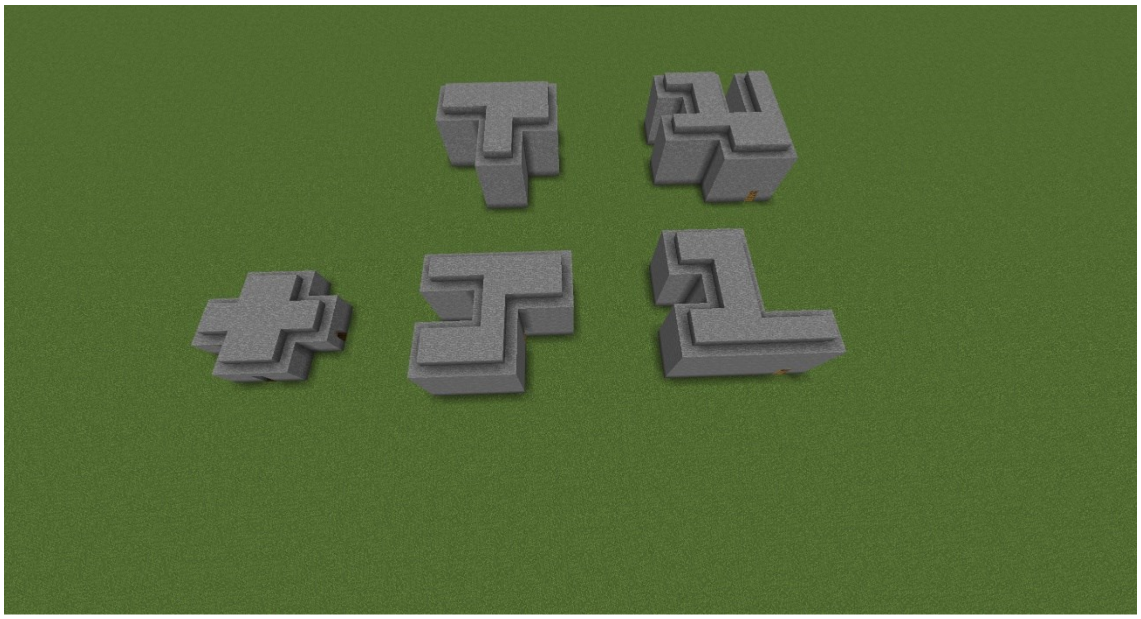


Figure 6-7: An example of 4 generated structures (on the right) from the "cross" template (on the left)

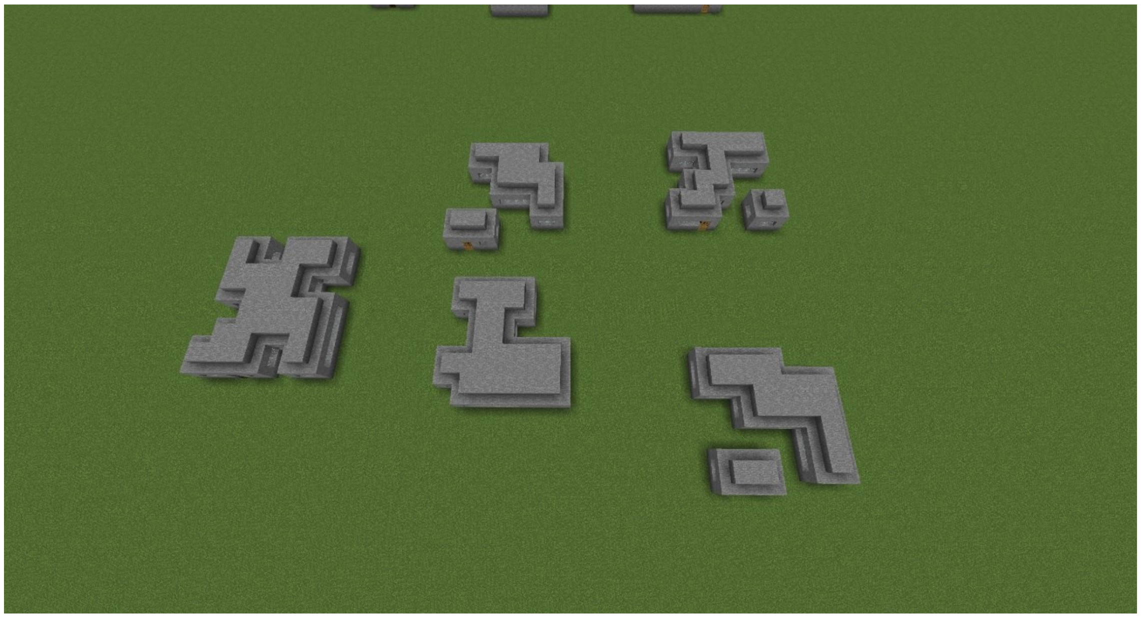


Figure 6-8: An example of 4 generated structures (on the right) from the "curvy" template (on the left) – due to the number of folds in the original template, the resulting structures tend to separate into multiple smaller structures

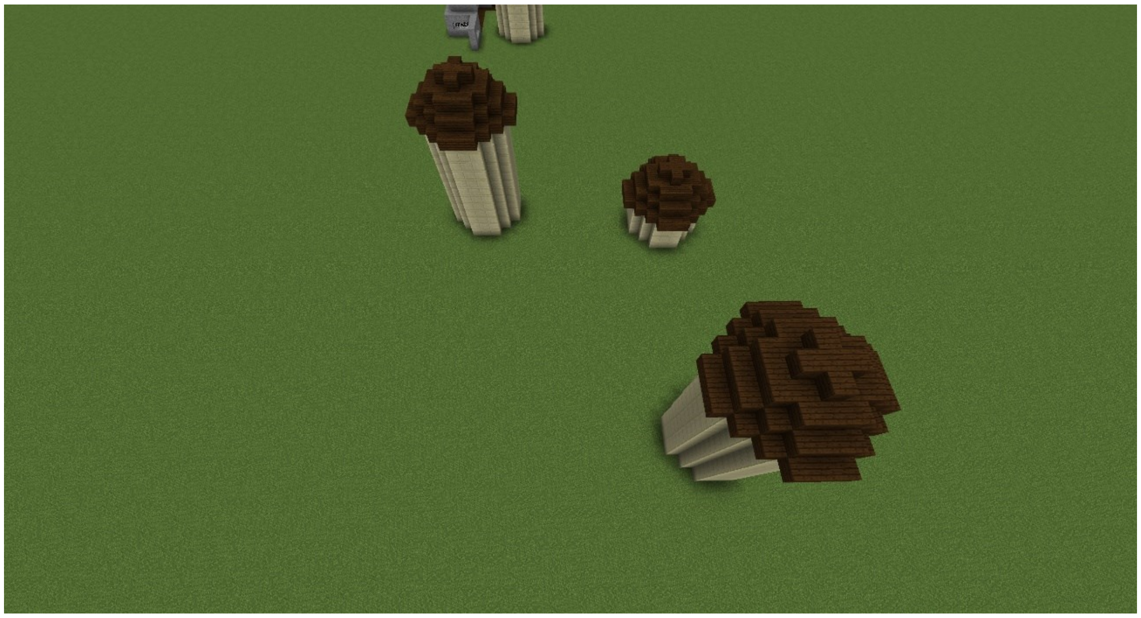


Figure 6-9: An example of 2 generated structures (on the right) from the "silo" template (on the left) – because of the small round shape of the original structure, the WFC algorithm is not able to extend the structure in depth or width – however, it can change the height

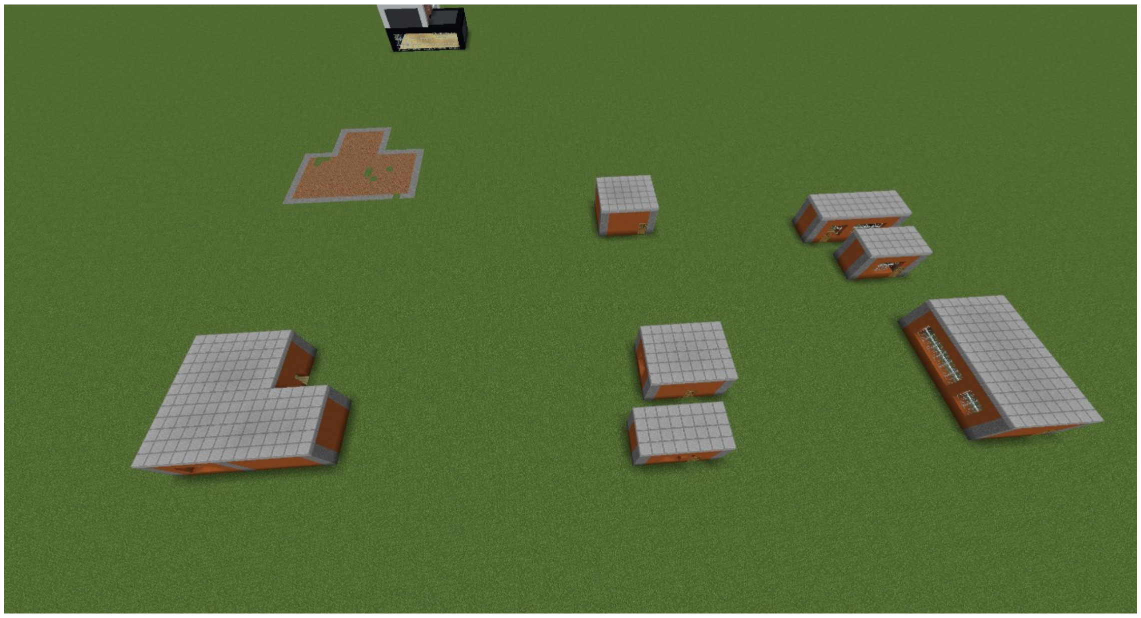


Figure 6-10: An example of 4 generated structures (on the right) from the "ter2" template (on the left)

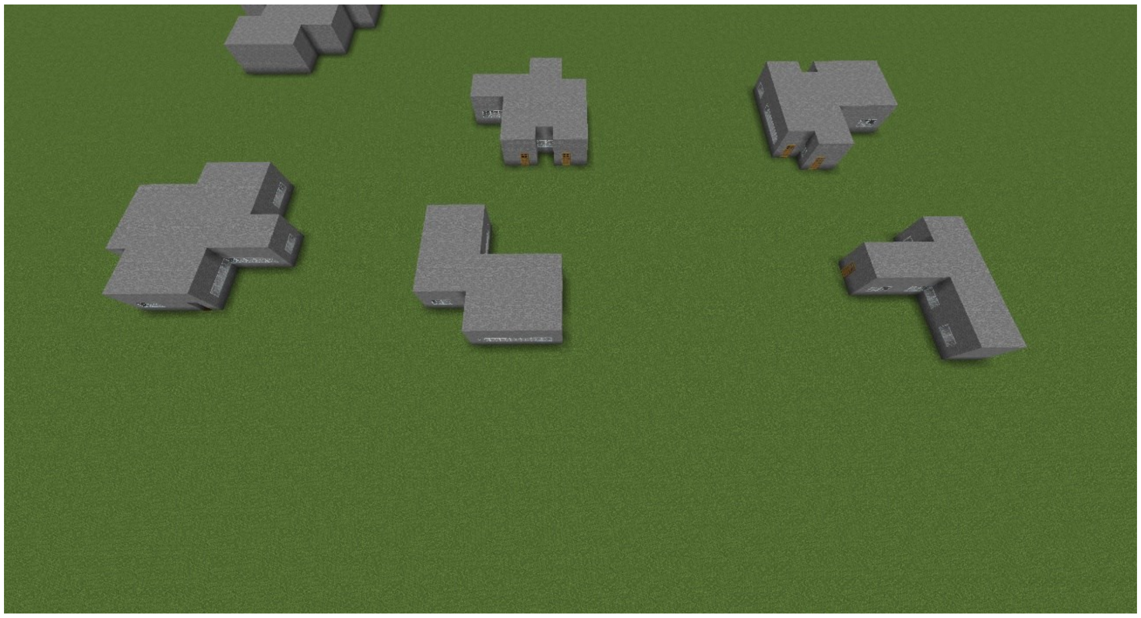


Figure 6-11: An example of 4 generated structures (on the right) from the "test4" template (on the left)

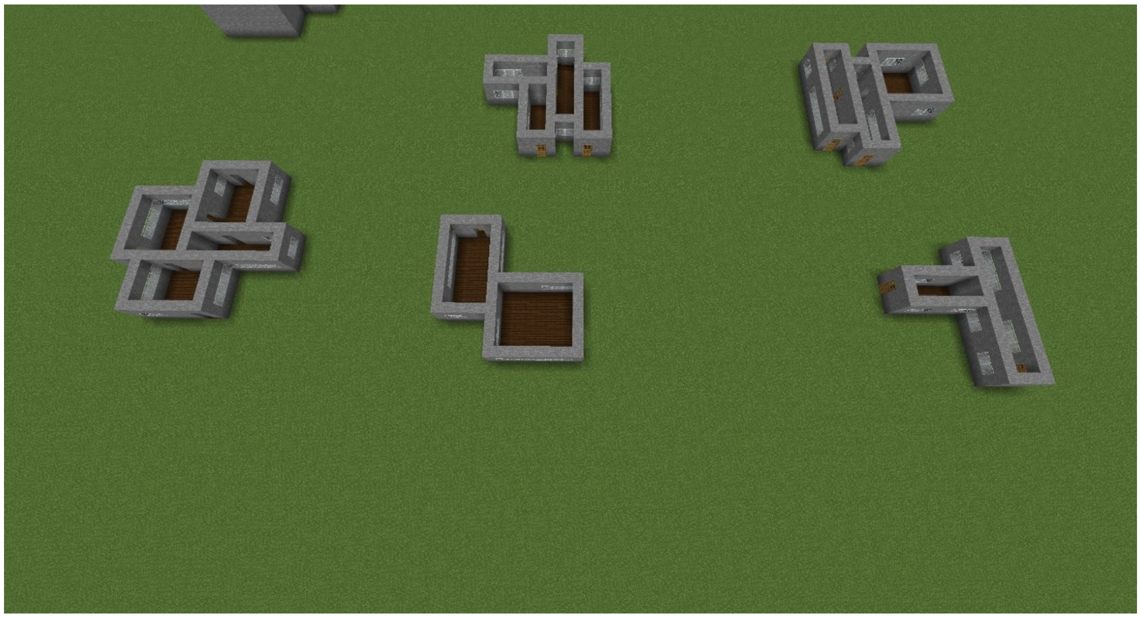


Figure 6-12: Generated structures from the "test4" template without a roof so that the inner walls are visible

An example video of the generation of a whole village can be found on YouTube (23). We will showcase village generation screenshots in subsection 6.2.3.

6.2.2 General evaluation

Our experiments show that even a simple Overlapping Model of the WFC algorithm (see subsection 3.2.2) is a viable option for PCG generation of 3D structures in Minecraft, as shown in Figure 6-13.

6.2.2.1 Strengths

The most significant strength of WFC is its ability to generate new diverse content that remains similar to the original template. If a structure can be created by a human, WFC can replicate the template and if used correctly, add variance to it.

WFC's universality ensures that it can be used in multiple phases of the generation process – from a simple division of the area to parcels to the generation of whole houses.

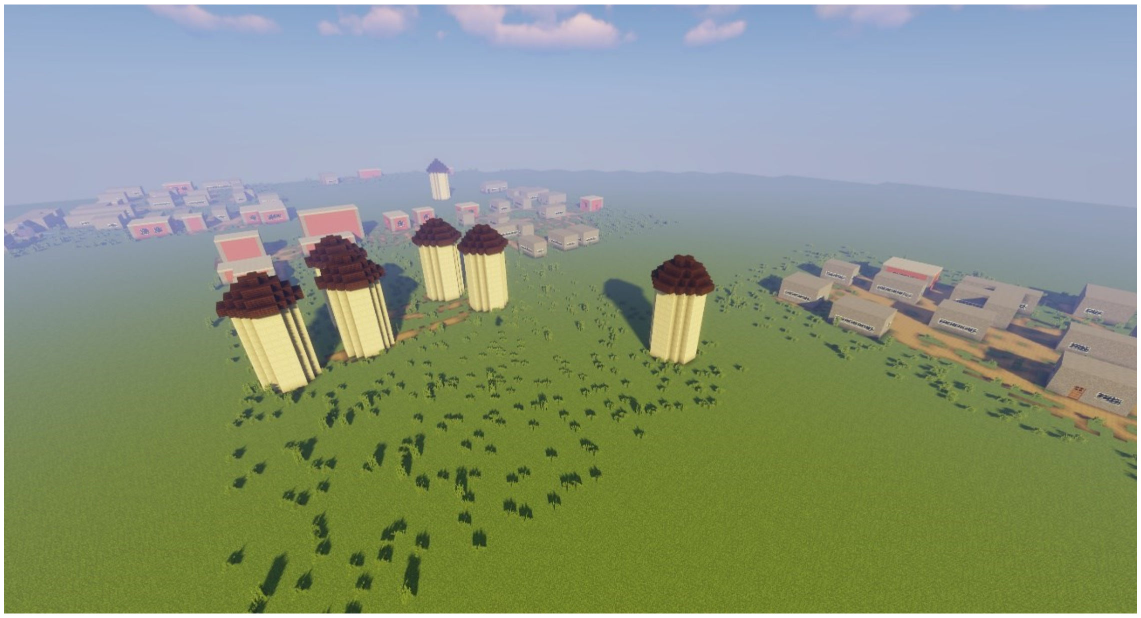


Figure 6-13: Diverse results of WFC Village generation with various parameters

6.2.2.2 Limitations

There are 3 main shortcomings to WFC generation.

First, the generation time and memory complexity depend on $outputSize \times extractedPatternCount \times dimensions$, which can grow very fast with an increasing number of dimensions that goes hand in hand with an increase of the number of patterns.

Second, unless we introduce new complex conditions into the core generation process, we cannot ensure that generated structure will be sound (there might be, for example, rooms without entrances). Another option (which we have used) is to perform some kind of postprocessing. However, if not done carefully, that can lead to unnatural-looking results.

Third, high template complexity reduces the diversity of generated structures. This issue forced us to simplify the houses to the bare minimum to achieve diverse results. We believe that this issue could be overcome if we created an algorithm to detect walls and separate the generation of inside decorations from the generation of a structure's marginal blocks. Further research is suggested to fully evaluate the feasibility of this solution.

6.2.3 Comparison

In this subsection, we compare our results with Mojang’s Minecraft villages as well as GDMC competitors’ submissions.

6.2.3.1 Minecraft

Even though Minecraft villages are simple and consist of 1:1 copies of predefined blueprints, Mojang addresses the issue by having multiple types of houses and multiple variants of blueprints of each type for each biome, as can be seen in Figure 6-14 and Figure 6-15.

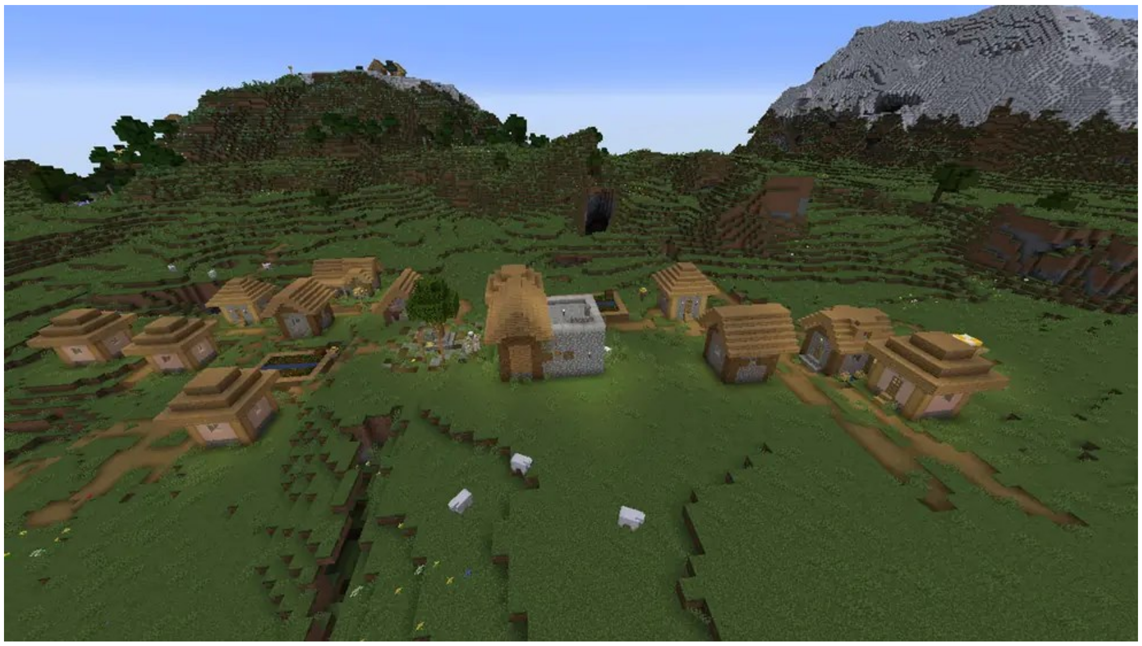


Figure 6-14: Minecraft Plains village (25)

Our approach offers higher diversity to generated houses as displayed in Figure 6-16 which was generated with only 3 templates. However, that diversity comes with the price of template simplicity. Our solution can be used as a substitute if additional postprocessing ensures more details in the generated structures. If we manage to solve the third issue from subsection 6.2.2.2, we believe that our solution will offer a reasonable alternative to Mojang’s implementation.

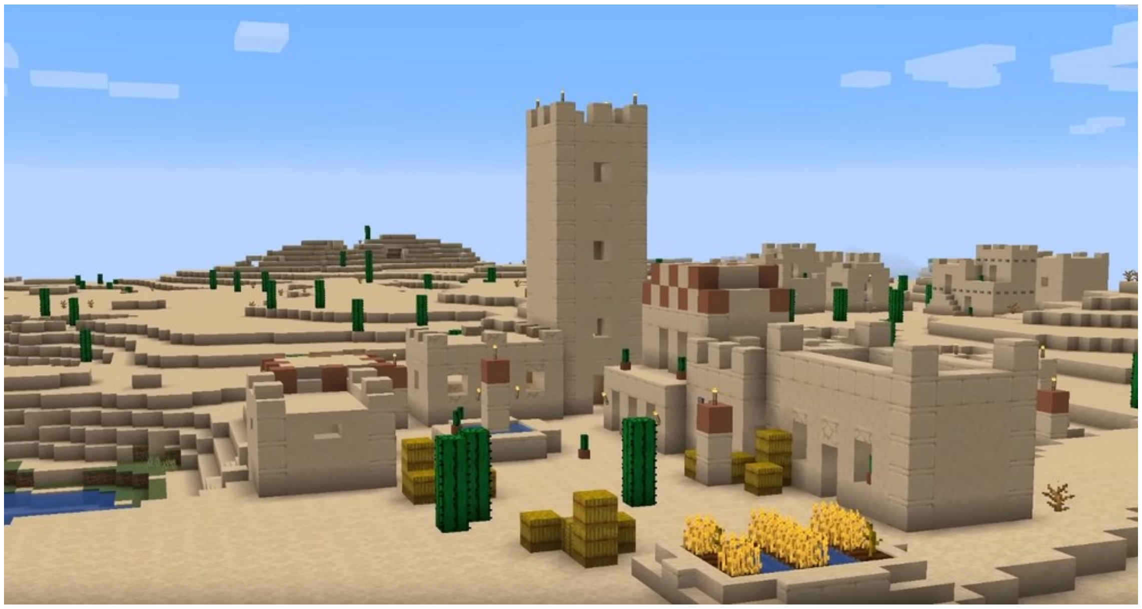


Figure 6-15: Minecraft Sand village (26)

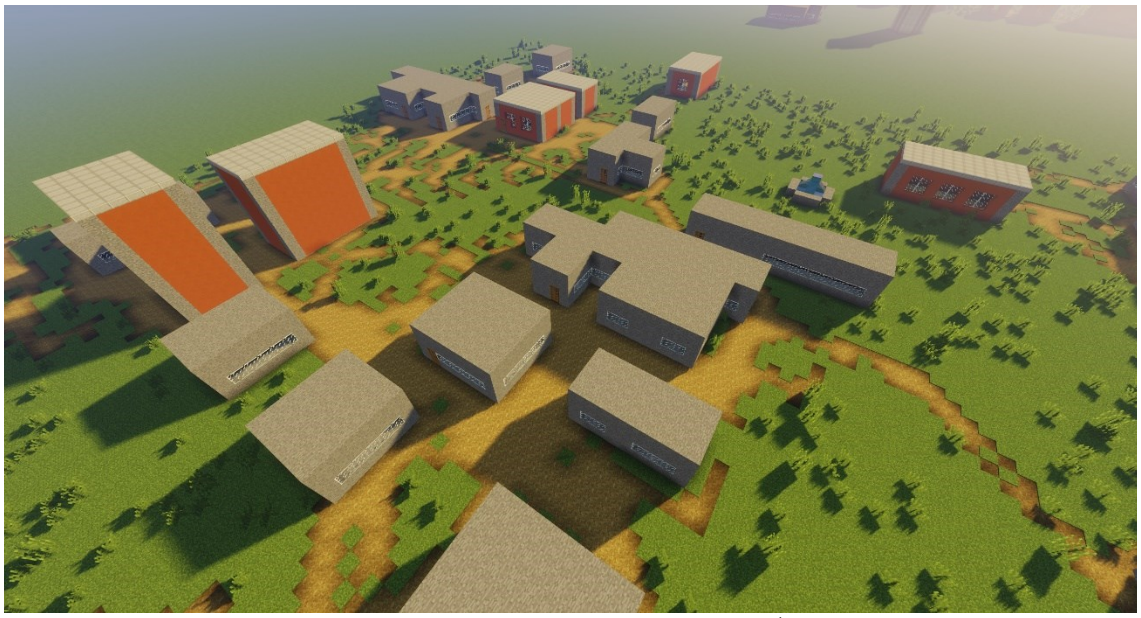


Figure 6-16: A WFC generated village[1]

[1] Visit https://www.youtube.com/watch?v=01oFJIyBDGA for the whole generation process

6.2.3.2 GDMC

Generative Design in Minecraft (GDMC) (18) is an AI settlement generation competition in Minecraft where participants compete to generate the most interesting settlement for a given area. Each settlement is judged according to various parameters (e.g.: adaptation to environment and terrain, or functionality from an embodied perspective). Results vary from participant to participant, but they usually use parametrized grammar for the generation process (19).

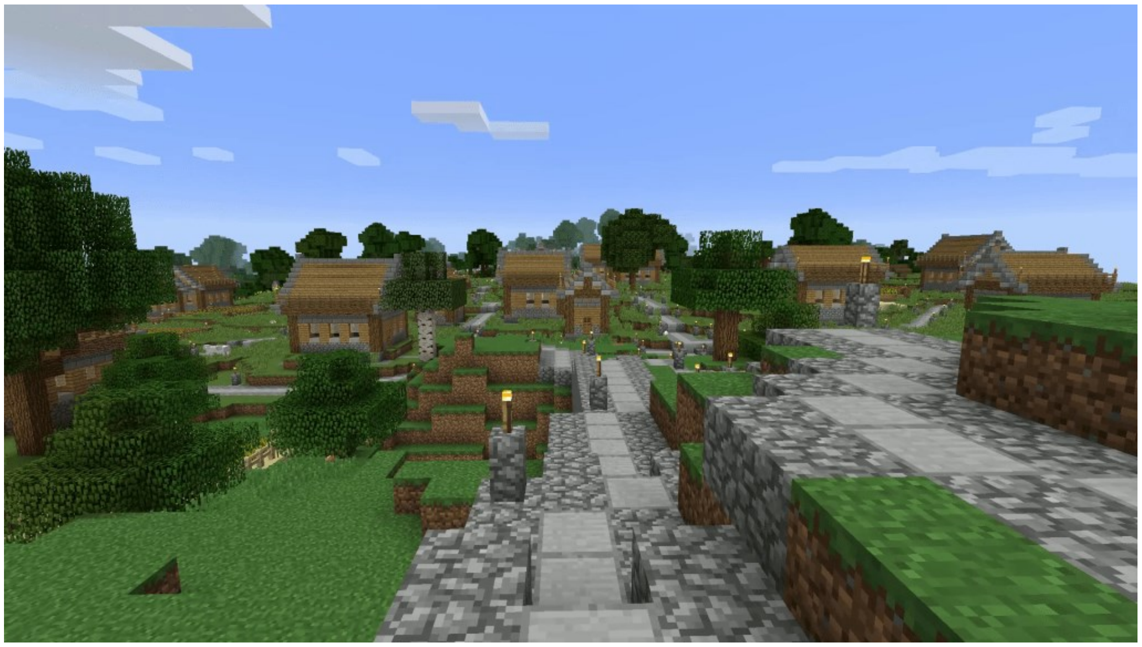


Figure 6-17: GDMC village – winner 2019 (27)

GDMC offers a feasible alternative to Mojang villages (see Figure 6-17), although the diversity of houses in most cases comes once again from different blueprints, which are translated by the grammar into the resulting structure. Even though the grammar offers variability to the resulting buildings, that variability is often limited.

We believe that WFC can help to increase building diversity. However, at the moment we fail to compete in every other aspect of the competition.

7 Conclusion

In this work, we explored the procedural generation of Minecraft villages utilizing the Wave Function Collapse algorithm.

We successfully reimplemented the original WFC library in the Kotlin language and generalized its interface and data structures. The library can run all the examples from the original repository that use the Overlapping model. Our implementation is compilable into both JVM and JS and provides the developer with more options to customize the generation process. Although our implementation proved less efficient than the original implementation, it compensates for the inefficiency with its modularity and extensibility.

We also created the Village Generation Library. The library serves as a middleware between the WFC library and the Minecraft mod and lifts as much logic as possible from the mod to Kotlin. The library provides a mapping between Minecraft blocks and integers and uses the WFC library to replicate Minecraft structures.

We successfully integrated the VGL into our Minecraft mod and demonstrated the capabilities of the WFC's Overlapping model in the Minecraft world. We show that the WFC provides a way to generate diverse structures based on simple templates. We also point out drawbacks of the WFC generation process and offer possible solutions to counter them.

7.1 Future work

There are various ways to enhance our work.

7.1.1 WFC library enhancements

The WFC library is missing the Tiled model from the original implementation as it was not necessary for our goals in this work.

There is the `CartesianNDWfcAlgorithm` class which is missing a `Model` counterpart.

Optional backtracking could be introduced into the `WfcAlgorithm` class as it should increase the success rate in attempts to generate more complicated content.

There is also an option to create a hybrid between Tiled and Overlapping models. A model which patterns would have size $N \times N$ with pattern overlap of size M where $M \ll N$. We suspect that this model could prove useful for the generation of structures with a high level of detail. We believe that pattern overlap of size 1 should be sufficient for generating structures using this model. If that assumption is correct, the model should relax the strictness WFC generation that comes from the Overlapping model which in most cases requires pattern overlap of size 2.

Optimization of the generation process could be done so that the library comes closer to the original performance.

7.1.2 Minecraft mod extensions

Minecraft mod could be improved by creating a specialized Minecraft `Model` which would be able to respect the orientation properties of blocks.

Further research could be done in the area of gradual generation of structures as mentioned in subsection 6.1.3.

The mod could be improved by searching for aesthetically pleasing templates that would work well with the algorithm.

Another improvement to the mod could be an integration with the Minecraft terrain.

With a bit of work, we believe that infinite villages could be achieved if the algorithm was run upon chunk loading.

Last but not least, we believe that the algorithm would be a fine addition to GDMC competitors. To achieve that, however, one would have to integrate our solution with multiple preprocessing and postprocessing procedures to ensure all GDMC criteria are met.

We believe that we achieved what we aimed to do. The WFC algorithm seems to be applicable in 3D space and we believe that in the upcoming years we will see an increase in the usage of the algorithm. We hope that this work helps with further research of the WFC algorithm and its uses, as it seems to have great potential.

8 Bibliography

1. Gumin, Maxim. Wave Function Collapse. *GitHub.* [Online] October 21, 2021. https://github.com/mxgmn/WaveFunctionCollapse.

2. Stamatescu, Ion-Olimpiu. *Wave Function Collapse.* Berlin : Springer, Berlin, Heidelberg, 2009. 978-3-540-70622-9.

3. Minecraft. *Minecraft.* [Online] Mojang, 1 1, 2022. [Cited: 1 1, 2022.] https://www.minecraft.net/en-us/about-minecraft.

4. Minecraft Revenue and Usage Statistics (2021). *Business of Apps.* [Online] Business of Apps, 2021. [Cited: 1 1, 2022.] https://www.businessofapps.com/data/minecraft-statistics/.

5. Survival. *Minecraft Fandom.* [Online] 12 11, 2021. [Cited: 1 1, 2022.] https://minecraft.fandom.com/wiki/Survival.

6. Java Edition version history. *Minecraft Fandom.* [Online] 12 17, 2021. [Cited: 1 1, 2022.] https://minecraft.fandom.com/wiki/Java_Edition_version_history.

7. Village/Structure/Blueprints. *Minecraft Fandom.* [Online] 11 9, 2021. [Cited: 1 1, 2022.] https://minecraft.fandom.com/wiki/Village/Structure/Blueprints.

8. Noor Shaker, Julian Togelius, and Mark J. Nelson. Procedural Content Generation in Games: A Textbook and an Overview of Current Research. s.l. : Springer, 2016, 1, 11.

9. Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. *Procedural content generation for games: A survey.* s.l. : ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 2013.

10. Černý, Vojtěch. Procedural Content Generation in Computer Games. *Game Development @ CUNI CZ.* [Online] Charles University, 2020. [Cited: 1 1, 2022.] https://gamedev.cuni.cz/study/courses-history/courses-2019-2020/procedural-content-generation-in-games-201920/.

11. Henderson, Roger Mohn and Thomas C. Arc and Path Consistency Revisited. *School of Computing, University of Utah.* [Online] 1986. [Cited: 1 1, 2022.] http://www.cs.utah.edu/~tch/CS4300/resources/AC4.pdf.

12. Isaac Karth, Adam M. Smith. WaveFunctionCollapse is Constraint Solving in the Wild. *Adam M. Smith, Ph.D.* [Online] 8 2017. [Cited: 1 1, 2022.] https://adamsmith.as/papers/wfc_is_constraint_solving_in_the_wild.pdf.

13. Gumin, Maxim. Basic 3D WFC. *Bitbucket.* [Online] 11 3, 2016. [Cited: 1 1, 2022.] https://bitbucket.org/mxgmn/basic3dwfc/src/master/.

14. Jakobs, Edwin. wfc. *github.* [Online] 11 2019. [Cited: 1 1, 2022.] https://github.com/edwinRNDR/wfc.

15. Village. *Minecraft Fandom.* [Online] 12 21, 2021. [Cited: 1 1, 2022.] https://minecraft.fandom.com/wiki/Village.

16. Anderson, Philip. Best Minecraft Village Mods. *PWRDOWN.* [Online] 8 2020. [Cited: 1 1, 2022.] https://www.pwrdown.com/minecraft/best-village-mods-minecraft/.

17. TheWeatherPony. Mo' Villages. *CurseForge.* [Online] 9 26, 2017. [Cited: 1 1, 2022.] https://www.curseforge.com/minecraft/mc-mods/mo-villages.

18. Generative Design in Minecraft . *The GDMC Competition.* [Online] 2021. https://gendesignmc.engineering.nyu.edu/.

19. Salge, Christoph & Green, Michael & Canaan, Rodrigo & Skwarski, Filip & Fritsch, Rafael & Brightmoore, Adrian & Ye, Shaofang & Cao, Changxing & Togelius, Julian. The AI Settlement Generation Challenge in Minecraft: First Year Report. [Online] 2020. https://www.researchgate.net/publication/338676099_The_AI_Settlement_Generation_Challenge_in_Minecraft_First_Year_Report.

20. Downloads. *Minecraft Forge.* [Online] 2022. [Cited: 1 1, 2022.] https://files.minecraftforge.net/net/minecraftforge/forge/.

21. Minecraft Fabric VS Forge – Which Mod Loader Is Better? *GamerTweak.* [Online] 12 18, 2021. [Cited: 1 1, 2022.] https://gamertweak.com/fabric-vs-forge-minecraft/.

22. Java Edition 12w07a . *Minecraft Fandom.* [Online] 6 10, 2021. [Cited: 1 1, 2022.] https://minecraft.fandom.com/wiki/Java_Edition_12w07a.

23. Mifek, Jakub. WFC. *YouTube.* [Online] 2022. https://www.youtube.com/playlist?list=PLugXGuvWOo3sy6vex5lgcfMQETEukP3WM .

24. Tick. *Minecraft Fandom.* [Online] 11 2021. [Cited: 1 1, 2022.] https://minecraft.fandom.com/wiki/Tick.

25. Village in Plains biome. *Minecraft Fandom.* [Online] https://static.wikia.nocookie.net/minecraft_gamepedia/images/c/c8/Village_in_Plains_biomes.png/revision/latest/scale-to-width-down/987?cb=20211228123753.

26. Village in Sand biome. *Minecraft Fandom.* [Online] https://static.wikia.nocookie.net/minecraft_gamepedia/images/8/86/New_Desert_Village_Architecture.png/revision/latest/scale-to-width-down/1000?cb=20181006022934.

27. GDMC. Winner 2019. *Generative Design in MInecraft Competition.* [Online] 2019. https://gendesignmc.wdfiles.com/local--files/wiki:2019-settlement-generation-results/winner.png.

9 List of Abbreviations

- API = Application Programming Interface
- JS = JavaScript
- JVM = Java Virtual Machine
- PCG = Procedural Content Generation
- VGL = Village Generation Library
- WFC = Wave Function Collapse

A. Attachments

In this appendix, we describe our attachments to this thesis.

A.1 Electronic attachment

In this section, we describe the electronic attachment. The attachment is split into multiple directories:

- WFC-Kotlin – contains source code for WFC library
- VGL – contains source code for Village Generation Library
- Minecraft-Mod – contains source code for the Minecraft mod
- jars – contains subfolders with built libraries and the mod
- documentation – contains subfolders with generated programmer documentation and a PDF of user documentation
- templates – contains templates we used in the Minecraft mod to generate structures presented in this work