Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Level Generation with Wave Function Collapse in Unity

Bachelor Thesis

for obtaining the academic degree of

Bachelor of Science (B.Sc.)

at the

University Of Applied Sciences - (HTW) Berlin

Faculty 4: Informatics, Communication and Economy

Course: ***International Media and Computing***

1. Assessor: Prof. Tobias Lenz

2. Assessor: Prof. Klaus Jung

Submitted by

Elisabeth Kintzel

s05741896

Elisabeth.Kintzel@Student.HTW-Berlin.de

Working Time

27.04. - 17.08.2023

Abstract

When it comes to developing video games, the task of manually crafting countless unique levels is quite overwhelming. Fortunately, there are algorithms, capable of creating procedurally generated levels, which can be used to automate this process. One of these algorithms is called Wave Function Collapse (WFC).

This paper aims to take a closer look at what makes WFC effective, what its limitations are, and how it can be extended. We implemented a variety of different modifications and extensions of the WFC algorithm and tested them with different test scenarios and requirements.

The goal is to investigate the potential of WFC, specifically for procedural level generation from a tileset, to make the process easier and faster.

Affidavit

I hereby declare that this thesis is the result of my own work.

I have not used any sources and aids other than those indicated. All sources and/or materials which were taken literally or analogously from publication or from other foreign statements are marked as such.

Furthermore, I confirm that I have not previously submitted this work to any other university or course of study as an examination, neither in identical nor in similar form.

13.08.2023

....

Date, Signature

Table Of Contents

Introduction.......... 1

1. Wave Function Collapse.......... 3

1.1 Background.......... 3

1.2 Procedural Generation.......... 5

1.3 Alternatives.......... 10

2. Implementation.......... 13

2.1 Base Implementation.......... 14

2.2 Variations.......... 20

2.2.1 Propagation Range.......... 20

2.2.2 Frequency Notes.......... 21

2.2.3 Connection Frequencies.......... 23

3. Generation of Tiles from Images.......... 25

3.1 Implementation.......... 25

3.2 Variations.......... 27

3.2.1 Separating Image Analyzer and Map Builder.......... 27

3.2.3 Grid Size and Offset.......... 28

3.2.2 Edge Wrap.......... 29

4. Testing.......... 30

4.1 Map Generation.......... 30

4.1.1 Test Conditions.......... 30

4.1.2 Test Results.......... 33

4.1.3 Test Evaluation.......... 39

4.2 Tile Generation.......... 41

4.2.1 Test Conditions.......... 41

4.2.2 Test Results.......... 42

4.2.3 Test Evaluation.......... 46

5. Conclusions.......... 47

5.1 Limitations and Revisioning.......... 48

5.2 Possible Continuations.......... 53

References.......... 54

Figures.......... 55

Appendix.......... 57

Link to the Github Repository.......... 57

List of Abbreviations.......... 57

Additional Data Tables and Diagrams.......... 58

Additional Maps.......... 63

Introduction

The creation of a video game requires a great volume of work and expertise in a variety of different subjects. From coding to art to marketing, there is an abundance of different skill sets needed to put together a functioning game. Luckily, not every single job has to be performed painstakingly by hand. Nowadays we have a vast collection of tools and devices at our disposal that can assist in the process.

Dungeon crawlers, for example, are games that have a lot of repetitive, maze-like levels, which have to be explored, looted, and of course survived. In such games, it is often necessary to repeat the same dungeon over and over again, either to find some rare treasure or to level up your adventuring abilities. While these dungeons seem to be the same in terms of difficulty, enemy types, length, or general design and style, getting met with exactly the same maze layouts every single time quickly becomes repetitive and boring. To combat that, the game designer could create a few different dungeon blueprints, but still, these would need to be repeated at some point. Fortunately, there is another option: procedurally generated levels, where each map is not completely designed and built by hand, but instead the layout is constructed by an algorithm.

There is a great range of different approaches and solutions to create procedurally generated levels, however, the algorithm we are going to take a closer look at in this paper is called Wave Function Collapse (WFC).

Scope

This paper aims to explore WFC, specifically in the context of procedural level generation from a tileset, to analyze what makes it effective, what its limitations are, and how it can be extended.

Naturally, the performance of such an algorithm can differ heavily depending on the scale and specific circumstances of the job it is supposed to fulfill. Therefore, we are going to experiment with variations of the algorithm itself, namely an increased propagation range, and output finetuning through tile frequencies and connection frequencies.

Then we are going to test each adaption under different conditions, including output scales, tileset sizes, and constraint complexities, to identify which attributes influence the WFC's performance the most.

Structure

The first chapter will go over the basics of Wave Function Collapse, how it works, where it originates from, and how it has been adapted into the game development field.

The next step is implementing this concept within a Unity project. Here, we are going to explain how the overall environment, in which the algorithm is supposed to work, has been set up. Then, we are delving further into how the WFC functions within this setting, as well as exploring the different alterations and optimizations tried. Afterwards, we will look into the possibility of generating a new tileset, based solely on an exemplary input image.

After the implementation section, we are focusing on the testing conditions that were experimented with. Here we are also taking a closer look at the tilesets we have been working with, in order to analyze the influence different tile characteristics have on the performance and effectiveness of the algorithm.

Lastly, we will be comparing the findings about each algorithm and test run we examined, so the final conclusions can be formed out of those results.

1. Wave Function Collapse

1.1 Background

The term wave function collapse originates in the quantum physics field, where it describes the collapse of a wave of possibilities, caused by being measured.

One of the most famous examples of this phenomenon is the Double-slit experiment, in which particles are shot at a barrier with two slits and caught at a screen behind it.[Fig.1] This experiment was originally done with photons, aka. light particles, but has since been repeated with other particles, such as electrons or protons. [1]

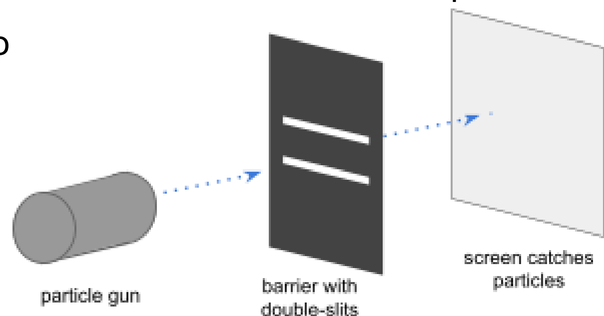

Fig.1: Double-Slit Experiment

When particles are being shot through a barrier like this, one would expect the pattern on the screen behind to look just like the openings through which the particles passed.[Fig.4] However, the particles on the screen seem to be arranged in an interference pattern,[Fig.2,3] a shape that typically occurs when two waves meet one another.

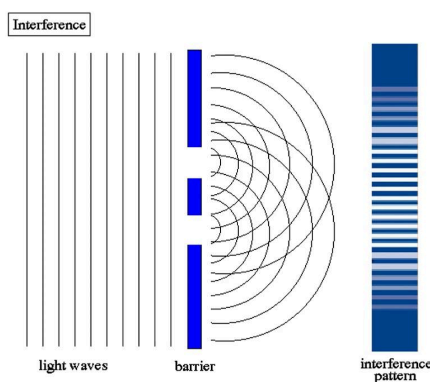

Fig.2: Wave Interference
[Bown: Diffraction and Interference Patterns - IB Physics (2021)]

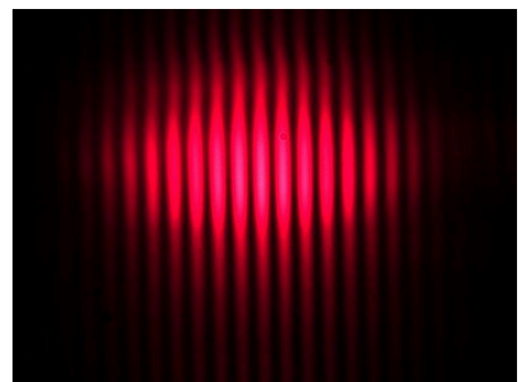

Fig.3: Interference Pattern
[A.Consoli: Definitively a wave... or not? (2022)]

At first glance, this outcome seems absolutely impossible. The particles formed this pattern, even though some of the spots are not reachable by a straight path through either of the slits. In an attempt to understand the movement of each individual particle, a detector is attached to the slits, which measures through which of the two the particle went. Doing this, however, suddenly changes the outcome drastically: the interference

pattern vanishes and instead, it gets replaced with the previously expected two-slit pattern.[Fig.4]

How exactly an object can behave like both a particle and a wave, depending on the situation, is still highly debated. However, one of the most commonly accepted explanations is known as the Copenhagen Interpretation. [2]

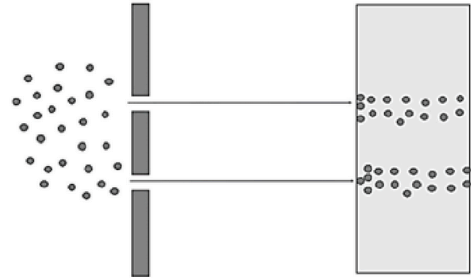

Fig.4: Expected Two-Slit Pattern
[V.Ranjbar: Quantum State Function, Platonic Forms, and the Ethereal Substance [...] (2022)]

According to this approach, the photon in an unobserved state, has a range of different positions in which it could exist. All these potential paths, the particle could take, form a wave of possibilities together.

In mid-flight, the photon is still in a superposition of existing in all places within the waves at once. Upon meeting the barrier, it can fly through either one of the slits, creating two new waves of possible trajectories. These two waves are then interacting with each other, creating the distinct wave interference pattern. Only when hitting the screen does the particle become visible and thus measured, causing the wave to collapse and forcing the photon to decide on one position.

By adding a detector to the slits, the photon's state gets measured even earlier, causing the wave to collapse at the barrier, thus creating the previously expected two-slit pattern.

To summarize, WFC is not about particles themselves interacting with each other. Instead, each possibility of one individual particle is influencing one another, until this wave of potential finally collapses through measurement.

1.2 Procedural Generation

The WFC algorithm was created by the Finnish programmer Maxim Gumin in 2016 as a way to generate images that are visually similar to a given input image. [3] It was published on GitHub as an open-source project, to encourage its usage and further development within the entire procedural generation community.

WFC stands out against other procedural image generation approaches by combining individual pixels into a larger tile, instead of taking every single pixel into consideration separately. Tiles are formed by clusters of multiple pixels and are repeated across the output image.

To ensure the local similarity between the input image and the generated output, only tiles, which can be found in the input image, are used for the generation. The algorithm does not create any new combination of pixels.[Fig.5]

Fig.5: Pattern Repetition
[M. Gumin: WaveFunctionCollapse Github Repository (2021)]

WFC works on the basis of constraint satisfaction, where a solution is constructed by fulfilling a set of given rules and restrictions. Each individual tile has its own connection rules, describing which patterns can appear next to it and which do not fit. When a tile is placed within the output image, the options of what can appear around it are limited by these rules. The algorithm builds the output tile by tile, in accordance with the set constraints. If the rules cannot be fulfilled, the output is faulty and needs to be redone.

Another distinct feature of WFC is its minimal entropy approach. To determine in which cell the next tile should be placed, the algorithm first calculates the entropy of each cell. Entropy is a property that describes the chaos, randomness, or disorder of a system. In

the case of WFC specifically, that means, the more possible tiles a single cell has left, the higher its entropy. To decide which cell should be filled in next, the algorithm searches for the one that has the least amount of possibilities left. That way, long lines and hard edges, like paths or walls, are usually built first. Only then gets the space in-between filled in.

The following flowchart is a step-by-step sequence of how the algorithm operates:

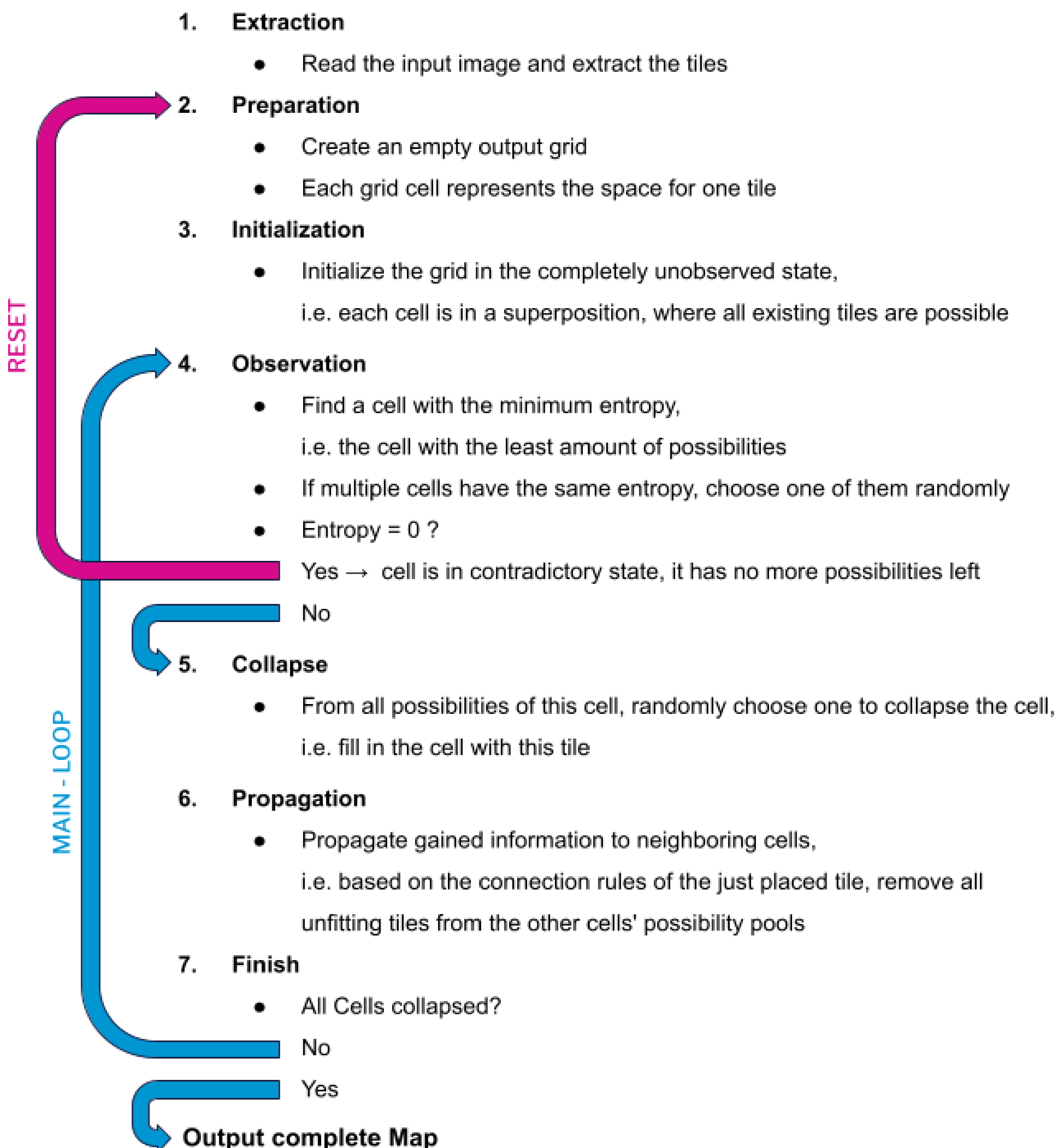

Fig.6: Flowchart - WFC working sequences

The name “Wave Function Collapse” suggests that its creator got partially inspired by the concept of the collapse of superpositions of unobserved states, found in quantum physics. However, some people believe this is an unfitting name, claiming that the algorithm only distantly reminds of actual quantum-physical concepts, and in reality is just an up-scaled constraint solver, similar to a sudoku or jigsaw puzzle solver. [4]
They claim the name is just a clever marketing ploy, to make the algorithm more attention-grabbing and interest-peaking. Despite this small controversy though, this strategy appears to be quite successful, since WFC quickly became a very popular, well-known algorithm within the procedural generation scene, that has since been expanded into various other tech fields as well.

Variations

Thanks to the algorithm's popularity, and its open-source availability, many people have experimented with it, creating a variety of different versions and adaptions.
Some of the modifications that are specifically useful for game development, have been implemented within the coding section of this paper, and are thus going to be explained in more detail in a later chapter.

However, games are not the only area people found a use for WFC. Due to its tiled approach and versatile constraint-solving nature, the algorithm can easily be altered to work with numerous different types of generative algorithms, creating a variety of interesting applications, which are worth being mentioned here.

Examples and demonstrations of some of these adaptions in action can be found in the footnotes.

Instead of just generating new textures for example, the algorithm can be used to extend a given image, either by enlarging its bounds or by replacing certain sections of the picture.

Expanding on this idea even further are programs, which enable a user to manually build part of an image, and then complete it procedurally.[1]

By expanding the algorithm into the 3rd dimension it is possible to generate impressive 3D models.[2] However, as a result of the much higher amount of cells and more complex tiles this requires, the processing time increases rapidly.

Luckily, there is an abundance of ideas and approaches on how to increase the algorithm's performance. For instance, by replacing the harsh reset with a backtracking option, or by using a simpler tileset for the generation process, which a decoder then converts into a more complex map.[3]

Furthermore, some creative programmers also found more artistic uses for WFC, using it to create paintings,[4] poetry,[5] or even music.[6]

Usage in Game Development

WFC proves to be a great tool for game development, offering effective solutions for various different aspects within the game creation process.

In the game Rodina, for example, WFC is used to create more interesting and varied wall decorations.[7] But the algorithm does not only offer the ability to generate textures, it can also create entire 3D models.

---

[1] WFC auto-completes a level started by a human:
→ https://i.imgur.com/X3aNDUv.gifv

[2] 3-dimensional tiles create large building models:
→ https://twitter.com/OskSta/status/787319655648100352

[3] Unconstrained WFC enhanced with Neural Decoder:
→ https://dl.acm.org/doi/pdf/10.1145/3472538.3472584

[4] WFC used to create dithering images:
→ https://twitter.com/voorbeeld/status/1073874337248239616
→ https://twitter.com/voorbeeld/status/1073875725499985926

[5] WFC imitates word placement and rhyme schemes of a poem:
→ https://twitter.com/mewo2/status/789167437518217216
→ https://twitter.com/mewo2/status/789187174683987968

[6] image inputs are converted into piano rolls
→ https://github.com/bbaltaxe/wfc-piano-roll

[7] game devs of “Rodina” talk about the new look of their buildings
→ https://steamcommunity.com/games/314230/announcements/detail/3369147113795750369

When researching about WFC in the gaming scene, you will inevitably run into the name Oskar Stålberg. The Swedish game developer experiments a lot with this concept, specifically in combination with games, sharing his knowledge and new accomplishments with the community. He has given multiple talks about the algorithm's functionality and opportunities, as well as his own history of implementation in his games.[8]

One of his most popular games, Townscaper, combines WFC with marching cubes on irregular grids, to create varied and interesting town buildings and decorations.[9]

In Bad North, another one of Stålberg's games, he utilizes WFC to generate environments. Each game level is located on a small island, containing cliffs, houses, vegetation, and similar adornments. The distinction here is that he altered the constraints of the algorithm to prevent inaccessible areas. Each part of the generated map is completely navigatable by the characters, without any obstruction.[10]

The final example is a small indie game, called Maureens' Chaotic Dungeon,[11] which has been created as part of a game jam. Because of a strict time constraint during its creation, the game itself is more a fun concept, than an actual finished, playable game. Nevertheless, it uses WFC in a very interesting, creative way, which justifies its referral. What makes this game so unique is how it uses WFC not to generate textures, models, or finished map layouts. Instead, it is used as a central gameplay mechanic.
In the game, the player can shoot a beam that essentially resets all cells back to their initial state, causing WFC to generate them anew. This way, the player can reroll certain game elements, like walls or ladders, to create new passages to previously unreachable areas. Due to the way WFC works, it is not possible to make any purposefully calculated changes to the environment, instead, the map layout gets altered in a very chaotic, random way.

---

[8] Oskar Stålberg talks about his game "Townscaper" and its use of procedural generation
→ https://www.youtube.com/watch?v=_1fvJ5sHh6A
→ https://www.youtube.com/watch?v=0bcZb-SsnrA
[9] WFC generates adaptive support beams to buildings for "Townscaper"
→ https://twitter.com/OskSta/status/1189902695303458816
[10] generated island maps are fully navigable in the game "Bad North"
→ https://twitter.com/OskSta/status/917405214638006273
[11] Game Jam entry "Maureens' Chaotic Dungeon"
→ https://andymakesgames.tumblr.com/post/182363131350/global-game-jam-2019-maureens-chaotic-dungeon

1.3 Alternatives

Model Synthesis

This algorithm was published in 2007 by the American computer scientist Paul C. Merrell. Model Synthesis (MS) is more focused on generating 3-dimensional models, but can also be used for 2D outputs. [5] It has since been made available as an open-source project as well. [6]

It has been credited as one of the resources Maxim Gumin used to base his work on, making WFC and MS each other's biggest competitors. Consequently, both algorithms are quite similar and are frequently compared to each other. [7]

One of the biggest differences between the two is the order in which they choose the next cell to collapse. While WFC uses a minimal entropy heuristic, MS approaches images in scanline order, which means the algorithm iterates through the image by starting in one corner and always continuing on to the next adjacent cell.

These different approaches can lead to quite different results, depending on the structure of the input. While MS results in a more rigid-looking arrangement, containing long, unbroken lines, the WFC output looks more organic and dynamic.[Fig.7]

However, the lowest entropy approach generally leads to more mistakes, and consequently, more necessary resets, causing MS to be the faster solution. [7]

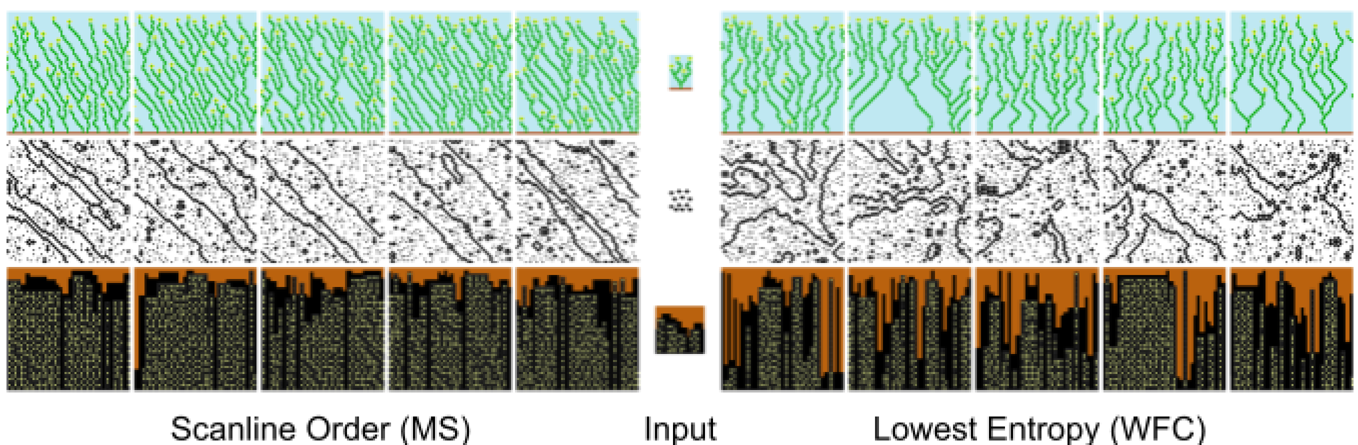

Fig.7: Directional Bias of Model Synthesis vs. Wave Function Collapse
[M. Gumin: WaveFunctionCollapse Github Repository (2021)] graphic modified

Since both algorithms are adaptable, there are enough solutions to deal with these issues quite effectively.

For instance, MS does not modify the entire output at once, but first separates it into smaller sections which are then generated consecutively. By adjusting the size of these blocks, the output's rigidness can be compensated.

This block working method is also part of the reason why MS has a much better performance time, since, in case of a reset, only the current block has to be redone. WFC, on the contrary, has to regenerate the entire output after every reset. This, naturally, leads to a much larger time increase, even if the same amount of resets would be required.

One solution WFC has to combat this is a farther propagation range[12]. By transmitting the newly gained information after a collapse, to not only the direct neighbors but also to cells beyond that, the reset rate can be significantly reduced.

Despite MS having been developed first, and arguably being even better suited for certain applications, WFC remains the more popular variant among the community, which has been adapted and experimented with far more frequently.

Texture Synthesis

Texture Synthesis (TS) is an entire research field in computer graphics, which is dedicated to the procedural generation of images. There are multiple different algorithms and approaches to this field, but generally, they all serve similar purposes, e.g. expanding images, filling holes in pictures, or generating new textures.[Fig.8]

The TS field is the predecessor to both MS and WFC. [8] However, where these two use a tiled approach to images, TS considers every pixel separately and compares it to not only their immediate adjacent neighbors but to an entire neighborhood of pixels.

The scanned neighborhood is compared to a matching one within the input image, to determine the color of the individual pixel.[Fig.9] [9]

---

[12] This is one of the adaptions, that have been implemented during this thesis' practical part, and thus will be inspected closer in a later section of this paper.

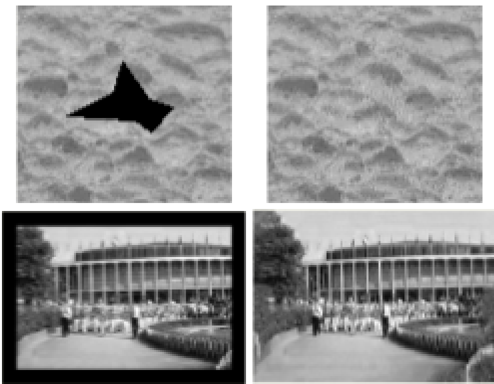

Fig.8: Texture Synthesis Examples
[A.A. Efros, T.K. Leung: Texture Synthesis by Non-parametric Sampling (1999)]

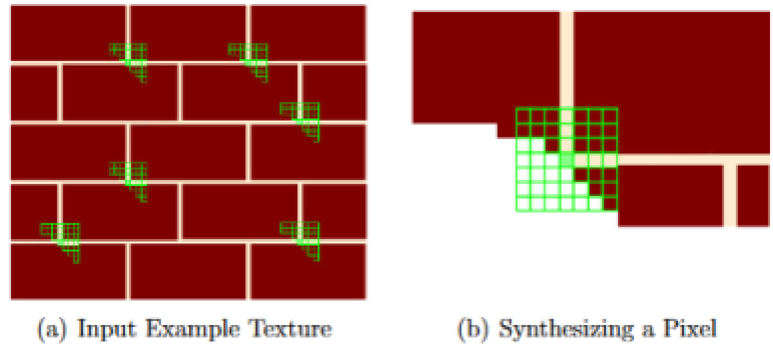

Fig.9: Comparison of neighborhoods to determine individual pixel
[P. C. Merrell: Model Synthesis (2009)]

This makes TS great when it comes to working with complex images like real-world photographs. WFC and MS, on the other hand, require the input images to be self-similar, which means they need to have the same repetitive patterns, making them specifically suited for working with pixel art and tilesets. Using TS for these applications would also be possible, however not very practical, since the smaller scale pixel comparison results in an unstructured-looking output.[Fig.10] [10]

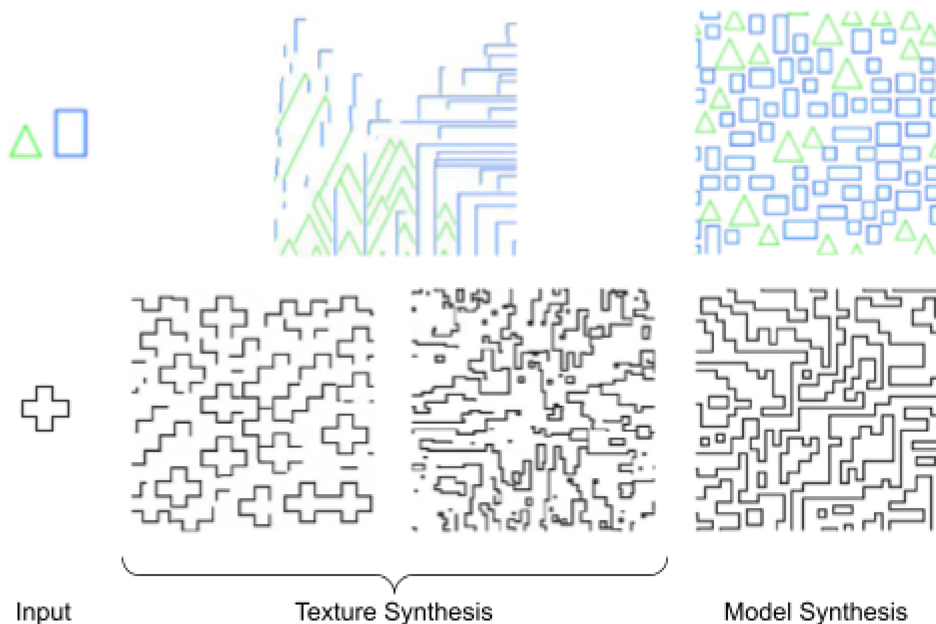

Fig.10: Comparison of Model Synthesis and Texture Synthesis
[P.C. Merrell: Model Synthesis (2009)] graphic modified

2. Implementation

The goal of this thesis is not to examine WFC in its originally intended purpose but to use it to create game levels from a deliberately crafted tileset. Therefore, we decided to code the entire program from the ground up, even though the original algorithm is available as an open-source project. This allows for more freedom and flexibility in tailoring the WFC idea to the Unity environment and the tiled level map concept. It also ensures a greater familiarity with every single facet of the code, which will be helpful in the eventual variations part of this project.

One aspect of tilesets is that they are generally created in an isolated form, where each tile has been arranged with a bit of blank space in between them.[Fig.11] Since the goal of these tile compositions is not to show how they each fit together, but only to provide a texture of every tile, they also do not necessarily fit together with their neighbors. Therefore the extraction from an image would not be possible in these cases.

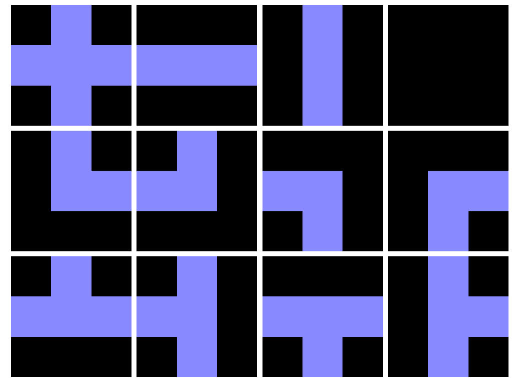

Fig.11: polished tileset has individual tiles separated from each other

For a complete implementation, however, that part of the algorithm will still be addressed in a later section of this paper, while the current chapter only focuses on generating an output, using a manually created tileset as an input.

The created code, along with the entire Unity project, can be found on GitHub[13], to individually review each feature again.

[13] https://github.com/chaotic-pan/WaveFunctionCollapse

2.1 Base Implementation

Setting the (Unity) Scene

The first task is to find an efficient way of creating a tile-based map layout. Unity already has a grid class and a tileset class integrated. However, both of these are polished tools that aim to make the creation of a tiled map easier for game designers, by making the process similar to painting entire tiles onto the grid. As a result, these tools lack flexibility and adjustability, making them unsuited for our intent of automating the map-building process.

The approach we chose instead, is to build the grid with the help of a layout group, which is a Unity object that automatically stacks multiple UI elements in an evenly-spaced grid pattern. This ensures that each cell is its own, separately accessible object, thus making it possible to assign a GridCell class to each instance, storing information like a list of all possible tiles for that cell, referred to as C-possible.[Fig.12]

Additionally, the layout group has a size constraints property, which regulates how many elements are stacked next to each other, before a new row is started, which makes the output's dimensions flexible and easy to adjust.

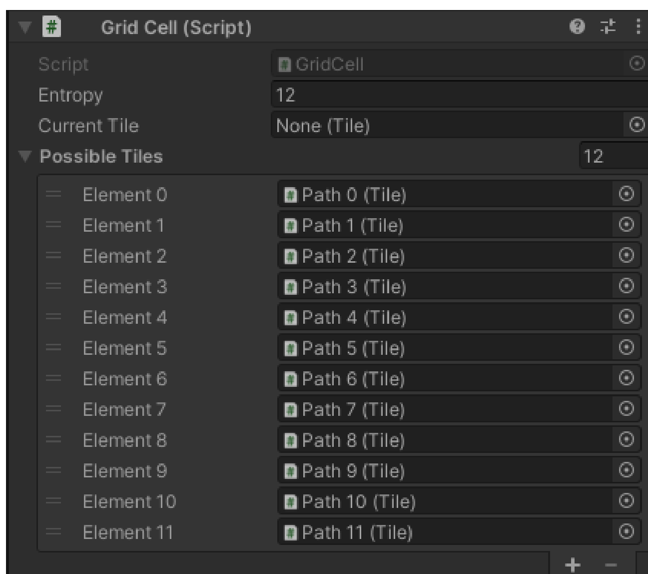

Fig.12: GridCell class inspector

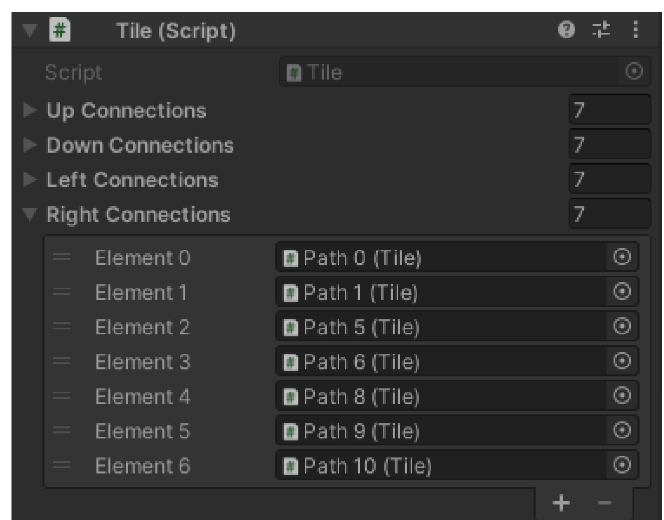

Fig.13: Tile class inspector

The tiles are created with prefabs, each holding the sprite of an individual tile. As with the cells, this has the advantage of each tile being its own object, which enables them to get individually accessed, and assigned classes. To create the connection rules between tiles, a Tile class stores a reference to all possible connections of the current tile (T-connect). Each tile needs to have 4 T-connect lists, to store the fitting tiles for each relative direction: up, down, left, and right.[Fig.13]

Another advantage of having the tiles as prefabs is that it makes them easy to instantiate as a child element of a cell. All the anchors and sizes have already been calibrated in advance, to ensure that the tiles are instantiated in the exact spot to form a seamless map together, without requiring any readjustments in position or scale.[Fig.14]

The grid itself works in a similar way. The cell object is a prefab, which can be instantiated as many times as required, while the layout group automatically arranges them in the grid formation.[Fig.15]

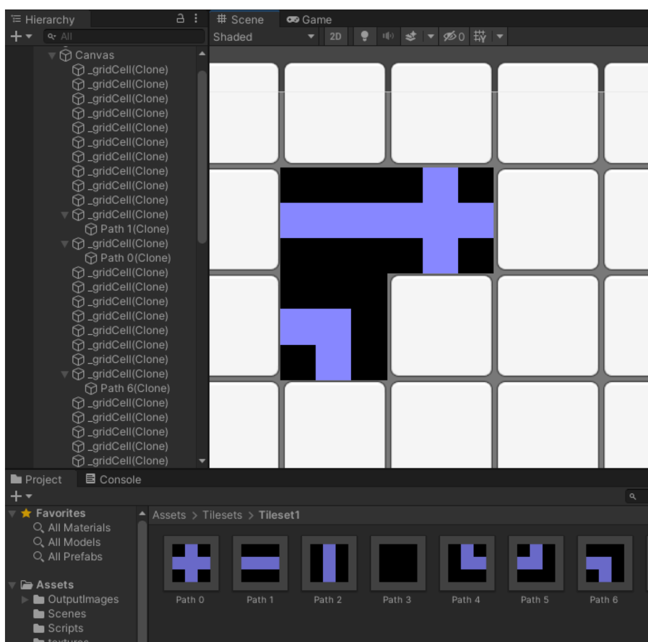

Fig.14: tiles form a seamless map together

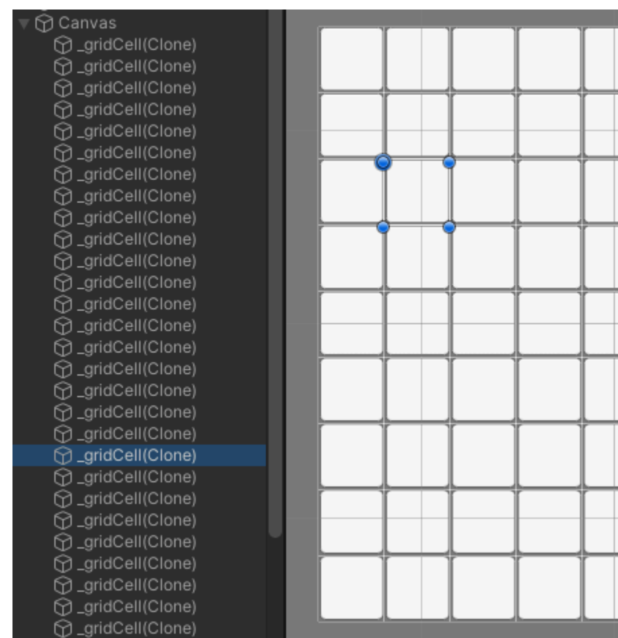

Fig.15: empty grid scene and hierarchy view

Scripting the Algorithm

Now that the scene is prepared, and the grid can be easily built up and filled in, the actual WFC comes into action. This section is separated by the algorithm operating sequences, previously introduced in Chapter 1.2.

Preparation + Initialization

The structure of the grid is represented in a 2-dimensional array. That way, the (x,y) coordinates that locate a cell within the grid, directly correspond to the [x,y] indices of the array. To prepare the grid, a double for-loop is iterating over the array, instantiating the cell prefab at every step, and immediately adding its reference into the current array position.

Each newly instantiated cell is also directly initialized, by inserting the entire tileset, containing a reference of each available tile, into C-possible.

```
for (int y = 0; y < rows; y++)
{
    for (int x = 0; x < columns; x++)
    {
        cells[x,y] = Instantiate(gridCell, transform).GetComponent<WaveCell>();
        cells[x,y].Init(tileset.tiles, x, y);
    }
}
```

Fig.16: Code Excerpt - creating and initializing the grid

Observation

In order to find the cell with the lowest entropy, first a reference value is created, which is initially set to a maximum value, unreachable by the actual entropy values.

Since it is possible for multiple cells to have the same entropy, additionally, a list is created, intended to store all of these same-valued cells. Next, a forEach-loop iterates through the grid array, comparing each entropy to the reference value.

In case of the current cell's entropy being lower than the comparison value, first, the reference gets replaced with the entropy's value. Then, the list gets cleared, to prevent any cells with a higher entropy still being in it. And finally, the cell gets added to the list.

In case of the cell's entropy and the reference value being equal to each other, the cell gets added to the list, without clearing it first.

If the entropy is higher than the reference, nothing happens, and the loop simply moves on to the next cell.

Additionally, there is a special case for negative values. To quickly identify collapsed cells, their entropy is set to -1. This way, they can be easily ignored here, since this method is supposed to only find the next un-collapsed cell.

Once the end of the loop is reached, the list now only contains cells with the same, minimal entropy. One of them is randomly chosen and handed over to the collapse method.

```
List<WaveCell> minCells = new List<WaveCell>();
int minEnt = tileset.tiles.Count;

foreach (var cell in cells)
{
    switch (cell.entropy)
    {
        case -1: continue;
        case var e:int when e < minEnt:
            minEnt = cell.entropy;
            minCells.Clear();
            minCells.Add(cell);
            break;
        case var e:int when e == minEnt:
            minCells.Add(cell);
            break;
    }
}
```

Fig.17: Code Excerpt - entropies being compared

Reset

At the beginning of the collapse method, the current cell gets checked for a contradiction. If its entropy equals 0, the method returns prematurely, and instead, the restart method is called.

There, the grid is set back to its initial state, by destroying all tile instances in the scene and re-initializing each cell with all possibilities once again.

```
foreach (var cell in cells)
{
    foreach (Transform child in cell.transform) {
        Destroy(child.gameObject);
    }

    for (int y = 0; y < rows; y++)
    {
        for (int x = 0; x < columns; x++)
        {
            cells[x,y].Init(tileset.tiles, x, y);
        }
    }

}
```

Fig.18: Code Excerpt - restart method

Collapse

If no reset is required, the collapse method resumes normally.

Out of all tiles in C-possible, one is randomly chosen and instantiated as a child element of that cell. Additionally, its entropy is set to -1, so that it can easily be filtered out in the observation method.

Propagation

The propagation method is called 4 times, once for each relative direction. As an input variable, the method takes one of the T-connects from the tile, which was just instantiated. It also requires the coordinates of the neighboring cell, to which the new information should be propagated.

```
ReduceEntropy(tile.RightConnections, x: cell.x+1, cell.y);
ReduceEntropy(tile.LeftConnections, x: cell.x-1, cell.y);
ReduceEntropy(tile.DownConnections, cell.x, y: cell.y+1);
ReduceEntropy(tile.UpConnections, cell.x, y: cell.y-1);
```

Fig.19: Code Excerpt - propagation call for each direction

After checking whether that cell even exists within the bounds of the grid, and if it has not been collapsed yet, the new possibilities are determined. For that, the intersect method is used, which is already implemented for List-datatypes. This method returns a new list that only includes entries that are present in both lists, T-connect and C-possible, the ladder of which is then replaced by this new list. Afterwards, the entropy is adjusted, by setting it to the new C-possible's size.

Finish

The entire algorithm is contained inside a while loop that checks whether the returned entropy of the observation method is zero or higher.

```
while (GetLowestEntropyCell().entropy >= 0)
{
    Collapse(GetLowestEntropyCell());
}
```

Fig.20: Code Excerpt - main loop

As mentioned, the observation method ignores negative values. However, if it cannot find any positive values, nothing can be returned at the end of the method. This occurs when all cells have collapsed and consequently have an entropy of -1.

In that case, the method returns the first entry of the grid array. Technically, every cell could be returned here, it simply takes the first one of the array; however, the choice here does not matter. Either way, a collapsed cell has to be returned, causing the while loop to detect an entropy of -1, therefore ending it.

Since the map consists of the instantiated tiles, which are already added to the Unity scene during the runtime of this algorithm, the map does not need to be output separately. The script simply ends, and the scene contains the finished map.

2.2 Variations

To make adjustments to the algorithm without discarding the old versions, the latest Map Builder class (MB) was duplicated whenever a new feature was implemented. This way each script is preserved, while the next version can build on top of it. It also made the eventual testing and comparing of the different versions much easier, since the individual scripts could simply be activated or deactivated as needed.

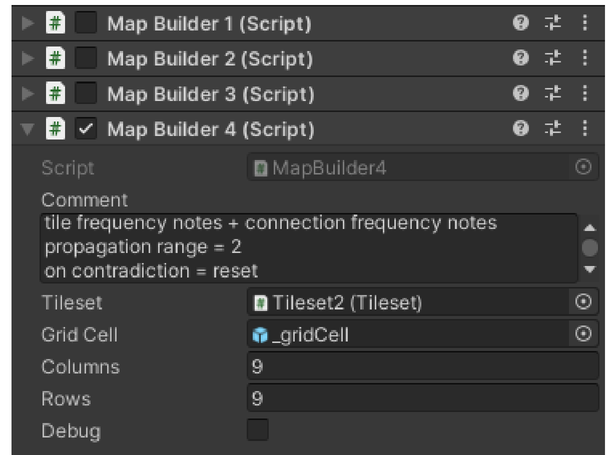

Fig.21: multiple versions of the Map builder script, each containing a short comment, detailing its features

2.2.1 Propagation Range

The base version of WFC only propagates to the direct neighbors of the collapsed cell. However, with more complex tilesets, or very large outputs, this can lead to a lot of required resets. One common way to reduce this fail quote is to propagate to even further cells.

When propagating to the second neighbor (N2), N2-possible needs to be intersected with the valid connection of the first neighbor (N1). However, since N1 does not have a tile yet, it therefore also does not have a concrete list with valid connections yet.

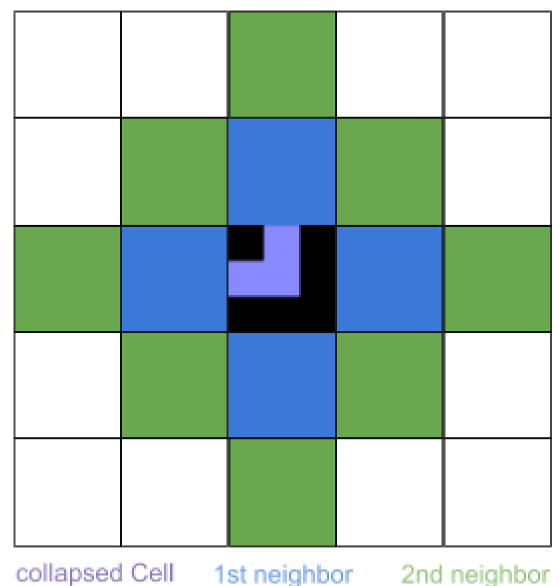

Fig.22: Visualization of neighbor count

To increase the propagation range, all entries of N1-possible need to be combined, to form a joined N1-connect. To achieve that, all T-connect entries of each tile within N1-possible are added to N1-connect. After filtering out all duplicates, it is now possible to intersect N1-connect and N2-possible.

Of course, this sequence has to be done four times per cell as well, to propagate in every direction.

```
foreach (var poss:Tile in cell.possibleTiles)
{
    right.AddRange(poss.RightConnections);
    left.AddRange(poss.LeftConnections);
    down.AddRange(poss.DownConnections);
    up.AddRange(poss.UpConnections);
}

right = right.Distinct().ToList();
left = left.Distinct().ToList();
down = down.Distinct().ToList();
up = up.Distinct().ToList();
```

Fig.23: Code Excerpt - creating a combined C-connect list for each direction

2.2.2 Frequency Notes

Generating a game level requires more room for fine-tuning the outcome than the basic WFC offers. Hence, frequency notes are a helpful expansion, to let a level designer control the look of the output a bit more individually, by adjusting how often a specific tile occurs.

Previously, when a cell was collapsed, one of its possibilities was chosen at random. The frequency notes are used to influence this randomness, by weighting each tile differently.

Each tile now has a new float variable, describing how often it should appear in the finished output. For example, if a certain tile has a frequency of 2, it should be chosen twice as often as other tiles, while a tile with a frequency of 0.5 only generates half as often.

This section of the code is based on the approach the tech-blog gridbugs used for their WFC. [11]

The frequencies are added up to create a value range. If all notes are still on the default value of 1, this range would simply equal the number of tiles in C-possible. If the frequencies vary, the individual ranges that correspond to a specific tile also vary in size.[Fig.24]

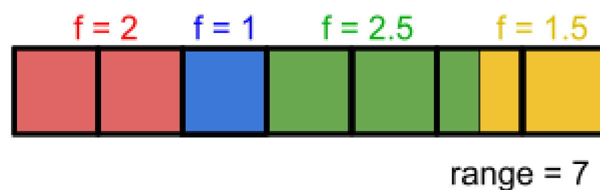

Fig.24: visualization of random range

Next, a random value within this range is chosen. To determine the corresponding tile, the value is compared to each individual tile frequency. While the frequency is smaller than the random value, the frequency gets subtracted from it. Once a tile is found with a higher frequency than the remaining random value, this tile is chosen for the collapse.

```
float rndRange = cell.possibleTiles.Sum(poss :Tile => poss.frequencyNotes);

float rndValue = Random.Range(0, rndRange);
int rndID = 0;

for (int i = 0; i < cell.possibleTiles.Count; i++)
{
    if (rndValue > cell.possibleTiles[i].frequencyNotes)
    {
        rndValue -= cell.possibleTiles[i].frequencyNotes;
    }
    else
    {
        rndID = i;
        break;
    }
}

var tile = cell.possibleTiles[rndID];
cell.SetTile(tile);
```

Fig.25: Code Excerpt - random selection with applied frequency notes

To make this process clearer to understand, the following figure includes an exemplary simulation of this process, based on the random distribution from Fig.24:

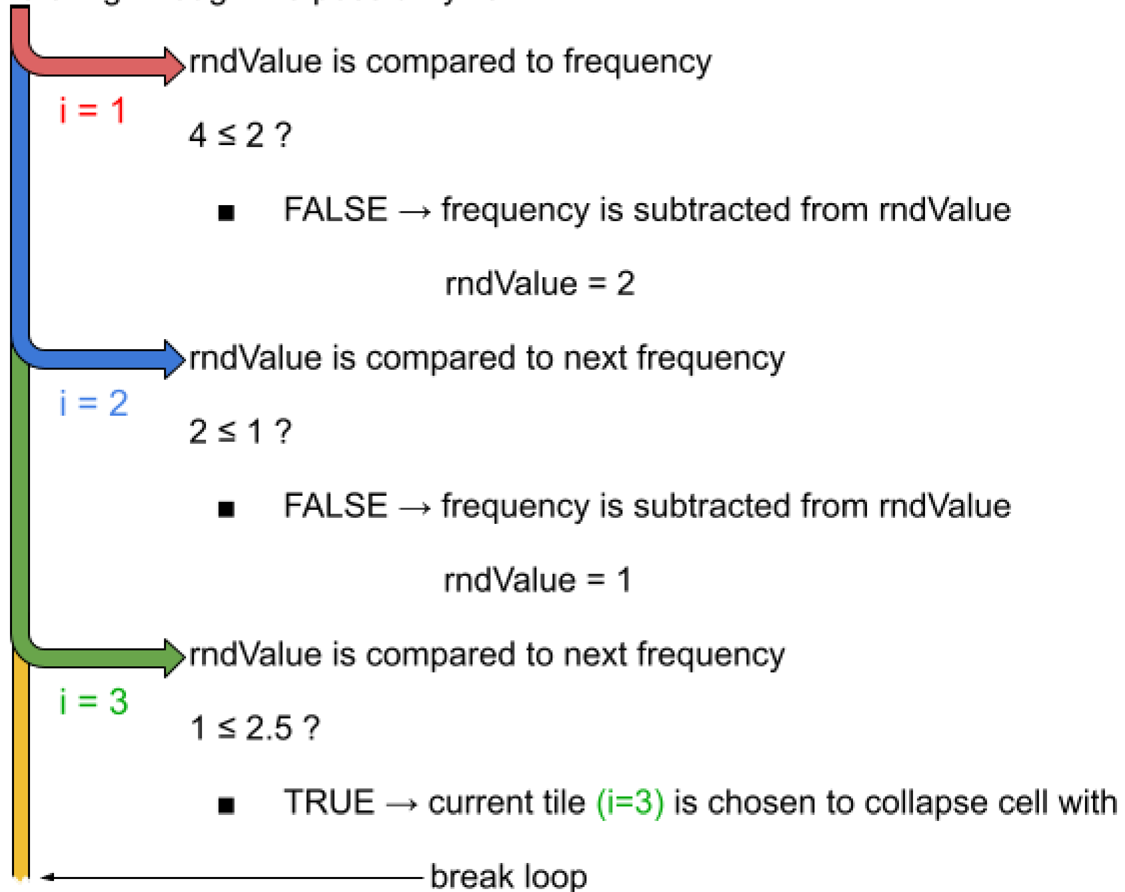

Fig.26: Flowchart - simulation of random selection with applied frequency notes

2.2.3 Connection Frequencies

This expansion is a fairly similar idea to the connection notes. Where previously each tile had a value of how often it should be placed, now every single connection is being weighted as well.

With this adaption applied, it is possible, for example, to have tile A spawn very rarely, except for when it is placed directly above tile B. In this example, the frequency of A itself would be set very low, but the frequency on the connection of B to A would be high.

Implementing this feature is fairly similar to the regular frequency notes.

First, a new field needs to be introduced, which stores the frequency value, for each entry in T-connect.[Fig.27]

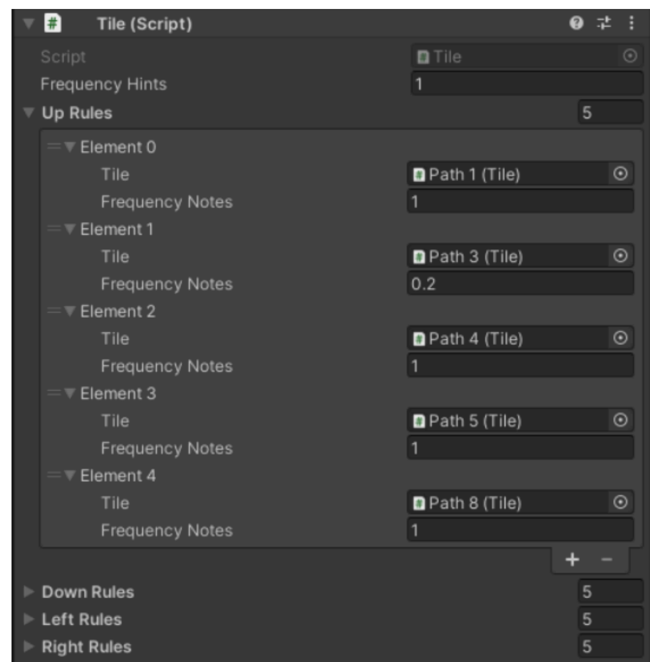

Fig.27: updated connection lists with frequency value

```
foreach (var poss in cell.possibleTiles)
{
    var freq = poss.frequencyHints;
    // multiple frequency notes with connection frequency
    freq *= MultiplyFrequency(poss, x+1, y, Direction.LEFT);
    freq *= MultiplyFrequency(poss, x-1, y, Direction.RIGHT);
    freq *= MultiplyFrequency(poss, x, y+1, Direction.UP);
    freq *= MultiplyFrequency(poss, x, y-1, Direction.DOWN);

    rndRange += freq;
}
```

Fig.28: Code Excerpt - tile frequencies being multiplied with connection frequencies;
the MultiplyFrequency method returns the product of all connection frequency values from the adjacent T-connects

To apply this new value to the calculation, the tile frequency has to be multiplied by the connection frequency of each N1-connect before it can be added to the random range.

Going back to the example, let us assume, that the cell which is currently about to collapse has tile A as a possibility, and borders to tile B. If A has a frequency of 0.1, and tile B has a connection to A with a frequency of 20, then these values would be multiplied, resulting in a total frequency of 2 for tile A in this cell. This value is then added to the random range.

3. Generation of Tiles from Images

As already mentioned, the initial first step of the WFC, in which an input image is analyzed and repetitive patterns are extracted from it, has been skipped in the previous chapter, due to the level-focused scope of this thesis.

However, since it is a crucial part of WFC, we have decided to implement it as well, to see which kinds of applications it can offer to the game-map-creating process.

3.1 Implementation

At the beginning of the image analysis, the input image is split into its raw pixel data, to easily access the individual colors and compare them to each other.

To iterate over the input image, a total of four nested for-loops are used: two of these loops iterate through the (x,y) coordinates of the entire image. However, they do not just go from pixel to pixel, instead, they iterate from tile to tile. Meanwhile, the other two are responsible for looping through the individual pixels within the tile.

For example, if the tiles, which are being extracted from the image, are each 5x5 px in size, then the first pair of loops jump around the image in steps of 5, to the beginning of each tile. Then, the second pair iterates from pixel to pixel within this 5x5 tile, adding each pixel data to a new texture. To prevent duplicates, the texture is first compared to all previously extracted tiles. Only if the texture does not match any of the other ones within this set, it is saved as its own image.

Additionally, each new texture gets an ID assigned. On one hand, this is used for the file name, since each newly saved texture naturally needs an individual name. On the other hand, this ID helps identify its position within the original image.

All tile IDs are stored in a 2-dimensional array, imitating the structure of the image. When a new tile is extracted, its ID is saved at the corresponding position within the array. In cases where a texture duplicate has been detected, the tile is not saved again. Instead, the ID of the matching one, which had already been saved previously, gets noted for the current position.[Fig.29]

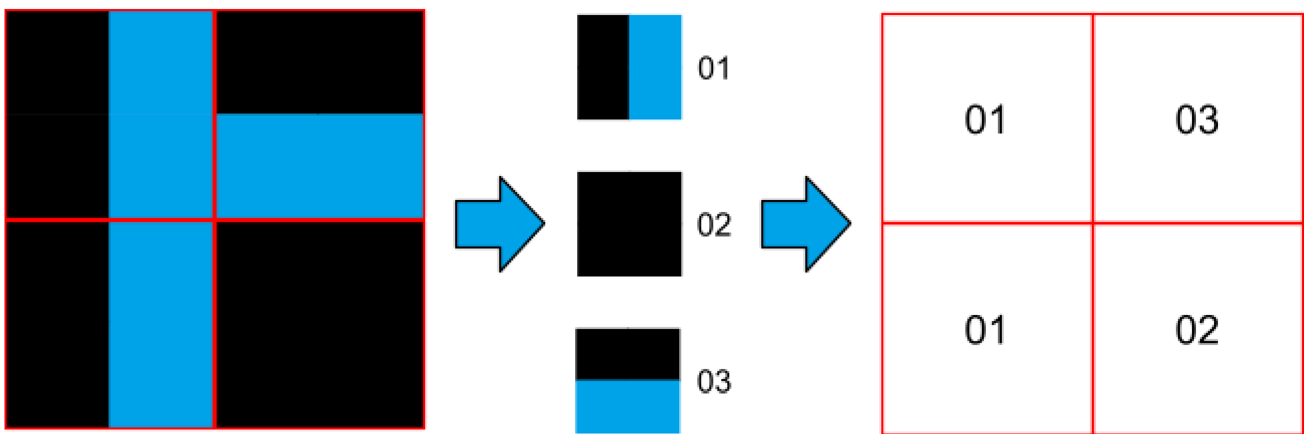

Fig.29: Starting at the bottom-left corner, the image gets split into sections. Each distinct section is saved as a tile and given an individual ID. The 2D array assigns the tile IDs to the corresponding position within the input image.

After all textures have been saved, they get turned into Tile prefabs. For that, a template Prefab is duplicated, and the new texture and ID are added.

To create the connection rules between the tiles, the position of the current tile is looked up in the array via its ID. By in- or decreasing the [x,y] coordinates in the array, it is possible to find the adjacent tiles in each direction and add them to the according T-connect. If a neighboring cell appears multiple times, the frequency of this connection is increased.

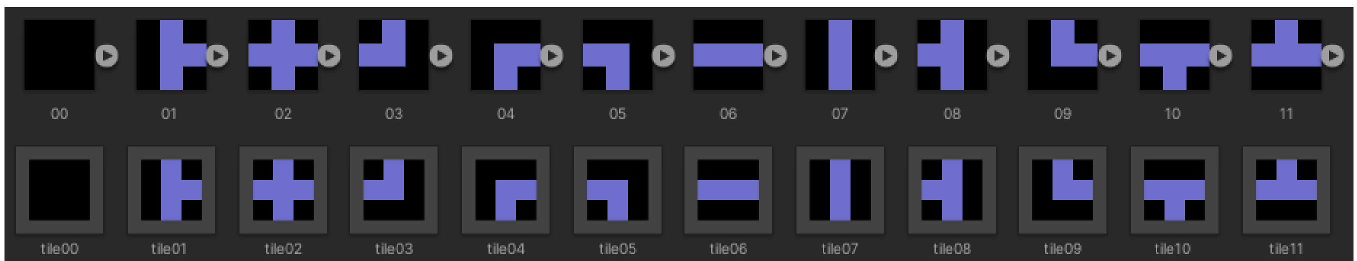

Fig.30: extracted textures (upper row) are turned into prefabs (lower row)

3.2 Variations

Unlike with the Map Builder algorithm, this feature was not first implemented as it is in the original and then altered. Instead, the variations have been implemented immediately, to work with the previously altered MB code.

That means that there are not multiple versions of the Image Analyzer class (IA), as is the case with MB. Instead, there is just the one IA class, already containing deviations from the original. These alterations will be detailed and explained in this chapter.

3.2.1 Separating Image Analyzer and Map Builder

For more flexibility, the IA has been completely detached from the MB and instead was developed in a separate scene, while the original WFC does both steps in the same class. The individual, already extracted tiles are saved as their own files and then put together in one unified tileset.

The tileset is a simple class that contains a list of all tiles.[Fig.31] By assigning this class to a new empty prefab, all created tiles can be easily saved persistently. The Map Builder requires such a class as an input variable, which makes it possible to quickly switch between tilesets. It also eliminates the need of having to analyze the same image again at the start of each generation process, since the tilesets are reusable.

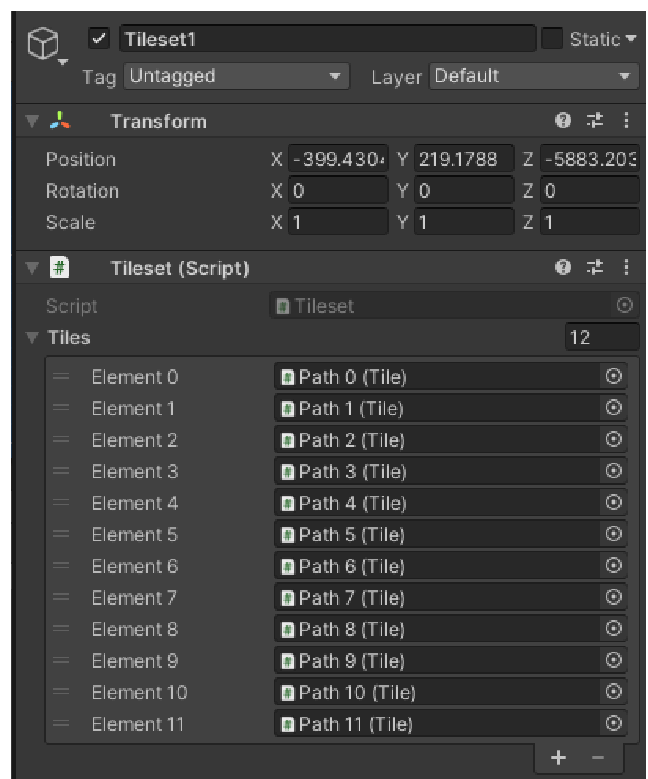

Fig.31: tileset class inspector

Additionally, saving all tiles persistently offers the possibility to manually modify them again, e.g. by adding more tiles, by adding new connection rules, or by adjusting the frequency notes.

3.2.3 Grid Size and Offset

Depending on the resolution of the input image, the dimensions of the individual tiles can vary greatly. Therefore, the size and offset have been made adjustable, by introducing a TileSize and TileOffset input variable.

TileSize describes the measurement of each tile in pixels. Since tiles are typically squares, this is only one singular int value, which is used for both the width and height of the tile.

TileOffset, on the other hand, describes how far away the new tile analysis starts from the previous one. By varying this value, it is possible to control exactly how much the tiles overlap. This, too, is just a singular int value, so there is no difference between horizontal and vertical spacing.

Below is a visualization of the influence these two variables have on each other.[Fig.32]
If the offset has the same value as the size, the tiles all start neatly next to each other, without any overlapping. If the offset is lower the tiles will overlap, thus creating a greater variety of tile options. Setting the offset larger than the size skips over some areas of the input image completely. This can be useful for inputs that already have the tiles separated individually.

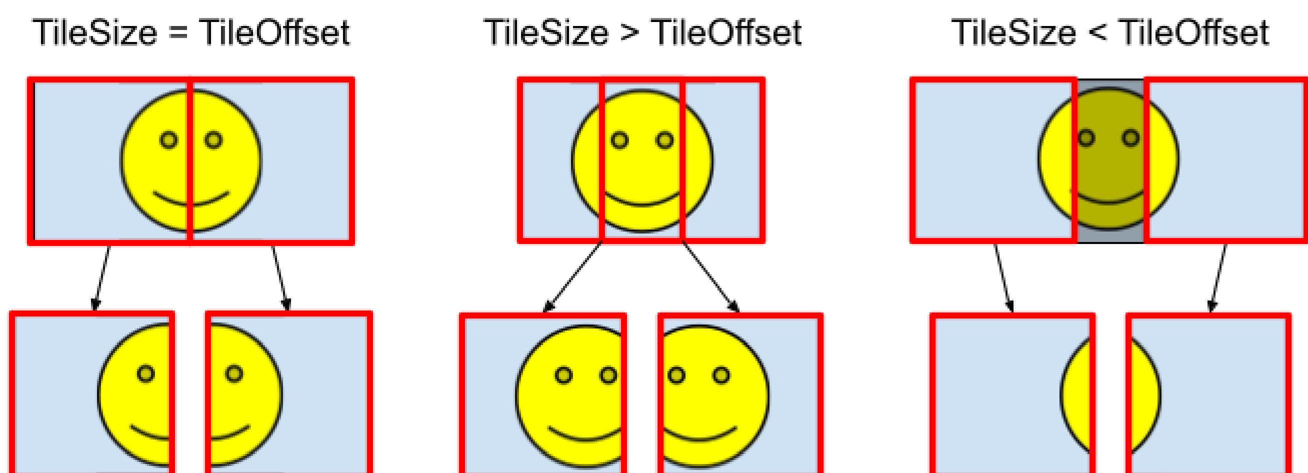

Fig.32: Visualization of correlation of TileSize and TileOffset

3.2.2 Edge Wrap

Making the tile size and offset independent of each, can cause the tiles to not match up correctly with the image size. This would lead to an outer edge of pixels being ignored since they cannot be put into a complete tile. To prevent this, the extraction loops back to the opposite side of the image again, to always form a complete tile.[Fig.33]

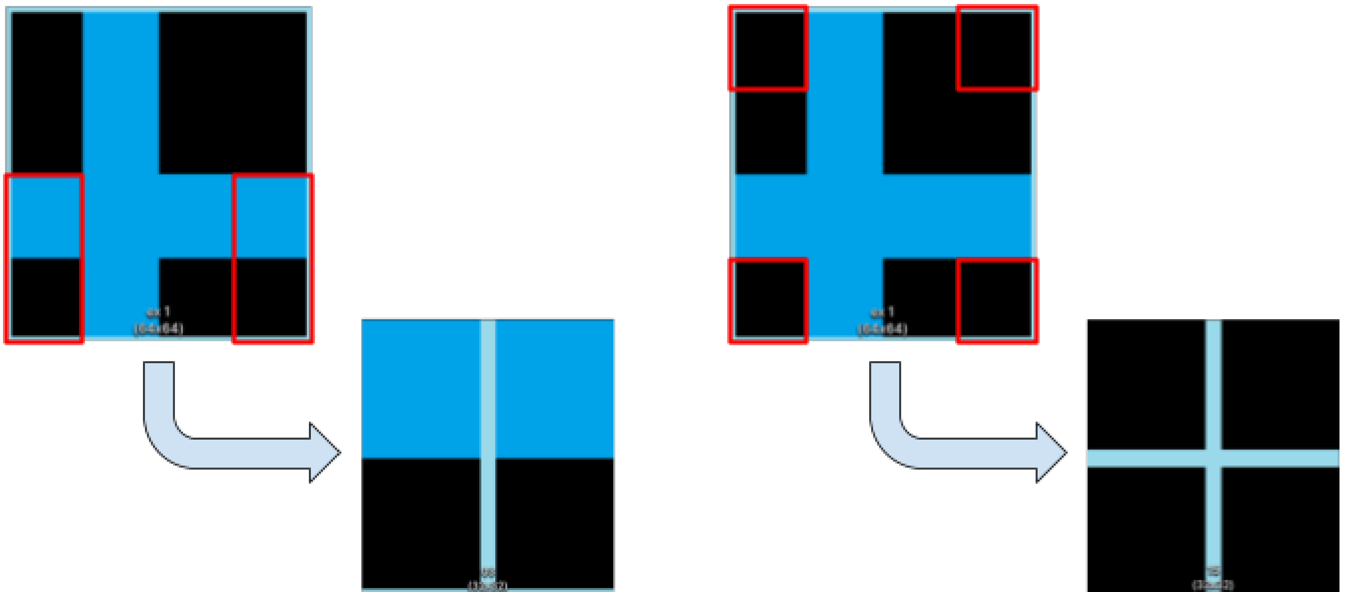

Fig.33: The tile creation wrapping around the input image's edges. The image has been given an outline, to make the wrapping visual in the extracted tiles. The red shows the tile's position in the input image.

Since not all input images necessarily have edges that can fit together like this, this feature was made optional. If the wrap mode has been deactivated, the program will first check if the input size and tile size are compatible. If not, the algorithm already stops there and simply outputs an error message to the console.[Fig.34]

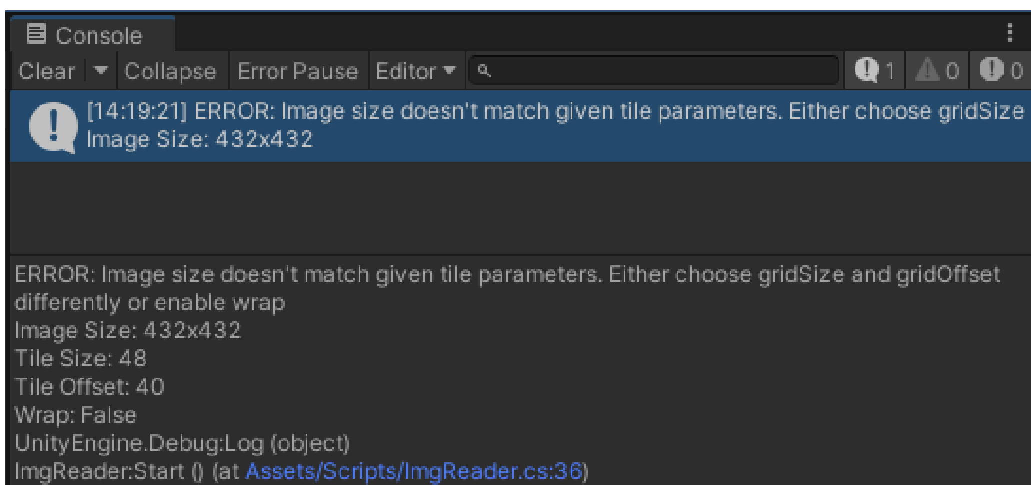

Fig.34: Error message for non-compatible tile-to-image ratios

4. Testing

4.1 Map Generation

4.1.1 Test Conditions

Since the Map Builder (MB) has been separated by its different versions, all of them can be tested individually, to compare their performances to each other. In total there are 4 MB classes, each consecutive class includes all features of the previous version, plus an addition. MB1 is just the base algorithm, without any modification or extensions. MB2 implements an increased propagation range, up to the second neighbor. MB3 has the frequency notes on the tiles themselves. And MB4 additionally includes the connection frequency notes. All variations use a reset in case of a contradiction.

To quantify the scripts' performances, a time tracker was added to the code. The timer starts after the empty grid has been built up, and stops again, once all tiles are successfully placed.

Whenever a contradiction is found, a counter is increased to measure the amount of necessary resets. Additionally, after every reset, a new timestamp is taken. This made it possible to compare times for a single run as well, by only measuring how long the last, successful iteration took. However, if no restarts happen, this last time and the total time are the same.

Fig.35: timestamps taken in code (left) and their output in the console (right)

For the tests two different tilesets are used, each with a different amount of tiles and restrictions. Generally, the two sets can be described as a simpler tileset (Tileset1) and a complex one (Tileset2).

Both sets have been created manually. Every single Prefab was created by hand, assigned a new tile texture, and given all possible connections.

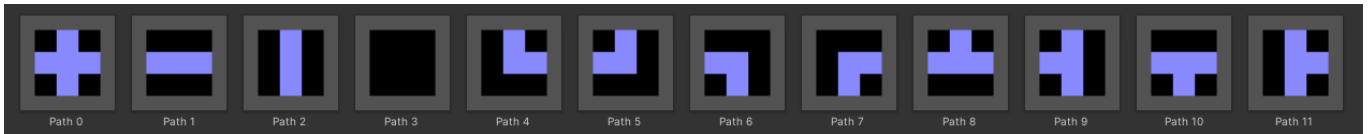

Fig.36: Tileset1

The first tileset has a total of 12 different tiles. Each of them contains only simple lines, bending in different directions.

This set was inspired by the example from the tech-blog gridbugs. [11] However, for the purpose of this project, it has been simplified, to create essentially a binary state for each tile side: either it has a line or it is blank.[Fig.37] The original tileset, on the other hand, had more variations, in which lines could also appear on the outer edges of a tile.[Fig.38]

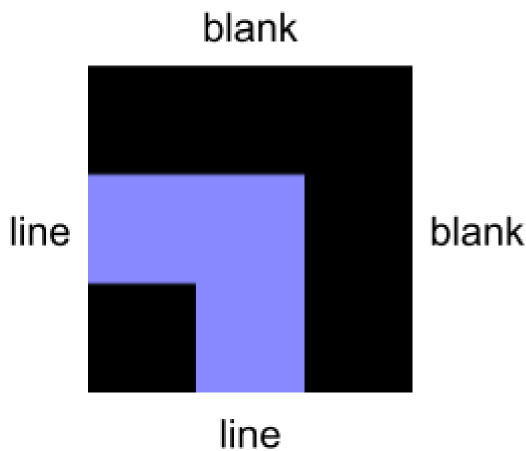

Fig.37: binary option for each tile side

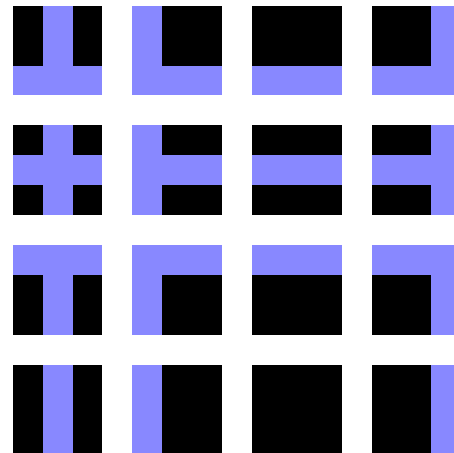

Fig.38: original tileset
[S.Sherratt: Procedural Generation with WFC (2022)]

This modified tileset has the advantage that it includes almost all available combinations of line/blank sides. Because of that, there are almost no tile arrangements possible, which would lead to a contradiction.

This made this tileset very useful for the initial testing alongside the development of the algorithm. It also poses a great comparison point against the more complex tileset.

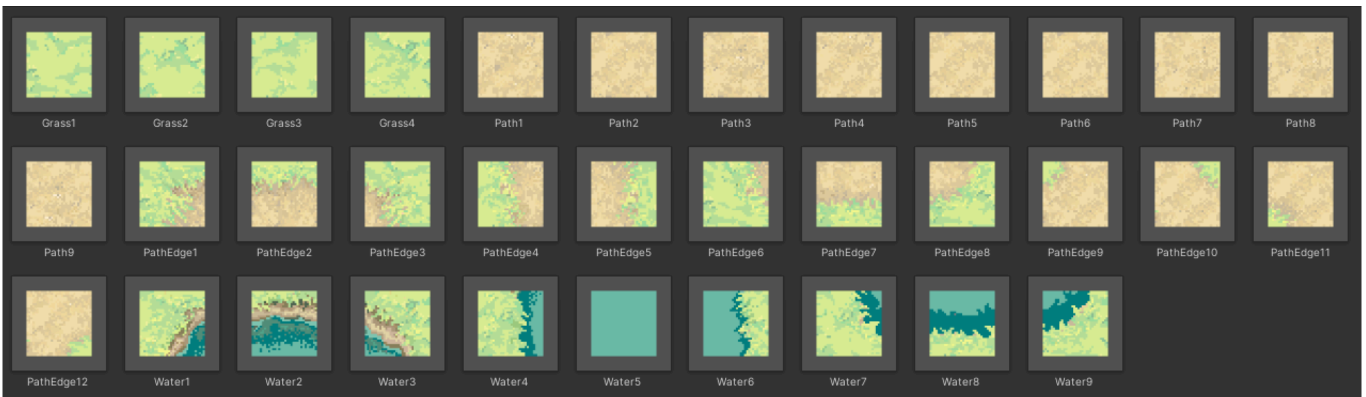

Fig.39: Tileset2

The texture for the second tileset was created by Brackeys, a YouTube channel, which makes freely available Tutorials about different Unity and Game Development topics.[12] The tutorial, in which this tileset was included, explained the use of the Unity-included tilemap tool. Since this tool was not used for this project, the tutorial itself was not used as a source.

This tileset has been intentionally chosen to be a bit more challenging. Unlike Tileset1, the sides of these tiles have more than just two options. Each side is roughly separated into 3 sections, each of which contains either grass, path, or water.[Fig.40] So, instead of each side having 2 options, now there are 27 different combinations for just a single side, leading to an enormous amount of possible tiles. As is apparent from Fig.39, not all imaginable combinations are included in this tileset. This leads to a lot of potential contradictions, which in turn require the algorithm to reset.

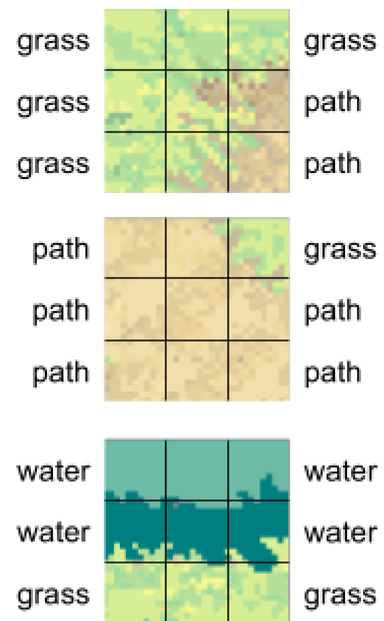

Fig.40: sides are separated into sections, each having 3 different options

The final characteristic that is notable within this tileset, are the multiple variations of the pure grass (Fig.39: Gras 1 - 4) and pure path tiles (Fig.39: Path 1 - 9).

While this does not affect the potential for resets, since all of these tiles have the same available connections, it does increase the overall amount of tiles, leading to higher calculation times.

4.1.2 Test Results

Each code variant is measured 10 times, and the recorded data is portrayed using bar diagrams. The bars display the average of all measurements, while the lines show the range between the minimal and maximal values. Alternative diagrams, plain data tables, as well as additional map outputs can be found in the appendix.

Run 1: MB1

The first test run used the base algorithm implemented in MB1. The output, which is being generated, spans a total of 20x20 tiles. One example map for each tileset can be seen in the figure on the right.[Fig.41]

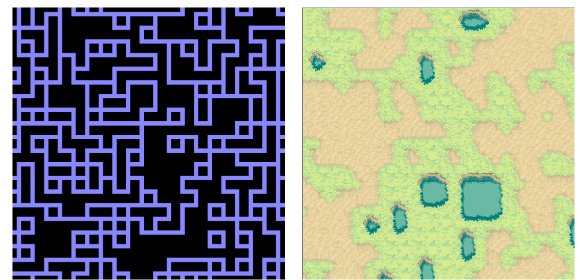

Fig.41: 20x20 outputs of MB1

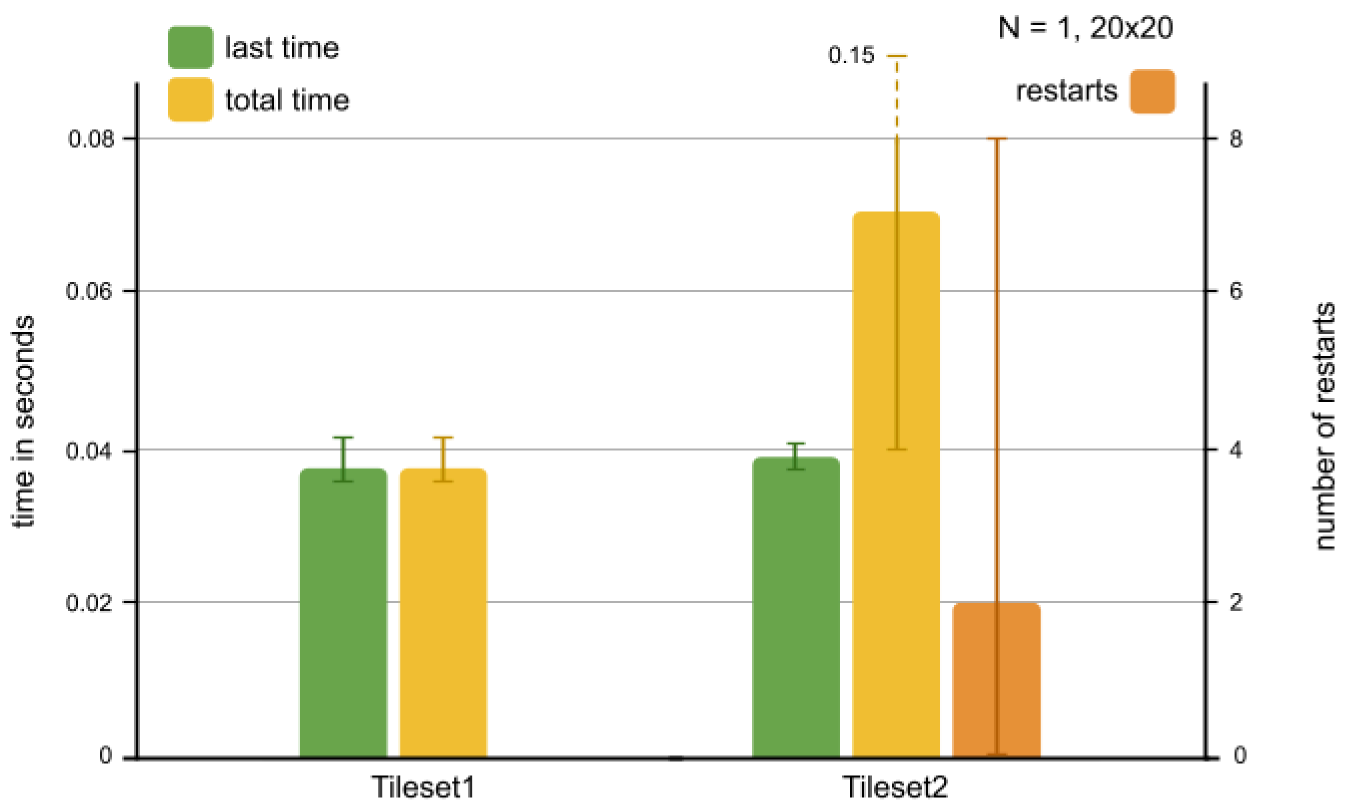

Fig.42: Diagram - Run 1: MB1

Interestingly, Tileset1 does not cause any resets at all, resulting in the total and last time being equal.

Tileset2 on the other hand, regularly runs into contradictions, causing a significantly higher processing time. Additionally, the number of resets for Tileset2 is very inconsistent, ranging from no restarts at all, up to 8. Consequently, the total time varies greatly as well.

When comparing only the last time of both sets, the simpler Tileset1 is a bit faster, however only marginally.

Run 2: MB1 vs. MB 2

When conducting the same test run with an increased propagation range to the second neighbor (N2), the reset rate for Tilset 2 completely drops to 0 as well, rendering the data for resets and last time irrelevant. Therefore, this diagram only compares the total time of MB2 in comparison to MB1.

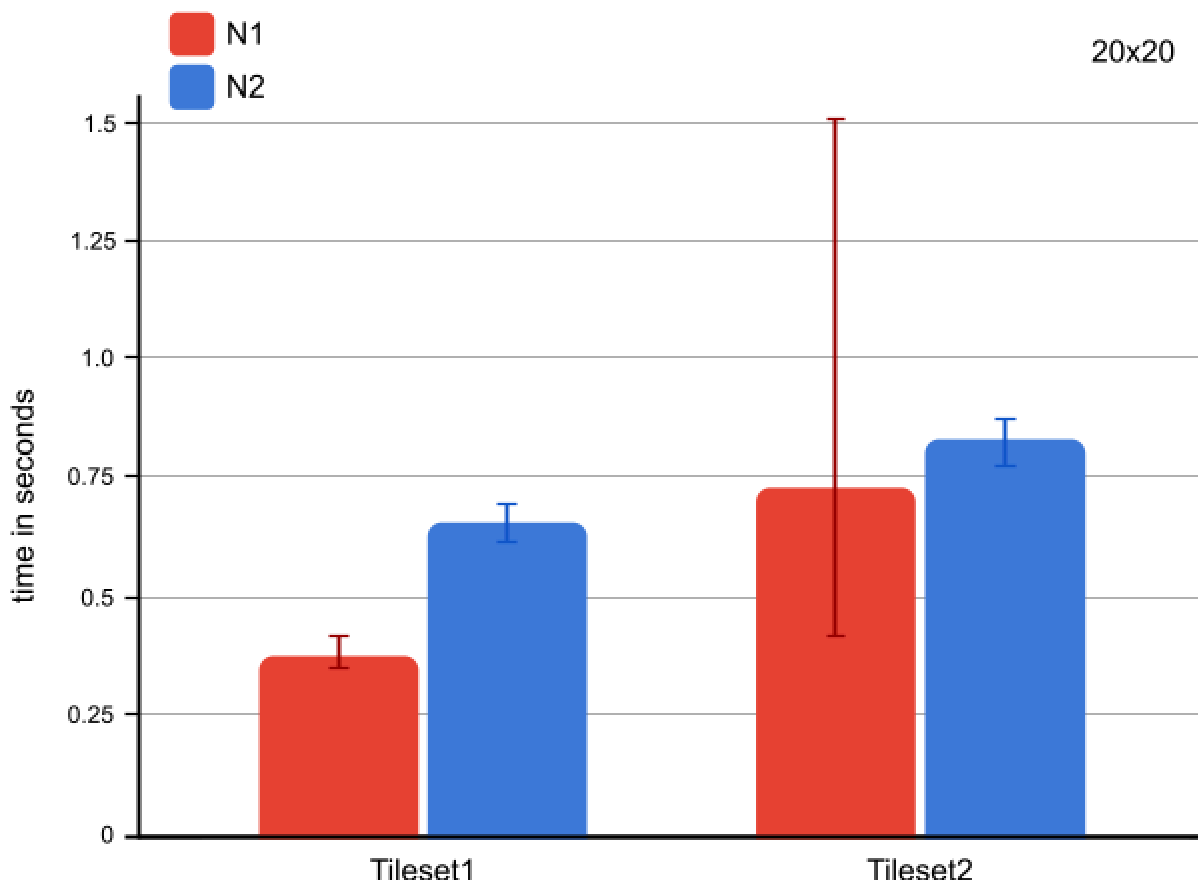

Fig.43: Diagram - Run 2: MB1 vs MB 2

What becomes apparent when looking at the result for Tileset1 is that an increased propagation range results in a higher processing time. This is not a big surprise, since the higher neighbor count requires more calculations to be made at each iteration. Therefore a time increase here is to be expected.

Looking only at the results for MB2, now the time difference between Tileset1 and 2 becomes more visible, whereas the gap between them in the previous run was only minimal, apart from the reset rate. Again, this is not particularly remarkable, since it seems only logical that the bigger tileset, with more complex constraints, requires more processing time.

However, what might come as a surprise is that even though the high reset rate was eliminated, the average time Tileset2 took with MB2 is still higher than with the first algorithm. Of course, that applies only to the average of all measured time, since the readings of MB1 include extremely high fluctuations. So, while most of the best cases of MB1 are better, it is quite unreliable and can cause catastrophic worst cases.

Run 3: MB1; increased Map Size

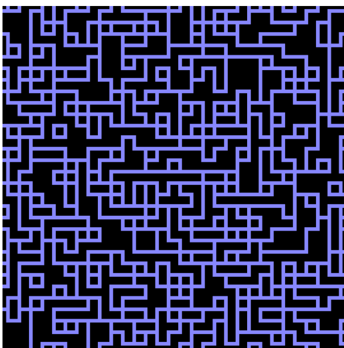

Fig.44: 30x30 outputs of MB2

To further investigate the influence of the reset rate on the processing time, the same two test runs have been repeated with a map size of 30x30.[Fig.44] All other parameters are kept the same, the only thing changing is the output size.

Note, that the following two diagrams have an algorithmic scale, unlike the other one in this chapter. This change was done due to the excessive jumps in the data points, within this test run, which would otherwise severely decrease the diagrams' readability. The same data sets with a linear scale can be found in the appendix.

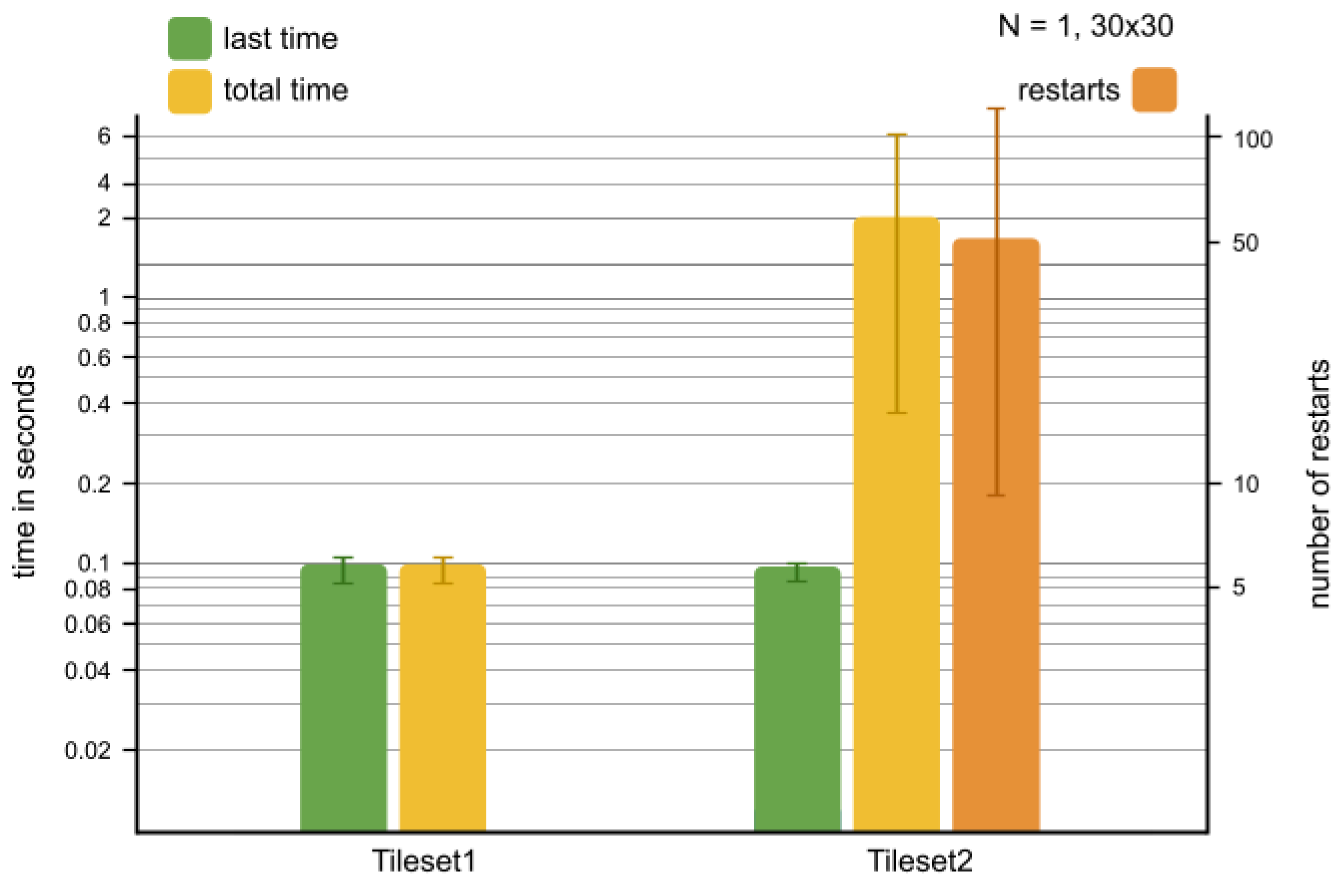

Fig.45: algorithmic Diagram - Run 3: MB1; increased Map Size

With just the simple propagation range, the results for Tilset1 did barely change, apart from a time increase proportional to the expanded size. Meanwhile the reset rate for Tileset2 skyrockets to over 100 at its worst, consequently also increasing the overall time massively.

Run 4: MB1 vs. MB2; increased Map Size

Using MB2 once again completely reduces the resets, even with the increased map size. Now the time difference makes the advantage of the second propagation over the first neighbor quite clear.

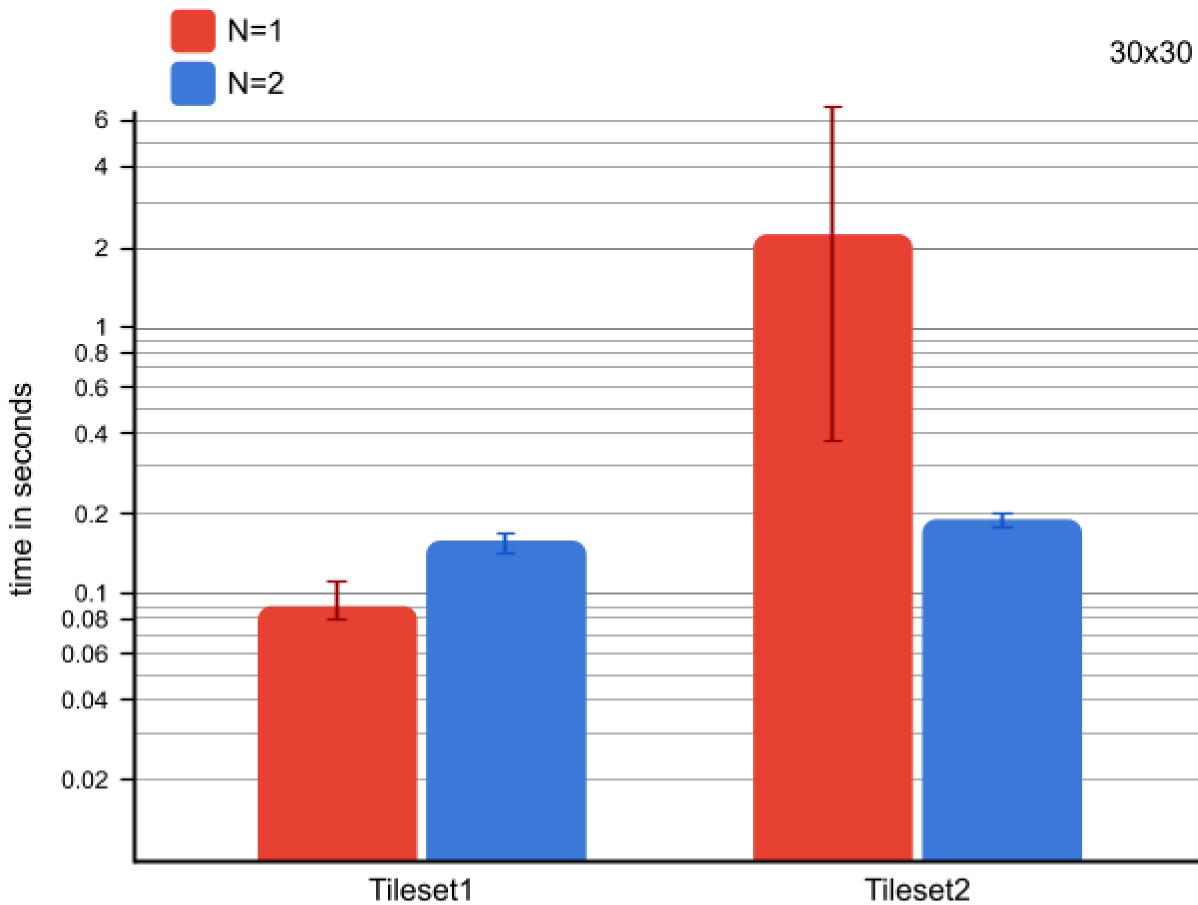

Fig.46: algorithmic Diagram - Run 4: MB1 vs. MB2; increased Map Size

Run 5: MB2 vs. MB3 vs. MB4

The final diagram compares the measurements from MB2, MB3, and MB4. Since 3 and 4 both started out as clones of MB2, they both include a propagation range of N2, causing neither of them to require any resets. Therefore, this diagram only portrays the total time of each algorithm, to see how much influence connection notes have on the performance time.

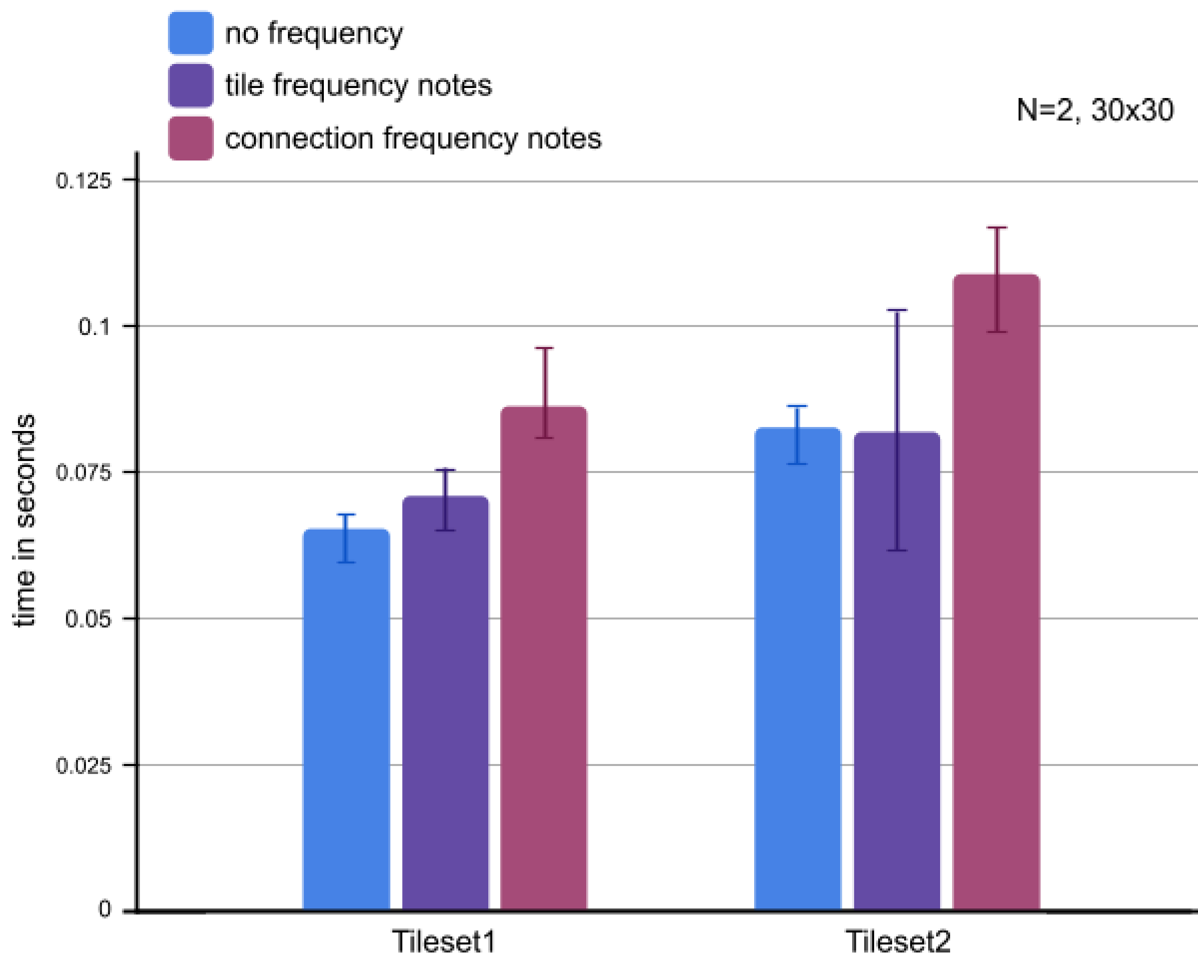

Fig.47: Diagram - Run 5: MB2 vs. MB3 vs. MB4

As is apparent from the data, each consecutive class required more time than its predecessor. Interestingly, the tile frequency notes only cause a relatively small time increase. Meanwhile, the connection frequencies caused a much larger jump.

In hindsight, neither of these revelations is particularly surprising, since the number of connections is significantly higher than the number of tiles. Therefore it is understandable that the increased amount of calculations also increases the time accordingly.

However, one thing that stands out is the very large variation Tileset2 has with MB3. This anomaly does not seem to have a clear explanation, and might just be a statistical outlier.

4.1.3 Test Evaluation

Comparing the first and second MB against each other clearly shows the effect of an increased propagation range on the algorithms' performance. While it can reduce the reset rate severely, it can also lead to a time increase.

On one hand, because of the extra step of iterating through N1-possible, to create the new connection list. Additionally, there is not only one N1-connect that has to be created, since one is needed for each direction.

On the other hand, increasing the propagation range naturally increases the number of cells that are propagated to exponentially.[Fig.48]

Both of these are responsible for drastically increasing the required calculating time.

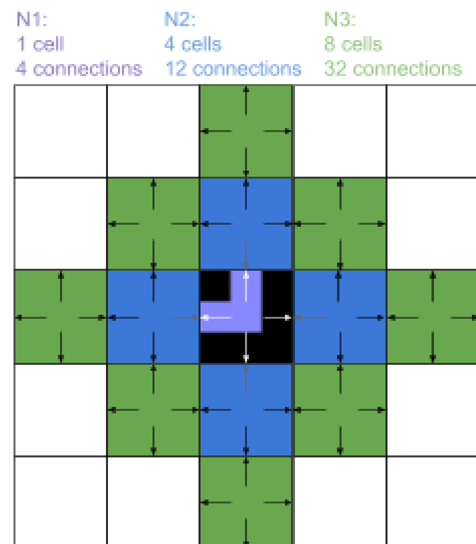

Fig.48: propagation range increases cell count

While it is possible to increase the range even beyond N2, this adaption should be considered thoroughly.

The N2 propagation still has the advantage of bordering directly on the just collapsed cell, which means they only need 3 propagation runs each, since the intersection gets aborted prematurely if the target cell is already collapsed. After all, that cell already has a tile, and thus does not need any more possibility calculations.

If the propagation range would be expanded, the farther neighbors would not necessarily have this advantage. Each of their adjacent cells could potentially still be empty, which forces them to go through all four propagation runs.

When implementing a higher propagation range, the time increases should be thoroughly waged against the reduced reset rate. During our test runs, the N2 propagation already led to the complete prevention of any restarts. Therefore, the decision was made not to implement a higher neighbor count.

The results of test run 5 show clearly how each new feature and addition requires more processing time. Therefore, every extension has to be investigated thoroughly, whether they are truly necessary and needed for the intended application of the algorithm.

Alternatively, it might be a good idea to have multiple versions of the algorithm available, each specialized for a different focus. This way it is possible to choose the most productive variation for the current task. For example, if you have a rather simple set, similar to Tileset1, and you already know you want to customize the look of the outcome quite finely through frequency notes, then omitting the N2 propagation might be worth considering. Otherwise, if you know your tileset is more complex, you definitely should use a higher propagation rate, which perhaps goes even beyond N2.

4.2 Tile Generation

4.2.1 Test Conditions

To test the accuracy of a generated tileset in comparison to a manually created one, one of the generated maps,[Fig.49] made during the earlier test runs, was used as an input image for the Image Analyzer script.

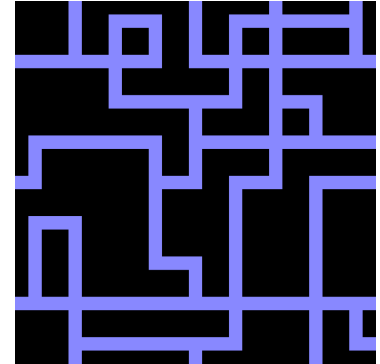

Fig.49: map used for tile generation

This map is 9x9 tiles large and contains each of the 12 different tile options of Tileset1 at least once. In total, the image is 432x432px, meaning each original tile takes up 48x48px. In the first test, the grid size and offset are both set to a value of 48, in an effort to replicate the original tileset as accurately as possible.

Afterwards, the generated tileset is put back into the MB script, to generate a new map from it, which is then compared to the maps created with the manual tileset.

In the second test, the input map remained the same, and the grid size also stays at 48px. However, this time, the offset is reduced to 36px, creating an overlap of ⅓ of a tile.

The purpose of this test is not to replicate the original tileset, but instead, to see what sort of variations different input variables can lead to, specifically what an overlapping tile analysis might result in.

This newly generated tileset is then used as input for the Map Builder as well.

4.2.2 Test Results

Run 1: Exact Replication

When recreating the original tileset as closely as possible, all tiles that were originally used to create the input image,[Fig.50] are also present in the newly generated tileset.[Fig.51]

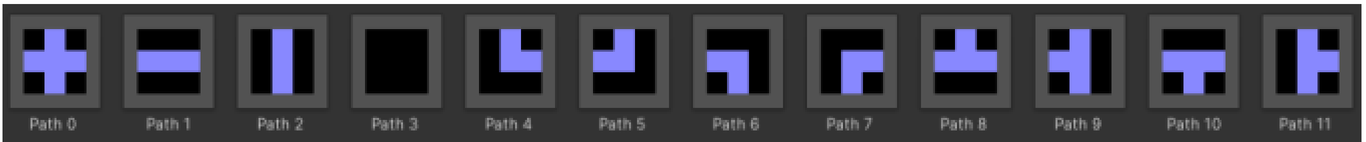

Fig.50: manually created tileset

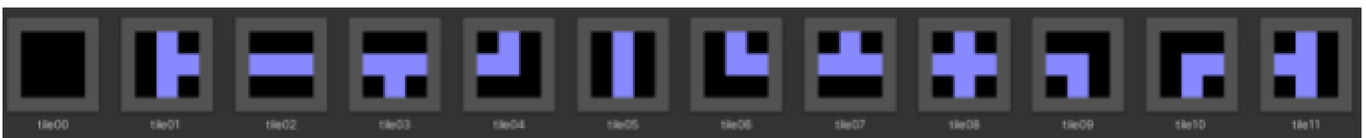

Fig.51: tileset generated from map

At first glance, this might lead to the assumption that the tileset replication worked very accurately. However, while the tiles themselves may look quite similar, the connection rules between them differ quite a lot. Below are the t-connect lists of both cross-shaped tiles (Path 0 of the manual tileset, and tile08 of the generated tileset).

Fig.52: connection rules of a manual tile (left) and a generated tile (right)

As can be seen, the generated tiles have fewer connections available than the original ones. Since the 9x9 map, used as an input, could not include every possible combination of tiles, naturally not all valid connections can be extracted from it.

In some cases, tiles even had zero valid connections for a specific direction. Since the input map did not have matching edges, the edge wrap mode had to be turned off for this test. However, this means that tiles, which only showed up on the edge of the map, had no adjacent tiles which could have been registered as a connection.

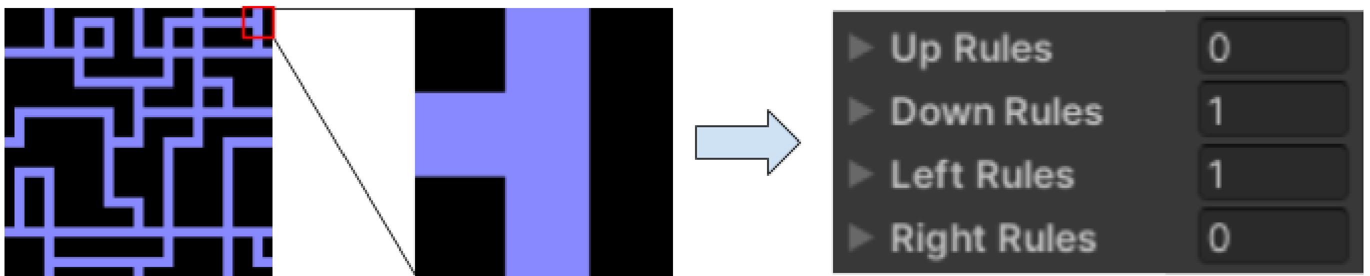

Fig.53: corner tile is missing connections

Due to these limited connection options, this set causes many contradictions when using it as an input for MB1, resulting in an average of 40 resets per map. Even with the second neighbor propagation, the algorithm still required one reset on average per map. Interestingly, when using this tileset in combination with MB4, the connection frequencies almost completely reduced the reset rate, resulting in only 1 in 10 maps being reset.

Nevertheless, each algorithm variation still managed to output a valid map eventually. Some of these outputs, and their different looks, can be seen in the figures below[Fig.54-56]

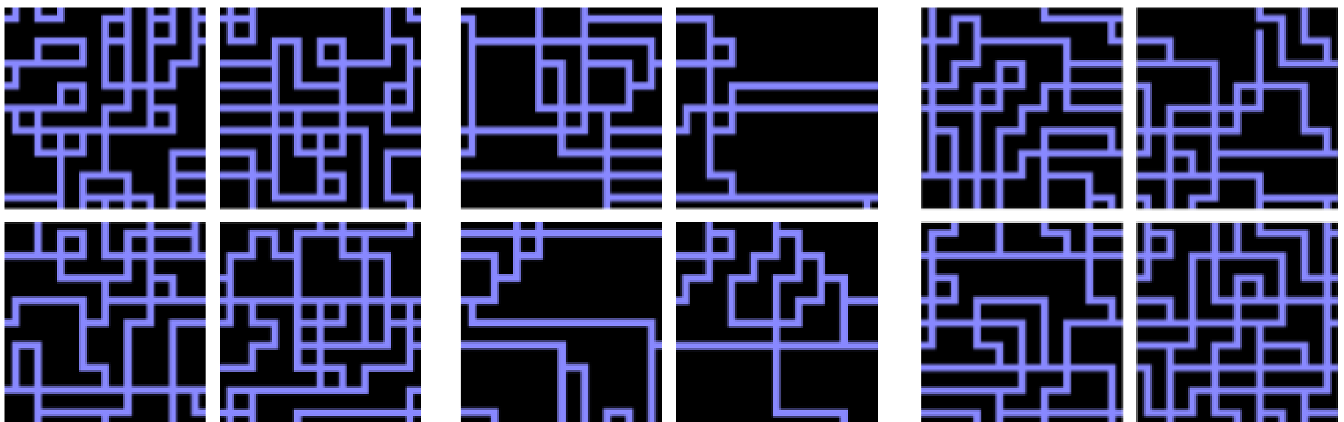

Fig.54: maps from manual tileset **Fig.55:** maps from new tileset in MB4 **Fig.56:** maps from new tileset in MB2

The first set of maps (left) are generated using the manually created Tileset1 as an input for MB4. The map used to extract the tiles[Fig.49] belongs to this set.

The next map set (middle) is the result of the newly generated tileset. These maps look quite different, having much more blank space and more long lines.

Finally, the third set (right) uses the generated tiles again, this time, without applying the frequency notes.

While this set looks fairly similar to the one made with the manual tiles, the pattern here seems a bit more repetitive, e.g. this "8" shaped pattern appears in all maps relatively unchanged.[Fig.57]

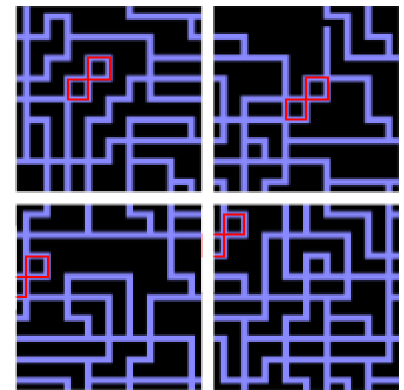

Fig.57: repetitive patterns

Run 2: Overlapping Tiles

When creating the tiles with an overlap, the Image Analyzer produces a much larger pool of possible tiles.[Fig.59] All tiles which are highlighted in blue, can also be found within the original tileset.

Interestingly, not all of the original tiles also exist in this new set. While most of them have been generated again, Tile 8[Fig.58] of the original set is missing.

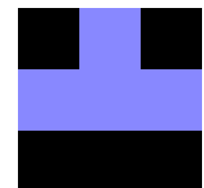

Fig.58: missing tile

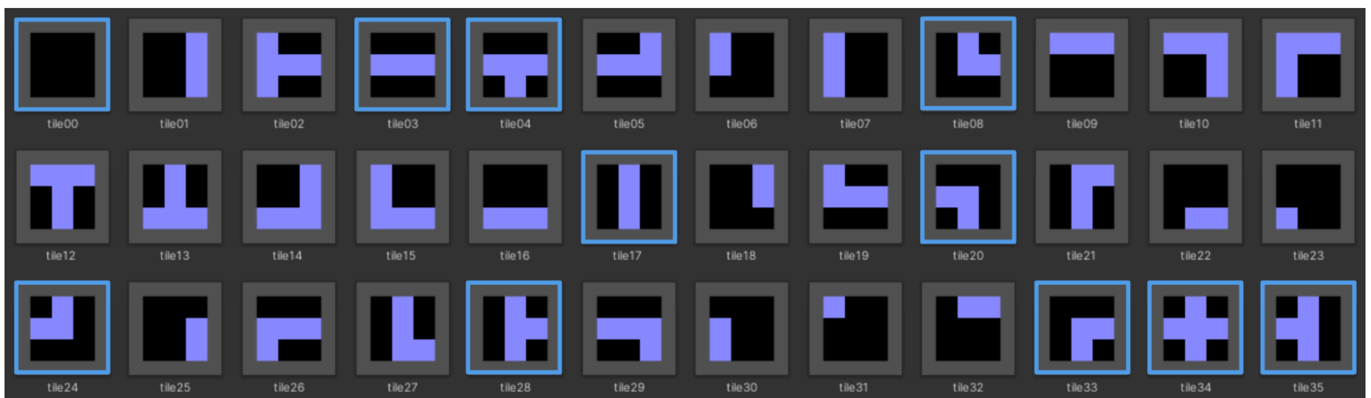

Fig.59: tiles generated from map with partial overlap

Generating a new map with this tileset results in very interesting looking results,[Fig.60,61] which definitely still remind of the original map, and yet offer a lot of new variation.

However, the same problem as with the previous test run remains: the input image only contains a limited amount of tile combinations, causing frequent blank spaces and more repetitive patterns.

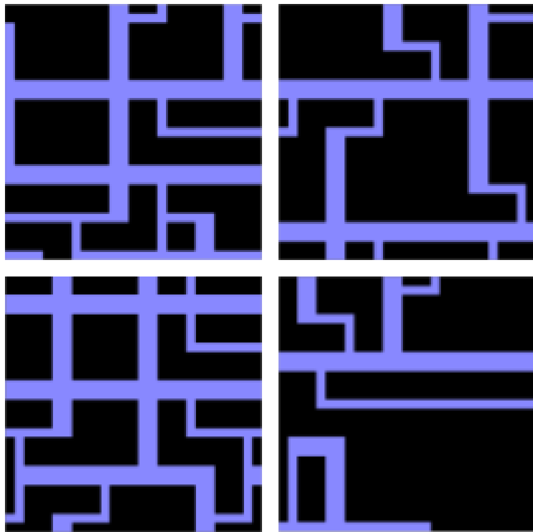

Fig.60: maps from overlapping tileset in MB2

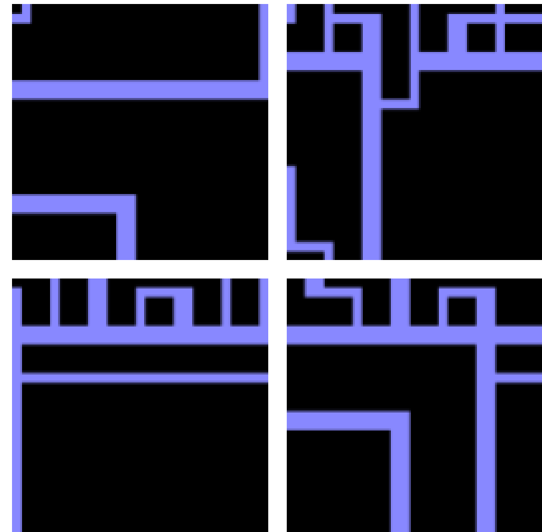

Fig.61: maps from overlapping tileset in MB4

Using just the first neighbor propagation of MB1 is not possible with this tileset, since the overlapping tileset causes a lot more contradictions than the previous one, leading to such a high amount of resets that we had to eventually terminate the program after several minutes runtime.

Luckily, it is possible to generate a finished map, by using the increased propagation range. Still, this leads to some resets, with an average of 10 resets for MB2, and 2.4 resets for MB4. As with the previous test run, the frequency notes help to lower the reset rate.

Since the overlapping tiles lead to such a different look, the resulting maps cannot be directly compared to the manual tileset maps,[Fig.54] causing this run to lack a proper comparison value made from a more complete tileset.

Therefore, we decided to duplicate the overlapping tileset and modify it, by adding all possible connections for each tile manually. The resulting map[Fig.62] still shows the varied patterns present in the previous two map sets. However now, the patterns appear less rigid and have less blank space between them.

While the increased number of connections available for each tile reduces the resets significantly, the high amount and variation of tiles still lead to some contradictions, even when using the second neighbor propagation. However, now this rate is less than 1 reset per map.

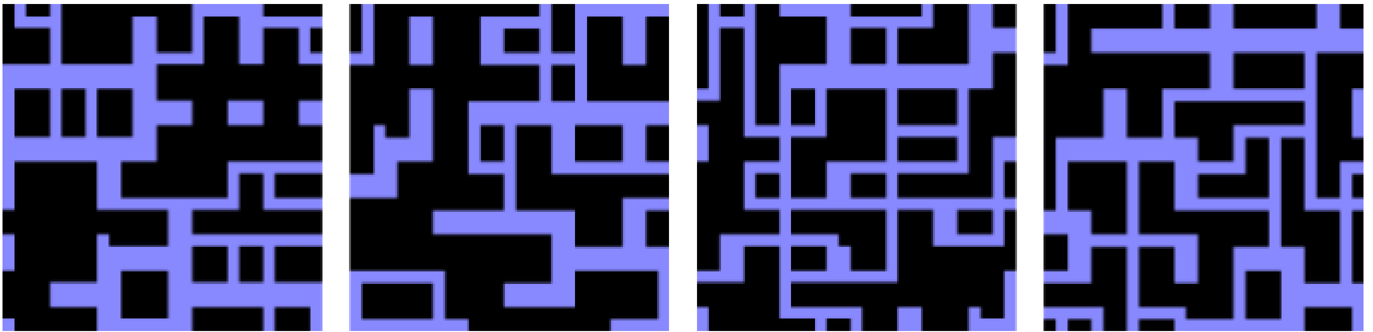

Fig.62: overlapping tileset with modified connections in MB4

4.2.3 Test Evaluation

While the generated tilesets hold some resemblance to the manually created ones, they seem to be less effective in their map generation, caused by incomplete tile connection lists. It appears that it is quite difficult to create an input image that includes all possible neighboring variations, from which all possible connection rules could be extracted.

However, the tile extraction still made the creation of a new tileset significantly easier and faster than creating all tiles by hand, since the prefabs are already generated, and equipped with the correct textures. That means this algorithm can be used to create a base tileset, which then can be manually adapted even further.

Especially interesting is this tool when it comes to creating variations of the original tileset, by extracting the tiles with some overlapping. Again, the resulting tileset is just a base, which still requires some enhancement, before it can be used effectively.

Overall, the Image Analyzer is not perfectly suited for game map creation on its own. However, with some manual adjustments afterwards, this tool can still be a helpful addition, making the tileset creation a bit faster and easier.

5. Conclusions

After implementing just some of the various thinkable expansions to the WFC algorithm, it becomes clear how much possibility lies within it.

WFC is perfectly suited to put a tileset together to form a map. Its constraints-solving approach makes it capable to create levels significantly faster and with less effort than a manual map building would require.

Additionally, certain extensions to the algorithm, like the frequency notes, allow for more flexibility and control of the outputs' individual look.

One of WFC's greatest limitations in this process is the integrity of the tileset. The complexity, time demand, and fail rate of the entire algorithm depend on what kind of tileset is used, how it looks, and what it wants to achieve. Luckily, the high adaptability of WFC is equipped to handle all kinds of characteristics and requirements a tileset might bring with it. More restricted connection rules, for example, can be easily combated with an increased propagation rate.

However, the quality of the tileset itself is almost as crucial for the outcome as the algorithm. Even the farthest propagation and most detailed frequency adjustments will not be able to create much variety from a severely limited, incomplete tileset.

Creating a good set, however, is more complex than it is with originally intended texture replication. While extracting tiles from an example image can be helpful and make the entire tile-creation process a bit faster, a good tileset definitely requires manual enhancement to take full advantage of everything WFC has to offer.

5.1 Limitations and Revisioning

After reviewing the work done for this project, a few points came up, which should have been handled differently or which could be improved upon. Unfortunately, the time constraint did not allow us to correct all of these oversights, and redo the according test run which would be influenced by the changes. Therefore, this chapter outlines what kind of miscalculations happened, and how they can be or have been corrected.

One of the biggest coding issues happens right at the start of the WFC class. As already explained previously, the entire algorithm is being kept alive using a while loop, which checks if all cells are collapsed yet. The method GetLowestEntrophyCell returns a cell, which is then immediately used in the if-condition check. However, since this cell needs to be accessed again, to hand it over to the Collapse method, GetLowestEntrophyCell is then called a second time, prompting it to compare all entropies again, just to return the same result.

```
while (GetLowestEntropyCell().entropy >= 0)
{
    Collapse(GetLowestEntropyCell());
}
```

Fig.63: Code Excerpt - double access of GetLowestEntrophy Class

Obviously, this is inefficient and can be easily avoided, by saving the method's return value as a variable, which is then accessed again.

Additionally, the GetLowestEntrophyCell method treats all cells with an entropy of 0 as regular cells. Only in the collapse method, the contradiction is detected and a reset is initialized. However, it would be much more efficient to immediately restart the map, as soon as a single cell with an entropy of 0 is found.

```
switch (cell.entropy)
{
    case -1: continue;
    case var e:int when e < minEnt:
        minEnt = cell.entropy;
        minCells.Clear();
        minCells.Add(cell);
        break;
    case var e:int when e == minEnt:
        minCells.Add(cell);
        break;
}
```

Fig.64: Code Excerpt - entropies of 0 are not treated separately

Luckily, both of these issues were easily fixed, resulting in the classes MapBuilder1_2 and MapBuilder 2_2. Both classes are duplicates from the previous MB1 and MB2, with the new fixes implemented.

Unfortunately, repeating all test runs from Chapter 4.1 was not possible in the remaining time. Instead, a new test run, comparing the performance time of the revisited classes, shows the potential time savings these improvements can achieve.

20x20

min.	Tileset1				Tileset2	
max.	MB1	MB1_2	MB2	MB2_2	MB2	MB2_2
	0.037	0.042	0.067	0.062	0.079	0.079
	0.042	0.036	0.078	0.060	0.092	0.083
	0.037	0.040	0.063	0.061	0.106	0.077
	0.041	0.037	0.067	0.068	0.097	0.076
	0.049	0.040	0.062	0.061	0.079	0.076
	0.041	0.037	0.066	0.064	0.105	0.077
	0.038	0.040	0.085	0.060	0.081	0.076
	0.044	0.041	0.070	0.060	0.089	0.076
	0.057	0.036	0.070	0.064	0.076	0.079
	0.042	0.036	0.062	0.071	0.079	0.072
ø	0.043	0.039	0.069	0.063	0.088	0.077

Fig.65: Table1 - test run after corrections

Tileset2, 20x20

min.	MB1			MB1_2		
max.	last time	total time	restarts	last time	total time	restarts
	0.039	0.158	5	0.041	0.054	1
	0.043	0.075	2	0.039	0.082	4
	0.103	0.277	3	0.042	0.086	2
	0.039	0.065	2	0.048	0.126	5
	0.043	0.127	4	0.046	0.086	1
	0.040	0.087	2	0.038	0.107	6
	0.044	0.055	1	0.043	0.056	2
	0.040	0.152	5	0.042	0.191	9
	0.041	0.041	0	0.039	0.070	1
	0.040	0.088	1	0.038	0.128	5
ø	0.047	0.113	2.5	0.042	0.099	3.6

Fig.66: Table2 - test run after corrections

Another coding mistake, which was only detected at the end of this project, happens within the IA class. Here, the x and y coordinates in some of the loops and accesses seem to be accidentally switched. Since this mixup seems to be consistently happening throughout the entire class, the cause of this issue could not be located in time.

Luckily, this mistake does not influence the output of the class itself, since the two coordinates seem to be switched in enough places to balance each other out.

The issue only occurs when using input images that are not square-shaped, aka. inputs whose height is different from their width. Since the x and y values of these images are not the same, they eventually lead to an out-of-bounds error. Luckily, the inputs, used in the test phase, were all squares, therefore this mistake did not influence the test results.

Nevertheless, it is important to keep in mind, when reviewing this project's code, that rectangular input cannot be used.

One aspect that caused a lot of trouble was turning the extracted tiles into reusable prefabs. While it is quite easy to do so manually, creating and modifying prefabs at runtime, often causes them to save incorrectly and discard most of the changes again, once the play-mode is exited.

We tried various approaches to save the prefab changes persistently, unfortunately though, we could not find a reliable solution. We were only able to create certain workarounds, e.g. creating one superfluous tile prefab, which is then immediately deleted again, which somehow prevents all previous prefabs from discarding their assigned textures again.

When it came to saving the contents of all T-connect lists, this little trick did not work, unfortunately. The only option we were able to find, that prevented all tile prefabs from discarding all connections, is clicking on one of the tile prefabs within this tileset, while still in play-mode. Opening even one of the prefabs apparently causes all of them to update again so that the connection stayed saved even after the play-mode is exited again.

Of course, this is not a viable solution to the problem, but it is the only fix we could find in the available time. We suspect that it is possible to first create all complete tile class

objects and only save each of them as prefabs at the very end of the algorithm. However, this would require restructuring the entire class, which the time constraint did not allow for, especially since there is no guarantee for this approach to actually work.

Therefore, for the time being, it is important to remember that the tiles must be manually accessed once while still in play-mode. The class outputs a warning to the console, reminding the user of this fact.

Due to a similar structuring mistake, when the individual tiles are extracted from the image, they are not tile objects yet, but only texture files. This means they do not have any of the tile variables yet, making it impossible to increase the tile frequency whenever a duplicated tile was found.

Solving this issue would require the tile textures to be turned into tile objects way sooner than they currently are. This issue was also noticed too late, where a complete restructuring of the class would have taken a considerable amount of time. Therefore we had to give up on this feature and continued our tests using only the connection frequency notes.

As previously explained, the grid on which the map is built is created by using a canvas with a layout group, which automatically spaces out child elements evenly. In hindsight though, this approach might not be the best option. The layout group is an UI element, which turns all its child elements into UI elements as well, meaning that this approach exclusively works in a strict 2D environment.

This did not create issues within the scope of this thesis, however, if the project would be expanded to include 3D elements, this aspect would need revisiting.

The test conducted in Chapter 4.2 ultimately concluded that extracting enough tile connection from a small input image is not possible. This issue could potentially be reduced by using a larger map than just the 9x9 input used in these tests.

The issue here is that the input map used is the result of the MB algorithm. Since the Unity scene does not portray the generated map pixel-perfect, it included small inconsistencies, causing every fourth row to take up one pixel more than the others. This is barely noticeable when just looking at the map, however, when extracting the tiles

from it, this inconsistency would have caused great issues. Therefore the used maps had to be first modified to conform to the actual tile size perfectly.[14] This process was even for the rather small 9x9 map quite time intensive, making it impossible to redo the tests again with a larger input image in the time remaining.

The final mistake was luckily caught and rectified in time, therefore did not influence any of the test runs. In an earlier version of the MB class, a faulty list-access was used to collapse the cells, using the following line of code:

```
tile = cell.possibleTiles[Random.Range(0, cell.possibleTiles.Count-1)];
```

The mistake here is using the list's count -1 as an upper limit for the random generator, which prevented it from accessing the last tile in C-possible. This mistake was only found after TileSet2 generated almost entirely giant ocean maps, with little to no land at all.[Fig.67] Due to the way, the tiles are named, all the water-closing tiles always are at the end of each C-possible, preventing this list access from being choosing them.

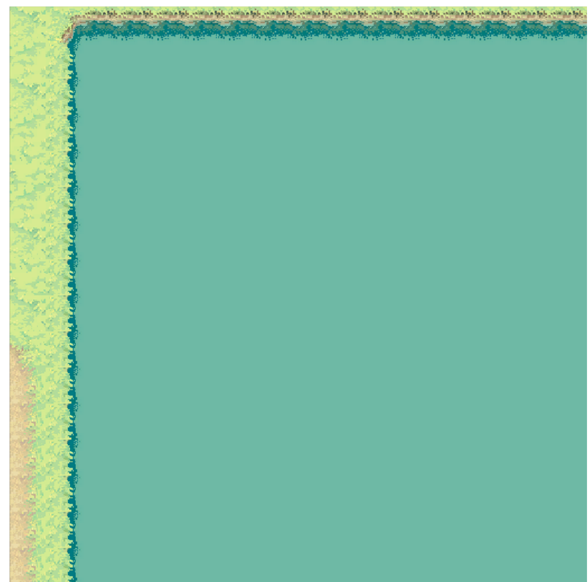

Fig.67: large ocean caused by incorrect list-access

Before the cause of this issue was found, however, we mistakenly assumed the tileset itself to be responsible for this phenomenon. Therefore, a new tileset was created in the first attempt to fix this problem. Tileset2.1 includes all the path and grass tiles of Tileset2 but misses all of the water tiles.

Luckily, the true cause of the mistake was found in time and could be corrected, which allowed us to use the original Tileset2 for our tests.

[14] The modified and original map can both be found in the appendix. However, they do look essentially the same, the only apparent distinction between them is the a few pixel difference in the size

5.2 Possible Continuations

Apart from fixing the mistakes mentioned in the previous chapter, there is a vast array of other continuations which are possible.

Each WFC variation implemented within the work of this thesis uses a hard reset in case of a contradiction. However, that is not the only option to deal with such a situation. Instead of discarding the entire map, it is possible to implement backtracking, to only revert the last few placed tiles, and try again from there.

Alternatively, it is possible to take some more inspiration from Model Synthesis and separate very large maps into individual blocks first, so that only a specific section of the entire output needs to be reset.

Another aspect that could be expanded is the tiles themselves. Currently, all tiles are used in exactly the orientation that they were created in. Rotation or mirroring each tile is a fast and efficient way to introduce more variety into the output maps, without having to modify the tileset itself. This addition might even help improve the accuracy of the Image Analyzer, by giving each tile not only its own connection but also the additional connection of each rotated or mirror version, found within the input.

To adapt the algorithm specifically for more advanced level maps, tiles can be equipped with more game objects. Currently, they are simply containing a texture, resulting in what is essentially a big background image. But it is entirely possible to add more elements, like colliders, triggers, and other types of interactables to them. It is even an option to add 3D elements to the tiles, without necessarily having to change the grid itself to 3D as well. Using the underlying grid as a sort of floor plan allows models like walls, furniture, plants, or other types of elements to be generated within the map as well.

But of course, expanding the dimensions of the grid to be 3D is possible as well.

References

[1] S. Eibenberger, S. Gerlich, M. Arndt, M. Mayor, and J. Tüxen, "Matter–wave interference of particles selected from a molecular library with masses exceeding 10 000 amu," *Physical Chemistry Chemical Physics*, vol. 15, no. 35, pp. 14696–14700, Aug. 2013. https://ui.adsabs.harvard.edu/abs/2013PCCP...1514696E/abstract (accessed Aug. 11, 2023).

[2] J. Faye, "Copenhagen Interpretation of Quantum Mechanics," *plato.stanford.edu*, May 2002. https://plato.stanford.edu/archives/win2019/entries/qm-copenhagen (accessed Aug. 11, 2023).

[3] M. Gumin, "mxgmn/WaveFunctionCollapse," *GitHub*, Apr. 20, 2020, https://github.com/mxgmn/WaveFunctionCollapse (accessed Aug. 12, 2023).

[4] H.Wolverson, "So what does WFC really do?" in "Wave Function Collapse - Roguelike Tutorial - In Rust," (2019) *bfnightly.bracketproductions.com*, https://bfnightly.bracketproductions.com/chapter_33.html#so-what-does-wfc-really-do (accessed Aug. 11, 2023).

[5] P. Merrell, "Example-based model synthesis," Jan. 2007. https://www.researchgate.net/publication/220791940_Example-based_model_synthesis (accessed Aug. 10, 2023).

[6] P. Merrell, "Model Synthesis," *GitHub*, Jul. 30, 2023. https://github.com/merrell42/model-synthesis (accessed Aug. 10, 2023).

[7] P. Merrell, "Comparing Model Synthesis and Wave Function Collapse," 2021. https://paulmerrell.org/wp-content/uploads/2021/07/comparison.pdf (accessed Aug. 12, 2023).

[8] P. Merrell, “Model Synthesis - Paul Merrell,” Jul. 05, 2021. https://paulmerrell.org/model-synthesis/ (accessed Aug. 11, 2023).

[9] A. Efros and T. Leung, “Texture Synthesis by Non-parametric Sampling,” 1999. https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/papers/efros-iccv99.pdf (accessed Aug. 10, 2023).

[10] P. Merrell, D. Manocha, A. Snoeyink, B. Watson, A. Lastra, and M. Lin, “5.1 Model Synthesis and Texture Synthesis” in “Model Synthesis,” 2022. https://paulmerrell.org/wp-content/uploads/2021/06/thesis.pdf#section.5.1 (accessed Aug. 11, 2023).

[11] S. Sherratt, “Collapse Cell” in “Procedural Generation with Wave Function Collapse,” *gridbugs*, Feb. 21, 2019. https://www.gridbugs.org/wave-function-collapse/#collapse-cell (accessed Aug. 11, 2023).

[12] Brackeys, “TILEMAPS in Unity,” YouTube. Jan. 31, 2018. https://www.youtube.com/watch?v=ryISV_nH8qw (accessed Aug. 12, 2023).

Figures[15]

[Fig.2] **Wave Interference**
Bown, “Diffraction and Interference Patterns - IB Physics,” 2021.
https://www.youtube.com/watch?v=elq0PbvAwic

[Fig.3] **Interference Pattern**
A. Consoli, “Definitively a wave... or not?,” 2022.
https://twitter.com/cons_ant_/status/1516436586484928516

[15] This section is listing the sources of the used figures. Note that most images have been created specifically for this paper, and therefore do not appear within this list.

[Fig.4] **Expected Two-Slit Pattern**
V. Ranjbar; “Quantum State Function, Platonic Forms, and the Ethereal Substance: Reflections on the Potential of Philosophy to Contribute to the Harmony of Science and Religion,” 2022.
https://www.researchgate.net/publication/367289856_Quantum_State_Function_Platonic_Forms_and_the_Ethereal_Substance_Reflections_on_the_Potential_of_Philosophy_to_Contribute_to_the_Harmony_of_Science_and_Religion

[Fig.5] **Pattern Repetition**
M. Gumin, “mxgmn/WaveFunctionCollapse,” GitHub, Apr. 20, 2020,
https://github.com/mxgmn/WaveFunctionCollapse/blob/master/images/patterns.png

[Fig.7] **Directional Bias of Model Synthesis vs. Wave Function Collapse**
M. Gumin, “mxgmn/WaveFunctionCollapse,” GitHub, Apr. 20, 2020. graphic modified
https://github.com/mxgmn/WaveFunctionCollapse/blob/master/images/directional-bias.png

[Fig.8] **Texture Synthesis Examples**
A. Efros and T. Leung, “Texture Synthesis by Non-parametric Sampling,” 1999.
https://people.eecs.berkeley.edu/~efros/research/EfrosLeung.html

[Fig.9] **Comparison of neighborhoods to determine individual pixel**
P. Merrell, “Model Synthesis.” 2009.
https://paulmerrell.org/wp-content/uploads/2021/06/thesis.pdf#subsection.2.2.2

[Fig.10] **Comparison of Model Synthesis and Texture Synthesis**
P. Merrell, “Model Synthesis.” 2009. graphic modified
https://paulmerrell.org/wp-content/uploads/2021/06/thesis.pdf#section.5.1

[Fig.38] **original tileset**
S. Sherratt, “Procedural Generation with WFC” *gridbugs*, Feb. 21, 2019.
https://www.gridbugs.org/wave-function-collapse/

Appendix

Link to the Github Repository

https://github.com/chaotic-pan/WaveFunctionCollapse

List of Abbreviations

Abbreviation	Explanation
WFC	Wave Function Collapse
MS	Model Synthesis
TS	Texture Synthesis
C	cell
T	tile
[C]-possible	list of all possible tiles for [a cell]
[T]-connect	lists of all possible connection for [a tile]; exists 4 times per tile, one for each relative direction
MB	Map Builder, the class that arranges tiles into a output map
N1	direct (first) neighbor of a cell
N2	second neighbor of a cell; direct neighbor of N1
IA	Image Analyzer, the class that extracts a tileset from an input image

Additional Data Tables and Diagrams

Map Generation

20x20

min. / max.	Tileset1			Tileset2		
	last time	total time	restarts	last time	total time	restarts
	0.037	0.037	0	0.037	0.081	3
	0.035	0.035	0	0.040	0.063	1
	0.035	0.035	0	0.039	0.039	0
	0.041	0.041	0	0.040	0.040	0
	0.040	0.040	0	0.037	0.132	3
	0.035	0.035	0	0.042	0.042	0
	0.039	0.039	0	0.035	0.156	8
	0.038	0.038	0	0.039	0.039	0
	0.037	0.037	0	0.040	0.048	1
	0.037	0.037	0	0.040	0.088	4
ø	0.037	0.037	0	0.039	0.073	2.0

Fig.68: Table - Run 1: MB1

20x20

min. / max.	Tileset1		Tileset2	
	N=1	N=2	N=1	N=2
	0.037	0.065	0.081	0.085
	0.035	0.066	0.063	0.085
	0.035	0.068	0.039	0.087
	0.041	0.068	0.040	0.085
	0.040	0.064	0.132	0.087
	0.035	0.066	0.042	0.077
	0.039	0.060	0.156	0.083
	0.038	0.066	0.039	0.080
	0.037	0.068	0.048	0.079
	0.037	0.064	0.088	0.082
ø	0.037	0.066	0.073	0.083

Fig.69: Table - Run 2: MB1 vs MB 2

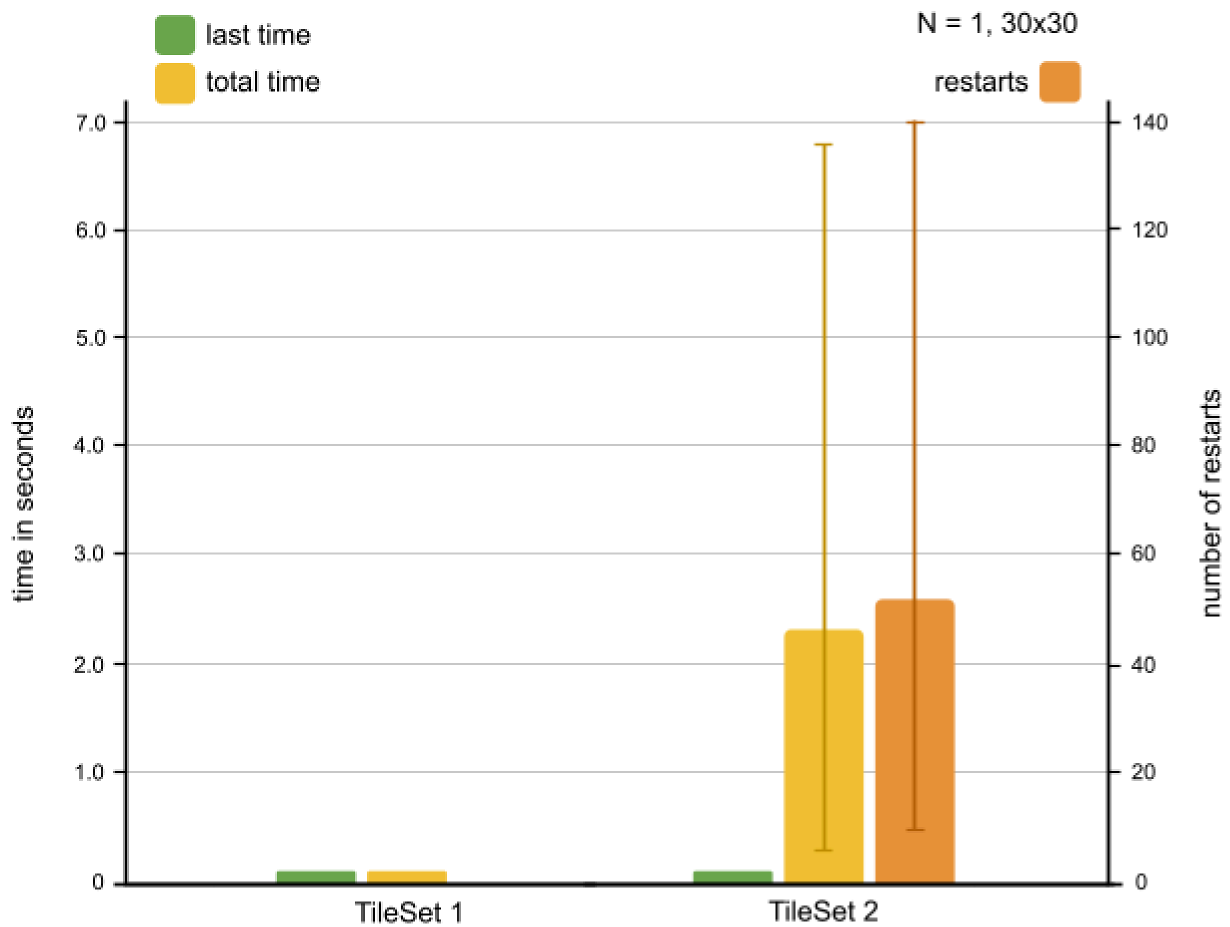

Fig.70: linear Diagram - Run 3: MB1; increased Map Size

30x30

min. / max.	Tileset1 last time	Tileset1 total time	Tileset1 restarts	Tileset2 last time	Tileset2 total time	Tileset2 restarts
	0.085	0.085	0	0.104	1.141	26
	0.109	0.109	0	0.088	0.601	18
	0.082	0.082	0	0.084	3.882	95
	0.097	0.097	0	0.092	0.375	9
	0.106	0.106	0	0.087	1.153	33
	0.094	0.094	0	0.095	0.800	24
	0.114	0.114	0	0.089	6.810	139
	0.092	0.092	0	0.091	2.490	65
	0.096	0.096	0	0.091	4.968	113
	0.101	0.101	0	0.086	0.670	17
ø	0.098	0.098	0	0.091	2.289	53.9

Fig.71: Table - Run 3: MB1; increased Map Size

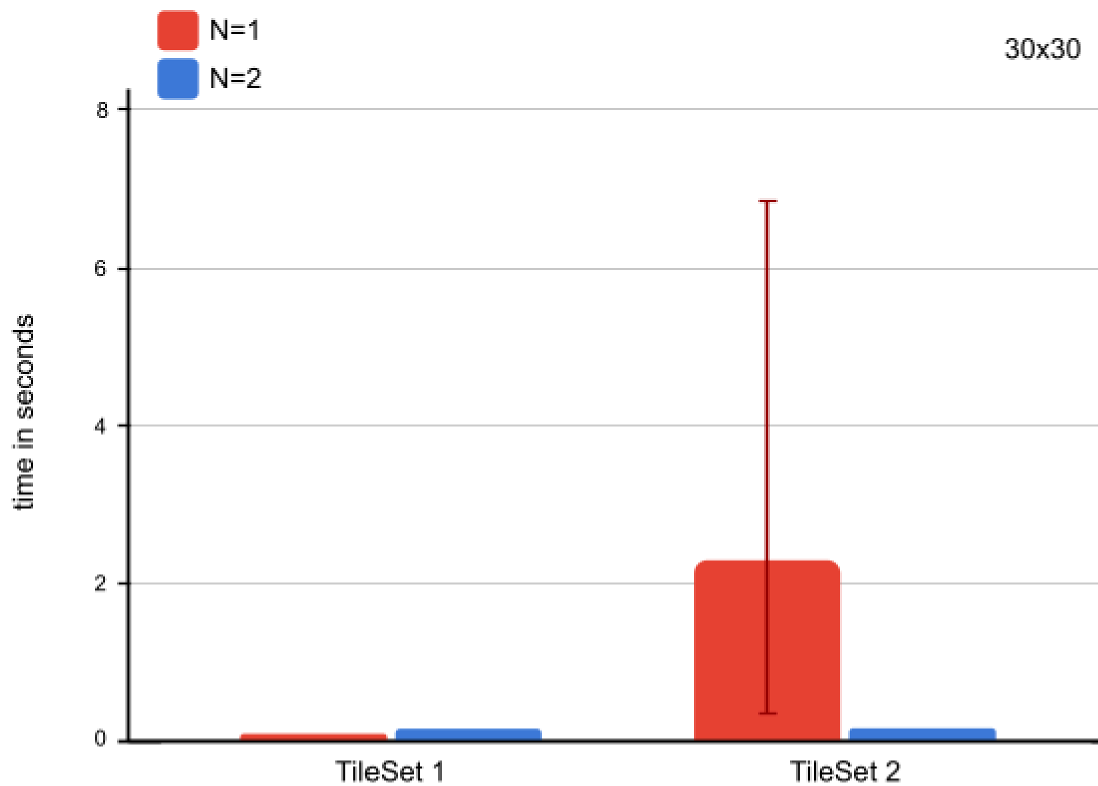

Fig.72: linear Diagram - Run 4: MB1 vs. MB2; increased Map Size

30x30

min. / max.	Tileset1 N=1	Tileset1 N=2	Tileset2 N=1	Tileset2 N=2
	0.085	0.176	1.141	0.193
	0.109	0.161	0.601	0.194
	0.082	0.150	3.882	0.193
	0.097	0.150	0.375	0.199
	0.106	0.148	1.153	0.181
	0.094	0.167	0.800	0.187
	0.114	0.154	6.810	0.196
	0.092	0.164	2.490	0.193
	0.096	0.175	4.968	0.196
	0.101	0.163	0.670	0.204
ø	0.098	0.161	2.289	0.194

Fig.73: Table - Run 4: MB1 vs. MB2; increased Map Size

20x20

min. / max.	Tileset1			Tileset2		
	no freq.	tile freq.	con. freq	no freq.	tile freq.	con. freq
	0.065	0.076	0.098	0.085	0.062	0.109
	0.066	0.076	0.089	0.085	0.070	0.110
	0.068	0.069	0.083	0.087	0.079	0.115
	0.068	0.071	0.081	0.085	0.103	0.106
	0.064	0.072	0.084	0.087	0.087	0.106
	0.066	0.071	0.086	0.077	0.066	0.117
	0.060	0.065	0.086	0.083	0.103	0.108
	0.066	0.070	0.091	0.080	0.086	0.107
	0.068	0.074	0.088	0.079	0.081	0.099
	0.064	0.067	0.081	0.082	0.082	0.112
ø	0.066	0.071	0.087	0.083	0.082	0.109

Fig.74: Diagram - Run 5: MB2 vs. MB3 vs. MB4

Tile Generation

9x9

min. / max.	manual Tileset		
	MB1	MB2	MB4
	0.009	0.015	0.020
	0.008	0.014	0.019
	0.008	0.014	0.021
	0.007	0.022	0.018
	0.013	0.014	0.028
	0.008	0.015	0.018
	0.014	0.034	0.025
	0.010	0.027	0.028
	0.010	0.028	0.083
	0.010	0.016	0.019
ø	0.010	0.020	0.028

Fig.75: Diagram1 - Run 1: Exact Replication

replicated Tileset, 9x9

min.	MB1			MB2			MB4		
max.	total time	resets	last time	total time	resets	last time	total time	resets	last time
	0.073	26	0.010	0.021	1	0.010	0.016	0	0.016
	0.042	13	0.008	0.014	0	0.014	0.014	0	0.014
	0.035	7	0.010	0.017	1	0.008	0.015	0	0.015
	0.054	20	0.005	0.027	2	0.010	0.016	0	0.016
	0.026	15	0.005	0.055	1	0.023	0.027	1	0.011
	0.377	114	0.011	0.012	0	0.012	0.015	0	0.015
	0.044	15	0.009	0.014	0	0.014	0.015	0	0.015
	0.110	13	0.008	0.013	0	0.013	0.016	0	0.016
	0.275	96	0.005	0.036	5	0.010	0.015	0	0.015
	0.175	73	0.005	0.023	1	0.010	0.019	0	0.019
ø	0.121	39.2	0.008	0.023	1.1	0.012	0.017	0.1	0.015

Fig.76: Diagram2 - Run 1: Exact Replication

overlap Tileset, 9x9

min.	MB2			MB4		
max.	total time	resets	last time	total time	resets	last time
	0.030	4	0.006	0.027	4	0.011
	0.024	7	0.008	0.025	4	0.010
	0.033	11	0.008	0.135	3	0.038
	0.026	3	0.009	0.015	0	0.015
	0.152	51	0.010	0.023	1	0.014
	0.043	7	0.007	0.015	0	0.015
	0.035	5	0.012	0.041	6	0.010
	0.026	2	0.010	0.017	1	0.012
	0.169	1	0.010	0.023	2	0.010
	0.040	8	0.009	0.020	3	0.008
ø	0.058	9.9	0.009	0.034	2.4	0.014

Fig.77: Diagram1 - Run 2: Overlapping Tiles

overlap_mod Tileset, 9x9

min.	MB4		
max.	total time	resets	last time
	0.031	1	0.017
	0.069	3	0.017
	0.070	1	0.030
	0.023	0	0.023
	0.029	0	0.029
	0.031	0	0.031
	0.023	0	0.023
	0.100	1	0.056
	0.023	0	0.023
	0.023	0	0.023
ø	0.042	0.6	0.027

Fig.78: Diagram2 - Run 2: Overlapping Tiles

Additional Maps

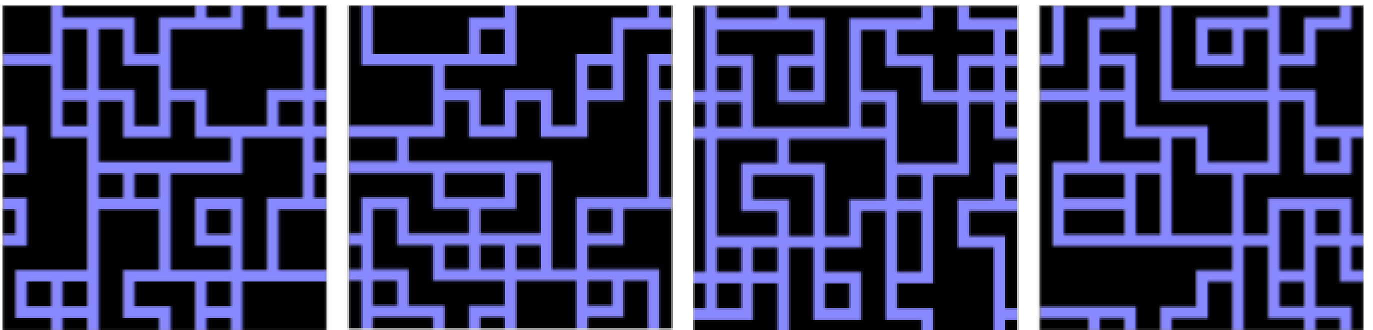

Fig.79: MapSet - Tileset1, MB1, 9x9

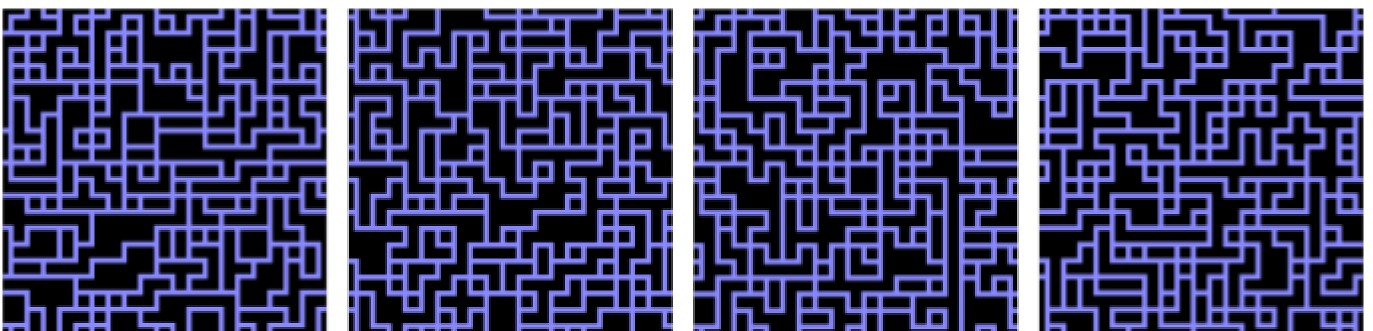

Fig.80: MapSet - Tileset1, MB1, 20x20

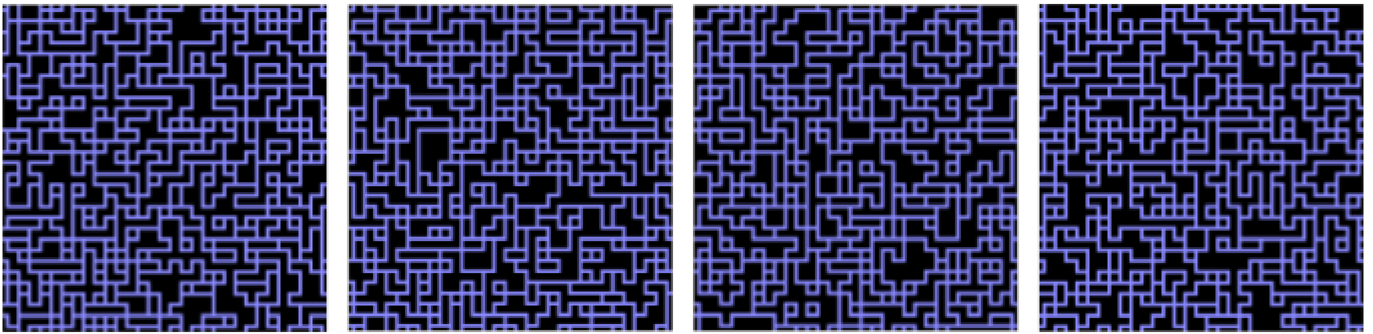

Fig.81: MapSet - Tileset1, MB1, 30x30

Fig.82: MapSet - Tileset2, MB2, 9x9

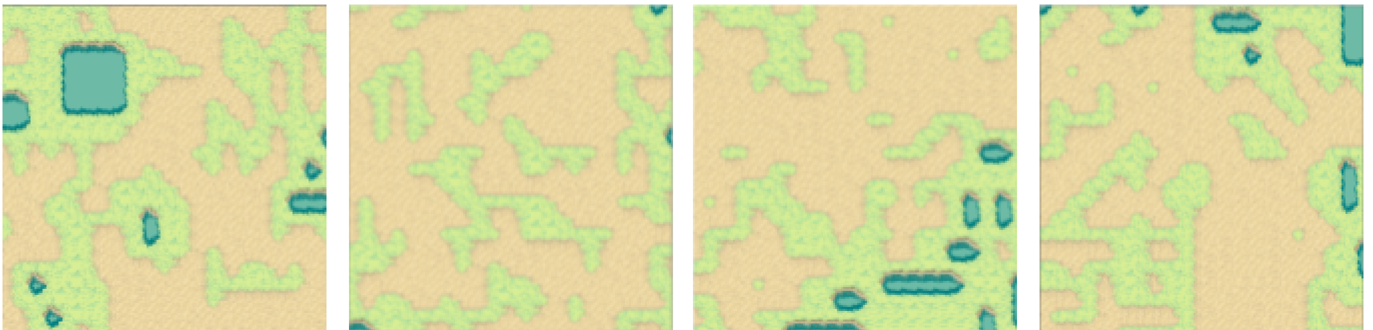

Fig.83: MapSet - Tileset2, MB2, 20x20

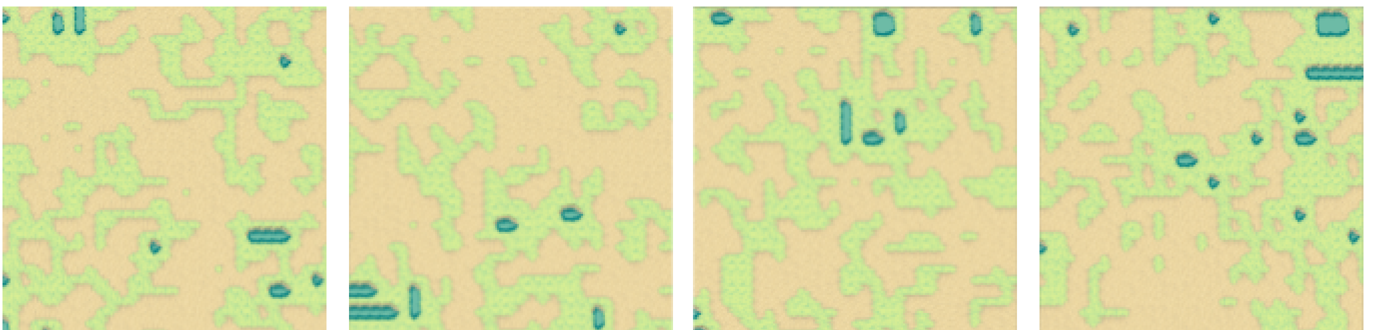

Fig.84: MapSet - Tileset2, MB2, 30x30

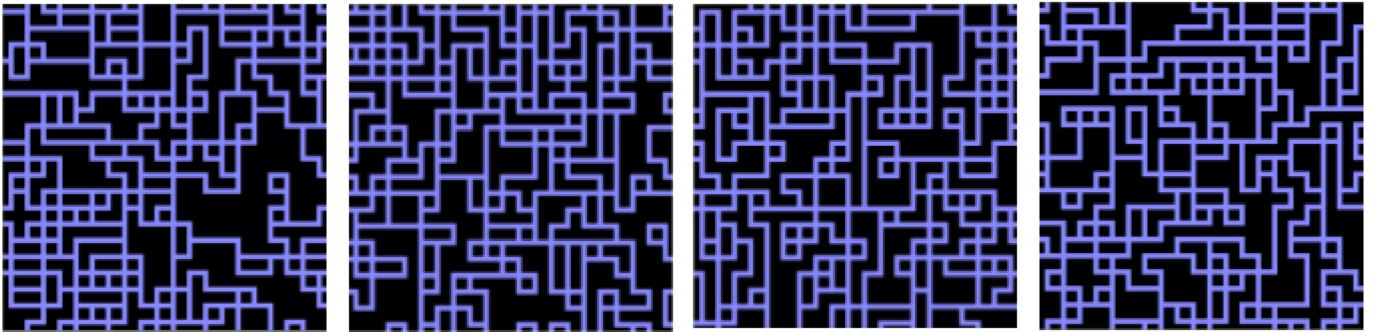

Fig.85: MapSet - Tileset1, MB3, 20x20, higher (2) freq. on blank and cross tiles

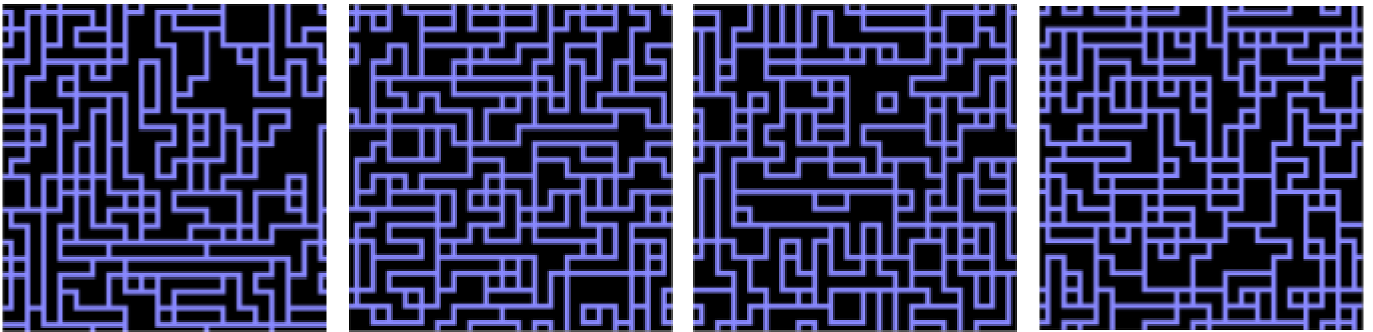

Fig.86: MapSet - Tileset1, MB3, 20x20, higher (2) freq. on straight tiles

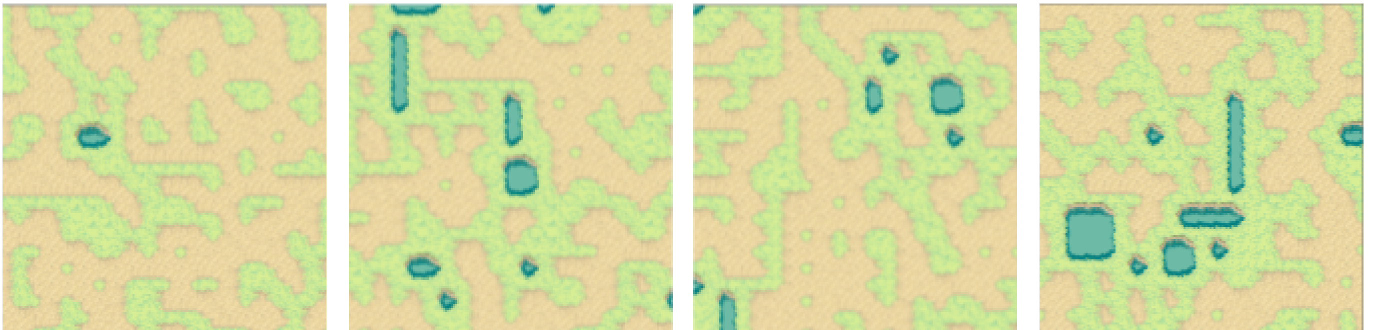

Fig.87: MapSet - Tileset2, MB3, 20x20, lower (0.1) freq. on pure grass and path tiles

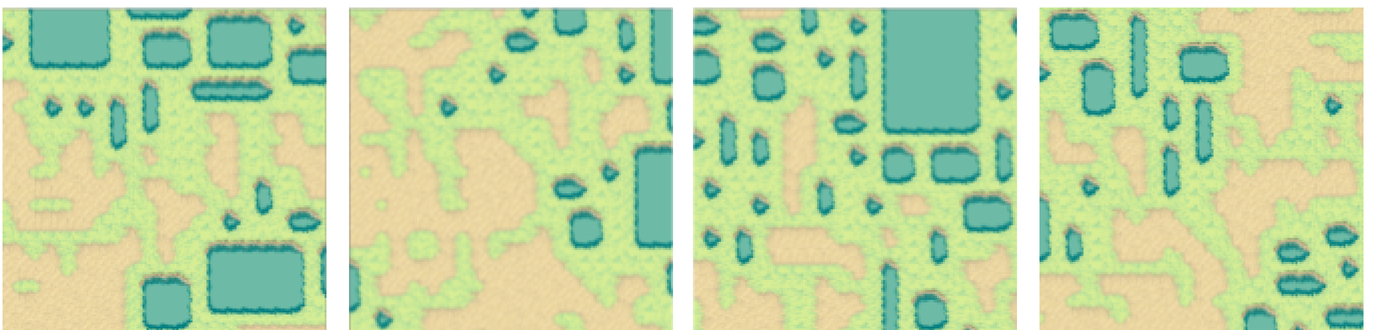

Fig.88: MapSet - Tileset2, MB3, 20x20, higher (2) freq. on water edge tiles, and very high (5) freq. on pure water tiles

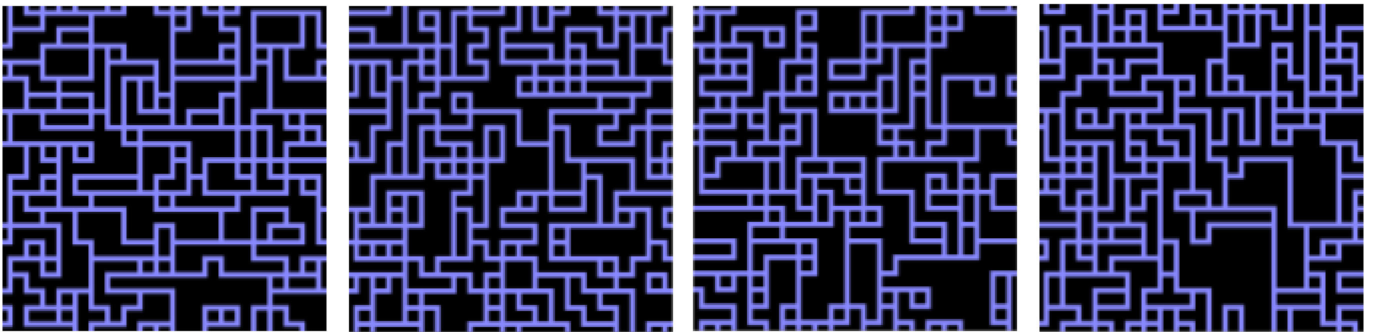

Fig.89: MapSet - Tileset1, MB4, 20x20, low (0.5) freq. on blank tiles, very high (5) connect-freq. for blank tiles next to straight, blank, and T tiles, high (2) connect freq. for straight tiles next to corners

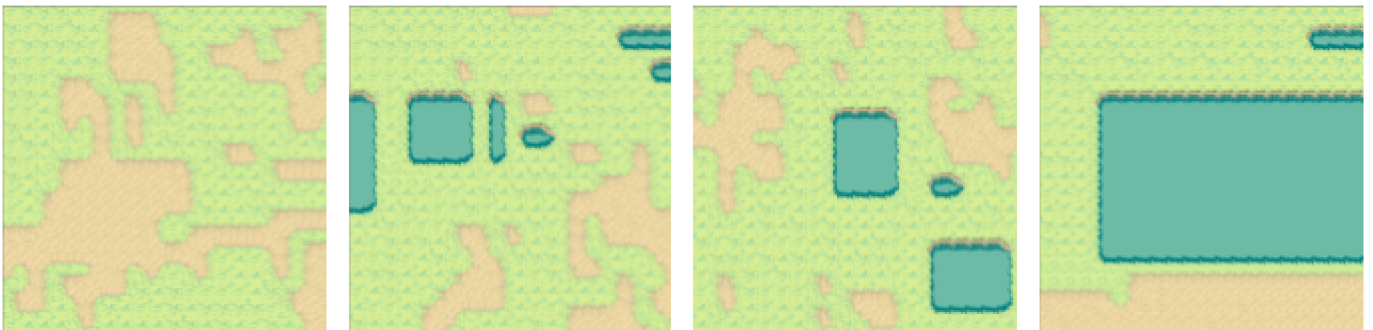

Fig.#: MapSet - Tileset2, MB4, 20x20, lower (0.2) connect-freq. for pure grass next tiles to each other, lower (0.1) connect-freq. for pure path tiles next to each other, lower (0.5) connect-freq. for path edges right next to each other (aka. more likely to have at least one pure grass tile between two separate paths), the same goes for water edges next to each other, the same goes for water edges and path edges next to one another, lower (0.1) connect-freq. for path corners next to the opposite corner (aka. more likely for paths to be wide enough to include at least one pure path tile), the same goes for water edges opposite of each other, higher (2) connect-freq. for inner path edges next to each other

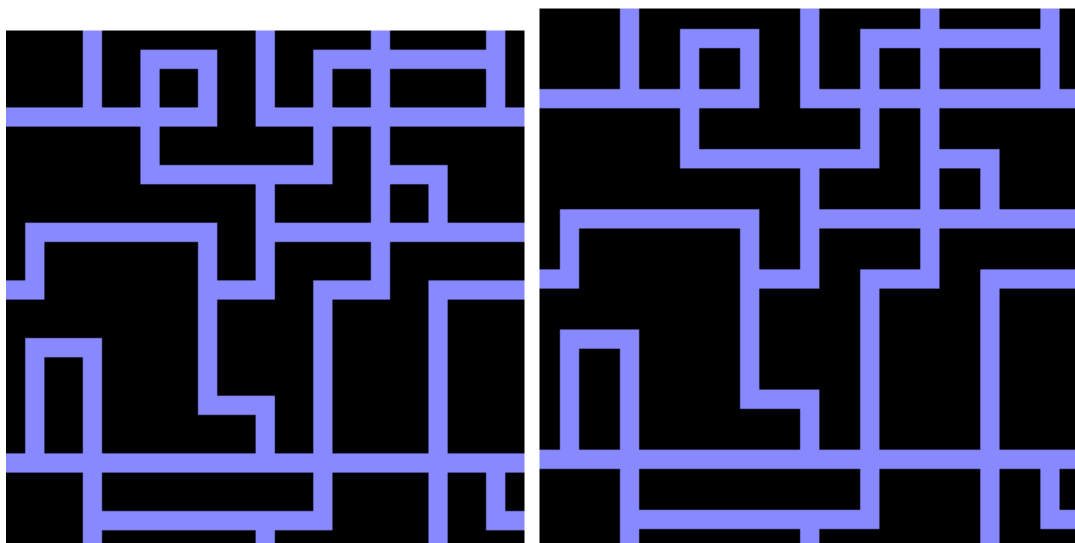

Fig.90: map used for tile extraction 432x432px (left) and the unedited version taken directly from the Unity Scene View with inconsistent tile size 450x450px (right)