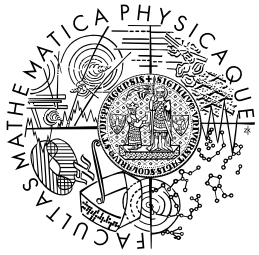

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jiří Filek

Hierarchical Version of the Wave Function Collapse Algorithm

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Vojtěch Černý

Study programme: Computer Science

Study branch: Visual Computing and Game Development

Prague 2023

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

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

In ..... date ....

Author's signature

I would like to thank my supervisor Mgr. Vojtěch Černý for his help during the development of this project and many great ideas. I would also like to thank my family, friends, and colleagues for their support.

Title: Hierarchical Version of the Wave Function Collapse Algorithm

Author: Bc. Jiří Filek

Department: Department of Software and Computer Science Education

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

Abstract: Maxim Gumin's Wave Function Collapse (WFC) is a recent algorithm used for procedural content generation. The algorithm uses constraint solving and local similarity to generate outputs. However, it struggles to generate large or complex outputs. We aim to generalize the original work to make the algorithm work hierarchically on several different granularities. We show that this approach is promising and yields better results than the original algorithm in several challenging domains. Our approach also provides better controllability of the outcome. The algorithm has applications in the field of procedural content generation to generate different kinds of 2D game levels. It can provide good variability for the players and save the time of game designers.

Keywords: Hierarchical Wave Function Collapse, Wave Function Collapse, Procedural Content Generation, Computer Games

Contents

Introduction

Maxin Gumin introduced the Wave Function Collapse algorithm (WFC) in late 2016. [Gumin, 2016b] It was a new approach to procedural content generation. It uses constraint solving to create its output. This is relatively rare in the field of procedural content generation. It opens a possibility for a lot of further research since it has not been explored enough. We will deal with it in much more in chapter 1.3.

The Wave Function Collapse algorithm was introduced for a 2D generation. It takes a bitmap as an input and generates outputs with similar characteristics. It is great for small and simple scenarios. The algorithm is also very flexible and can generate totally different kinds of outputs. However, it struggles to generate complex and structured outputs. It is also very hard to control to generate outputs satisfying given constraints. To alleviate this issue, it has already been combined with other methods used in procedural content generation. [Minini and Assunção, 2020]

Goals

The primary goal of this thesis is to combine the Wave Function Collapse algorithm with itself. We mean that we can create several instances of the Wave Function Collapse algorithm, each generating totally different kinds of outputs. Then we can combine those instances instead of using other procedural content generation methods. We will combine them in a hierarchical way.

Since it is whole new research, we will need to discover many improvements to make our algorithm controllable and applicable to challenging problems. We designed two such example problems: a World Map and Dungeon generation.

Another important aspect is to compare it with the regular Wave Function Collapse. It is important to compare both the quality of generated maps and the performance of both algorithms. This way, we can determine if it is a promising area for further research or, rather, a dead-end. Wave Function Collapse is a constraint solver, which makes it computationally intensive. Therefore, we also need to explore its performance since another significant performance cost might make the algorithm unusable.

Finally, we aim to provide a decent user interface for our hierarchical version. The algorithm itself is very difficult to control. Having its hierarchical version difficult to set up might also make the algorithm unusable because the learning curve would be too steep.

Structure

In chapter 1, we will discuss the background information about procedural content generation, and mainly Wave Function Collapse algorithm. In chapter 2, we will discuss the hierarchical version of the algorithm. We will also focus on many modifications of Wave Function Collapse to make it better for our use case.

In chapter 3, we will demonstrate the usage of the hierarchical algorithm. We will apply it to generate a World Map and a Dungeon, discussing and explaining every design decision we make. This chapter is significant for understanding how to apply the hierarchical algorithm to different problems. We will show and discuss generated outputs in chapter 4. We will also compare them with regular Wave Function Collapse outputs regarding quality, satisfying design requirements, and performance. Finally, we will discuss the Unity implementation in chapter 5.

Related Work

We are trying the generalize the Wave Function Collapse algorithm. There were already many generalizations. Many authors have implemented the 3D Wave Function Collapse algorithm. [Gumin, 2016a] It is not really important for us since we will focus only on its 2D version.

Some authors have found that the WFC is hard to control. Therefore, they were looking for ways how to make the WFC more controllable. [Cheng et al., 2020] Our hierarchical version tries to achieve a similar goal of creating structured outputs.

Joseph Parker has implemented WFC in Unity. [Parker, 2016]. This is important for us because we base our implementation on his framework.

The most important related work was probably done by Brian Bucklew, who combined WFC with other PCG methods for the game Caves of Qud [Bucklew, 2019a] [Bucklew, 2019a]. The core idea behind Hierarchical Wave Function Collapse is closely related to his work.

Otherwise, our work is new and highly experimental. Therefore, it visits many unexplored areas.

1. Background

This chapter will introduce the related helpful theory for the following chapters. It serves as a building block for this thesis. We will start with a brief introduction to Procedural Content Generation 1.1. After it, we will shortly discuss a Constraint Satisfaction Problems 1.2, which is closely related to the Wave Function Collapse algorithm. Finally, we will discuss the Wave Function Collapse 1.3 algorithm itself.

1.1 Procedural Content Generation

Procedural Content Generation (PCG) could be defined as: "PCG is the algorithmic creation of game content with limited or indirect user input." [Togelius et al., 2011] The content can mean almost anything. It ranges from the maps, through loot and items, to whole stories or game mechanics.

In comparison with procedural art, the PCG for games is more strict. The generated content must satisfy some constraints to be helpful. For example, a generated map must support other game mechanics, and its major parts should be reachable.

There are several reasons why to use PCG. Firstly, it can speed up the process of making a game. People are expensive, and it can replace long, repetitive work. Secondly, some problems are only solvable with PCG. We could not have potentially infinite levels in Minecraft without PCG. Some designers would need to place down all the blocks, which is infeasible. Thirdly, the PCG can greatly improve the replayability of the game. Finally, it can help us to understand design constraints and intentions. We understand those concepts much more when they are required to be expressed in some formal language. [Shaker et al., 2016]

The help of PCG is suitable for both small and large game studios. The smaller studios could use a wild approach to generate large parts of the game. On the other hand, large AAA studios are using PCG to place roads, rivers, and vegetation on large open-world maps. The designer can draw their general layout, and the PCG method will handle all the repetitive tasks and details. [Werle and Martinez, 2017]

1.2 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is closely related to the Wave Function Collapse. The Wave Function Collapse is a special case of the CSP. Therefore, we will start with the more general CSP.
A constraint satisfaction problem has three components:

X is the set of variables $X_1, ..., X_n$
D is the set of domains $D_1, ..., D_n$, one for each variable
C is the set of constraints that specify allowable combinations of values

For variable X_i, the domain D_i consists of a set of allowable values $v_1, ..., v_n$. Constant C_i has a tuple of variables and relation that specifies which values are allowed simultaneously. Each CSP is defined by an assignment of values to the variables. The assignment is consistent if it does not violate any constraints. The assignment is complete when it assigns a single value to each variable. The solution to a CSP is a consistent and complete assignment. [Russell and Norvig, 2010]

When solving the CSP, we can use constraint propagation. It means we can use constraints to reduce the domain for a variable, given a partial assignment. This process can also transitively reduce legal values for another variable. We can look at the CSP as a graph, where each variable is a node and binary constraints define edges. A common approach is to enforce arc consistency. It means that for each edge (X_i, X_j) and every value in D_i, there exists a value in D_j that satisfies the binary constraint on the arc (X_i, X_j). To enforce this consistency, we remove all the values in D_i that do not have any supporting value in D_j. [Russell and Norvig, 2010]

Algorithm 1 is called AC-3, and it resolves the arc consistency. We could also enforce higher levels of consistency. A path consistency is similar, but it works on three variables at a time. The number of variables that we are considering simultaneously is called an order. If we enforce the consistency of order N, it solves the whole CSP immediately. On the other hand, enforcing higher levels of consistency is much more computationally intensive, and arc consistency is a good compromise. [Russell and Norvig, 2010]

To solve the CSP, we can combine search with constraint propagation. It means that we iterate between two steps. We pick a value for some variable, assign it the value, and propagate new constraints with AC-3. We can repeat this process until we have a solution. If AC-3 finds a contradiction, we must backtrack and try different values for the variable.

There are two common heuristics. The first one is to choose the most constrained variable. That is a variable X_i with the least legal values in D_i. The

Algorithm 1 AC-3 [Russell and Norvig, 2010]

Input: csp (X, D, C)
$queue \leftarrow$ all $edges$
while !$queue.empty()$ **do**
  $(X_i, X_j) \leftarrow queue.pop()$
  $revised \leftarrow false$
  for all x in D_i **do**
    if no value in D_j supports x **then**
      $D_j.delete(x)$
      $revised \leftarrow true$
    end if
  end for
  if $revised$ **then**
    if $D_i.size() == 0$ **then**
      return $false$
    end if
    for all X_k in $X_i.neighbors \backslash \{X_j\}$ **do**
      $queue.push((X_k, X_i))$
    end for
  end if
end while
return $true$

idea is that the CSP is likely to fail on this variable in the future. The second heuristic is to use the least constraining value. That is the value that rules out the fewest option from its neighbors.

1.3 Wave Function Collapse

The Wave function collapse (WFC) is a quite recent algorithm used for procedural generation. Maxim Gumin introduced the algorithm in late 2016. Most of the resources about Wave Function Collapse are available on his GitHub page [Gumin, 2016b]. The standard version of the algorithm works on a two-dimensional grid. It takes a bitmap as an input a produces a different bitmap as an output. An interesting feature is that the output bitmap can be much larger than the input bitmap. Those two bitmaps are locally similar.

Local similarity is defined as:

- (C1) for a given positive integer N, each $N \times N$ pattern that is in the output has to be somewhere in the input
- (C2 weak) probability of the pattern appearing in the output should be close to the density of the pattern in the input

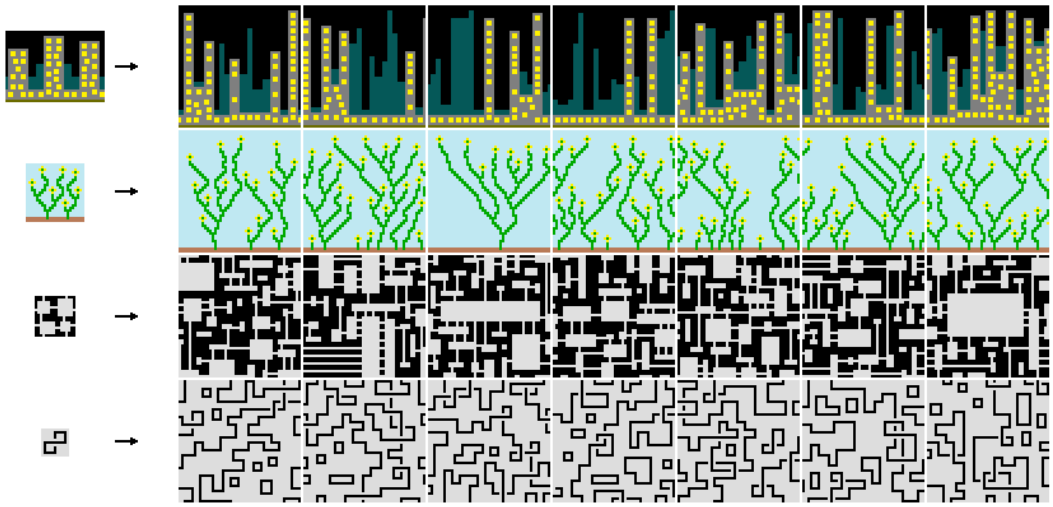

Figure 1.1: Examples of Wave Function Collapse; on the left side, is the input bitmap and on the right side are several generated outputs. [Gumin, 2016b]

The definition of local similarity needs a positive integer N. This parameter can influence the outcome of the algorithm dramatically. If the N is large, the algorithm tends to copy large chunks of the input to the output. It also adds significant performance costs. Taking N too small can create degenerate output due to under-constraining. The most common option is to take $N = 3$, which is a good compromise between the variety of outcomes, control, and performance. The algorithm must also know the output size as *width* and *height* parameters.

Figure 1.1 shows the usage of the Wave Function Collapse algorithm in some examples. It shows that the WFC algorithm is applicable to a large domain of problems. We can generate totally different kinds of outputs with a single algorithm. This example is taken from the WFC repository [Gumin, 2016b].

The Wave function collapse algorithm has two models, which we will introduce in the following sections. Those models are the Overlapping Model 1.3.1 and the Simple Tiled Model 1.3.2.

1.3.1 Overlapping Model

The first model is the Overlapping model or the Overlap model. It can be described with the following Algorithm 2.

At the start, the algorithm does a preprocessing step. It extracts all patterns from the input bitmap B, assigns them weights, and finds which patterns can be placed next to each other. Weights are obtained from the number of appearances of a given pattern on the input. Then it initializes the *propagator*. It is a two-dimensional array initialized to an uncollapsed state. There can be any pattern from B in each cell.

Algorithm 2 Wave function collapse (Overlap model)

1: **Input:** width W, height H, pattern size N, input bitmap B
2: $patterns \leftarrow all\ N \times N$ patterns from B
3: $weights \leftarrow \#occurrences$ of each pattern
4: $constrains \leftarrow$ which patterns neighbors which in B
5: $propagator \leftarrow array\ (W \times H)$ each contains a list of $patterns.size$ of indices

6: // position (w, h) is collapsed $\iff propagator[w, h].size \leq 1$
7: **while** *not collapsed* **do**
8: **if** $exist\ (w, h) \in propagator : propagator[w, h].empty$ **then**
9: **return failure** {random restart here}
10: **end if**
11: $(w, h) \leftarrow$ uncollapsed position from propagator with minimal entropy
12: $selected \leftarrow$ random weighted pattern from $propagator[w, h]$
13: $propagator[w, h] \leftarrow selected$
14: propagate this with *constrains*
15: **end while**
16: **return** *propagator*

After the preprocessing step, there is the main loop. In the loop, we will always pick an uncollapsed cell and collapse it. An *uncollapsed cell* is defined as a cell with at least two possible patterns. The collapse selects a possible pattern randomly distributed according to weights for a given cell. After selecting a single pattern in a cell, we must propagate this information into adjacent cells. The propagation works the same way as in a regular constraint satisfaction problem using the arc consistency and the AC-3 algorithm 1. Patterns that are no longer supported will be pruned. This propagation process is recursive, and it can propagate into distant areas. It is also the most significant performance bottleneck.

Algorithm 2 can fail on line 9. It happens if, for some cells, there does not exist any possible pattern. It means that each pattern was banned by some neighbors. For this reason, we are all always collapsing the cell with a minimal Shannon entropy 1.1.

$$H(P) = -\sum_{i=1}^{n} P(x_i) \log P(x_i) \tag{1.1}$$

$$cell = \underset{c \in uncollapsed}{\operatorname{argmin}} H(c) \tag{1.2}$$

The minimal entropy heuristic is similar to the most constrained variable heuristic used in CSPs. Therefore, a random restart of the algorithm works relatively well, as failures do not often occur with good inputs. Alternatively, we can use backtracking to solve this issue. Picking a cell to collapse on line 11 and the collapse itself on line 12 is usually done using a `Observe()` function.

The propagation of new constraints on line 14 is usually done by a `Propagate()` function.

We cannot use the second common heuristic from the CSP to select the least constraining variable. This way, we would improve our chances for a successful collapse. However, most outputs would be very similar, which we want to avoid in the PCG. Selecting the weighted random pattern also helps to satisfy the (C2) constraint in the definition of local similarity – patterns appear with a similar distribution in both the input and the output.
We have mentioned several times that WFC and CSP are closely related. Now we can define the WFC as a CSP:

X — a set of all cells
D — a set of patterns from the input
C — neighbors are four adjacent positions, and supported values are extracted from the input

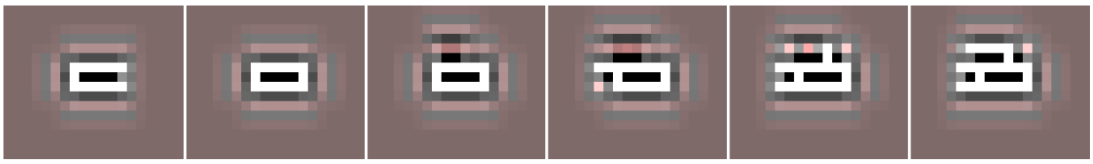

Figure 1.2: Consecutive updates of the output, uncollapsed cells shows the average value of remaining possible colors [Karth and Smith, 2017]

Since the algorithm is incremental, we can use its output for a nice visualization of cells that are already collapsed. We can see an example of it in figure 1.2. Each following image shows one iteration of the algorithm. A single step can collapse multiple tiles due to constraint propagation. [Karth and Smith, 2017]

1.3.2 Simple Tiled Model

The Overlapping model stores the whole $N \times N$ pattern in each cell, which makes the algorithm non-trivial to modify. The Simple tiled model works differently. It stores only tiles from the input into each cell – a 1×1 pattern. It uses adjacency rules to constrain its neighbors. This means that we only look, at which patterns can be placed next to each other in four cardinal directions. It is equivalent to considering only 1×2 patterns. The input is often given as a *.xml* file because it is easy to describe. For each tile, we only need to store which tiles are adjacent in that four directions. Weights can also be set manually here for each tile. We can easily generate this file from an input bitmap.

The Simple Tiled model is generally less powerful than the Overlapping model. However, it is simpler to use and modify. Therefore, more relevant research was done using the Simple tiled model. [Cheng et al., 2020]

Algorithm 3 Wave function collapse (Simple Tiled model)

Input: width W, height H, input *.xml* file I
$tiles \leftarrow$ all tiles from I
$weights \leftarrow$ tile weights from I
$constrains \leftarrow$ which tiles neighbors which in I
$propagator \leftarrow array\ (W \times H)$ each contains a list of $tiles.size$ of indices
// position (w, h) is collapsed $\iff propagator[w, h].size \leq 1$
while *not collapsed* **do**
  if $exist\ (w, h) \in propagator : propagator[w, h].empty$ **then**
    return failure {random restart here}
  end if
  $(w, h) \leftarrow$ uncollapsed position from propagator with minimal entropy
  $selected \leftarrow$ random weighted tile from $propagator[w, h]$
  $propagator[w, h] \leftarrow selected$
  propagate this with $constrains$
end while
return *propagator*

We can see that Algorithm 3 for the Simple tiled model is almost the same as Algorithm 2 for the Overlapping model. Firstly, prepossessing differs as we read those values from a file instead of extracting them from the bitmap. However, the idea of prepossessing it is the same. Secondly, we use tiles directly instead of patterns, as mentioned above.

Oskar Stålberg has made a very nice online interactive demo for the Simple Tiled model that runs in a browser. It demonstrates how the algorithm works and shows the remaining possible tiles for each cell. [Stålberg, 2017]

1.4 Inputs

We have described the idea behind the WFC. Surprisingly, the Wave Function Collapse algorithm takes a lot of inputs. We will focus on them more in this section. We will describe inputs to the original implementation. [Gumin, 2016b]

Inputs to the WFC algorithm including their types

- Input Bitmap (bitmap – two-dimensional array of byte)
- Width (unsigned integer)
- Height (unsigned integer)

- N for the Overlapping model (small unsigned integer greater than 1, usually 2 or 3)
- Seed (integer)
- Periodic item input (bool)
- Periodic output (bool)
- Symmetry (integer in range $[1, 8]$)
- Iterations (unsigned integer)

We have already discussed the core inputs: bitmap, width, height, and N. Now, we will discuss the more special inputs.

The WFC algorithm uses pseudo-random numbers to generate different outcomes. It takes a seed parameter to make the algorithm's results reproducible. The value `seed = 0` is special and is used to generate pseudo-random output that is not reproducible.

The periodic input states whether the input should also be sampled periodically over its edges. We discovered that the periodic input works well and is active in all our usage.

The periodic output is similar to The periodic input. It states whether patterns on the output that has to be in the input also reach over borders. We discovered that the periodic output also works well and is active in all our usage. Especially in combination with borders, which we will discuss in the next chapter.

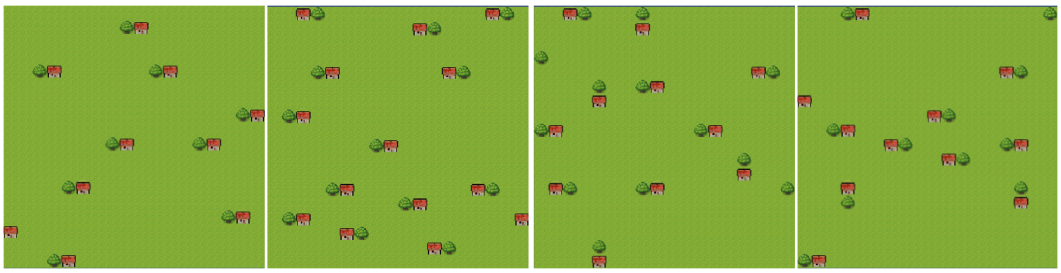

Figure 1.3: Different symmetries, uses values from left 1, 2, 4, and 8

Symmetry is crucial. It states from how many directions should be the input sampled. Each number adds either a new rotation or a new reflection. This is a great way to get more patterns from the input. It results in more variability in the output. However, if some tiles are not symmetric, the symmetry can make the outputs unusable. The influence of symmetry is showcased in figure 1.3.

Finally, the last parameter is iterations. The WFC algorithm will run only for a given number of iterations. This is helpful for the visualization of the progress

of the generation. The value `iteration = 0` again has a special meaning. It means unlimited number of iterations and does the whole collapse at once.

2. Analysis

The previous chapter showed us how the Wave Function Collapse algorithm works. In this chapter, we will analyze the hierarchical usage of the algorithm and its consequences. We will need to do much work to make the algorithm usable.

2.1 Motivation

We have the basics of the WFC algorithm. Therefore, we can finally focus on the motivation behind its hierarchical usage. We already know that the WFC algorithm is very flexible. It also implies that the algorithm is relatively hard to control.

Brian Buclew have mentioned several issues with the WFC algorithm. [Bucklew, 2019a] [Bucklew, 2019b] He is one of the authors of a successful game, Caves of Qud, which uses WFC extensively.

WFC difficulties:

1. Homogeny – the output of the WFC is homogenous, it does not have any high-level structure and it feels the same everywhere
2. Overfitting – if the input is complex, the exact shape will likely be copied to the output
3. No postprocessing – there is no general way to ensure the connectivity of different areas

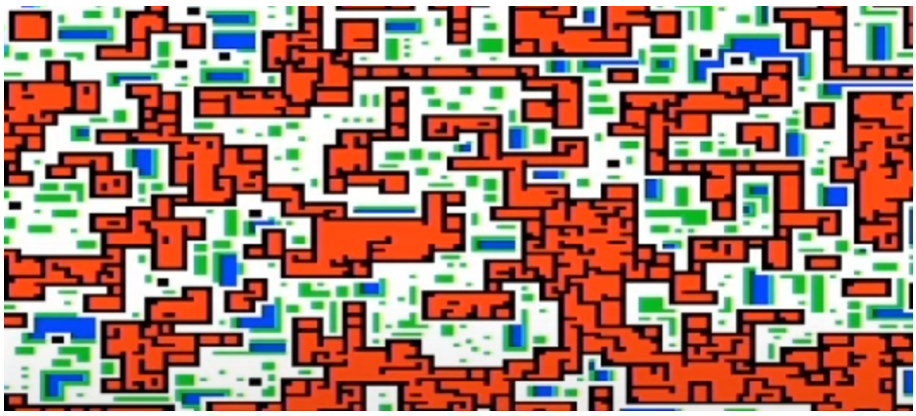

Figure 2.1: Example of homogeny [Bucklew, 2019a]

Figure 2.1 shows an example of homogeny. There is a lack of any high-level structure. The whole output looks the same. It is a common issue since WFC propagates constraints locally and cannot ensure high-level structures.

There is a simple solution for homogeny. We can split the whole output into several regions and use different WFC for each region. Each region will be homogenous, but the overall output will have a structure.

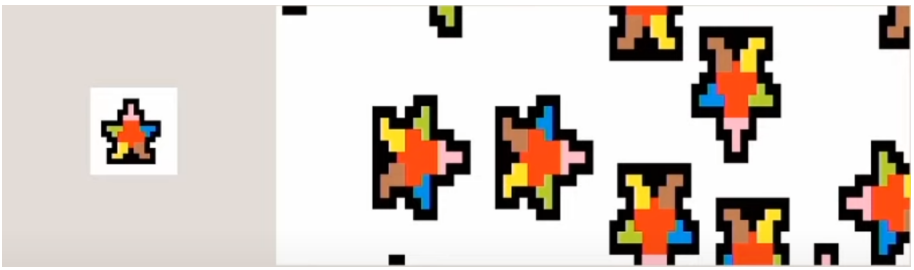

Figure 2.2: Example of overfitting [Bucklew, 2019a]

Figure 2.2 shows an example of overfitting. The shape of the input is way too complex. Therefore, it has to be copied to the output (it can be rotated or reflected). Otherwise, it would not satisfy the conditions of local similarity, and the generation would fail. This is very bad because the output lacks variety.

Due to overfitting, we need to avoid very complex shapes on the input. Alternatively, we would need to have many different complex shapes which allow their combinations. However, it is more computationally intensive. A possible solution is to avoid details and add them as postprocessing or with another WFC pass.

The final issue about the lack of postprocessing is trivial to solve. We can add a postprocessing step at the end. Postprocessing can be used, for example, to connect rooms.

A general solution to work efficiently with WFC is to combine it with different PCG methods to cover its weaknesses. It is an already-known approach. [Bucklew, 2019a] [Minini and Assunção, 2020] Our approach is slightly different. We will combine the WFC algorithm with its other instances.

2.2 Hierarchical Wave Function Collapse

The Hierarchical Wave Function Collapse (HWFC) is a new algorithm introduced by this thesis. The primary idea behind it is simple. The Wave Function Collapse algorithm uses only local constraints, so it works locally. It is tough to enforce global constraints in the algorithm. Therefore, we can use very specific WFC to generate an abstract top layer that deals only with a general layout. Next, we can use the output from the abstract layer and run a Wave Function Collapse with a different input on each of them. We can also subdivide the work into more layers to avoid overfitting.

For example, generating a world map with a standard Wave Function Collapse is extremely difficult, maybe even impossible. However, with the Hierarchical

Wave Function Collapse, we can use the first layer to generate a layout of oceans and continents. In the second layer, we can divide the continents into biomes. Finally, we can generate the output for each biome over a given area in the third layer. We will discuss it much more in the following chapter.

We have not found any papers dealing with the hierarchical usage of the Wave Function Collapse algorithm. On the other hand, as we have already mentioned Brian Bucklew gave a talk at the Game Developers Conference 2019 Bucklew [2019b] and at Roguelike Celebration 2019, Bucklew [2019a] about their game Caves of Qud. The recordings from both talks are available on YouTube. In Caves of Qud, the map is situated on a two-dimensional grid and procedurally generated at runtime using the Wave Function Collapse algorithm. They used the Overlapping model with $N = 3$. It might be a bit surprising since the Simple Tiled model is probably more popular among successful games using WFC. [Stålberg, 2018] [Stålberg, 2020] For some features, they used an approach similar to the hierarchical idea. When generating ruins, they picked an area and ran the WFC over it. Finally, they placed it on the primary background WFC which generated the whole map.

Figure 2.3 demonstrates the core idea behind HWFC. At first, we generate a simple general layout. It is shown in the top part. Then, in the next layer, we can run different WFCs over each generated area. This way, we will avoid homogeny. We can add more layers to add details and avoid overfitting if necessary. More layers can also be helpful if we would like to generate even more structured outputs.

However, the most considerable advantage of using WFC hierarchically is that it dramatically reduces the complexity of the input bitmap. It means we can split one huge input bitmap with thousands of tiles into several smaller ones, each containing hundreds of tiles. These smaller maps are much easier to design and adjust when necessary. We will discuss this further in chapter 3.1, giving particular examples.

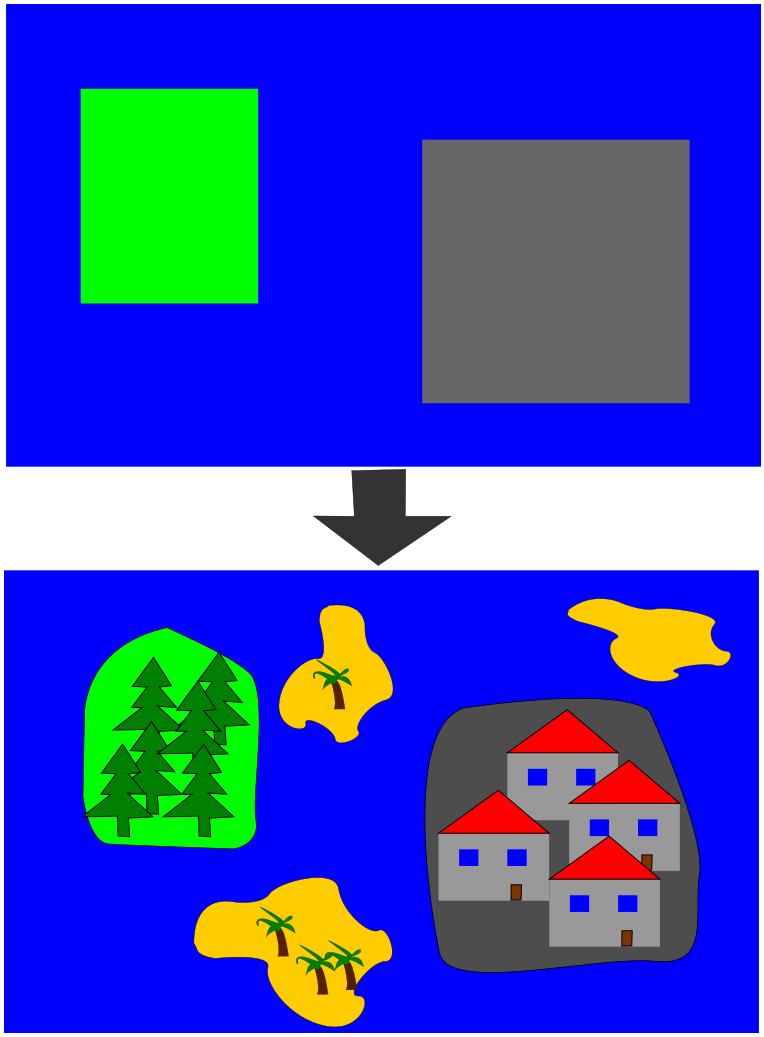

Figure 2.3: The core idea of the HWFC; start by generating a simple layout on top, and continue by adding details with more specialized WFCs on bottom

2.2.1 Enforcing Minimal Size

The idea of the hierarchical Wave Function Collapse algorithm is simple. However, there are several issues that we need to solve. Firstly, we want to run a new WFC on the part of the output of some other WFC. Therefore, it might cause an issue if the given part of the output is too small. Running a WFC over 5×5 area makes little sense as it can not generate any meaningful output. Furthermore, in some cases, the WFC will fail every time with such a small output. The minimal meaningful size of output varies heavily for each WFC. In general, the Overlap Model requires a larger output than the Simple tiled model.

We can enforce the minimal size directly using the Overlapping model with $N = 3$ by stacking different tiles next to each other. The exact pattern is shown in figure 2.4.

Using this pattern, we can guarantee any minimal size. The minimal size will be at least $2k \times 2k$, where k is the number of different stacked tiles. The

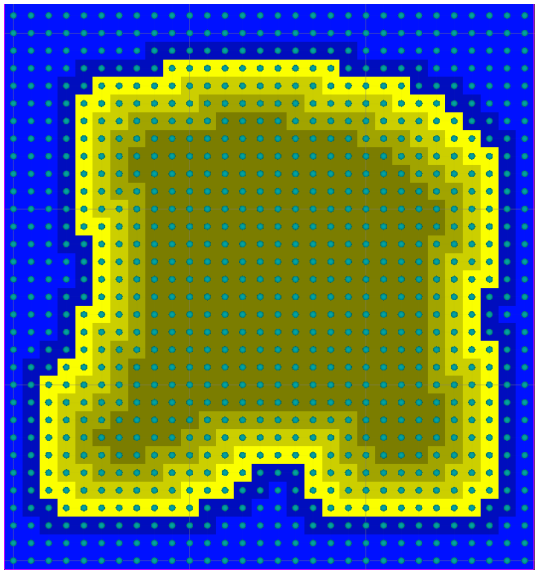

Figure 2.4: Expanding pattern for the Overlapping model $N = 3$, stacking several one tile width layers must also stack them on the output

middle can expand further if we leave at least the 3×3 pattern from the middle tile. Leaving the middle as only 2×2 will block the option for expanding. Then the whole pattern might be copied to the output. We can see that all tiles have different colors. It is not an issue since we only use them as a layout for WFCs in the next layer. Therefore, we will generate a different output over this one.

The simplest way, to ensure the minimal size using tile stacking is with a purely rectangular input. Figure 2.4 shows, that this method can be easily used even for more general shapes.

In this example, tiles have different colors for clarity and better visualization. Postprocessing could convert those tiles into the same color if necessary. However, the idea is that we will run another WFC on top of this one, so it is rarely required.

The expanding pattern might look performance intensive because it contains many tiles and even more patterns. In practice, it is not an issue. The problem is relatively heavily constrained because we have to generate those stacked layers one-by-one, and we cannot skip any of them. Therefore, once we start generating the pattern, the algorithm often collapses multiple tiles simultaneously.

2.2.2 Upscaling

It might be beneficial to generate the first abstract layer smaller. In the next step, we can upscale it. It will significantly help us since it simplifies achieving minimal sizes of areas, as mentioned in the previous section 2.2.1.

Here the idea of upscaling is straightforward. We can duplicate each tile into a 2×2 pattern or even larger. The duplication generally does not work very

well since we get rougher edges. However, here we will run another WFC on this abstract layer. Therefore, it will not be visible in the output. Also, we can make those rough edges smoothed using a simple 3×3 filter. Furthermore, generating smaller outputs has significant performance benefits since the runtime heavily depends on the size of the generated output.

We need to consider whether it is useful to upscale the map performance-wise since we will run another WFCs on the large upscaled version. This works well because the generated shapes are much smaller. However, there can be one large shape covering the majority of space shape. For example, when generating continents, there might be a single connected ocean covering the whole map. It is visible in figure 2.3. We can overcome this issue by dividing the ocean into four smaller chunks and running WFC on top of each chunk. Usually, these types of backgrounds are relatively easy to connect, so we can do it. On the other hand, splitting up continents would be much more difficult since it would be hard to connect them seamlessly.

2.3 Modifying Wave Function Collapse

The Wave Function Collapse algorithm is very powerful and flexible, but it is difficult to use and control. In the following section, we will discuss possible modifications to the algorithm. Those modifications are required for its usage in the HWFC. We will focus primarily on the Overlapping Model 1.3.1 with $N = 3$. We have found that this one works best for our requirements.

2.3.1 Early Tile Placement

The idea of Early Tile Placement is simple. We can collapse some cells manually before we run the algorithm itself. Preplacing a tile can be quite useful since we can guarantee to have some pattern on the output. For example, the early placement of a water tile guarantees that there will be a lake or a sea on the output.

Early tile placement is relatively straightforward to implement. We can pass a different list containing pairs $[position, pattern]$ to the algorithm. When the algorithm selects a cell to collapse, it will first check the list and collapse those cells. It is straightforward to do using the Simple Tiled model, where we can pass $[position, tile]$ directly. This idea was already implemented. [Cheng et al., 2020]. It gets more difficult for the Overlapping model. The original implementation converts all possible patterns into unique numbers. Then it internally works with just a grid of numbers for much better performance. This representation

is converted back to the bitmap once the algorithm finishes. Therefore, we need to specify the whole $N \times N$ pattern and transform it correctly into the internal representation of the algorithm.

Algorithm 4 WFC - Early Tile Placement (Overlapping model)

```
1: Input: width W, height H, pattern size N, input bitmap B,
   early placement list P
2: patterns ← all N × N patterns from B
3: weights ← #occurrences of each pattern
4: constrains ← which patterns neighbors which in B
5: propagator ← array (W × H) each contains a list of patterns.size of indices

6: // position (w, h) is collapsed ⟺ propagator[w, h].size ≤ 1
7: while not collapsed do
8:    if exist (w, h) ∈ propagator : propagator[w, h].empty then
9:       return failure {random restart here}
10:   end if
11:   if !P.empty then
12:      (w, h) ← P[0].position
13:      selected ← P[0].pattern
14:      P.remove(0)
15:   else
16:      (w, h) ← uncollapsed position from propagator with minimal entropy
17:      selected ← random weighted pattern from propagator[w, h]
18:   end if
19:   propagator[w, h] ← selected
20:   propagate this with constrains
21: end while
22: return propagator
```

In Algorithm 4, we can see how we can achieve Early tile placement. It is very similar to the original Wave Function Collapse algorithm 2. The major difference starts on line 11 where we first take cells to collapse from P. This version will modify the input list of early placement P. This might cause trouble when the algorithm fails and we need to do a random restart. If we would like to avoid this, using an *index* in the list is also sufficient.

We need to remember that doing this will significantly influence the output. It will ensure that some tiles look exactly the same as we set. Therefore, doing it on a large scale can hurt the variability of outputs. Moreover, it will influence the selection of cells to collapse. The minimal entropy heuristic will always select cells around already collapsed cells. Placing several tiles around different areas can cause a collapse of the map from multiple directions. It can avoid generation patterns containing large structures that spread over larger areas. Those structures are already constrained a lot from different sides. In comparison, the

standard WFC will pick a random cell and slowly expand the area around it. Here larger structures are less over-constrained.

2.3.2 Borders

In some situations, it is beneficial to set the borders of the result manually. For example, if we want to make a room over the whole output, we can place wall tiles around the edges. Another example could be generating ocean. There we can place the water tiles around the edges to avoid having parts of the islands cut-offed by the end of the output. We can use the knowledge of Early tile placement to add the borders. That means we can place tiles around the edges of the output.

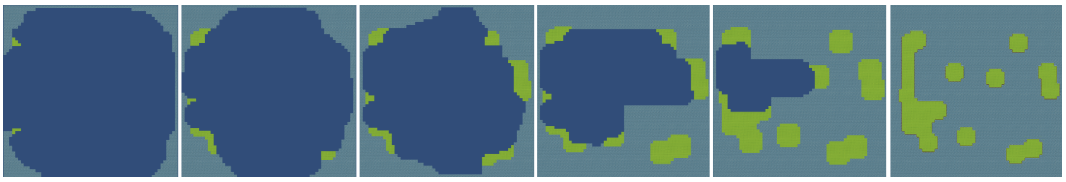

Figure 2.5: Impact of borders on the generation; each figure advances by 200 iterations

We can see an example of borders in figure 2.5. As mentioned in the previous section 2.3.1, this will influence the output. We will place tiles all around the edges, and the minimal entropy heuristics will start the collapse from there. It will further collapse all edges into the middle. This type of collapse has a consequence as it tends to avoid larger shapes that reach over many tiles because they can already be constrained for more directions. This effect is significant while using borders since the middle parts are already somewhat constrained from all four sides.

Figure 2.6: Standard generation without borders; each figure advances by 200 iterations

Figure 2.6 shows the same example as figure 2.5 but without borders. In these examples, it took one more patch of iterations to collapse the borderless example. It makes sense since placing borders already collapses a significant amount of cells. More importantly, the general shape of the collapse is different. Here, we usually get some area which is expanding further.

We can overcome the different nature of the collapse by adjusting the input accordingly. That means making patterns on the inside even larger than we would

regularly do. The weights of those interesting patterns will be larger than fillers, and it will balance out the side effects of using borders.

2.3.3 Non-rectangular Output

The standard Wave Function Collapse algorithm expects rectangular output $Width \times Height$. Generalizing it to non-rectangular output is necessary for the HWFC. In our case, the non-rectangular output is still a subset of grid tiles. There are three primary ways to do it, each with different advantages and disadvantages. The first option is to divide the output into smaller rectangular areas. The second option is to modify the algorithm to directly output a more general shape. The third option is to generate a rectangular output and follow it with postprocessing, where we can throw away unwanted tiles.

Dividing the area into only rectangular areas is simple. However, this can produce many small areas, and it would be extremely difficult to make the output nicely connected between those areas. Therefore, this approach is insufficient.

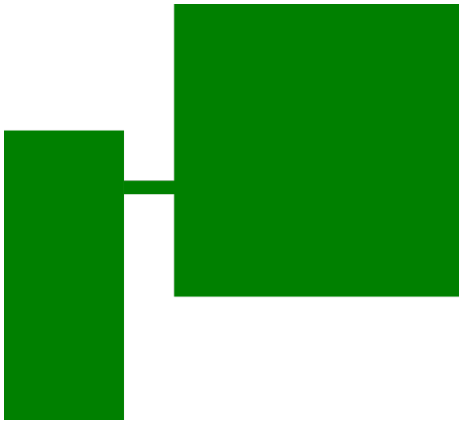

Figure 2.7: Complex shape with a narrow path in the middle

Suppose we would like to generate non-rectangular output directly. Then we must pick a different data representation instead of a simple two-dimensional array. Probably the best is to keep this whole array and add a bitmap describing which cell is not in the input. The next step would be to adjust the $Observe()$ method to pick only cells we want to collapse. The $Propagate()$ method must also be modified to only propagate to output cells. However, the direct generation has one big issue. There might be a path that is wide of just two tiles. This example is shown in figure 2.7. If we used the standard Overlapping model with $N = 3$, then the algorithm would fail every time. This situation can easily happen – for example, imagine a narrow wall between two rooms. If we used the Simple Tiled model, then this situation could not happen, and this approach might be optimal.

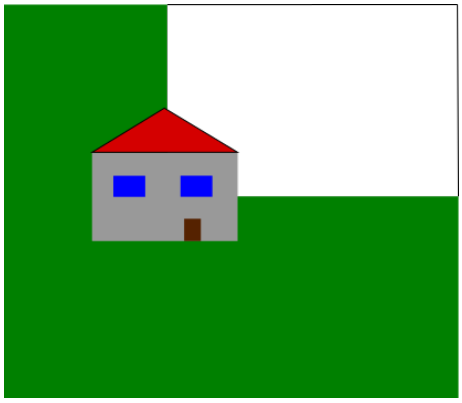

Figure 2.8: output over discarded area

The final option also has some issues. We can spend many resources generating parts of the output that will be later discarded. An even larger issue is that we can generate a house at the borders and discard a part of it, leaving degenerate output. Figure 2.8 demonstrates this situation. However, it can be overcome using Early tile placement 2.3.1. We can use some neutral pattern for discarded areas – for example, solid walls or grass. Therefore, we have chosen this option because it is the most reliable.

2.3.4 Auto-tiling

Wave function collapse thrives in keeping the structure of the input. For example, we can surround an island with special borders – each individual tile facing a given direction. Then the WFC will generate an output preserving this constraint.

On the other hand, this has one issue. Border tiles have fixed orientation, so we cannot use the *symmetry* settings in the WFC algorithm. The *symmetry* will sample the input with up to 8 rotations and reflections. Doing so will increase the variety of the output with relatively simple input. Therefore, we need to design more complicated inputs to produce variate outputs.

The alternative is to use auto-tiling. Auto-tiling means using one general tile, such as an island or wall, for the generation and later changing it to the proper version based on its surroundings. When the entire map is generated, the changes must be done as a postprocessing step. For this to work, we need to identify generated tiles. Fortunately, we need this information for the hierarchical process.

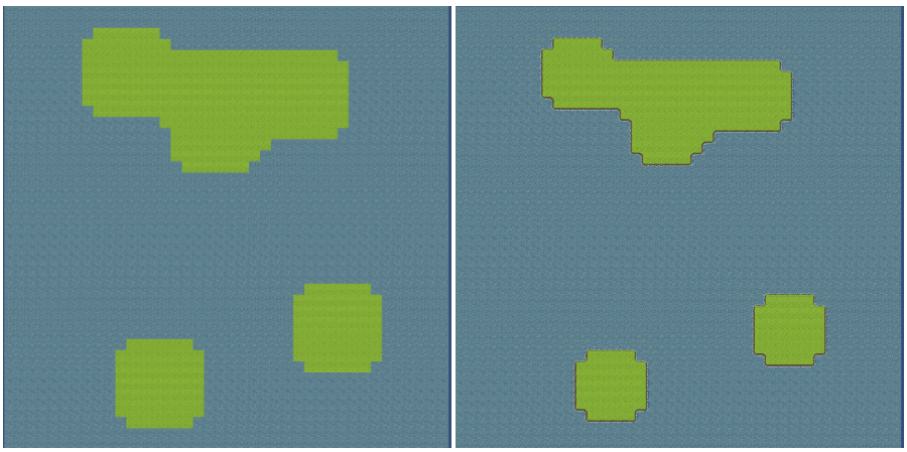

Figure 2.9: Example of auto-tiling; left is without auto-tiling, right is with applied auto-tiling

Figure 2.9 shows an example of auto-tiling. The right image has a smooth transition between the ocean and the islands. In our case, the islands get smaller, but it depends on assets.

3. Experiments

In this chapter, we will in-depth analyze the use of the hierarchical version of the Wave Function Collapse algorithm for generating different types of content. The primary purpose of this chapter is to figure out how difficult it is to generate content with the HWFC. We will later compare each of them with outputs obtained from the standard single-layer WFC. In all examples, we will use the Overlapping model 1.3.1 with $N = 3$ if not stated otherwise.

Firstly, we will explore a world map in the section 3.1. Later, we will look at generating a dungeon in the section 3.2.

3.1 World Map

This thesis aims to generate a simple world map since that would be almost impossible using regular WFC. The main reason is that a world map is heavily non-homogeneous, containing large continents with several different biomes and small islands. A single-layer WFC tends to generate a homogeneous output. This example can be found in the scene *WorldMap*.

We are using two tilesets for the world map. Those tilesets are available at https://opengameart.org/content/overworld-grass-biome and https://pipoya.itch.io/pipoya-free-rpg-world-tileset-32x32-40x40-48x48. Assets are distributed under CC0 license.

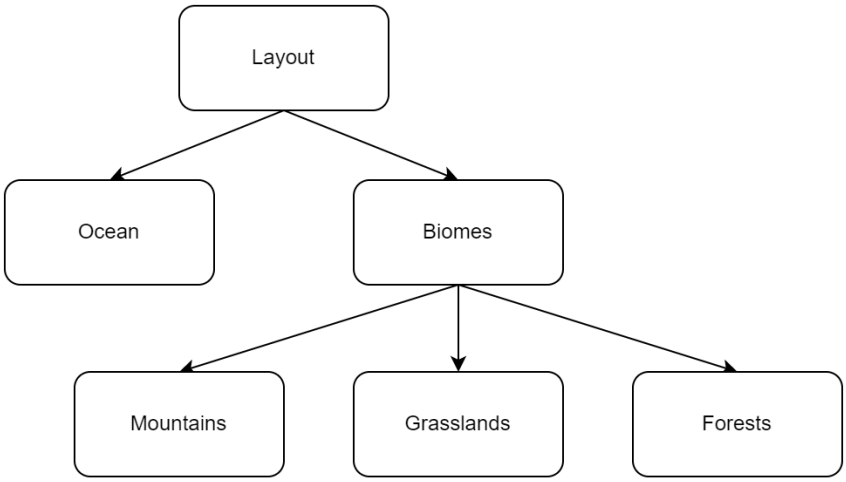

Figure 3.1: World map overview

Figure 3.1 shows the overview of the world map. Each box represents one WFC. Their placement represents layers. Arrows show their dependencies — the

WFC at the end of the arrow runs on top of the output from the other WFC.

3.1.1 The First Layer

The first layer is abstract, and we will call it the abstract layer. It tries to spit up the map into continents and oceans. One possible input for the abstract layer is shown in figure 3.2. The size of the input is 35×35 tiles. It is essential to think about the size of the input. Manually drawing the input is quite labor-intensive. Each input has a small cyan circle over each tile for better visualization. Outputs do not have those circles because it would be distracting, and we do not need them there.

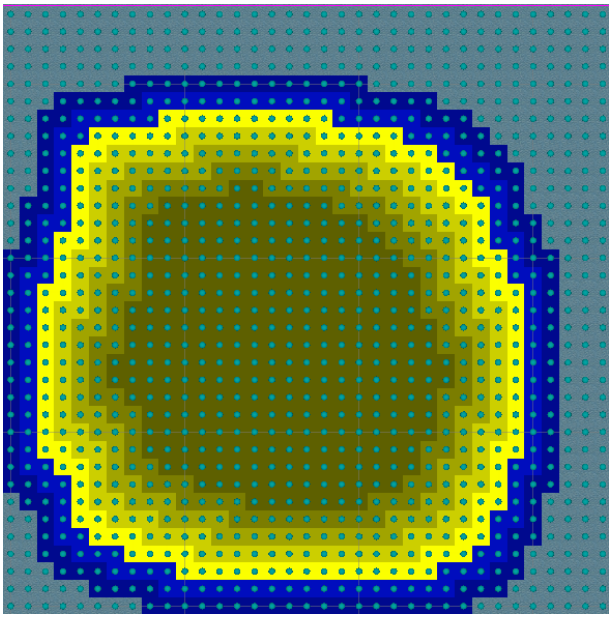

Figure 3.2: Abstract layer input

Two generated outputs can be found in figure 3.3. The size of each is 80×60 tiles. We use this size in our world map example. Therefore, those are much larger than the input. Usually, we will show more than one output. The reason is that we want also to demonstrate the variability among our outputs.

The blueish tiles specify an ocean. The yellowish tiles define continents. Both use the enforcement of the minimal size technique discussed in section 2.2.1. The yellowish tiles enforce the minimal size of each continent. The more tiles we stack, the larger those continents will be. The optimal size depends heavily on the size of the generated output. In our case, the output size is 80×60 tiles, as mentioned above.

Different types of water tiles are used to enforce a distance between continents. Otherwise, some continents could be right next to each other. One general ocean tile and two specific padding around the continent guarantee a 6 tile gap. We

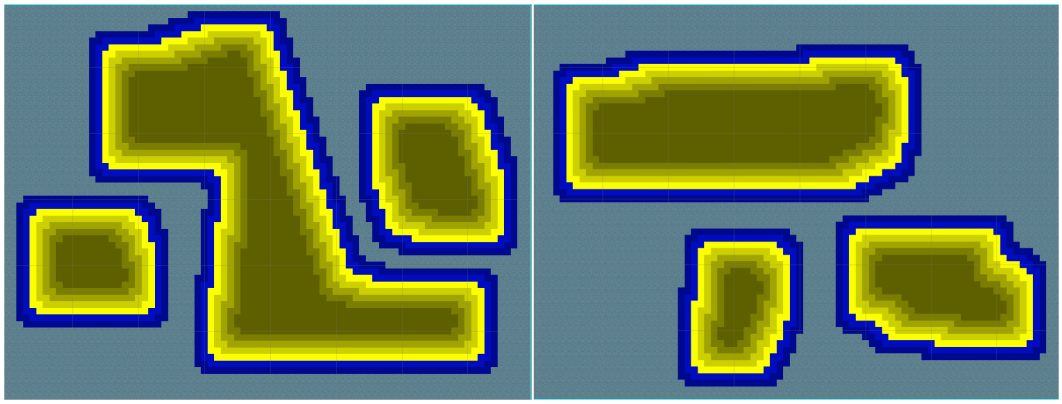

Figure 3.3: Abstract layer outputs

also place water tiles around the edges early to avoid having continents across borders.

We can take advantage of the map upscaling since the output from the abstract layer will not be a part of the final generated map. Using the upscaling for the abstract layer simplifies the input dramatically. Otherwise, we would need to provide much larger input with much more stacked tiles to generate sufficiently large continents. In our case, we use an upscale factor of two with smoothing around rough edges. Therefore, the whole output from the abstract layer is very similar to one shown in figure 3.3, only it is 160×120 tiles large instead.

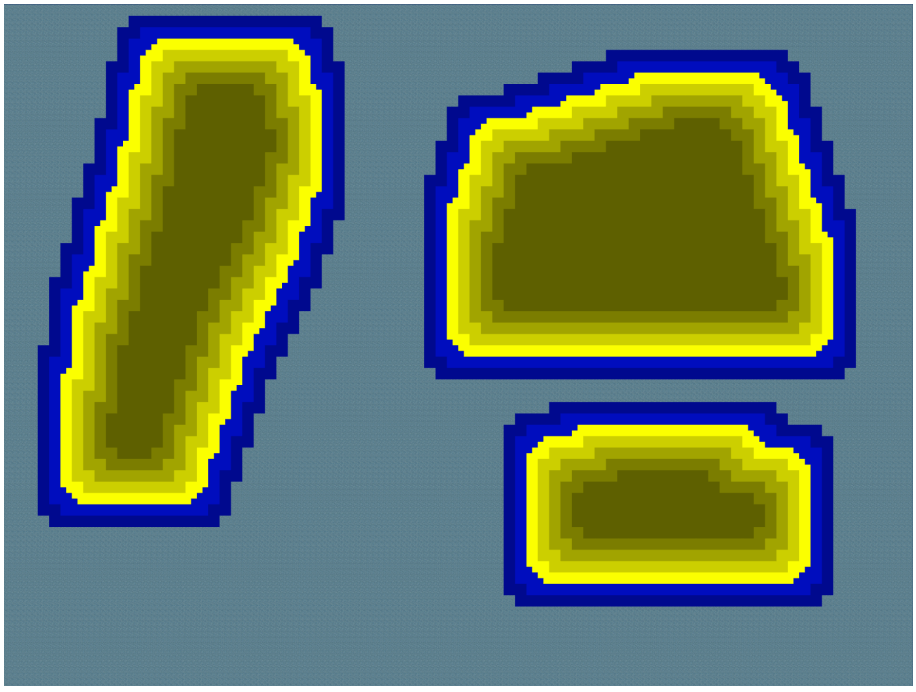

Figure 3.4: World map example ($seed = 1$) after the abstract layer is generated

Figure 3.4 will be our running example. It shows the output looks after the first layer. The size of the output is 160×120 tiles. Using this example, we will

demonstrate the influence of each WFC on the generation.

3.1.2 The Second Layer

In the second layer, we will generate oceans with islands. Also, we will split continents into biomes. Therefore, we have two WFCs in the layer called *Sea* and *Biomes*.

Sea

The *Sea* WFC is pretty simple. An example of the input is in figure 3.5. In our case, the input's size is only 30×30 tiles. This WFC will be visible in the final output since we will not run any more WFCs on top of this one.

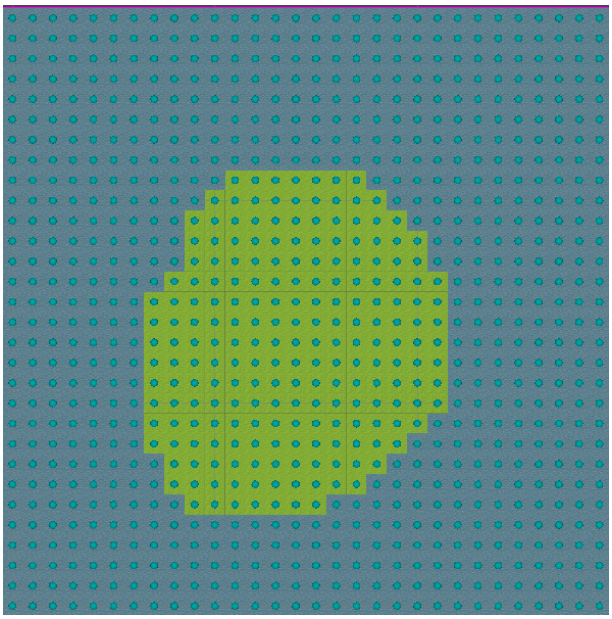

Figure 3.5: Sea input

We can see two examples of *Sea* outputs in figure 3.6. Each is 60×60 tiles since the generated output can be enormous. The size of the *Sea* WFC is the same as that of the final output. Therefore, we want to have this WFC simple. The first reason is that having fewer patterns will make it run faster since we need to propagate less information. The second reason is that we do not want this WFC to fail because, due to its size, it can take a significant portion of the total run-time.

Similarly to the abstract layer, several tiles are stacked to enforce the minimal size of the island. In our case, there are three grass tiles. Those tiles need to have the same color since they will be visible in the final output. The amount of water around the island influences the island's density. Adding more water will

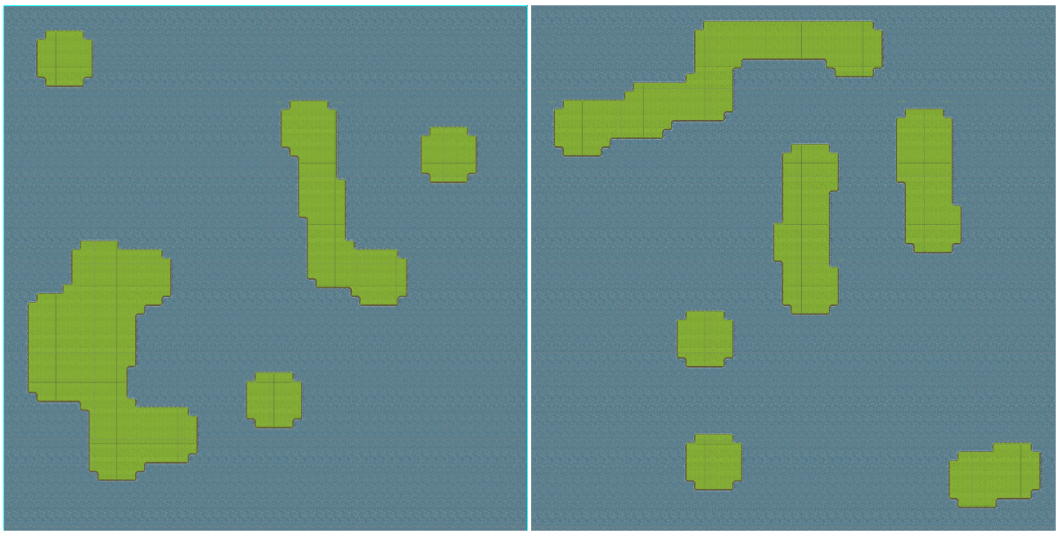

Figure 3.6: Sea outputs

reduce the amount and the size of generated islands. In our case, there is quite a lot of water since we do not want islands everywhere. The *Sea* WFC also uses auto tiling to put nice edges around islands and to provide seamless transitions towards the water. We can see it in figure 3.6. Auto tiling gives us the option to use higher *symmetry* settings. We could avoid using auto tiling. However, we would need to provide more detailed input containing several islands. Otherwise, we would lose some variety among generated islands.

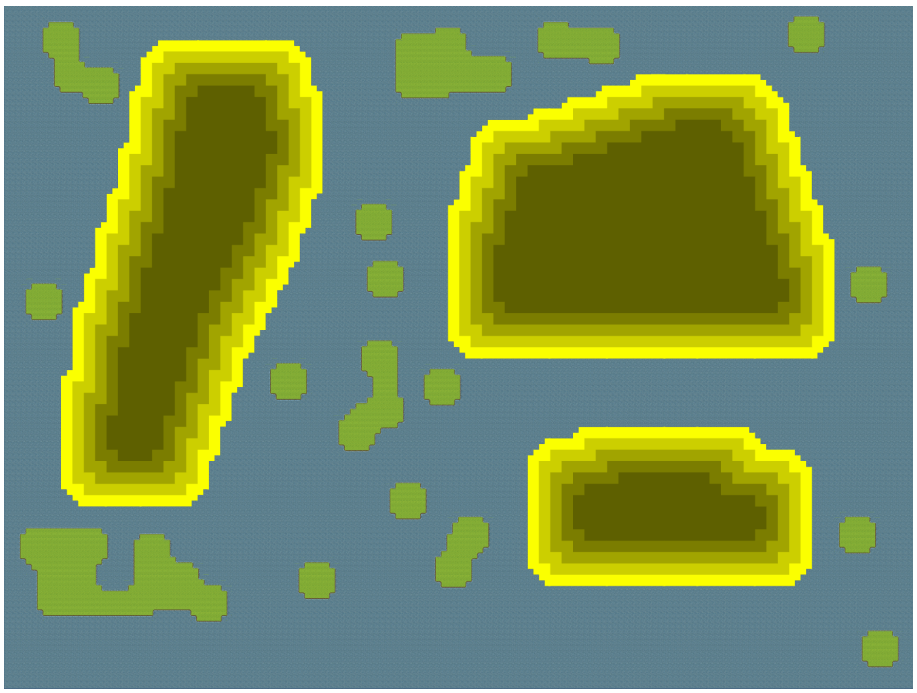

Figure 3.7: World map example ($seed = 1$) after the sea in generated

Figure 3.7 shows the same world map as figure 3.4. This time, after the sea, is

generated. It spawned many islands of different sizes. We can see that it cutoffs blue parts of the continents. As mentioned earlier, those serve as a delimiter between continents and are part of the sea.

Biomes

The other WFC in this layer is *Biomes*. An example input is shown in figure 3.8. Similarly to the abstract layer, we will run more WFCs over this one. Therefore, it will not be visible in the final output. This input is more complex, containing 40×40 tiles.

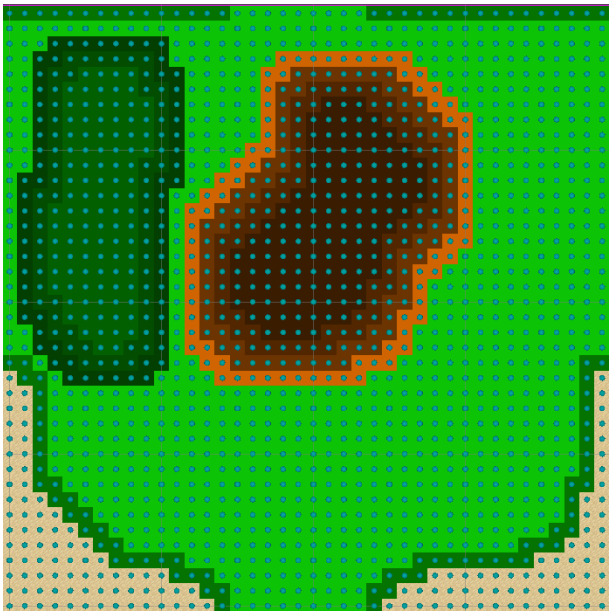

Figure 3.8: Biomes input; here is yellow sand at the bottom, green grasslands as the majority of input, brown mountains, and dark green forests

There are four parts to this input: sand, grasslands, forests, and mountains. We want to have some sand around the edges of each continent. Therefore, we use sand as borders and for discarded areas. However, we want to avoid having those continents only from the sand. Therefore, there is a bit of sand placed in flexible patterns. By flexible pattern we mean, that almost any 3×3 shape can be formed out of the sand using $symetry = 4$. Most of the input is grass. We will guarantee to have sand around the edges by placing it early. Then a large amount of grass will force the WFC to quickly switch towards grass as the collapse gets further away from the edges. Most of the generated output will be grasslands. Therefore, we again enforced the minimal size of forests and mountains to be 6×6 and 8×8 tiles, respectively. We are again using slightly different colors for better visualization.

Examples of outputs for *Biomes* are in figure 3.9. The size of each output here is 50×50 tiles. A larger size is because *Biomes* will cover the whole continent.

Figure 3.9: Biomes outputs

There are small chunks of sand over the continents. Those are generated as a consequence of the sandy edge. In this case, we do not might them. They help to break down the monotony of the generated continents. Alternatively, postprocessing could be used to remove those chunks if required. We are using a second tile for *Grasslands* – the darker green around the edges. Its purpose is to keep those sandy chunks further away from each other.

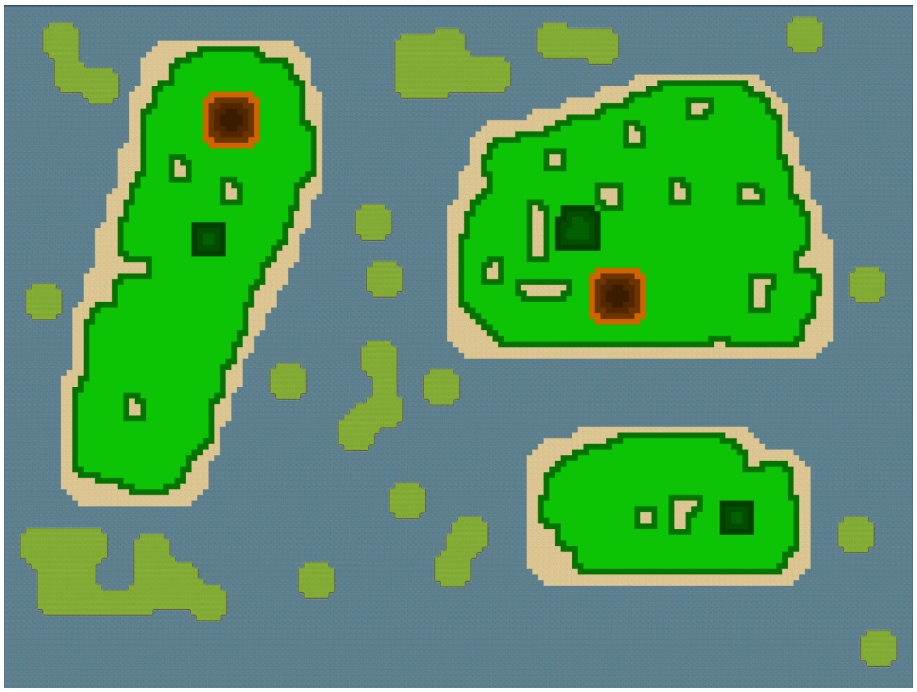

Figure 3.10: World map example ($seed = 1$) after the biomes are generated

Figure 3.10 shows the running example after generating the biomes. It is actually the output of the whole second layer. In this case, there are three

instances of the *Biomes* WFC, one over each continent. The generated ocean is the final. In the third layer, we will finish the generation of continents. As mentioned above, the generation of the ocean takes much more time because we run it over significantly larger area than continents. We showed only one example of the output here. The main reason is that the map is quite large, so putting two next to each other could make it hard to see. The other reason is that this is just the middle step of the generation, and we will focus more on the final results later.

3.1.3 The Third Layer

The third layer is the final layer for our world map example. Its purpose is comparable to the standard single-layer WFC because it generates the final output. There are three WFCs in this layer, namely *Grasslands*, *Mountains*, and *Forests*. One huge advantage of the hierarchical approach is that those WFCs are almost independent, and we can design each without looking at others. We only need to keep this in mind to guarantee smooth transitions between them.

Grasslands

We will start with the *Grasslands* WFC. This WFC is the most complex. The input is 35×35 tiles. We can see its example in figure 3.11. It also generates villages and paths between them. It might be better to generate paths as a post-processing step since that would give us more control our the output. However, we want to push the WFC algorithm to its limits.

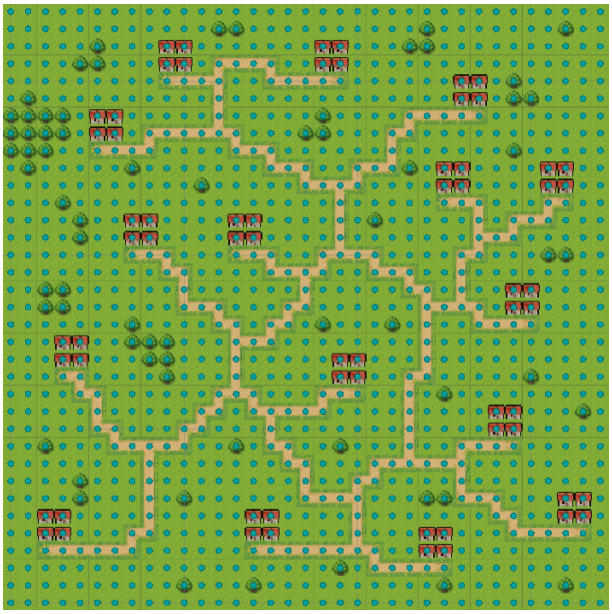

Figure 3.11: Grasslands input

The first crucial observation is that under each village is always a unique end-of-road tile (either left or right). Under each village, there is precisely one of them. It forces each village to be connected to some path. Otherwise, we could generate a village with two path tiles underneath that would not lead anywhere. The second observation is that the path cannot suddenly end. This forces each village to connect with at least one other village. More precisely, the village can also be connected to itself via loops of the path, but this rarely happens with our input. Not all villages will be connected to a single pathway network. That would constrain our algorithm way too much.

In the input, we have a high density of villages. However, we can expect a lower density in the output since WFC tends to avoid more complex structures. Moreover, the village is relatively complex as it also needs a path to different villages. Auto tilling could be used again to generate a correct path shape. However, here it does not allow us to use higher values *symmetry* because of the villages. Therefore, we are not using auto-tilling for the generation of paths. There are also some trees in the *Grasslands* to add more variety.

We need to be careful when generating non-rectangular shapes. We will place a 3×3 pattern of solid grass in areas that will be later removed. It is crucial because we want to avoid having a village or a path suddenly cut-offed.

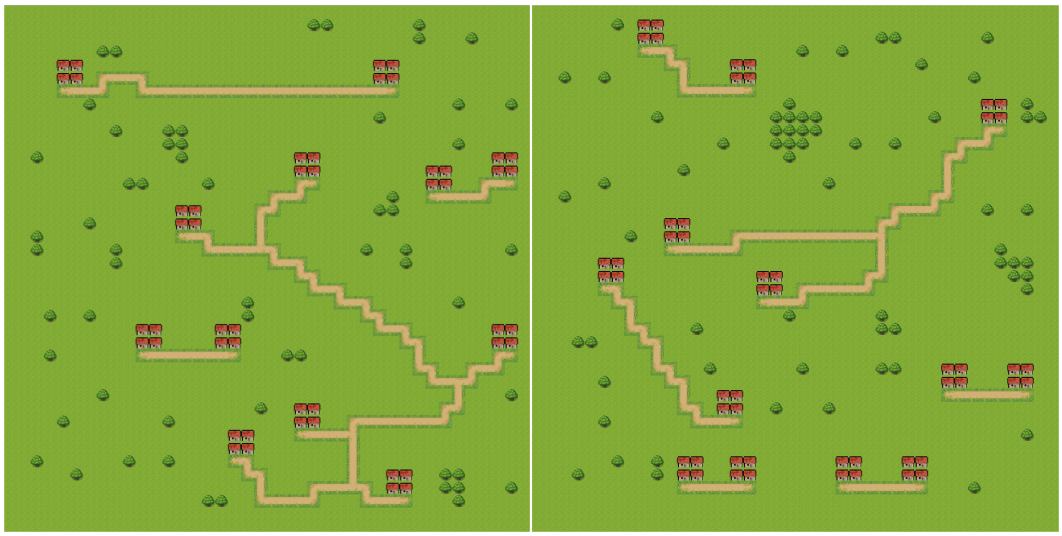

Figure 3.12: Grasslands outputs

Figure 3.12 shows two examples of generated grasslands. The size of each output is 40×40 tiles since those will cover most of the continent. Especially in the right one, we can see numerous pairs of close villages connected with a path. It will be useful when we add mountains or forests into the middle of the grasslands. Then it would be very challenging to connect multiple villages far away.

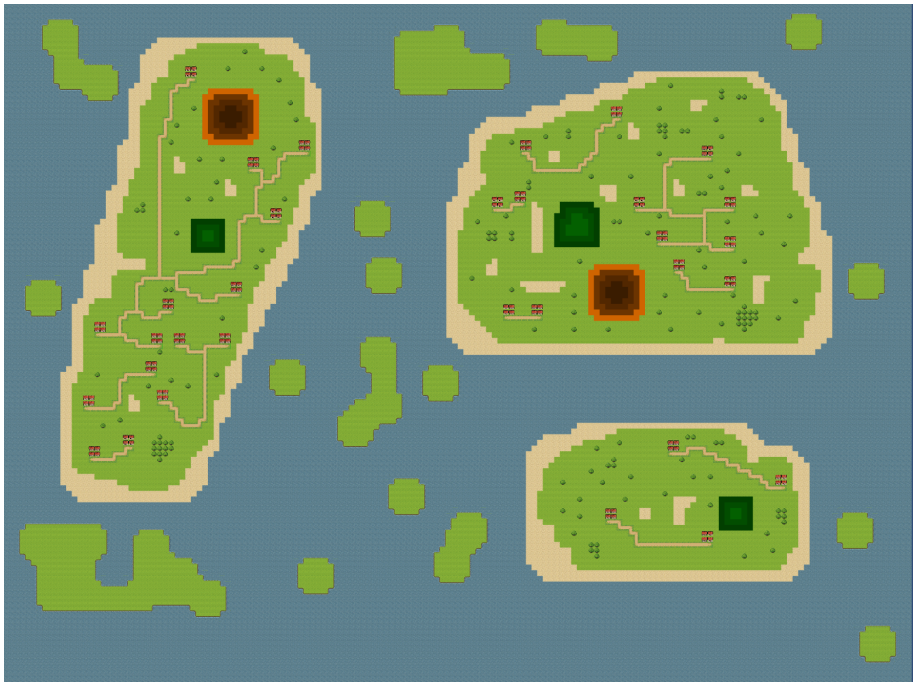

Figure 3.13: World map example ($seed = 1$) after the grasslands are generated

Figure 3.13 shows the world map example after the grasslands. We can see that most of the generation is already done. The grasslands can generate complex path networks even if they are limited by sandy parts, mountains, and forests in a way.

Mountains

The two remaining WFCs are much more straightforward in comparison to grasslands. *Mountains* WFC will generate simple mountains on the continent.

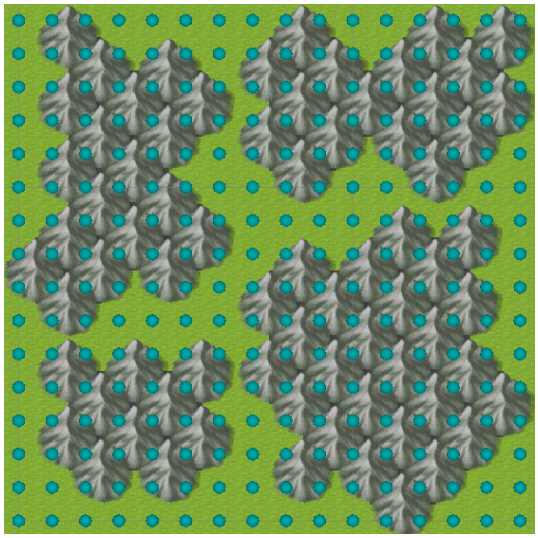

Figure 3.14: Mountains input

We can see the input for *Mountains* in figure 3.14. The size of the input is only 16×16 tiles. It will early place grass tiles around the edges to make the transitions between *Grasslands* seamless. The critical point is that *Mountains* uses the $N = 2$ instead of the regular $N = 3$. Firstly, we can afford it since a hill is created of 4 tiles with an ability to connect more of them, and for it, $N = 2$ is sufficient. Secondly, it even helps us because we need to place only the 2×2 pattern of grass around the edges instead of the regular 3×3 pattern. This approach makes the area covered by hills larger. Finally, $N = 2$ allows the input to be smaller and simpler.

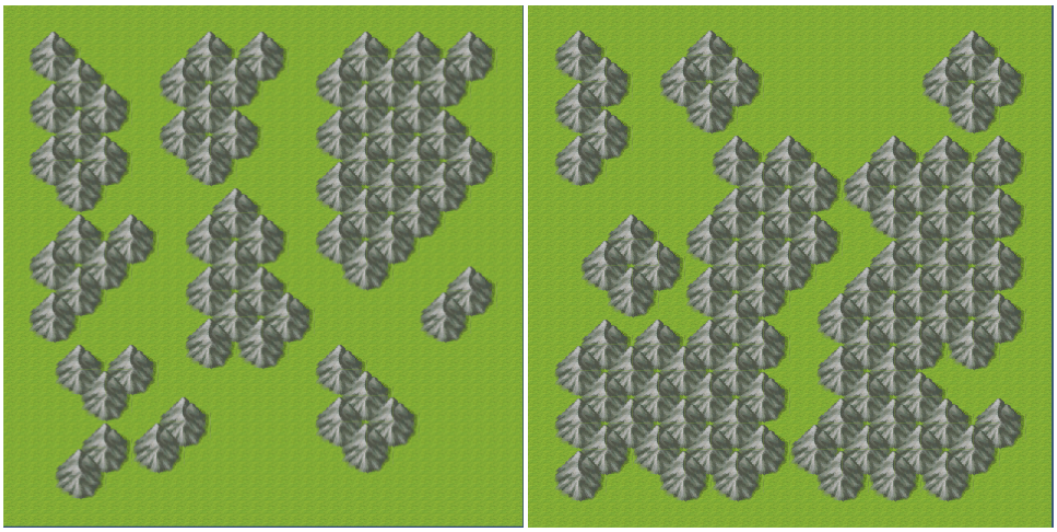

Figure 3.15: Mountains outputs

Two outputs of *Mountains* WFC are in figure 3.15. The size of each is only 20×20 tiles. This WFC will usually run in smaller areas, so we do not need a large sample. There is still a relatively large area covered by grass. We do not mind it since it blends smoothly into the final output.

Figure 3.16 shows the example with mountains. It is a minor change since mountains cover a tiny part of the continents. However, they still add variety, and those small WFCs are very fast.

Forests

Finally, the last WFC is *Forests*. This one is for sure the simplest.

We can see the input in figure 3.17. It has only 12×12 tiles making it the smallest input. It is also almost trivial, as the layout of the trees does not matter. We mostly care about their density.

Similarly to *Mountains*, it again uses the $N = 2$ as it is sufficient here. *Forests* is the only WFC that does not early place any tiles. It is just not required here. There are not only trees but also some grass. We do not want

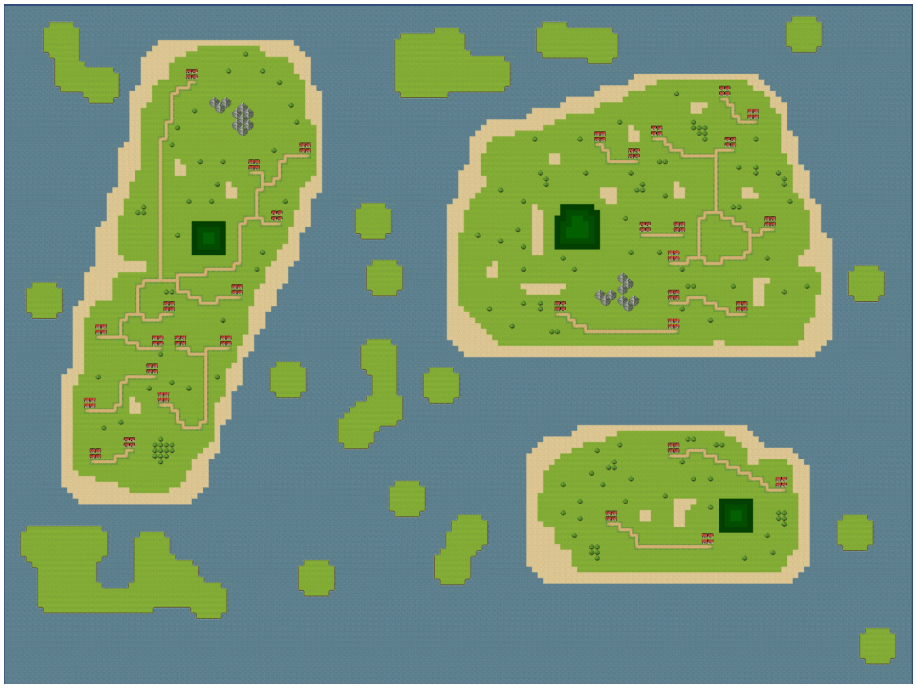

Figure 3.16: World map example ($seed = 1$) after the mountains are generated

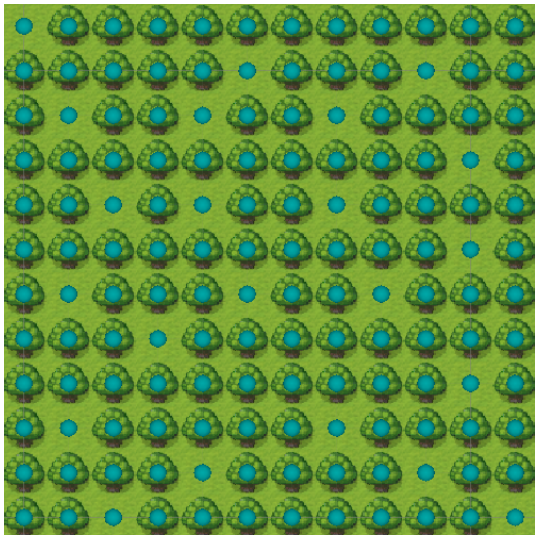

Figure 3.17: Forest input

to have solid chunks of trees only. Besides that, it again helps with smoother transitions between *Grasslands*.

It might be tempting to avoid this WFC and add large areas of trees directly into the *Grasslands*. However, it will still tend to avoid large areas of trees that are not all over the map. Although using the concept of tile stacking to ensure the minimal size could help, splitting it into two WFC also reduces the overall complexity. The other advantage of breaking it into more WFCs is that we can get more variety over those forests. Otherwise, it would need to be a large part of diverse forests into *Grasslands*. That would hugely increase the overall size

and complexity of *Grasslands* since it would need to achieve a similar density for villages and paths compared to trees. The main idea of Hierarchical Wave Function Collapse is to reduce this complexity and to use several much simpler WFCs.

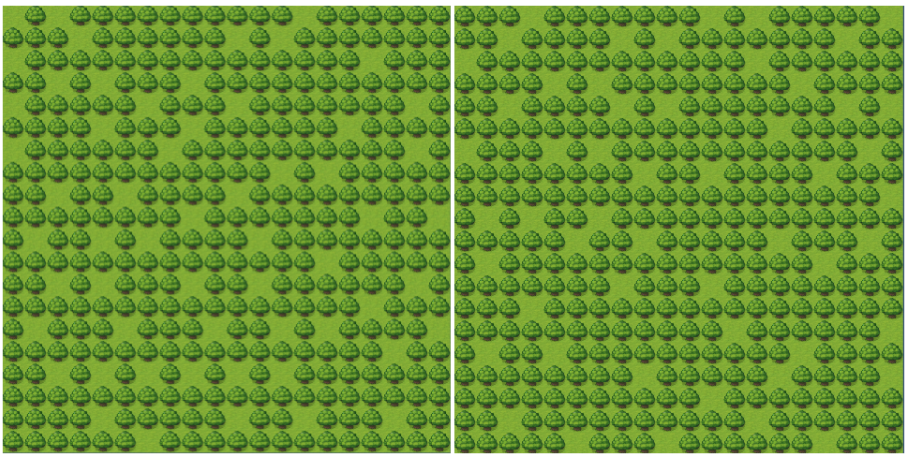

Figure 3.18: Forest outputs

Two outputs of *Forests* are in figure 3.18. The size of each is 20×20 tiles. However, even smaller outputs would be sufficient. We are consistently generating these over smaller areas.

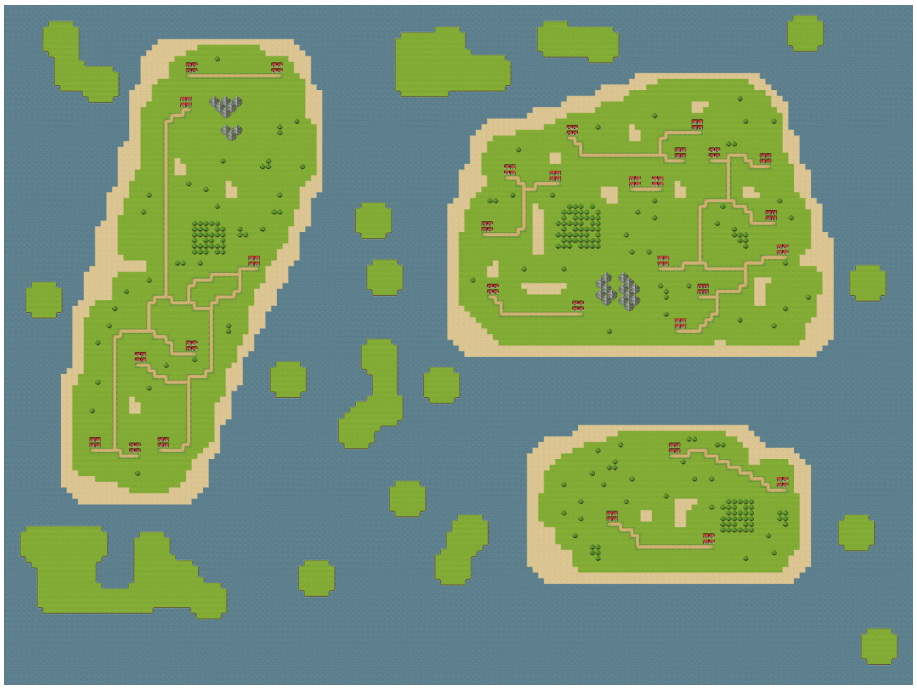

Figure 3.19: World map example ($seed = 1$) after the forests are generated

Figure 3.19 shows the world map after adding the forests. It is the final version of the world map since there is no postprocessing. We will show much more of them and discuss the outputs of the HWFC in the next chapter.

3.2 Dungeon

A dungeon is the second example we will try to generate using the Hierarchical Wave Function Collapse. On the one hand, a dungeon seems to be much less complex and structured than a world map. It is partially true. On the other hand, our dungeon has some properties that are hard to achieve with the WFC algorithm. Our dungeon needs to meet the following design criteria.

Dungeon design criteria:

- All areas have to be reachable
- There is a single large open boss room in each dungeon
- There are both Cavern-like and Mansion-like areas

For all of the reasons mentioned above, we will need to combine HWFC with quite a lot of postprocessing. This example can be found in the scene *Dungeon*.

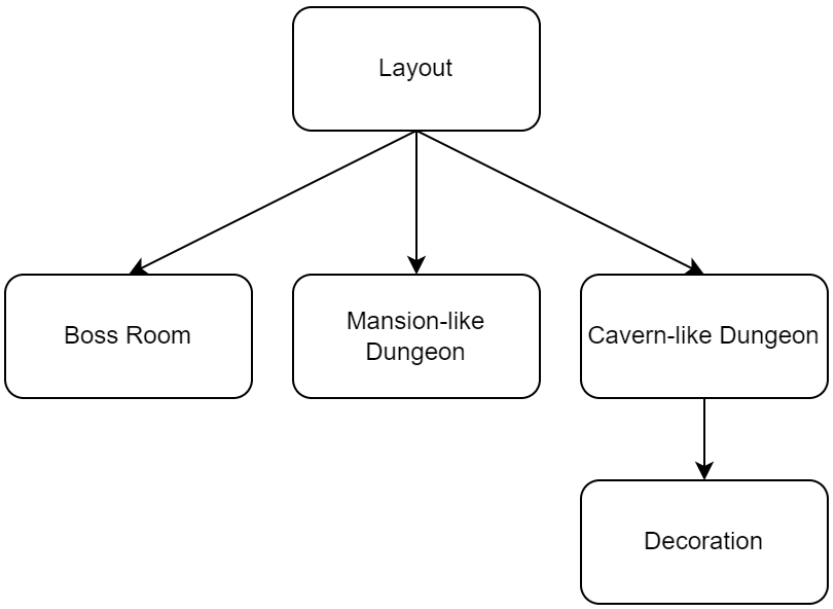

Figure 3.20: Dungeon overview

Figure 3.20 shows the overview of the dungeon generation and its layers. We are again using three layers, and we will look at them more profoundly in the following sections.

All assets used for the dungeon are handmade.

3.2.1 The First Layer

The first layer has the same purpose as the first layer of the World Map. It divides the map into large regions, each serving a different purpose. We have three types of regions. The first one is a boss area. The second one is used for a cavern-like part of the dungeon. The third one is used for a mansion-like part of the dungeon. We will deal more with them in the following subsection. The input for the abstract layer is shown in figure 3.21. The size of this input is 40×40 tiles.

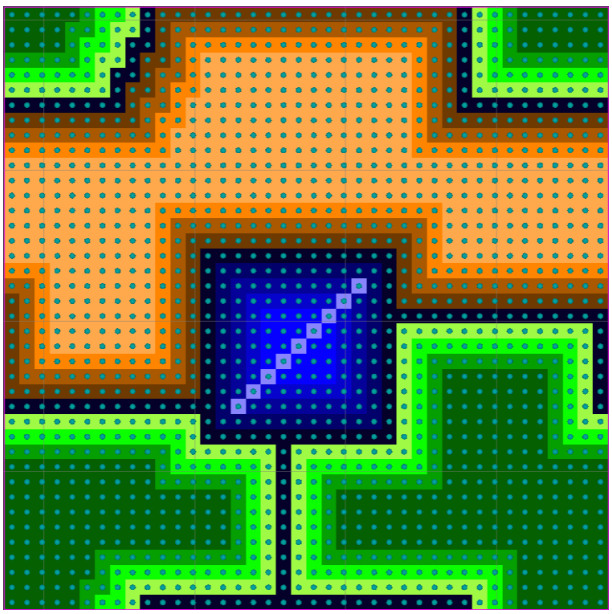

Figure 3.21: Dungeon — Abstract layer input

The darkest blueish tile makes a gap between two adjacent areas. None of WFC will be run on top of it. There is one fundamental difference compared to the World map. In the world map, we used an ocean as a default tile and generated continents on top of it. Here, we want to generate regions, and none is a default. Therefore, we choose a different approach to designing the WFC input. The input might look strange. We are heavily utilizing the impact of the periodic input. Therefore, we can connect areas over the edges. We also want those layouts for rooms to be irregular and roughly similar in size.

Two outputs are shown in figure 3.22. The size of each output is 50×50 tiles. We will again upscale it by a factor of two. To guarantee a boss room, we can preplace one of its corners at a pseudo-random position close to the corner of the map. The position must be directly determined from `seed` 5.3.2 to make the results reproducible. This will force our map to have at least one boss room. However, there is a chance that we will have more than one boss room. It will be just a big open room, so we do not mind it. Alternatively, the boss room tile

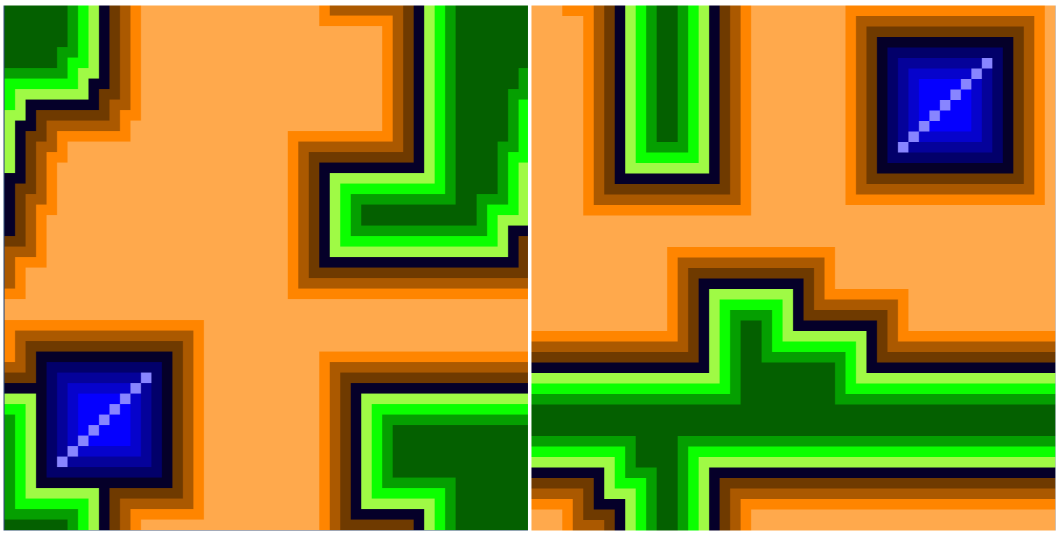

Figure 3.22: Dungeon — Abstract layer outputs

could be banned from the generation in areas further away. This feature is not hard to implement and is very similar to the Early tile placement 2.3.1. It was even implemented for the Simple Tiled model [Cheng et al., 2020]. We choose not to implement this feature. Adding more features would make the HWFC even more difficult to use since we would need to specify an extra parameter for each WFC.

There are custom tiles around one diagonal of the boss room. Those tiles will force the room to be a square. It is undesirable to have a long but narrow boss room. There is no straightforward way to force the aspect ratio of the room close to one. Therefore, we will force it to be precisely one all the time.

Similarly to the world map, we will show the progress of the HWFC in the dungeon. Figure 3.23 will serve as our running example for it. It shows the output looks after the first layer. We have again used upscaling with a factor 2. The size of the output is 120×100 tiles.

Figure 3.23: Dungeon example ($seed = 1$) after the abstract layer is generated

3.2.2 The Second Layer

The second layer places down all rooms. Therefore, it has a significant impact on the final result. We have three types of rooms, which we will describe in the following subsections. All of the WFCs are using the same assets. The major differences are in their structure.

Boss Room

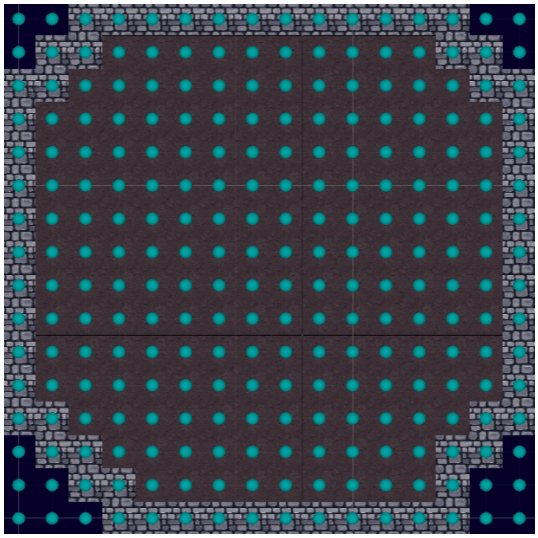

Figure 3.24: Boss room input

The boss room should be a large open area ideal for a boss fight. Therefore, we want to fill most of the region. We can see the input in figure 3.24. The input is tiny, only 16×16 tiles.

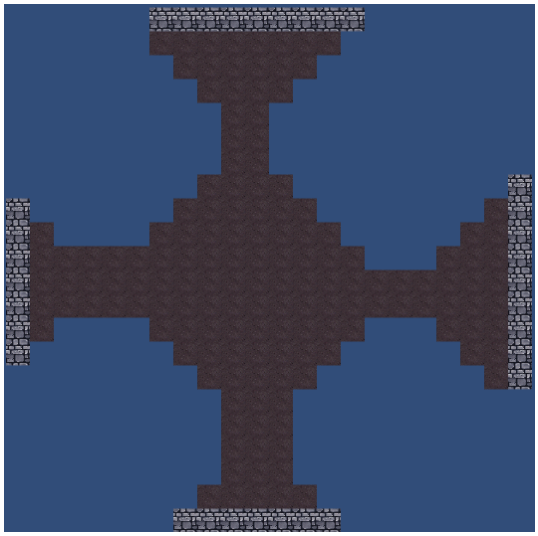

Figure 3.25: Boss room output after a single iteration

Most of the work is done by an intelligent early tile placement. It places a pattern with a wall tile around the bottom edge's middle section and the right edge's middle section. Those patterns have ground tiles attached to the middle of the room. After a single propagation, most of the room is filled with ground tiles. Since we are using periodic input and output, the WFC is forced to place walls on the remaining two edges. Those are the only possible patterns. The example of an input after a single iteration is in figure 3.25. The blue color is the background. It represents tiles that are not yet collapsed. The rest of the middle area will be filled by ground in a few more iterations. The WFC can only influence the shape around corners. This makes the small size of the input sufficient.

Two example outputs are in figure 3.26. Each of them is 22×22 tiles large. We picked this size since it is the actual size in the final dungeon from the abstract layer after an upscaling. Both outputs are quite similar. Our approach dramatically reduces the possible variety of generated outputs. We compensate by using the `symmetry = 8` option to get more variety. Furthermore, it always generates a large open room which is desirable.

We could also cover less area which the preplacement of walls. It would make the WFC effectively run on a larger area. However, it might produce degenerate outputs if we pick the area to be too small.

Figure 3.27 shows the running example of the dungeon after the boss room is generated. We can see it in the bottom right corner. However, the remaining two

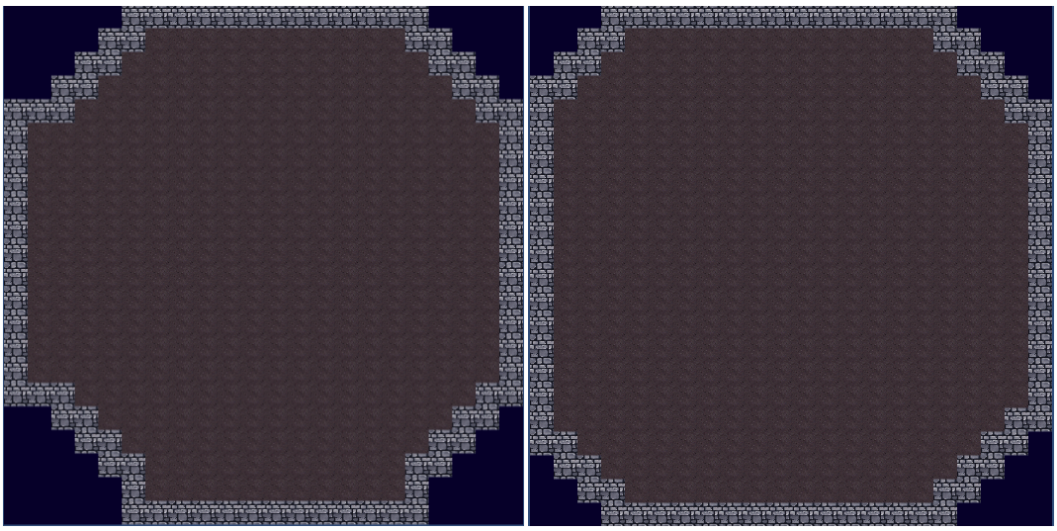

Figure 3.26: Boss room outputs

Figure 3.27: Dungeon example ($seed = 1$) after the boss room is generated

WFC will have a much greater impact on the result since they cover significantly larger areas.

Mansion-like Dungeon

Mansion-like part of the dungeon should look like a mansion. It should have simple, mostly rectangular rooms connected with straight paths.

These rooms are easily generated with the WFC shown in figure 3.28. The input size is only 16×16, and we use `symmetry = 4` to increase variety. We are

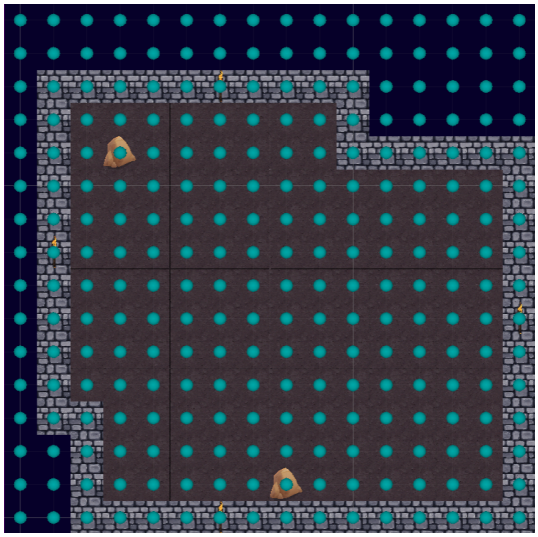

Figure 3.28: Mansion-like dungeon input

stacking three different ground tiles to enforce the minimal size of the room. It will make the inside at least 6×6 tiles large. Some walls also contain torches, which are not in the cavern-like sections of the dungeon. There are also some rocks to add variety. It is possible to generate paths with WFC. [Scurti and Verbrugge, 2018] However, it is tricky because we want to connect the whole dungeon, and the grasslands example has already demonstrated its difficulty. Therefore, we will add them at the end with postprocessing.

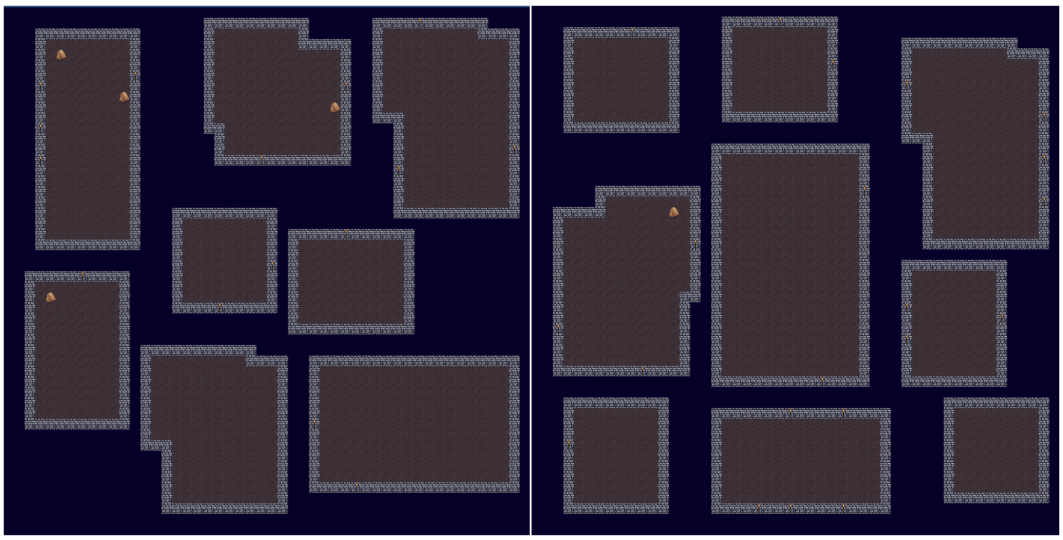

Figure 3.29: Mansion-like dungeon outputs

Figure 3.29 shows two examples of generated outputs. The size of each is 50×50 since those will have a decent size on the final output. We can see that it generates multiple tightly packed rooms. And those rooms to either rectangular or almost rectangular. It is ideal for the mansion-like part of the dungeon.

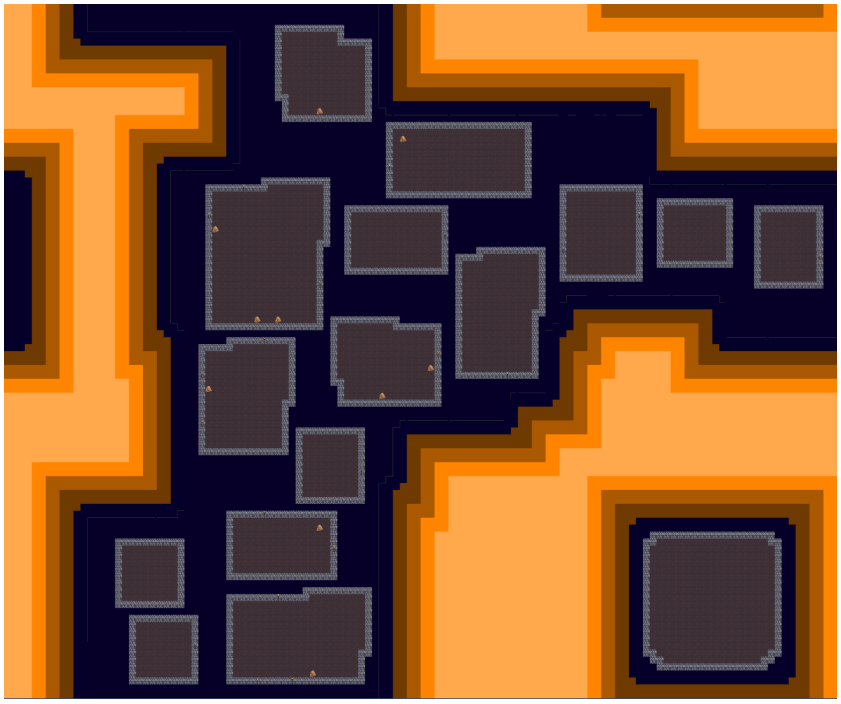

Figure 3.30: Dungeon example ($seed = 1$) after mansion-like parts are generated

Figure 3.30 shows the dungeon after generating all mansion-like segments. In this example, most of the dungeon is made of mansion-like rooms. Therefore, it has a very significant impact on the overall output.

Cavern-like Dungeon

The final WFC in the second layer is a cavern-like part of the dungeon. It means many more open areas and irregular rooms.

For the generation, we are using a highly irregular room. The input is shown in figure 3.31. Its size is 30×30 tiles, making it much larger than the mansion-like input. As always, we use the tile stating of different ground tiles to enforce its minimal size. Here we have three ground tiles again stacked. We get the most variety in the `symmetry = 8`.

This WFC uses postprocessing that connects those cavern-like parts. Parts are connected by wide paths to make them blend in. The width of paths is 5 tiles inside plus walls. Figure 3.32 shows an example of the generated map. The left image is without the postprocessing, and the right one has already added paths. The output size is 60×60 tiles since the cavern-like parts are usually significant. Since figure 3.32 shows only one output, there is one more example in figure 3.33 to show more variety. The generated maps are good as they provide a single large open area.

The postprocessing is complex. The area where we run this WFC is not

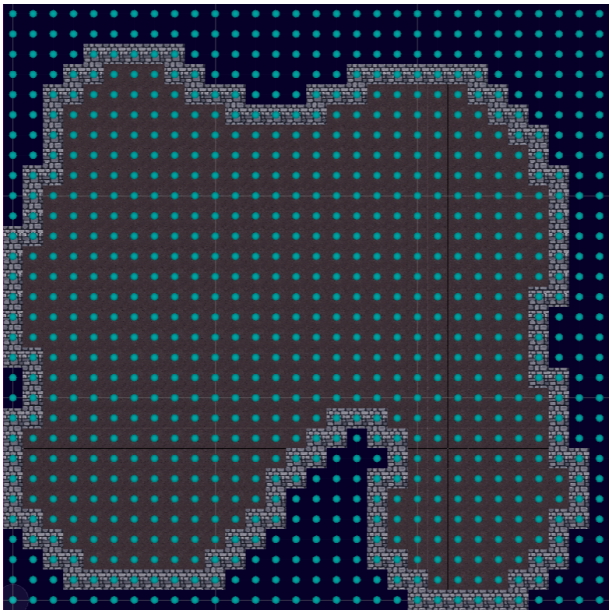

Figure 3.31: Cavern-like dungeon input

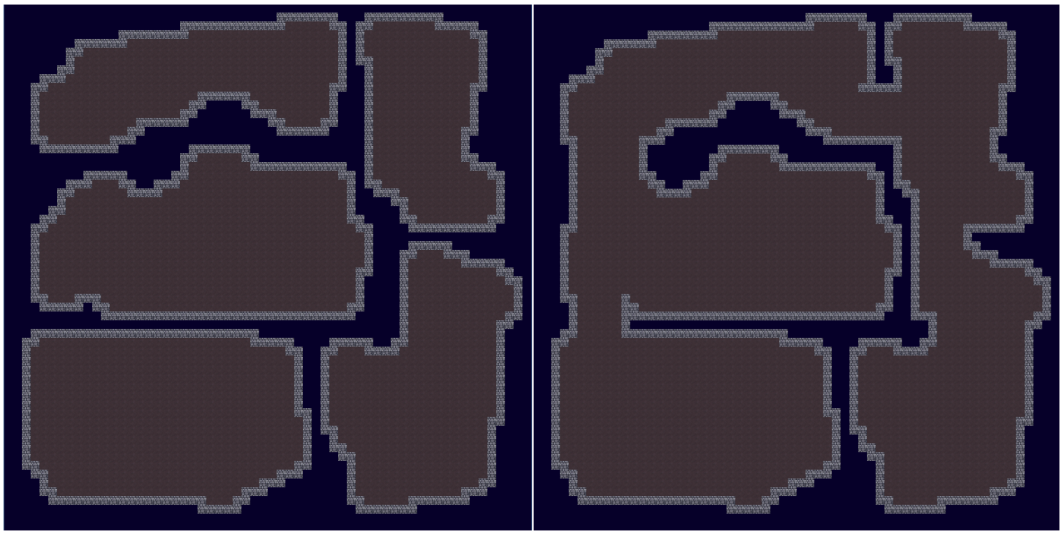

Figure 3.32: Cavern-like dungeon output 1 – left generated map, right is after postprocessing

rectangular most of the time. However, we cannot create a path across a different area. Those paths would be removed in the next step, and we would get hanging paths at the area's borders. Therefore, those paths must be created only on top of the given area. Fortunately, the postprocessing can access a bitmap that specifies which tiles are inside.

Figure 3.34 shows the dungeon after generating all cavern-like segments. In this example, there are three, each having a decent size. We can clearly distinguish between the mansion-like and cavern-like portions of the dungeon. However, it still appears that those parts belong together.

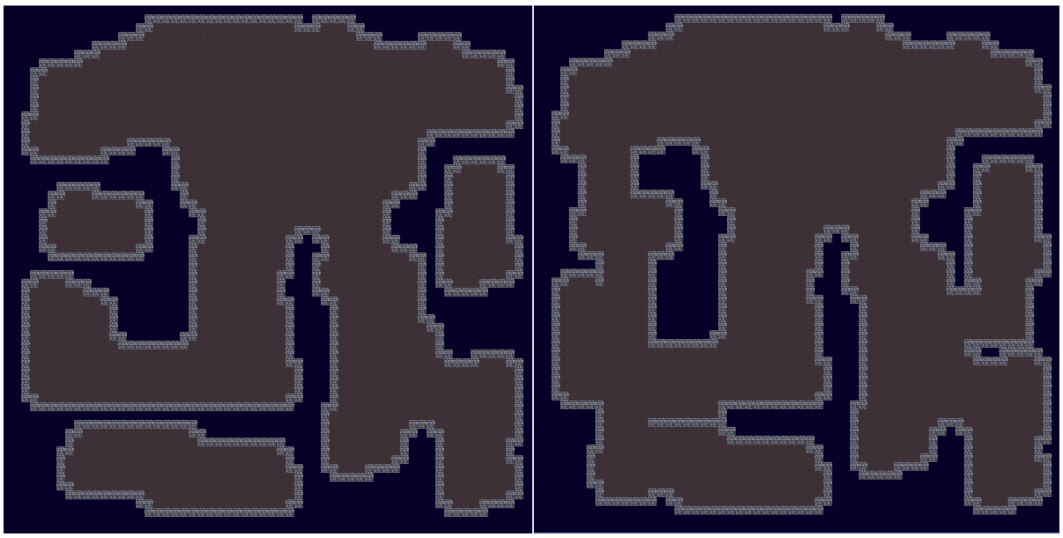

Figure 3.33: Cavern-like dungeon output 2 – left generated map, right is after postprocessing

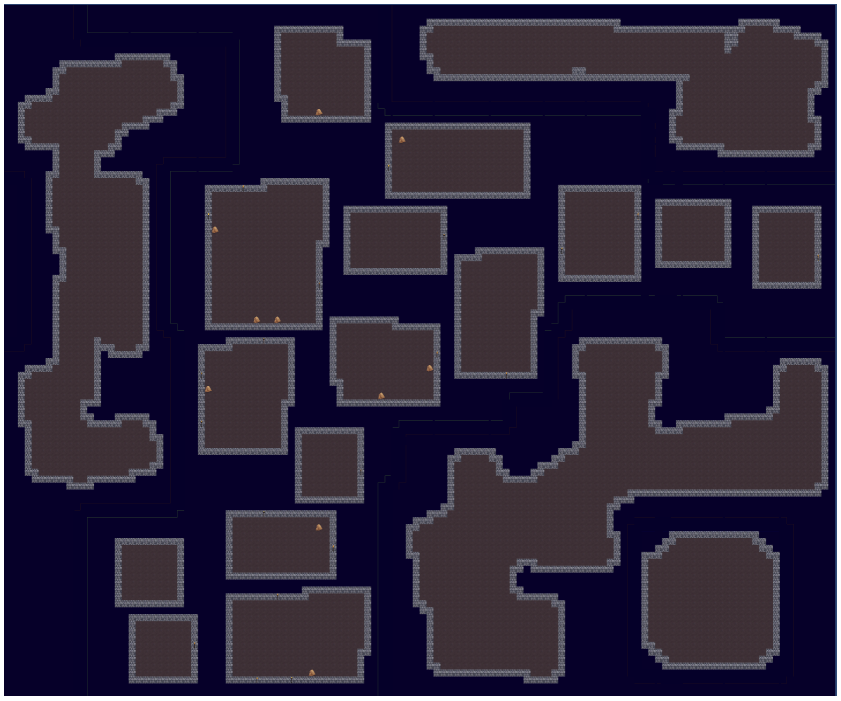

Figure 3.34: Dungeon example ($seed = 1$) after cavern-like parts are generated

3.2.3 The Third Layer

The third layer is straightforward. It runs only on top of the Cavern-like parts of the dungeon, over the ground tiles. Its purpose is to add details such as rocks, plants, and mushrooms. We could add those details directly to the Cavern-like WFC, but it would cause overfitting. Shapes on the output would closely match shapes in the input. It would kill the variety, and using an extra layer is a simple workaround.

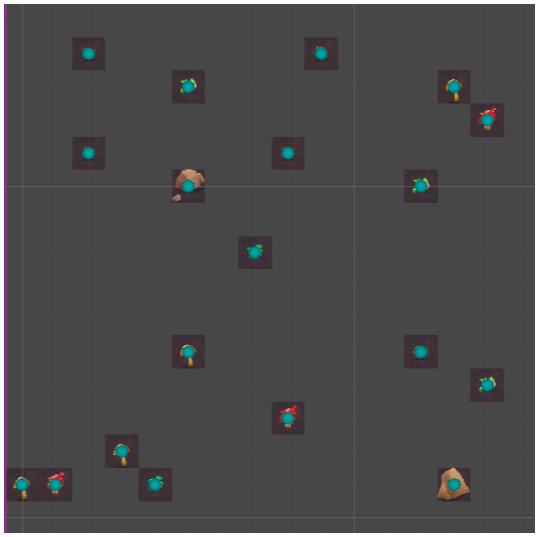

Figure 3.35: Decorations Input

We can see the input in figure 3.35. The size of the input is 20×20 tiles. Since the input is straightforward, we can easily tweak it to fit our needs. Increasing and decreasing the density of decorations is trivial. It might be surprising, but this WFC still used the `N = 3`. Otherwise, if we would use the `N = 2`, some decorations might be too densely packed. Most of the area is left unfilled.

All decorations have a background. It has a good reason. We need to flatten the map into a single layer because it allows us to do simple postprocessing over the whole output. If we need to have multiple backgrounds, it is easy to modify the export of the HWFC. Our proposed solution is to mark those top tiles as transparent. And then, we could merge those transparent tiles with a tile under them and export the resulting tile.

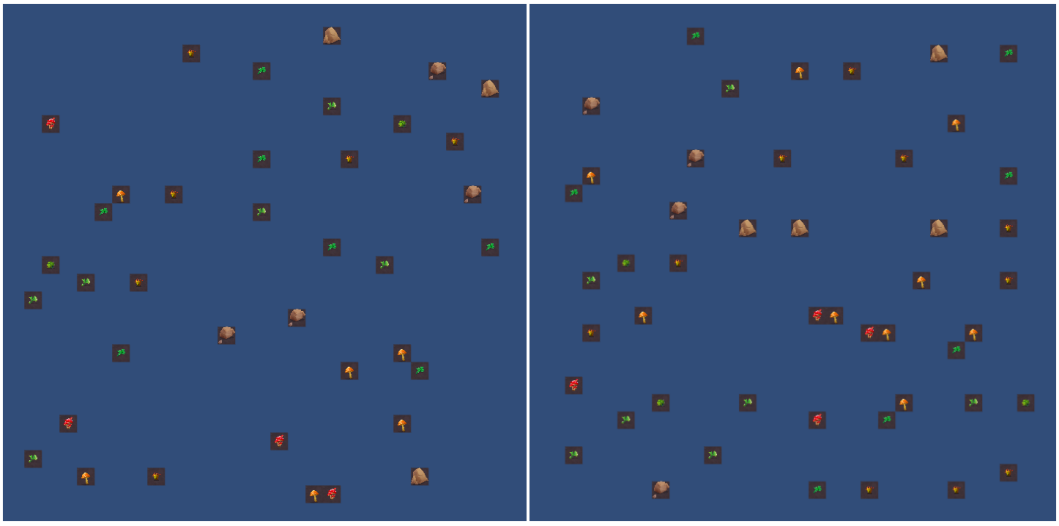

Figure 3.36: Decorations output

Figure 3.36 shows two examples of outputs. The size of each is 30×30 tiles.

We can see that most of those outputs are blue. The blue color represents a background. Therefore, those tiles are not filled.

There is one more detail about this WFC. It does not have to use any early tile placement. However, we can place an empty 3×3 pattern in areas we do not need to fill. The reason for it is not the quality of the output but the performance. The rectangular cover of the Cavern-like areas can be massive. Therefore, this WFC is relatively expensive. Moreover, it is much faster to collapse tiles with the preplacement. We will discuss this more in the implementation chapter, namely algorithm 5.

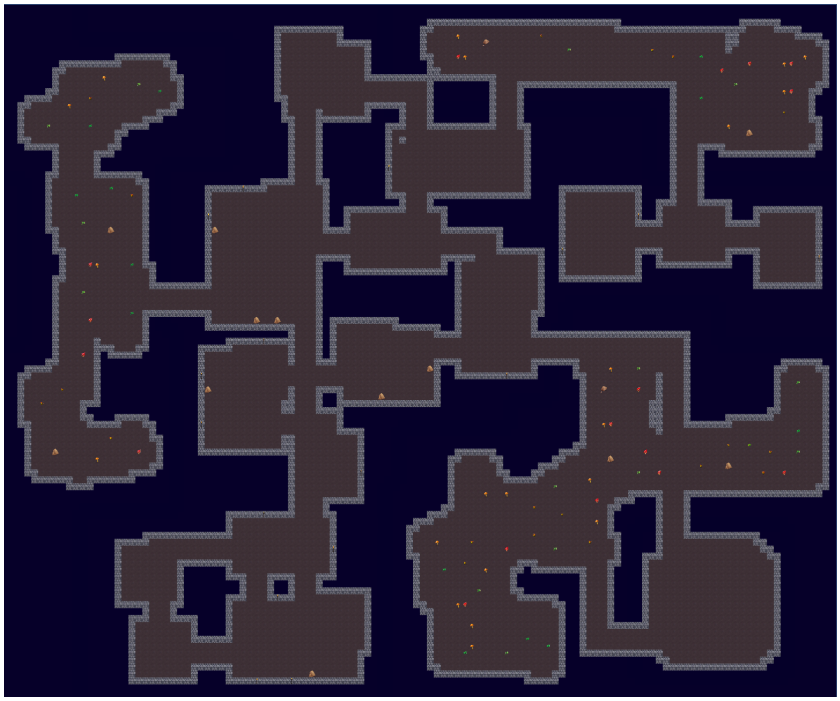

Figure 3.37: Dungeon example ($seed = 1$) after decorations are generated

Figure 3.37 shows the final version of the dungeon. It places decorations into the cavern-like part. Then it connects all rooms via postprocessing. This final postprocessing works on the whole output. Therefore, it can connect regions generated by different WFCs.

4. Results

In the previous chapter, we have seen how to use the HWFC algorithm to generate outputs. In this chapter, we will look at the outputs of the HWFC and compare them to the regular WFC. Besides that, we will also compare the performance of both algorithms.

4.1 World Map

We will start with the world map described in section 3.1. Generating a world map is a very challenging task for the WFC algorithm.

4.1.1 Generated Maps

We used three layers and six different WFC to generate the World Map. Now, we will see and discuss the final results. We can set a seed for the whole Hierarchical WFC generation to make our results reproducible. There are also multiple generated results to show their diversity. We have already seen one final output in the previous chapter in figure 3.16. That output used $seed = 1$. Therefore, we will use seeds starting from 2, and each subsequent HWFC gets a seed incremented by one. This way, we ensure that results are not cherry-picked, and even the worse ones will be shown. Those outputs demonstrate the average outcome.

As mentioned in previous sections, each generated map has 160×120 tiles. That size is quite huge for a WFC algorithm since the algorithm thrives in smaller scopes.

We can see the first example in figure 4.1. We will not place multiple maps next to each other here since all the details would be barely visible. There are three continents, one large and two smaller. The continents cover a good portion of the map. One thing that might not be optimal is the long island in the bottommost part of the map. It has an unnatural shape. Besides that, the generated map is solid and shows a good variety. It is widely non-homogeneous, ranging from large continents to small islands.

The second output is in figure 4.2. There are four smaller continents in this one. Each continent is relatively simple since they are small. Still, each continent is different. The sea, with smaller islands, covers a significant portion of the output. Overall, this output is very different from the first one but solid. Different outputs are very important in PCG. We want to avoid a situation where all maps would feel the same.

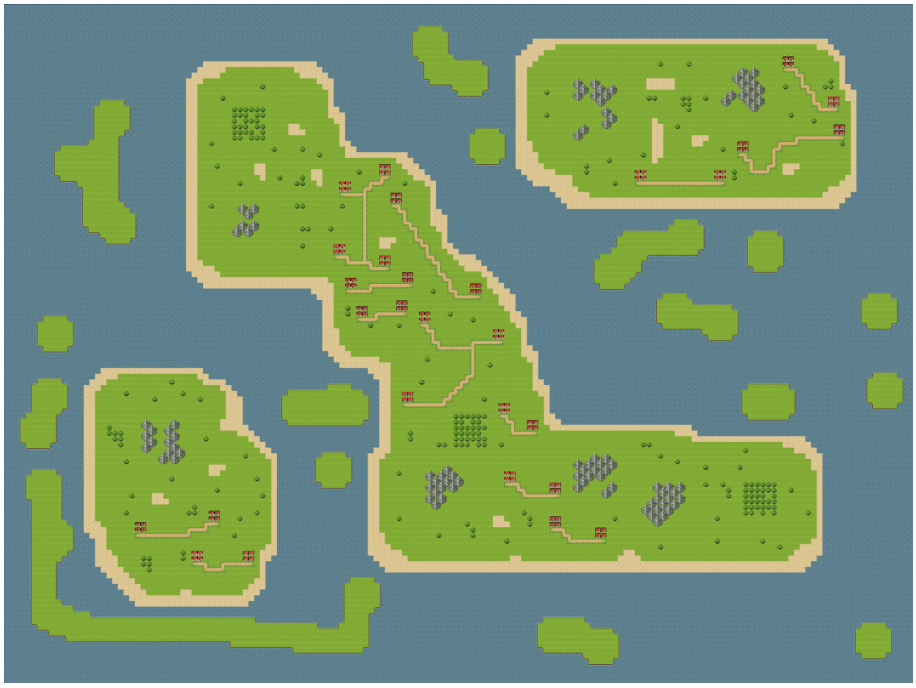

Figure 4.1: World map output ($seed = 2$)

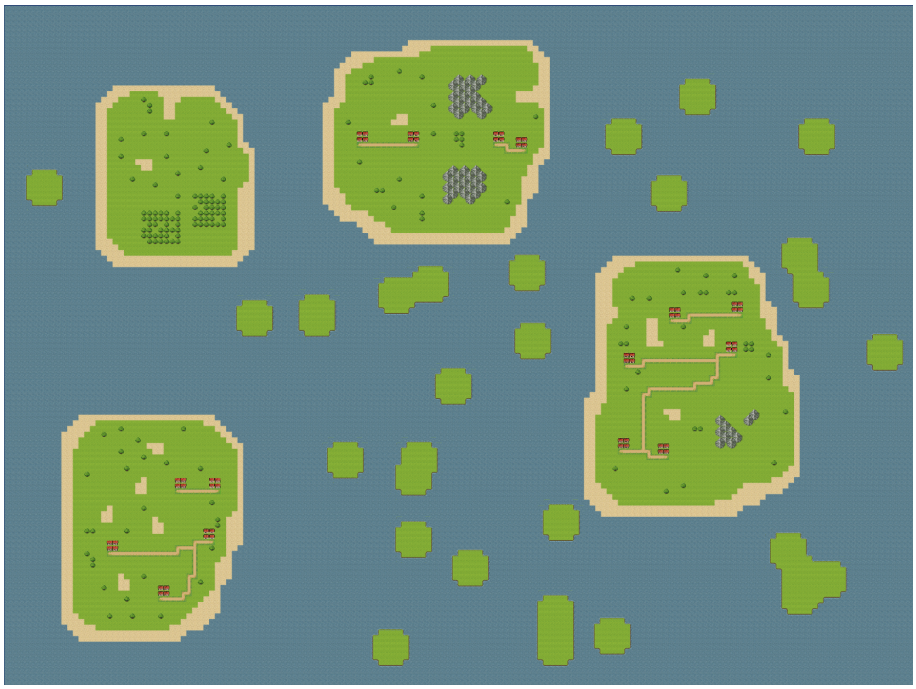

Figure 4.2: World map output ($seed = 3$)

We can see the third output in figure 4.3. There are large continents. However, each continent looks different. The topmost continent has long paths covering it. Again, the generated map looks solid.

The fourth output is in figure 4.4. This one has an enormous continent in the middle. Although the continent is huge, it still looks well and is non-homogenous. Having huge continents might be bad since there is a greater chance that the

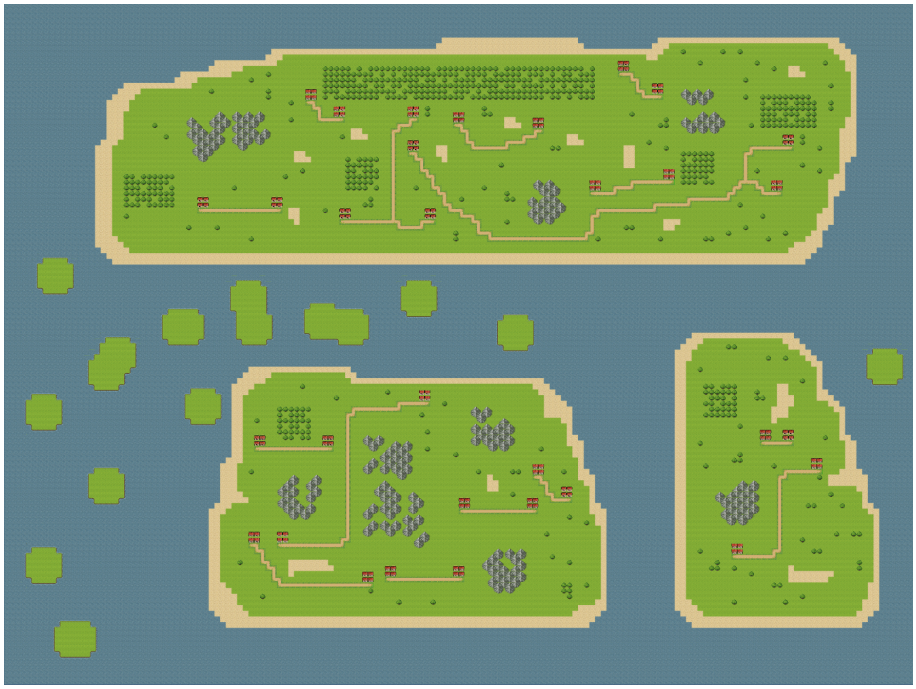

Figure 4.3: World Map output ($seed = 4$)

Figure 4.4: World map output ($seed = 5$)

generation will fail. And it takes some time to generate the whole continent again. In this case, it needed three attempts to generate the grasslands. However, this map is probably the best one so far.

The fifth and final output is in figure 4.5. It contains two large continents nicely filling the map. Again, it is very solid. It does not make sense to show more outputs here. However, many more maps could be generated with the pro-

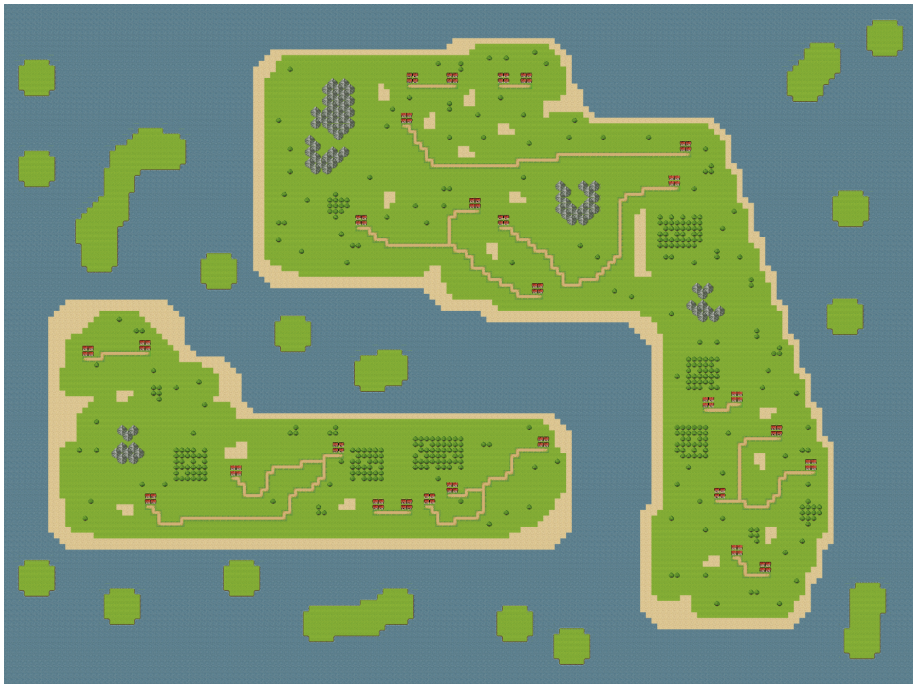

Figure 4.5: World map output ($seed = 6$)

vided application. Those generated outputs used *seed* values from two to six. Therefore, the average output with a random *seed* can be comparable. Overall, the quality of outputs is relatively high and even consistent. We have not encountered a single output that would be terrible. It is also relatively fast to generate. We will focus more on performance in the following section. Having these properties, it looks like this example could be easily used even for a runtime generation. Adding new assets into the third layer is simple if more variety is required. Alternatively, adjusting previous layers to change some generation properties is also straightforward.

4.1.2 Comparisons

We found out that world maps generated by the HWFC are all pretty good. However, we need something that we can compare with it. A good candidate for comparison is the regular single-layer WFC. However, comparing the HWFC with a single-layer WFC is extremely hard to do right. We want it not to be biased toward the HWFC.

The obvious solution would be to hand-design an input for the single-layer WFC. Although this would be as fair as possible, it has a significant issue. We would need a huge and precise input for complex, highly structured data like the world map. The input size would probably need to be similar to the desired output size. Otherwise, it would be very hard to achieve a good variety. In our

case, it is around $160 \times 120 = 19200$ tiles. We can not count on the upscaling property of WFC if we want to generate a heavily constrained output. Manually placing almost 20000 of tiles would be an enormous task. Especially if we would like to tweak some part of the input, a large area would need to be redrawn. Using tricks like enforcing minimal size in a single-layer WFC is also possible. However, it would be much more difficult for such a huge input.

Therefore, we want to avoid this large hand-made map at any cost. We can prevent it by taking the output of the HWFC and feeding it as an input to the single-layer WFC. We obviously need to flatten the output of the HWFC to pick only tiles visible in the final output, but it is straightforward. On the one hand, these comparisons will be slightly biased toward the HWFC. On the other hand, if we spend $100\times$ more time on creating the input for the regular WFC, it would be heavily biased towards it. This is the first point where HWFC significantly outshines a regular WFC. We have no idea where and how to start if we would like to create an extensive, heavily structured input with a single-layer WFC. Therefore, the best we can do is to feed it a large, good-looking map and hope it will work.

The quality of the HWFC output could significantly influence our WFC's quality. Therefore, we will try it with different inputs. Firstly, we want to focus on the quality of the HWFC compared to the regular WFC. This is subjective. However, we could quickly tell if there are significant differences. Later, we will compare other properties, such as performance.

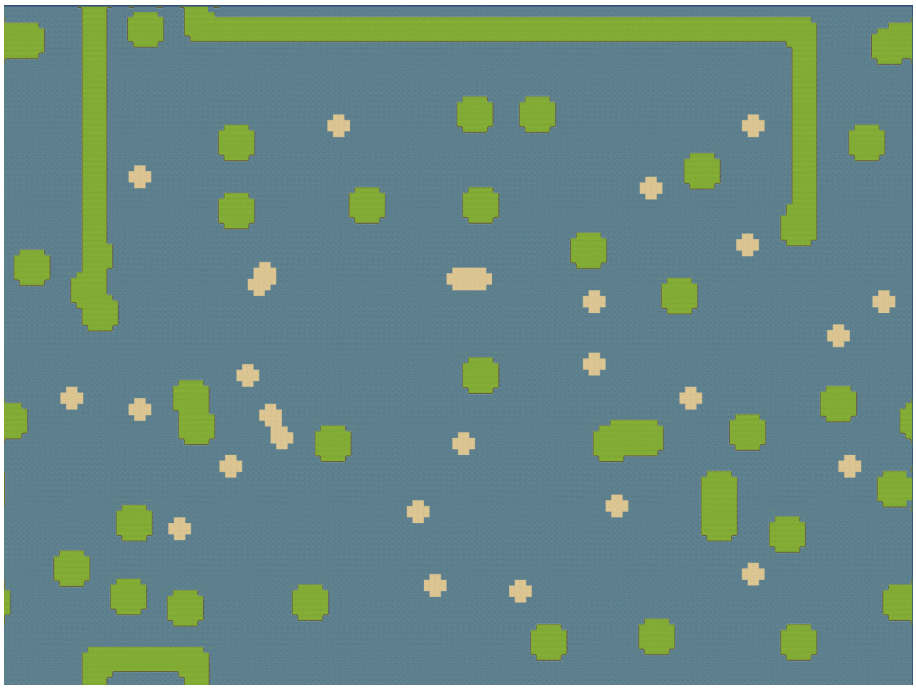

Figure 4.6: World map single-layer ($seed = 2$)

We will start with a World Map from figure 4.4 as this seems to be the best overall. We will use an Overlap WFC with $N = 3$ as usual for comparisons. Again we will start with s $seed = 1$ for the single-layer WFC. The first output is in figure 4.6. It uses $seed = 2$ since the generation with $seed = 1$ has failed. Sadly, it is terrible. The output is totally degenerate. It tried to generate the sea. Its easiest filler is many tiny islands. The reason for it is that generating highly structured output is hard.

Figure 4.7: World map single-layer ($seed = 8$)

The next successful generation uses a $seed = 8$. It is shown in figure 4.7. It is pretty decent. However, it is not a world map. The map is homogeneous, and there is only the *Grasslands* biome.

The next successfully generated maps have $seed = 10$, $seed = 14$, and $seed = 16$. However, they all look almost identical as 4.6. The $seed = 19$ contains only grasslands and looks as 4.7. We have tried each seed all the way to $seed = 100$, with some more successful generations. However, they all were very similar to either 4.6 or 4.7. There was none one having most sea and continent simultaneously. This is the issue with the homogeny of WFC, as it is hard to transition between them.

We have found out that the pure single-layer WFC is out of luck. Clearly, we need a better approach. However, we can still improve it with the early tile placement. Placing down both part of a water area and part of a continent could do the trick. We can place a 3×3 pattern of grass in the middle and a 3×3 pattern of the sea at one edge. Although this idea seems promising, it fails miserably. We

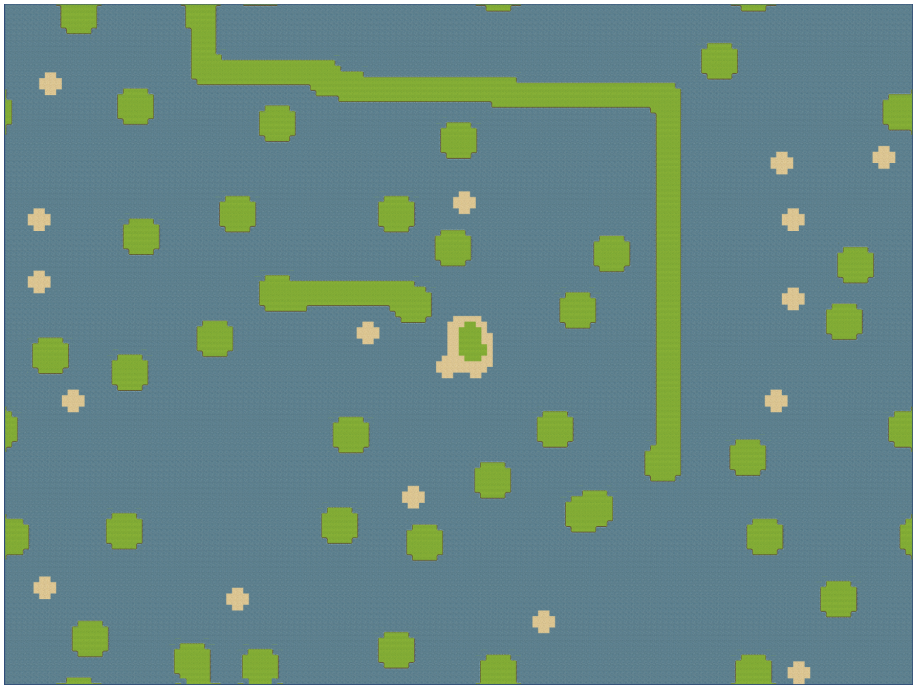

Figure 4.8: World map single-layer, preplaced grass and water ($seed = 2$)

can see an example in figure 4.8. There is a tiny continent directly in the middle. We could easily overlook it as another island. This pattern keeps occurring very frequently with our setup. It has a simple explanation. The entropy of cells next to the water tile is lower than the entropy of cells next to the grass tile. Therefore, the algorithm will keep expanding the water area. In the end, it will try to collapse the area around the grass. This will very often result in a tiny continent. It is easier to generate while satisfying all the constraints enforced by the ocean around it. We have tried all values up to $seed = 50$ without any success.

We can still help it by placing down a village instead of grass. The entropy around the village is smaller. Therefore, there is a chance that the area around it will be expanded. An example is in figure 4.9. The first successful generation has used $seed = 5$. Unfortunately, the entropy of the water is still lower, and it tries to generate the continent very late. The resulting continent is larger and looks much better, but it is still way too small.

We have tried each map all the way to $seed = 100$. Most of them failed or were similar to figure 4.9. Figure 4.10 shows probably the best map we got. It does not have any ugly structures. However, it is still much worse than every HWFC output.

Our last chance is to place grass in the middle and hope for a good result. This has two advantages compared to the pure WFC without any tile placing. Firstly, the sea is more common, so there is a larger chance to transition to the

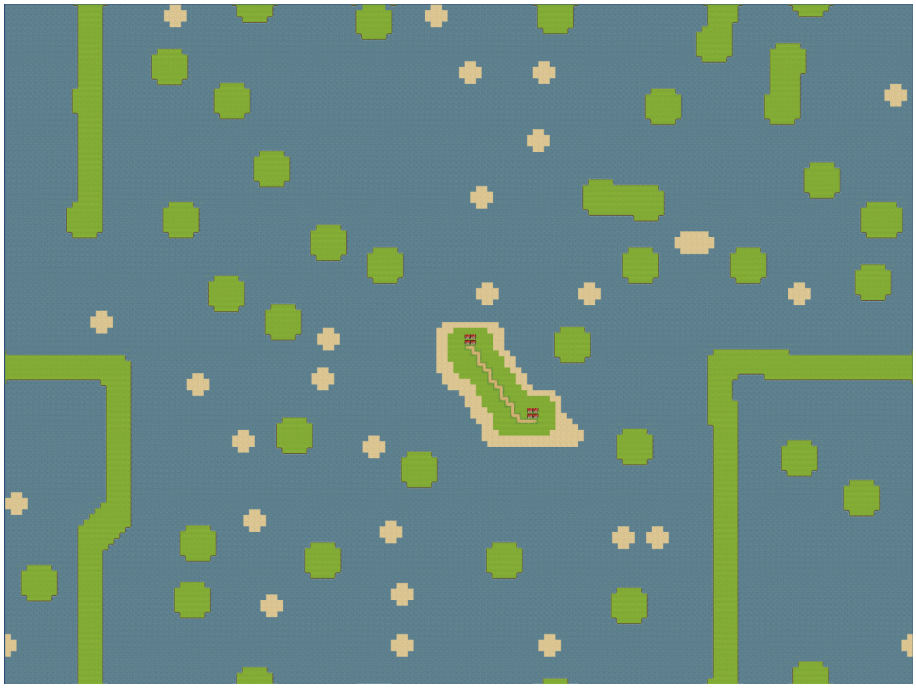

Figure 4.9: World map single-layer, preplaced village and water ($seed = 5$)

Figure 4.10: World map single-layer, preplaced village and water ($seed = 96$)

sea. Secondly, we will start the collapse from the middle instead of a random point. This gives us a better chance to have water around the edges, with should look better. We expect most of the maps to be very similar to the one in figure 4.6.

We have tried each map up to $seed = 100$. There were several ones having both continents and sea at the same time. Unfortunately, most of them failed in

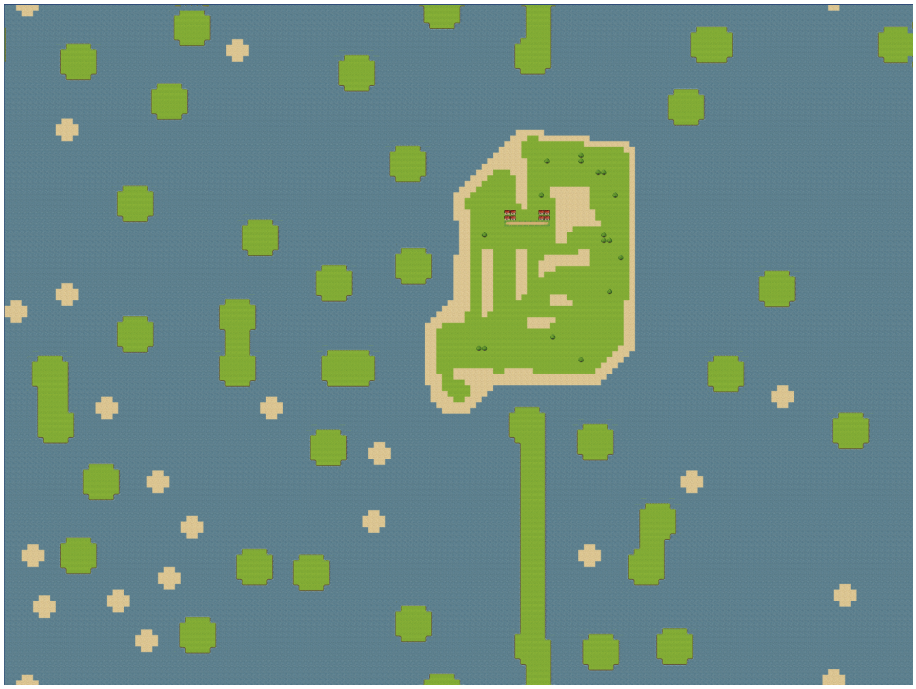

Figure 4.11: World map single-layer, preplaced grass ($seed = 22$)

the end. Figure 4.11 shows one successful output having both. This output is not so bad, but it still gets crushed by any HWFC output.

Unfortunately, we could not generate good world maps with a single-layer WFC. However, it is not too surprising. Generating a world map is extremely challenging for the WFC because it is very structured. It is most likely possible to generate good looking world map with a single-layer WFC, but it would take an enormous effort to do so by designing a perfect input. At that point, it does not make sense to use WFC anymore. We should use different PCG methods instead.

Increasing N

Another approach is to increase the parameter N. It increases the viewing window size and forces it to copy larger parts of the input to the output. Therefore, it could improve the output quality at the cost of variety. However, even setting `N = 4` crashes the application. The reason is simple. Our input map is enormous, and it contains many different patterns. Memory consumption can increase exponentially with N. Profiling shows that even with `N = 3`, the application takes over $2.5GB$ of memory. Therefore, if we want to find all patterns of size 4×4, we run out of memory, and the application crashes.

4.2 Dungeon

Similarly to the world map, we will now analyze generated dungeons. We use the dungeon discussed in section 3.2, including its postprocessing.

4.2.1 Generated Maps

Again, we are using seeds to make our results reproducible. The size of generated maps is 120×100 tiles. Generated maps include postprocessing that adds paths between areas of the dungeon.

We have already seen one generated dungeon in figure 3.37. That dungeon uses $seed = 1$. Therefore, we will again start with $seed = 2$ and increment it by one for each output.

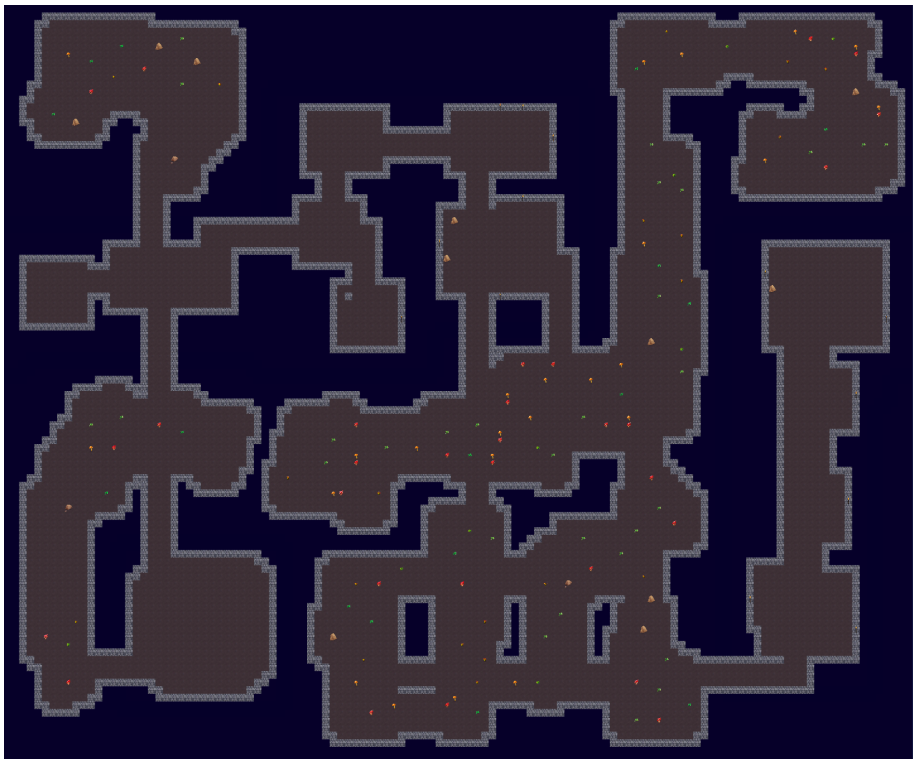

Figure 4.12: Dungeon ($seed = 2$)

Figure 4.12 shows the first output. We can see a boss room in the bottom-left corner. There are two ways into the boss room. This might not be optimal. However, it is caused by postprocessing. Therefore, it could be avoided by using smarter, more complex postprocessing without affecting the rest of the generation. Besides that, the dungeon has both cavern-like and mansion-like sections in a good balance. It would probably create a good gameplay experience.

Figure 4.13 shows the second generated dungeon. In this case, the boss room is in the upper-right corner. There is only one path to it which is probably the

Figure 4.13: Dungeon ($seed = 3$)

best. This dungeon is very large. It is generated to the edges and covers a lot of space inside. The only weakness of this dungeon is the long straight path at the bottom. However, it is again caused by the final postprocessing. The issue is not a fault of the HWFC and could be avoided using better postprocessing. Otherwise, this dungeon is very solid.

The third example is shown in figure 4.14. There are again two paths to the boss room. Besides that, we can see large open cavern-like parts next to the boss room. This would surely provide a totally different gameplay experience compared to the small mansion-like areas at the bottom. Again, the largest issues are the long straight path created by our postprocessing.

Figure 4.15 shows the fourth generated dungeon. This time the majority of the dungeon is cavern-like. This dungeon is quite different from the previous ones, lacking larger mansion-like structures. However, it still looks interesting and would provide a different experience.

Finally, figure 4.16 shows the fifth and last example. Cavern-like parts of the dungeon cover the majority of the map. Then, there is a mansion-like section near the boss room. This time, those sections are separated. The map is very solid, potentially the most realistic.

However, some outputs might be much worse. Figure 4.17 shows one of them. The tiny room in the bottom-right corner is reachable only through the boss

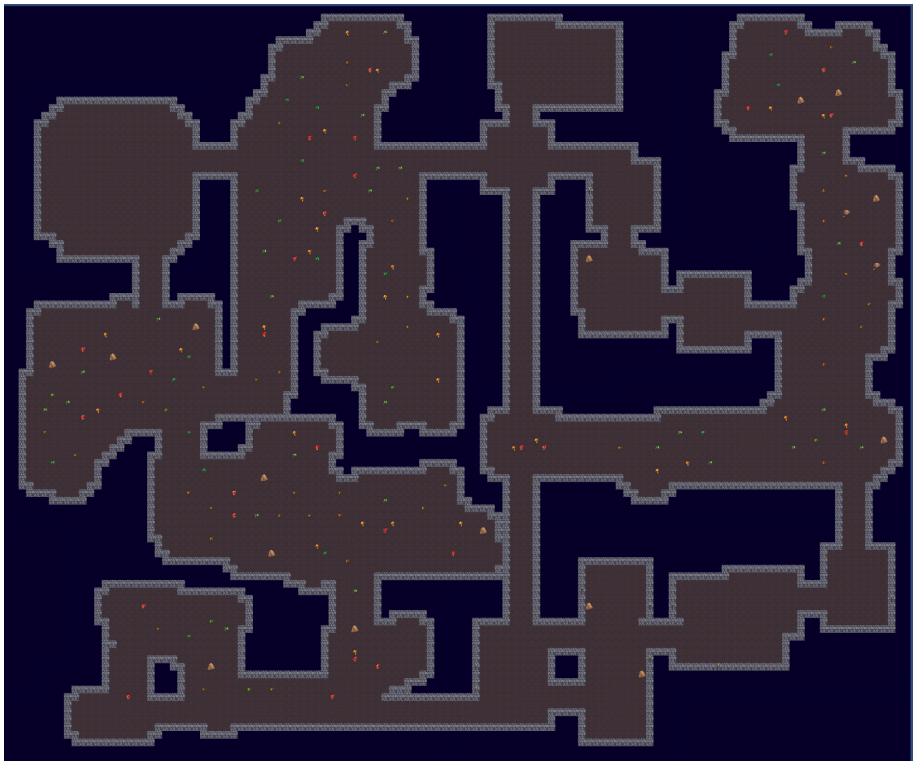

Figure 4.14: Dungeon ($seed = 4$)

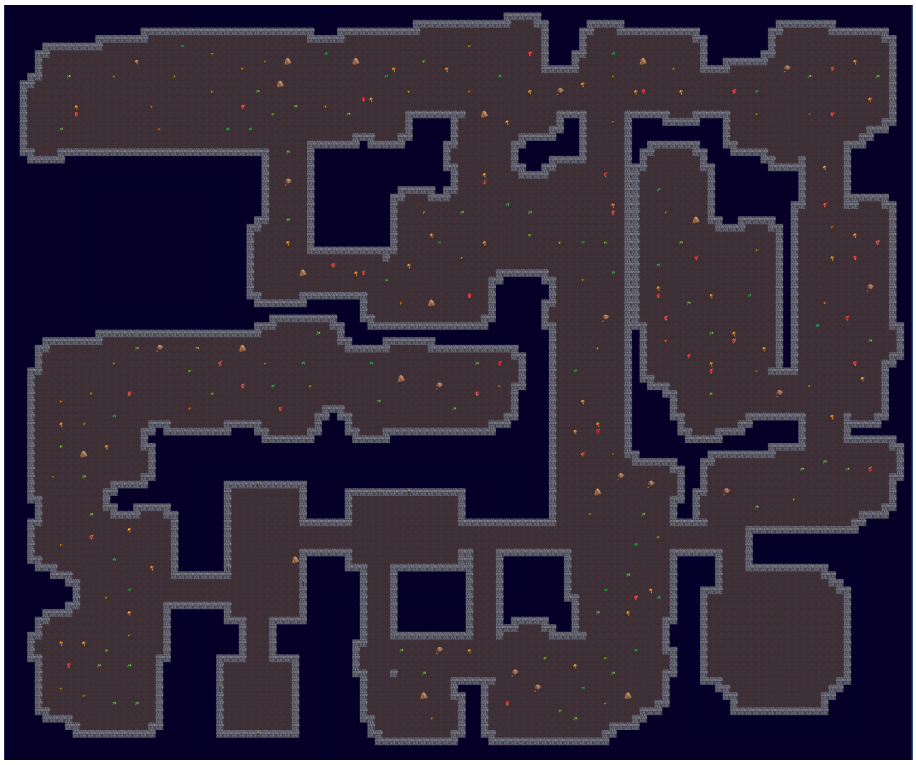

Figure 4.15: Dungeon ($seed = 5$)

room. This is the first seed where it happened, so it is relatively rare. It could be terrible design-wise. However, it is rare and could be avoided with better

Figure 4.16: Dungeon ($seed = 6$)

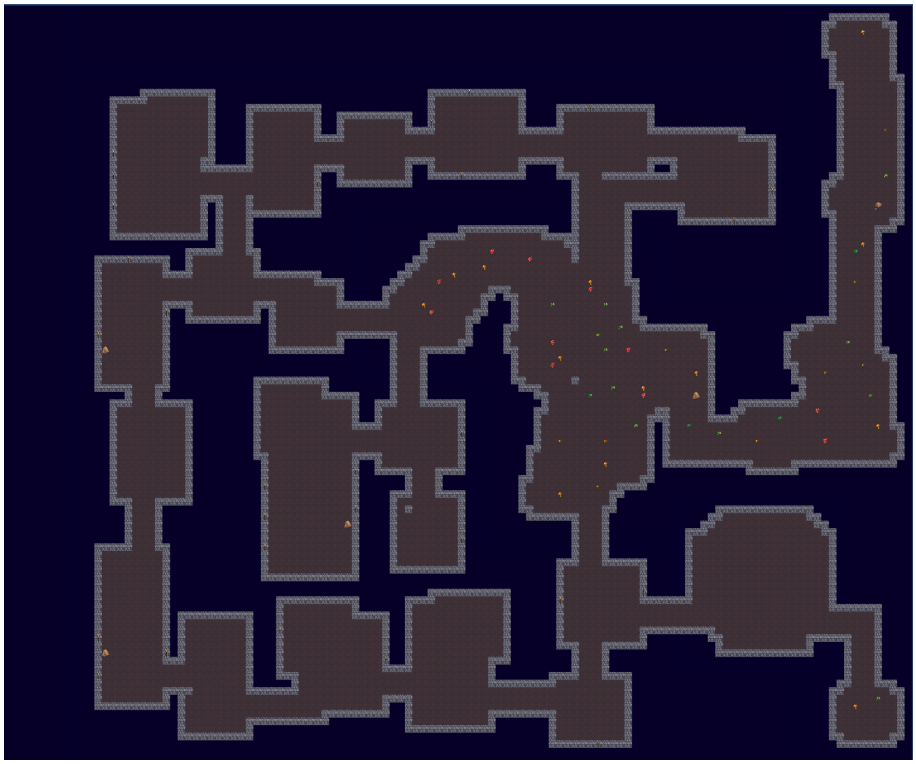

Figure 4.17: Dungeon ($seed = 11$)

postprocessing. Overall, this dungeon is not great, as most of it is filled with mansion-like parts.

Obviously, the quality of generated dungeons depends heavily on our design

criteria. Here, all maps satisfy all of ours, which is very important. On the other hand, our criteria are relatively vague. Similarly to the world map, we used each seed value from 2 to 6. Therefore, those dungeons should be close to the average expected output. It is crucial that the quality of generated maps is again consistent. Besides that, those maps have a similar structure but are all different, which is very important. It means that the HWFC for a dungeon generation could even be applicable for runtime usage. There are some outliers, for example, figure 4.17. However, that could be solved via postprocessing without affecting the algorithm.

4.2.2 Comparisons

This section will show examples of single-layer WFC and compare them with the HWFC. We will start by taking the example 4.13 as an input for the single-layer WFC. It does not really matter which input we choose because they are all quite consistent. This one gives us a good balance between the cavern-like and mansion-like parts. It also covers a majority of the map. Again it is good to have a large input since the generated dungeon is relatively complex.

Figure 4.18: Single-layer dungeon without postprocessing ($seed = 1$)

Figure 4.18 shows the first example. Here, we have used preplacement around borders. Otherwise, some rooms could continue on the other side of the map. We use the `symmetry = 8` for maximum variety. There is no postprocessing.

Therefore, some areas are not connected. There are also several dead ends and narrow paths.

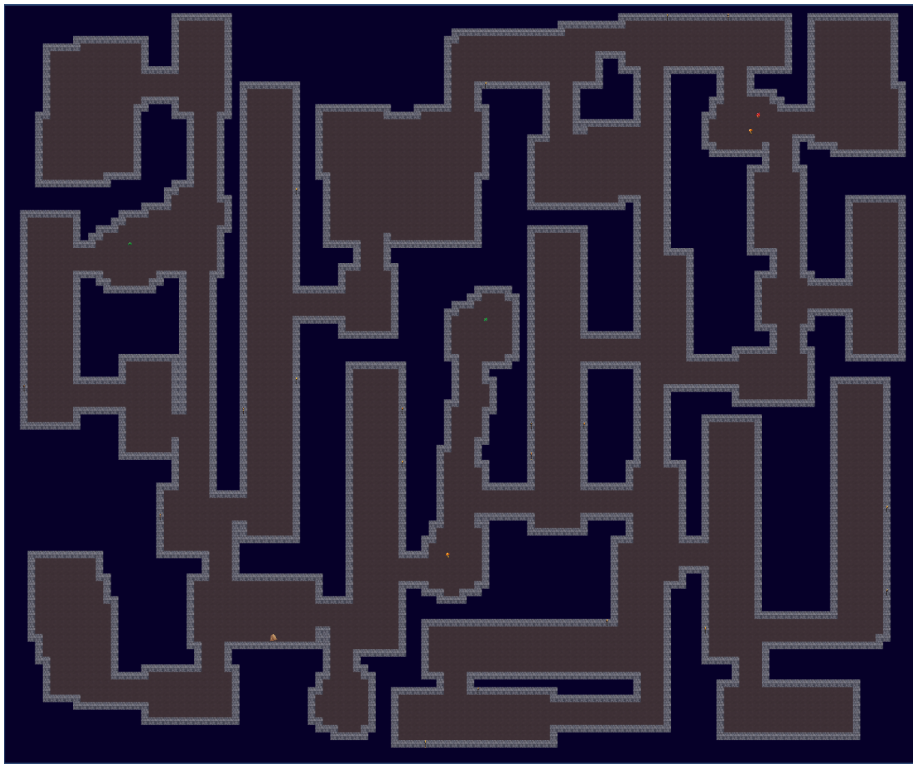

Figure 4.19: Single-layer dungeon with postprocessing ($seed = 1$), the input is without paths

An obvious improvement is to include postprocessing that adds paths. We have this functionality implemented into the HWFC so that we can use it here. However, we will not upscale the map after the first layer and use it only to run one postprocessing. We will also take input without added paths. Figure 4.18 shows that they clearly harm the final outcome. And we will add them with postprocessing anyway. Figure 4.19 shows an example of a dungeon with paths added with postprocessing. This dungeon is much better. However, many small long rooms make it quite unnatural. Dungeons in figure 4.18 and figure 4.19 use the same seed but are very different because figure 4.18 has paths as a part of the input. Anyway, this approach seems promising, so we will generate more outputs to get a better sample size than one.

Figure 4.20 shows a second example with the same settings. Those only differ in a seed. The quality of generated maps is very similar. This one is decent but mostly a mansion-like dungeon with narrow rooms.

We have tried many more seeds with similar results. Either the generation has failed, or there were long, narrow rooms that looked unnatural. However, here we can at least compare it to the HWFC. The HWFC is clearly better, giving us more variety. The single-layer WFC does not satisfy the design criteria for

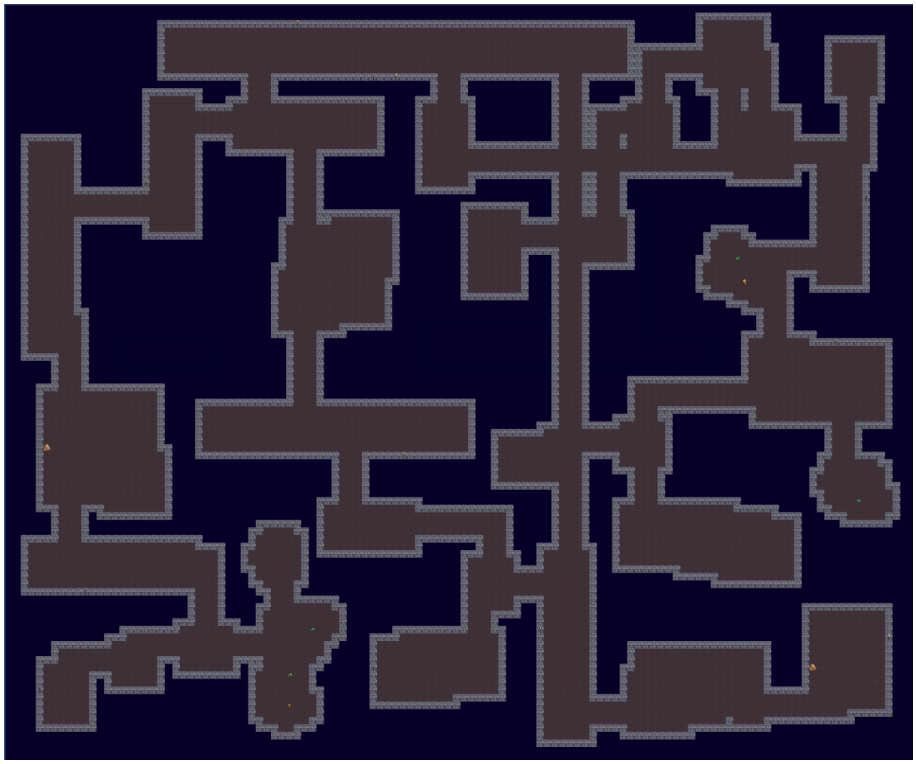

Figure 4.20: Single-layer dungeon with postprocessing ($seed = 2$)

generating different types of rooms. Here, we have only the mansion-like parts, making the whole dungeon homogeneous.

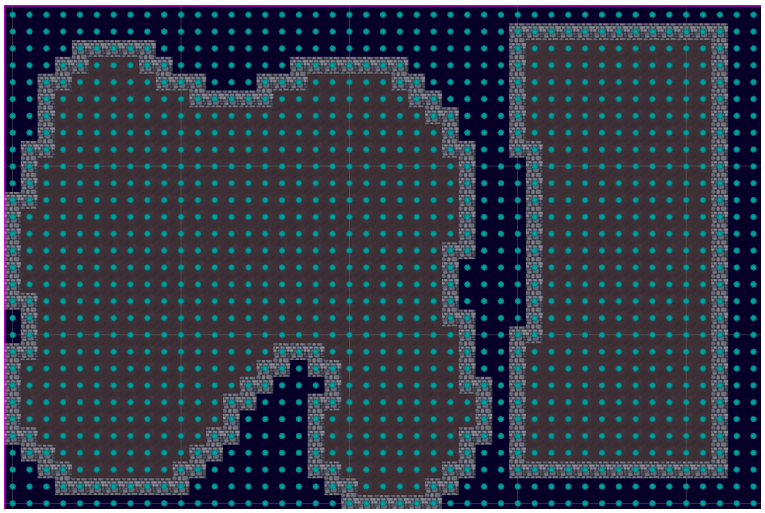

Figure 4.21: Handmade input to the single-layer dungeon

We can try one more strategy. The dungeon is far less structured than the world map. Therefore, we can create the input manually. We will start which the input for the cavern-like portions shown In figure 3.31. And we will extend it with mansion-like parts. We will ignore the boss room. This way, we will not fulfill all our design criteria, but we have a chance that the output will be decent.

Figure 4.21 shows the map we will use as input for the next generations. The cavern-like part is on the left, and the mansion-like part is on the right.

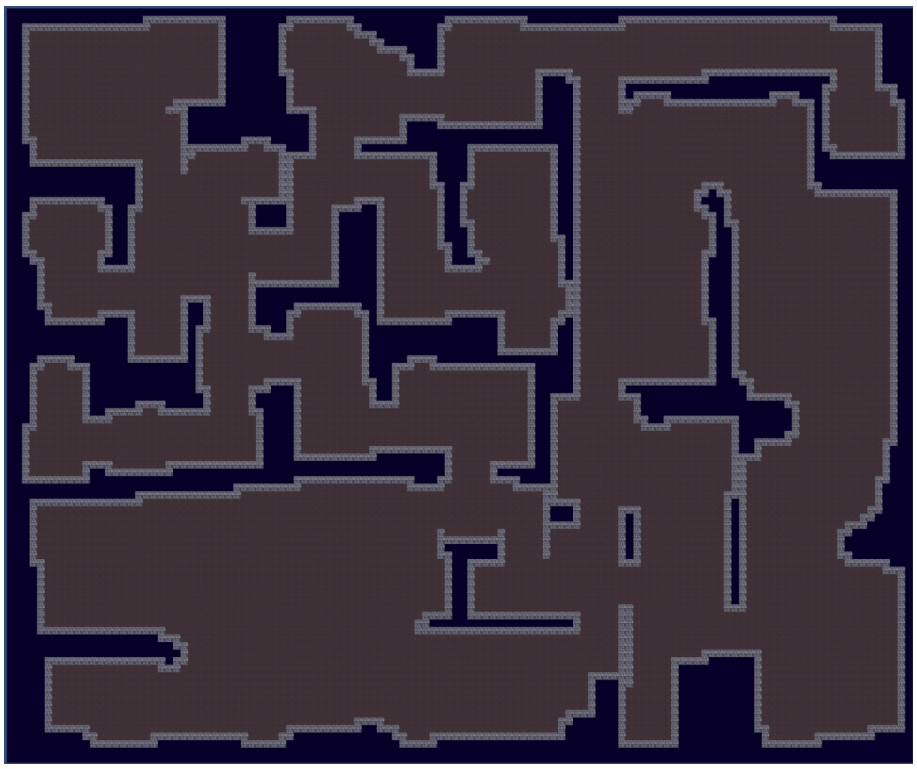

Figure 4.22: Single-layer dungeon with handmade input and postprocessing ($seed = 1$)

Figure 4.22 shows its first example. We can see that the output is way better compared to previous results. This output is visually nice and even comparable to the outputs of HWFC. Therefore, we will generate more outputs.

The second example shows figure 4.23. The room on the left might be way too large and open. Otherwise, this output is decent. Therefore, we will show one more example.

Finally, the third example is shown in figure 4.24. The output is fine. Similarly to the HWFC, our generation on the simplified handmade input is pretty consistent.

However, there are inconveniences. The single-layer WFC can only generate the layout. We have tried to generate decoration, but the generation was over-constrained, and the results were terrible. The same applies to the boss room. In the HWFC, we had regions with different feels. Here, the whole map is a mix between cavern-like and mansion-like parts. It lacks any high-level structure, and the map is homogeneous – the core problem for a WFC. The single-layer WFC also does not meet our design criteria. It would be extremely difficult to modify the WFC in a way that will satisfy them.

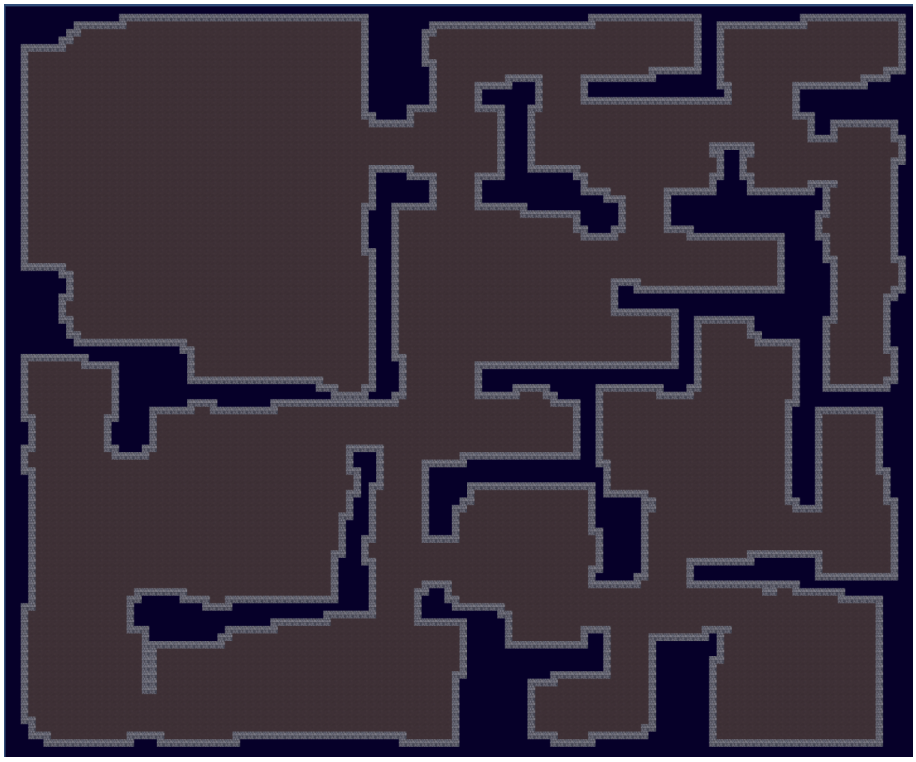

Figure 4.23: Single-layer dungeon with handmade input and postprocessing ($seed = 2$)

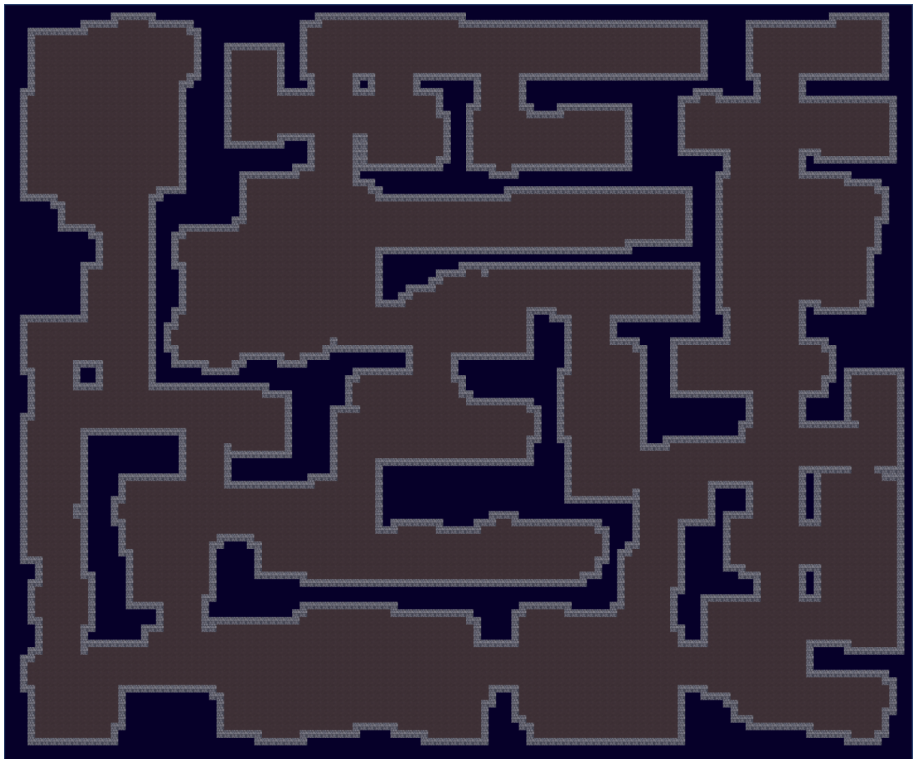

Figure 4.24: Single-layer dungeon with handmade input and postprocessing ($seed = 3$)

We have figured out that the single-layer WFC works for less structured content like the dungeon. However, in our case, the quality of outputs is still a bit worse than those generated by HWFC. The single-layer WFC does not generate structured content to fulfill only very simple design criteria. However, the generated maps can look good and be usable with sophisticated postprocessing.

4.3 Performance

The performance is vital since the WFC algorithm itself is computationally intensive. Therefore if the HWFC is much worse, it could make the algorithm impractical.

CPU	AMD Ryzen 7 5800X
RAM	2×16 GB DDR4, 3600 MHz, cl 18
GPU	Nvidia GeForce RTX 3060 Ti 8 GB

Table 4.1: Hardware used for experiments

For all experiments, we have used the hardware shown in figure 4.1. Our measurements are CPU bound. Therefore the CPU is by far the most important component to get comparable results. In fact, it depends primarily on its single-threaded performance.

We have gathered results from 100 successful runs of each algorithm. If the generation failed, the seed was incremented, and the generation was started with a new seed. The same applies to the HWFC. However, there it means that only one WFC instance in the pipeline has failed, and this instance will be run again.

The measurements were run on a standalone build of the game. To gather the data, we used Unity's `Time.realtimeSinceStartup` to avoid possibly inaccurate measurements due to a drop in FPS.

4.3.1 World Map

For the world map, we are comparing HWFC, a regular single-layer WFC, and a single-layer WFC with grass placed in the middle because it usually generates more complex outcomes.

Results are shown in table 4.2. The HWFC is surprisingly the fastest on average. However, for the single-layer WFC, it is caused by failed generation. There is time without fail, and the single-layer WFC is much faster than HWFC. It does not make sense to collect this metric for the HWFC, and it does not fail much anyway. The standard deviation is much smaller for the HWFC. The

	HWFC	WFC	WFC with grass
Average time [s]	19.04 ± 4.16	23.61 ± 19.32	33.49 ± 29.01
Min time [s]	14.56	8.91	8.79
Max time [s]	36.33	97.46	143.86
Average time (no fails) [s]	—	9.39 ± 0.26	9.35 ± 0.21
Average failures	0.93 ± 1.22	2.06 ± 2.45	3.32 ± 3.48
Min failures	0	0	0
Max failures	5	13	18

Table 4.2: World map performance

maximal time for a successful generation is by far the lowest. The reason for it is simple. The single-layer WFC tries to generate a large complex map. Therefore, there is a decent chance that the generation will fail. At that point, we need to start over with the whole generation. Indeed, it is shown in the maximum failures. It means the maximum number of fails in a row for the WFC. In the case of the HWFC, it shows the maximum number of failures during a single generation. The HWFC usually fails for a smaller WFC, so its runtime is not affected much.

It is handy that the HWFC is much more consistent. It has by far the lowest maximal time. It makes the algorithm a good candidate for runtime usage. Single-layer WFC has huge spikes. Each has several attempts, much longer than the maximum for HWFC.

4.3.2 Dungeon

In this section, we will compare the performance of HWFC and single-layer WFC in the dungeon example.

	HWFC	Handmade WFC
Average time [s]	7.39 ± 1.21	6.92 ± 0.5
Min time [s]	5.2	5.85
Max time [s]	12.17	8.38
Average failures	0.23 ± 0.58	0.0 ± 0.0
Min failures	0	0
Max failures	3	0

Table 4.3: Dungeon performance

Table 4.3 shows the results. This time, we did not include the single-layer WFC generated on top of the output of HWFC. It was the worst by far since many generations have failed. The dungeon generation was way faster than the

world map. It has two reasons. Firstly, the dungeon is way smaller – 120×100 instead of 160×120 tiles. The second reason is that the dungeon is less structured.

We can see that the dungeon generated with the handmade WFC is faster than the HWFC by about 7%, but HWFC is faster in the best case. Those results are very close. The main difference is that the single-layer WFC has no failures. The reason is that the input is very simple. It contains only 5 tiles – 3 ground tiles to enforce the minimal size of rooms, 1 wall tile, and 1 tile for a background. Those tiles are also in very flexible patterns, making the generation hard to fail. The HWFC has way more tiles. Only the background has 10 different tiles. The experiment is biased towards the handmade single-layer WFC as it generates only the layout. Generating decoration increases the number of patterns which slows down the algorithm. It also increases the chance of failure.

We have found out that the runtime of both algorithms is very similar. The splitting of the output to several regions done by HWFC greatly impacts the runtime. Therefore, it balances out the generation of more layers. If we increase the output size, it will get even better for the HWFC since the runtime of the WFC algorithm scales non-linearly with the size.

5. Implementation

In this chapter, we will explore the implementation itself, including all the details. The project is implemented in Unity 2019 and is available on GitHub — https://github.com/fileho/Hierarchical-Wave-Function-Collapse. As a baseline it uses a Unity package developed by Joseph Parker Parker [2016] which contains a nice wrapper around WFC to allow the usage directly in Unity.

Using Unity has several advantages. Firstly, it helps a lot with simplifying the input as it can be created directly in Unity instead of using some external editor. Secondly, most of the games that are already using some version of WFC are implemented in Unity. Finally, the original implementation is created in *C#*, so we use it directly with wrappers. It does not make any sense to implement the WFC again. The original solution is well-tested and used across many applications. For a *C#* solution, it is also well-optimized. However, C++ implementations are significantly faster. [Ernerfeldt, 2016]

5.1 Wave Function Collapse Unity Wrapper

This package [Parker, 2016] simplifies the implementation of WFC in Unity a lot. It uses the standard WFC algorithm [Gumin, 2016b] and provides wrappers around those models. It is quite an early port that was already used in several games [Parker et al., 2016], [Parker et al., 2017], or [Parker, 2018]. We have heavily modified this wrapper so it works better for our use case. This package is distributed under the MIT license as the WFC algorithm.

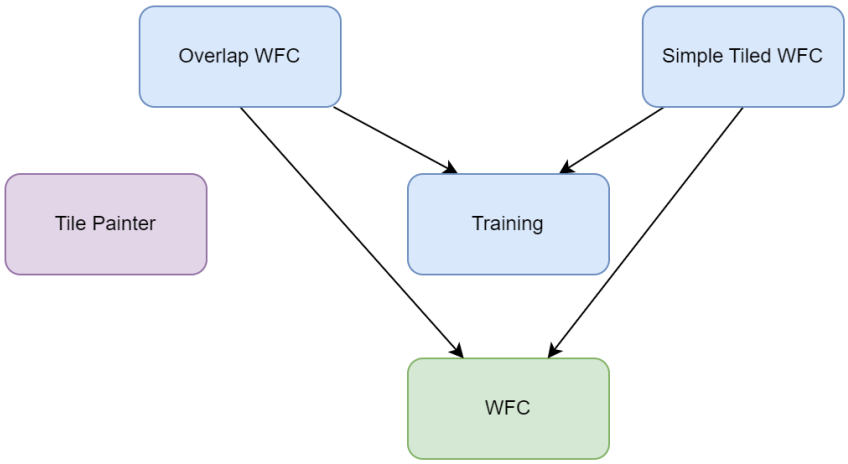

Figure 5.1: WFC Wrapper overview

Figure 5.1 shows an overview of the package. It contains the regular imple-

mentation of the Wave Function Collapse algorithm, we will call it `WFC` [Gumin, 2016b], and four other components. Those are `OverlapWFC`, `SimpleTiledWFC`, `Training`, and `TilePainter`. The package represents every tile as a whole `GameObject`. It is extremely flexible since we can use it for larger objects like a whole room. It also simplifies the Hierarchical implementation since we can add any data to each `GameObject`. However, using a `GameObject` has significant memory and performance consequences. Using a `Tilemap` would be much more efficient for tiles. Therefore, it might be useful to transform these `GameObject` into `Tilemap` as postprocessing [Filek et al., 2022].

5.1.1 Tile Painter

The `TilePainter` component helps to create an input for the WFC. For us, it is the most important part of the package. It heavily utilizes a custom `Editor`, which allows us to draw directly in the scene view.

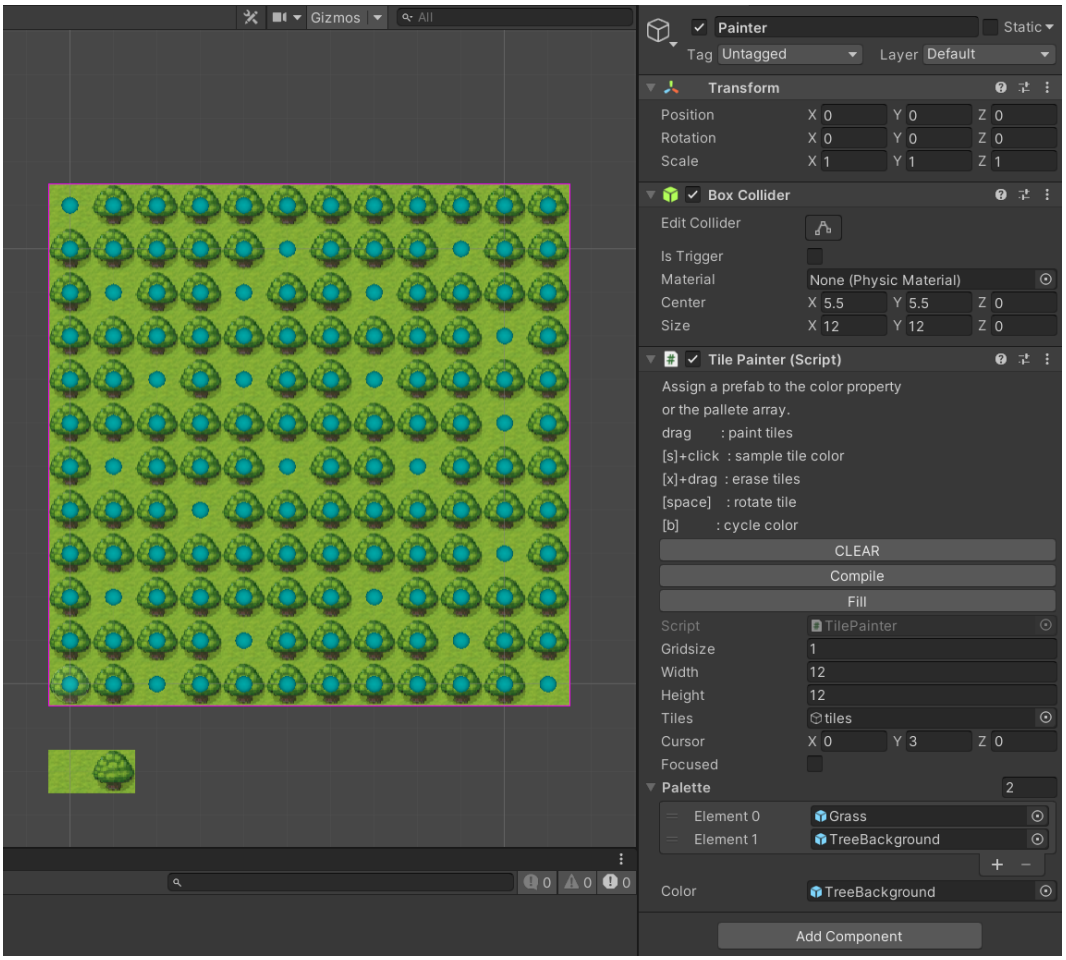

Figure 5.2: Tile Painter

An example of the `TilePainter` is shown in figure 5.2. We can see a drawn map on the left side in the scene view. Implementation-wise, it takes a list of

prefabs to instantiate to the scene. We can see those prefabs in the `Palette`. Then we need to select an active prefab that will be placed into the scene. This active prefab is the `Color`; in our case, it is *TreeBackground*.

The placing of prefabs to then the scene is done via drawing over the canvas. Therefore, the canvas needs to have attached `Collider` to register mouse events. We can also adjust the canvas size by setting `Width` and `Height` parameters. The `TilePainter` instantiates all spawned `GameObject` as children into a single `GameObject` that is attached as its child.

We have added some quality-of-life changes. Firstly, the *Fill* button fills the entire map with the selected prefab. Secondly, the *Compile* button will try to compile the map. We will discuss what it means in the next section.

5.1.2 Training

The `Training` component specifies an input to the WFC. We can attach it to the child `GameObject` created by the `TilePainter`. It extracts the drawn tiles and then converts them to a representation that the WFC uses. That is a `byte[,]` used for bitmaps. It is also useful to convert the resulting `byte` produced by the WFC algorithm back original `GameObject`.

It has one critical function `Compile()`. It scans placed tiles and extracts this information from them. It assigns a unique `byte` value to each unique prefab. Then it creates a `byte[,] sample` of the same size from the input. Alternatively, we can export the input as a *.xml* file of the Simple tiled model.

5.1.3 Model Wrappers

The `OverlapWFC` component and the `SimpleTiledWFC` component are wrappers around the Overlapping model 1.3.1 and the Simple Tiled Model 1.3.2, respectively. They derive from `MonoBehaviour` so we can place them into the scene. We can set up all the standard parameters, such as *width*, *height*, or N in the case of the Overlapping Model. We must also set its `Training`, which specifies the input. When we call the `Generate()`, it creates a particular model of WFC, forwards its parameters, and runs it. When the generation is finished, it extracts its result and draws the output to the scene. There is one more useful feature. It allows us to limit the number of iterations that the algorithm will execute. When the generation is not finished, it runs more iterations in the `Update()`. This can highly improve responsiveness. Otherwise, it could freeze the application until the generation has finished.

The algorithm can spawn tens of thousands of `GameObject` that cause some issues. One problem is that the scene can be over 100 MB large. We can over-

come those huge sizes by using algorithmic compression. We can only store the parameters and seeds for the generator and generate the map when required.

5.2 Modifications

To support all the features mentioned in previous chapters, we had to modify the WFC implementation and the wrapper. Primarily we need to add support for the Early Tile Placement and the Postprocessing. We discuss them in the following sections.

5.2.1 Base WFC

The first change was to add the `BaseWFC` component. It is derived from the `MonoBehaviour`, so it can be attached to a `GameObject`. This class contains all the shared data and functionality for both wrappers. Then, we can derive `OverlapWFC` and `SimpleTiledWFC` from the `BaseWFC`.

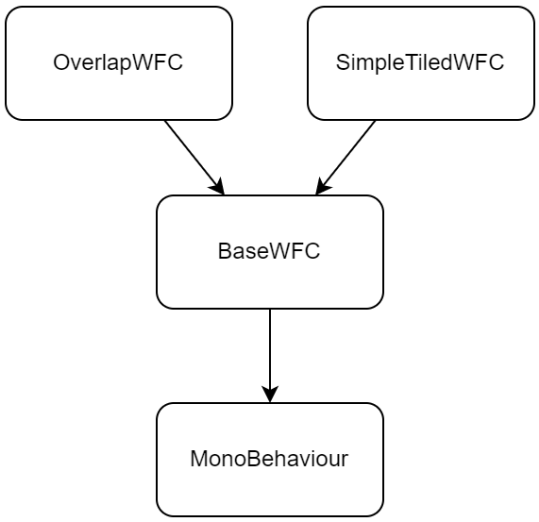

Figure 5.3: WFC Wrapper — inheritance

Figure 5.3 shows the scheme of inheritance. The base class `BaseWFC` allows us to use both wrappers interchangeably for our hierarchical WFC.

5.2.2 Early Tile Placement

The most crucial new feature is the Early tile placement 2.3.1. The modification to the WFC is simple — it takes an extra list of $pattern, position$ pairs. It is handled by a `Predetermited` class. Then in the `Observe()`, it checks whether

there is some cell for early collapse, picks it, and sets its pattern accordingly. The `Propagate()` does not need any modifications. The tile is represented as its index in the `Training` component. As a quality of life change, this index is the same as the tile index in the `TilePainter`.

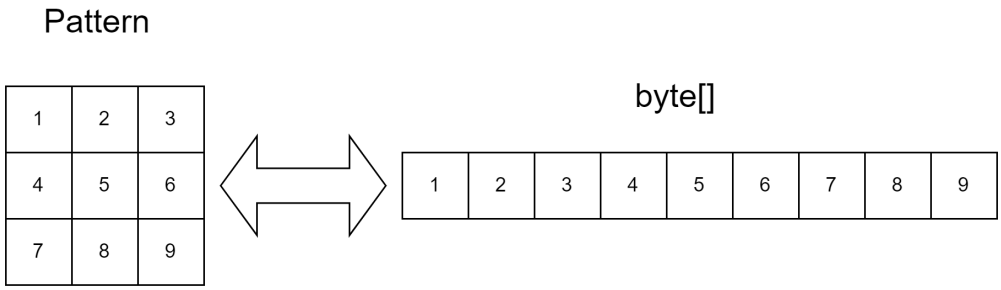

Figure 5.4: Predetermined pattern for the Overlapping model

A bit tougher is to specify the pattern for the Overlapping model. The whole pattern represented as `byte[]` must be correctly encoded into a single `long` value. WFC uses this representation for better performance. This transformation is done internally in the constructor of the `OverlappingModel`. However, the order of those tiles is not intuitive. Therefore, the `Pretetermined` takes a `byte[]` and handles this transformation internally. If we want to create a pattern, we can specify it row by row as shown in figure 5.4.

We want the Early Tile Placement to be easily specified in the `OverlapWFC` or the `SimpeTiledWFC`. Therefore, we have created a `Preplacement` script which derives from `ScriptableObject`. It contains two virtual functions.

Public virtual functions

- `virtual void Run(OverlapWFC)`
- `virtual void Run(SimpleTiledWFC)`

This way, any custom script deriving from `Preplacement` can override the behavior for both types of WFC. Besides that, it can get all the required information from its argument. Most critically, it can get the size of the WFC. It provides us with great flexibility. We can call `preplacement.Run(this)` from our WFC wrappers and the overload resolution will handle calling the appropriate version.

The primary use of the early tile placement is on large chunks. The reason is that in the hierarchical approach, we need to fill areas covered by different WFCs to avoid degenerate outputs. Using this observation, we can significantly improve the performance of the early tile placement. We can start by collapsing all predetermined cells, and only after that can we propagate everything information. We

Algorithm 5 WFC - Early tile placement (optimized version)

Input: width W, height H, pattern size N, input bitmap B,
early placement list P
$patterns \leftarrow all\ N \times N$ patterns from B
$weights \leftarrow \#occurrences$ of each pattern
$constrains \leftarrow$ which patterns neighbors which in B
$propagator \leftarrow array\ (W \times H)$ each contains a list of $patterns.size$ of indices

// position (w, h) is collapsed $\iff propagator[w, h].size \leq 1$
while *not collapsed* **do**
  if $exist\ (w, h) \in propagator : propagator[w, h].empty$ **then**
    return failure {random restart here}
  end if
  if $!P.empty()$ **then**
    while $!P.empty()$ **do**
      $last \leftarrow P.size() - 1$
      $(w, h) \leftarrow P[last].position$
      $selected \leftarrow P[last].pattern$
      $P.remove(last)$
      $propagator[w, h] \leftarrow selected$
    end while
  else
    $(w, h) \leftarrow$ uncollapsed position from propagator with minimal entropy
    $selected \leftarrow$ random weighted pattern from $propagator[w, h]$
    $propagator[w, h] \leftarrow selected$
  end if
  propagate this with *constrains*
end while
return *propagator*

can afford such an approach since we do not need information about unsupported patterns during the collapse of predetermined tiles. Therefore, we can propagate constraints only at the end of this step. It saves us a lot of work since many of the cells to propagate are already collapsed.

We can see the optimized version of the algorithm 5. The main difference is the addition of a `while` cycle on line 12. Another critical detail is the `if-else` block around the `while` cycle. This is crucial as we must first propagate all the constraints before collapsing a normal cell. Otherwise, there is a significant chance that we would pick a wrong value, and the algorithm would immediately return a *failure* in the next step. We are also removing elements from the back of the list since this does not force a move of all elements as removing from the front.

It has a flaw. When we collapse and propagate to a large part of the map, it is a significant amount of work that can freeze a game for a short duration. On the other hand, it speeds up the generation a lot, so it is, in our case, probably worth it.

5.2.3 WFC Postprocessing

We can run a custom postprocessing script after each WFC. This script needs to have all information about the result of the generation. Therefore, we must pass the entire result of `GameObject[,]` into it. Then the script can do any modifications to our results.

Implementation-wise it is almost the same as the Early Tile Placement. We have a `WFCPostprocessing` script derived from `ScriptableObject`. It contains a `virtual void Run(GameObject[,])`. We can override this function in any further derived class.

There are two custom postprocessing that we are using right now. The first one is `RemoveSmallRooms`, which removes areas that are too small. It is helpful since some WFC will always fail in tiny areas. The second is `ConnectRooms` which is much more complex. Its purpose is to connect rooms. It is used in the dungeon generation to add paths in Cavern-like areas.

5.2.4 Runtime

We can limit the number of iterations that the algorithm executes per update. Otherwise, we could not see how the map was generated. An even larger reason is that it would totally freeze the game for a couple of seconds.

However, the original implementation has a major issue. When we want to draw the generated parts, it will loop over all tiles, look if it is already collapsed,

and draw it. We want to collapse a low number of tiles each update, so this has significant performance implications. However, there is an easy fix. The WFC can store the collapsed tiles in each iteration. Then, we could loop only over those tiles. This way, the draw will access every tile only once, which is optimal.

This sounds easy, but collapsing a single tile can collapse more tiles around it. Therefore, we cannot add those tiles in `Observe()` because not all collapsed tiles will be there. However, we can postpone it to the `Propagate()`. It works because if the tile is collapsed in a given iteration, the `Propagate()` function has to access it and propagate this information further.

We could use threads to generate WFCs. In the case of Unity, we would need to use its job system. However, moving the implementation to jobs is not trivial. Besides that, a significant amount of work would be required to synchronize results with the main thread. In unity, only the main thread can instantiate objects.

5.3 Hierarchical Controller

The `Hierarchicalcontroller` is the main script handling the HWFC. It holds layers of WFC. Then it generates the given layers one by one. When one layer is done, it will start the next layer on top of its output. It is done by instantiation of appropriate WFC to each location.

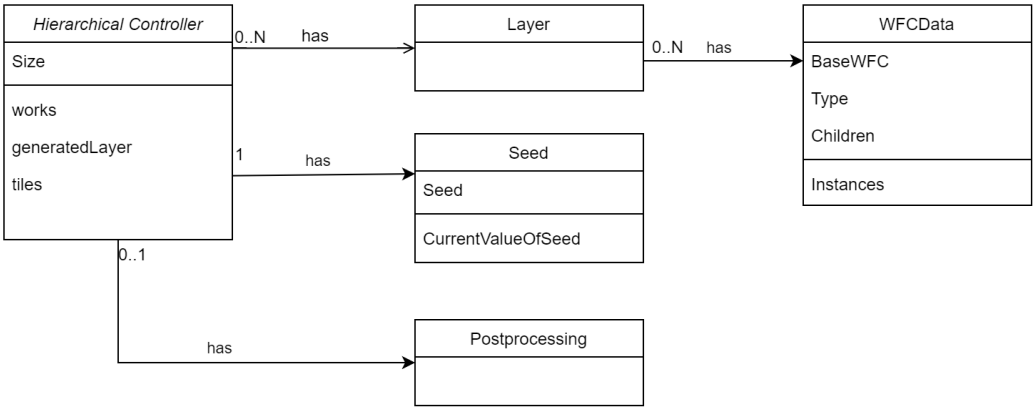

Figure 5.5: Class diagram of the `Hierarchicalcontroller`; arrows mean has instances of relationship

Figure 5.5 shows the class diagram of the `Hierarchicalcontroller`. It heavily utilizes composition.

Individual WFC place down prefabs. Those prefabs should have attached the `TileType` component. It holds a single `int` value. Based on this value, the Hierarchical Controller identifies regions for the subsequent layers. Each WFC is

stored in the structure called `WFCData`. It contains the following variables.

WFC data

- `BaseWFC wfc` — the WFC itself
- `List<int> children` — indices of its children in the next layer
- `List<int> values` — values of `TileType` on which we can run instantiate this WFC; it is important the determine shape of each region
- `List<BaseWFC> instances` — all active instances of this WFC; it is used internally and not specified by the user, so it is hidden

We could represent `List<int> values` only as a single `int` value. However, using `List<int> values` gives us greater flexibility and a higher chance to reuse some prefabs for more WFCs.

With the knowledge of the `WFCData`, we can define a class `Layer` representing one layer. It contains a `List<WFCData>` and some extra helper functions. Therefore, the whole structure of WFCs is `List<Layer> layers`. Our structure for layers is linear. We can see an example of it in figure 5.6. Each color represents one layer. Links represent dependencies between subsequent layers. In our case, those links are indices to the next layer.

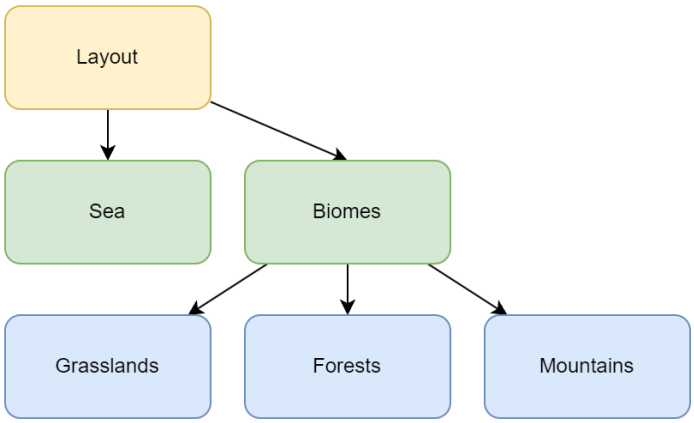

Figure 5.6: Linear structure of WFCs

We could also use a tree-like recursive structure. There, each `WFCData` would hold a `List<WFCData> children` directly instead of their indices in the next layer. However, we have tried this approach, and it was very difficult to set everything up properly in the inspector. We would need to create whole custom editor windows with a tree-like representation, which would have an even steeper learning curve.

We need to specify the following properties:

- `vector2 size` – the size of the output; the resulting output will be twice as large is each dimension due to upscaling
- `int seed` – the seed used for the generation; 0 is a random seed
- `List<Layer> layers` – all WFCs we want to use in a format discussed above
- `Postprocessing postprocessing` – the postprocessing that covers that whole output and is run as a last step; this field is optional

The public API of the `Hierarchicalcontroller` is straightforward. It has one function responsible for the whole generation.

Public API

- `public void StartGenerating(bool incrementSeed = false)`

Algorithm 6 HWFC – Main loop

1: **Input:** *width w, height h, List¡Layer¿ layers*
2: *Clear all generated WFCs*
3: *Run the layout WFC*
4: *Wait until it is done*
5: **for all** *layer* ∈ *layers* **do**
6: *GenerateLayer(layer)*
7: *Wait until all WFCs are done*
8: **end for**
9: *Export the output into a flat single-layered output*
10: *Run postprocessing if specified*

Algorithm 6 shows a pseudo-code of the main loop. It is executed after calling the `StartGenerating()`. There are two important note about this algorithm. The first one is that we need to wait after each WFC until it is done. This makes the implementation much more difficult. We are using callbacks to do so. When each WFC is created, we register an `OnGenerationDone()` of the `HierarchicalController` to an event in the WFC. This way, we get informed when the WFC finishes. We can count how many WFCs need to be generated in each layer. When all required WFCs are done, we will call an `UpdateLogic()`, which moves at the next step as shown in algorithm 6.

The second note is the `GenerateLayer()` on line 6. This function is crucial for the HWFC. In fact, it does not take a `layer` but its index. The reason is that we also need access to the previous layer.

Algorithm 7 HWFC – `GenerateLayer()`

```
1: Input: int layerIndex
2: for all prevLayerWFC ∈ layers[layerIndex − 1].layer do
3:   for all wfcInstance ∈ prevLayerWFC.instances do
4:     for all wfc ∈ prevLayerWFC.children do
5:       for all area ∈ FindAllPatterns(wfcInstance, wfc) do
6:         Instatiate new WFC over the area
7:         Set its position, size, and bitmap of discarded positions
8:         Register it for a callback when it is done
9:       end for
10:    end for
11:  end for
12: end for
```

The algorithm 7 might look scary since there are four nested for-loops. However, we usually need to iterate only over a few elements in each of them. The first one extracts all WFCs from the previous layer. The second one extracts all WFC instances of each WFC. Each WFC can potentially have more instances. For example, we can have more continents, as was shown in figure 3.10. The third iterates over all WFCs that are children of the selected WFC in the previous layer. That means grasslands, forests, and mountains for the biomes WFC in our world map example. Finally, the fourth finds all matching consecutive areas and instantiates proper WFC instances over each of them. We find those consecutive regions using BFS, which is probably the simplest option.

There are some public functions that are not expected to be called from the code. The purpose of those functions is to generate given layers from the inspector manually. This way is cumbersome but very helpful for debugging. Those functions are all called from the implementation and usually have major preconditions.

Public functions usable from the inspector

- `public void GenerateLayer(int layerIndex)`
- `public void UpscaleMap(int scale = 2)`
- `public void Clear()`
- `public void ExportMap()`

The `GenerateLayer(int layerIndex)` generates a given layer. However, it has a precondition that all previous layers must be completed. Otherwise, the behavior is undefined.

The `UpscaleMap(int scale = 2)` upscales the whole map by a given scale. Obviously, this scale has to be positive. However, the implementation supports the upscaling only directly after the first layer. It could be generalized to work after any layer, but we did not find a case where it would be helpful. Therefore, it could be called only directly after the generation of the first layer.

The `ExportMap()` flattens all layers into a `GameObjects[,]`. Ultimately, we do not need intermediate layers and only care about the result. This function also applies the postprocessing on our result if one is provided.

Finally, the `UpdateLogic()` works only in the game mode. In the editor, it uses a macro to exit and does not mess with a manual per-layer generation. It is a reason why using the HWFC in a game mode is strongly preferred since it is much more convenient.

The `HierarchicalController` has `public UnityEvent generationDone` and invokes it at the end of the generation. This way, we can subscribe to it and know when the generation finishes. We have used it to automate the generation of many maps for our benchmarks.

5.3.1 Editor

The HWFC has a steep learning curve. Therefore, it is crucial to make its specification as easy as possible. One of the essential features of our implementation is a custom editor. It is called `HierarchicalControllerEditor`, and it dramatically simplifies the setup of the HWFC.

The example of the custom editor is shown in figure 5.7. It primarily deals with layers. As mentioned earlier, the layer needs a `List<WFCData>`. However, there is an abstraction that holds it in a `Layer` class. We want to make setting values in the HWFC as easy as possible, so this `List<WFCData>` is exposed directly to the user without the need for the `Layer`. We can see the whole representation of the second layer in figure 5.7. It is reasonable to specify, and it is readable. There are also buttons that allow us to add and remove layers with the restriction that we always require at least one layer.

The first layer is unique. It is represented as a regular layer since it simplifies the implementation. However, it must contain only one WFC. Besides that, some parts of the `WFCData` are irrelevant. The `List<int> values` is useless since it is not running over any other WFC, and `List<int> children` must contain all WFC indices in the second layer. Therefore, we need to specify only the `BaseWFC wfc`. Our custom editor does this and represents it as a `ROOT` variable.

The editor also provides buttons for the manual generation of layers. How-

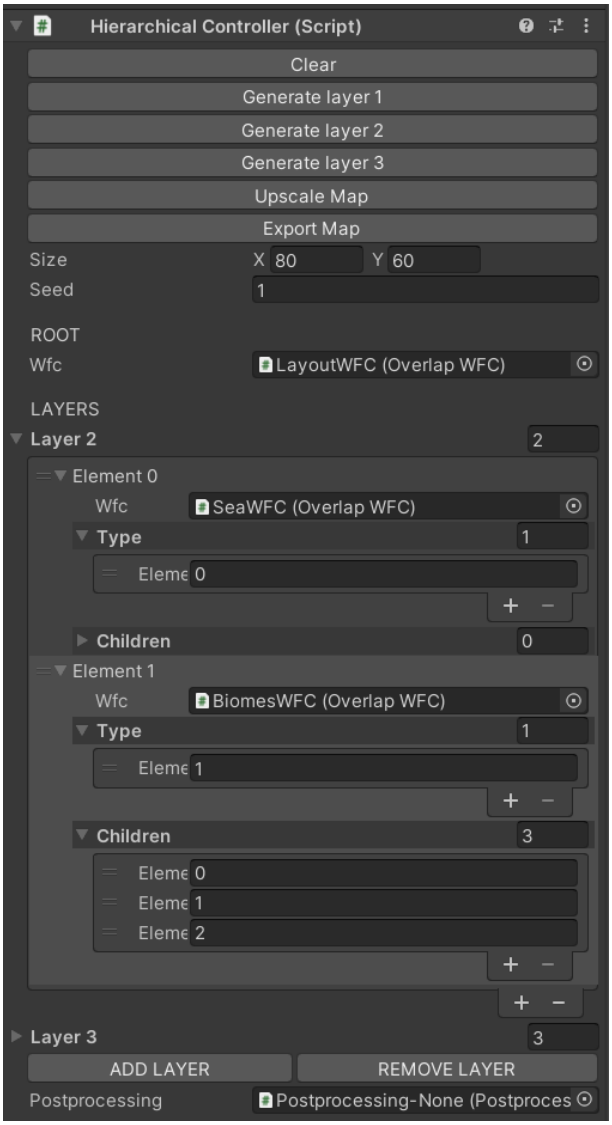

Figure 5.7: Editor of the Hierarchical Controller

ever, those buttons could be used with caution since they have preconditions, as mentioned earlier. Besides that, the generation is much slower if we do not run the application. Still, those options might be handy for debugging. The first button is *Clear*, which clears the whole input. Then there are N buttons. Those are *Generate layer 1 – Generate layer N*. Each is responsible for the generation of an appropriate layer. The displayed buttons are dynamic based on the number of layers we have specified. After it, there is an $UpscaleMap$ button, which upscales the map after the first layer. Finally, the last button is *Export Map*, which creates a single-layer representation of the output.

Finally, the editor also simplifies the definition of seed. It exposes a single `int` value directly from its encapsulated representation.

5.3.2 Seed

We want to provide an option to specify a seed to the Hierarchical Wave Function Collapse to make our results reproducible. However, there are a few issues. Firstly, we need to know how seeding works for the standard WFC. Indeed, there is an option to specify an integer as a seed, and the value `seed = 0` means a random seed.

We need to specify a `seed` for the `HierarchicalController`. The trivial solution would be to assign its value to every WFC instance we will use. However, there are two problems with this approach. The main problem is that if some WFC fails to generate, then the whole Hierarchical WFC will fail to generate. This is an enormous problem since there is a pretty solid chance that one of our many WFCs could fail. This has an easy solution. If a WFC fails to generate and its `seed` $\neq$ `0`, we can simply increment the `seed` and rerun the WFC.

The other problem is that some abstract WFC could generate two exactly the same patterns. Then when we run two instances of WFC from the next layer over it, we would get the same output for both of them. This would also imply for each following layer. Therefore, if we have two same size continents in the World Map example, then both would look exactly the same. We can again solve this issue by increasing the `seed` for each generated WFC. This can still cause the same issue. If the first WFC fails, it will increment the `seed` itself and will again get the same results. However, this is very easy to adjust. We just need to add a larger number to the `seed` for each WFC. If we would take `seed += 10` for each WFC, then we would need at least 10 failed generations to be stuck on the same seed again.

The last thing to consider is that we need to save the initial *seed* value. The reason is that we might want to re-generate the output. And the algorithm has the same `seed`, so it should behave the same.

Conclusion

In this thesis, we have introduced and implemented the hierarchical version of the Wave Function Collapse algorithm. We have also shown its capabilities for generating complex non-homogenous outputs like the world map or a dungeon satisfying given design criteria. We had to experiment and implement multiple improvements like the Early Tile Placement and borders, work with non-rectangular output and enforce its minimal size, and use per-layer and final postprocessing to make it all work.

We have demonstrated that the Hierarchical Wave Function Collapse in complex scenarios outshines the standalone Wave Function Collapse. The HWFC can work even in situations where the use of the regular WFC is infeasible. Besides that, upscaling the first layer and intelligent use of the Early Tile Placement allows us to achieve comparable performance to the standalone WFC. We have also shown that the HWFC has the potential for runtime usage, but more research is still required. The runtime was very close on average but did not spike nearly as high as the standalone WFC, which is critical for the runtime usage.

However, the primary advantage of the HWFC lies in spitting the huge and very complex input into several smaller ones. Creating one large, very complex input is an extremely challenging task. On the other hand, spiting it into smaller ones has two advantages. Firstly, it is much easier to create. Secondly, we can modify the individual outputs later and easily iterate the generator. Modifying the complex huge input of a standalone WFC is also very challenging. This division of the input makes its creation much more manageable or even possible.

Outputs from the HWFC algorithm are very consistent and have a similar level of quality. Furthermore, ínputs have a similar structure, but each output feels unique. It is caused by the generation in layers. The first layers significantly influence the shape of the final output, and the later layers add details.

The HWFC also has some disadvantages. The most obvious is its complexity. Even the standalone WFC has a very steep learning curve. It is easy to make inputs and start generating some outputs. However, it gets challenging if we want those outputs to satisfy some design criteria. The learning curve is even steep for the HWFC. Especially the user has to think about layers and how to spit the task to create the desired output. On the other hand, enforcing the design criteria is much simpler for the HWFC.

Further Work

The thesis focuses mainly on experimenting and making the HWFC work. Therefore, it is open to much further work.

The most obvious is to generate more esthetically pleasing outputs. This task should be mainly about good assets. The other option is to generate an output of larger-scale structures. It means using more complex objects instead of tiles or bitmaps as assets. We could, for example, use whole rooms or major parts of them. This way, it generates much larger and more complex outputs without any performance cost.

Another option is to focus on the runtime use of the HWFC. This is challenging since it is required to ensure a high quality of generated levels. It could be achieved by the generate and test method. However, the WFC algorithm is slow, so this approach is probably, not optimal.

Another task is to improve the performance of the HWFC algorithm. The easiest way is to implement it in C++ for Unreal Engine 5. Using C++ allows us to write way faster code. However, the most probable place where the HWFC could be used is some experimental game developed in Unity. There is still a place to optimize our Unity version. We are not using the job system. We are drawing new parts of the output each frame, which is possible only from the main thread. Therefore, heavy synchronization would be required. However, drawing the output might not be desirable (if the player should explore the map). In that case, using the job system could speed up the generation by a significant amount.

Probably the greatest challenge is incorporating the HWFC into a full game that can be commercially successful. This should be possible, as shown in the Dungeon of Chaos game. [Filek et al., 2022] However, there is still much work for a larger-scale commercial game.

Another possibility is incorporating the hierarchical idea into a 3D WFC. The hierarchical idea will work the same way. However, 3D WFC is even more complex and harder to use. It would be a large task to make it work well and produce good outputs.

Bibliography

Brian Bucklew. Dungeon Generation via Wave Function Collapse. Roguelike Celebration, Oct 2019a. URL https://www.youtube.com/watch?v=fnFj3dOKcIQ. Accessed: 15/4/2022.

Brian Bucklew. Tile-Based Map Generation using Wave Function Collapse in 'Caves of Qud'. Game Developers Conference, 2019b. URL https://gdcvault.com/play/1026263/Math-for-Game-Developers-Tile. Accessed: 15/4/2022.

Darui Cheng, Honglei Han, and Guangzheng Fei. *Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm*, pages 37–50. Entertainment Computing – ICEC 2020, Jan 2020. ISBN 978-3-030-65735-2. doi: 10.1007/978-3-030-65736-9_3.

Emil Ernerfeldt. Wave Function Collapse in C++. GitHub, Oct 2016. URL https://github.com/emilk/wfc. Accessed: 3/5/2023.

Jiří Filek, Martina Fusková, and Shivam Sharma. Dungeon of Chaos. GitHub, Oct 2022. URL https://github.com/fileho/Dungeon-of-Chaos. Accessed: 17/1/2023.

Maxin Gumin. Basic 3D WFC. Bitbucket, Nov 2016a. URL https://bitbucket.org/mxgmn/basic3dwfc/src/master/. Accessed: 3/5/2023.

Maxin Gumin. Wave Function Collapse. GitHub, Sep 2016b. URL https://github.com/mxgmn/WaveFunctionCollapse. Accessed: 17/2/2022.

Isaac Karth and Adam Smith. Wavefunctioncollapse is constraint solving in the wild. In *The International Conference*, pages 1–10, Aug 2017. ISBN 978-1-4503-5319-9. doi: 10.1145/3102071.3110566.

Pedro Minini and Joaquim Assunção. Combining Constructive Procedural Dungeon Generation Methods with WaveFunctionCollapse in Top-Down 2D Games. Nov 2020.

Joseph Parker. Unity Wave Function Collapse. itch.io, Oct 2016. URL https://selfsame.itch.io/unitywfc. Accessed: 12/3/2022.

Joseph Parker. Bug with a Gun. itch.io, 2018. URL https://selfsame.itch.io/bug-with-a-gun. Accessed: 21/02/2023.

Joseph Parker, Ryan Jones, and Oscar Morante. Proc Skater 2016. itch.io, 2016. URL https://arcadia-clojure.itch.io/proc-skater-2016. Accessed: 01/02/2023.

Joseph Parker, Douglas Fileds, Joshua Suskalo, Ramsey Nasser, and Tims Gardner. Swapland. itch.io, 2017. URL https://arcadia-clojure.itch.io/swapland. Accessed: 19/02/2023.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence a Modern Approach*. Pearson, 3 edition, 2010. ISBN 978-0-13-604259-4.

Hugo Scurti and Clark Verbrugge. Generating Paths with WFC. *CoRR*, abs/1808.04317, 2018. URL http://arxiv.org/abs/1808.04317.

Noor Shaker, Julian Togeliusm, and Mark J. Nelson. *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems. Springer International Publishing, Cham, 1st ed. 2016. edition, 2016. ISBN 3-319-42716-4.

Oskar Stålberg. Wave - by Oscar Stålberg, May 2017. URL https://oskarstalberg.com/game/wave/wave.html. Accessed: 8/4/2022.

Oskar Stålberg. Bad North, Aug 2018. URL https://www.badnorth.com. Accessed: 10/8/2022.

Oskar Stålberg. Townscrapper, June 2020. URL https://www.townscapergame.com. Accessed: 10/8/2022.

Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2nd International Workshop on procedural content generation in games*, PCGames '11, pages 1–6. ACM, 2011. ISBN 9781450308724.

Guillaume Werle and Benoit Martinez. 'Ghost Recon Wildlands': Terrain Tools and Technology. Game Developers Conference, Feb 2017. URL https://www.gdcvault.com/play/1024029/-Ghost-Recon-Wildlands-Terrain. Accessed: 25/2/2023.

List of Figures

List of Tables

List of Abbreviations

WFC — Wave Function Collapse algorithm 1.3.1
HWFC — Hierarchical Wave Function Collapse 2.2
PCG — Procedural Content Generation
CSP — Constraint Satisfaction Problem
API — Application Programming Interface

A. Attachments

This chapter describes attachments to the thesis.

A.1 Unity Project

The Unity project is placed in a folder *source*. It contains the implementation of HWFC and the examples we used in the thesis. It used Unity version 2020.3.19f1 LTS.

The application can also be downloaded at GitHub:
https://github.com/fileho/Hierarchical-Wave-Function-Collapse.

A.2 Build

The build application is placed in a folder *bin*. It contains a build for the Windows platform. For other platforms, building the application from the Unity project is necessary.

The application contains both examples: the world map and the dungeon. It is important to fill in the *seed*. It should be a non-negative integer. $Seed = 0$ generated pseudo-random output. We can generate and inspect many outputs of the HWFC using this application. More information is in the user documentation.

A.3 User Documentation

User documentation is placed in *documentation* folder. It describes how to use both the build and the Unity project.