BSc in Computer Science

School of Computing, Science and Engineering

Final Year Project Report

Procedural Content Generation Aided by Wave Function Collapse

Author: Robert Andrei Popovici @00600031

Supervisor: Dr. Norman Murray

2021-2022

Abstract

In 2016, Maxim Gumin released a procedurally generating algorithm that makes use of notions from many different areas of programming. With the main two areas being constraint solving and texture synthesis. Most game developers have treated this algorithm as a “black box” where their involvement with the actual development of it has been minimal, while some have gone on to make very specific versions of the algorithm proprietary to their games. This work sets out to recreate the algorithm in a 3D format as an extension of Unity to aid with development of procedurally generated worlds as a general solution. The final solution managed to create varied, expansive 3D models given a much smaller input training data by extracting constraints from the input and using a solver to create the model given the constraints. The current implementation does come with some caveats, as it currently stands the height of the model cannot be changed and the output is very dependent on the input data given. Finally, it cannot create structures that are meant to be logically connected (eg. a road network) as the algorithm focuses on finding groups of tiles that can be patterned such as wang tiles.

Table of Contents

1.Introduction

1.1 Motivation

The main reason for undertaking the project is the fact that I found the idea behind the WFC algorithm interesting as it can procedurally generate seemingly complex worlds from a basic set of constraints or even just a sample image. Another great pro is the fact that I get to familiarize myself with Unity even further by being able to engage with the Editor and Engine more deeply. While it may seem like a challenging algorithm, the implementation shouldn't be too complex even for 3D worlds and the main foreseeable issue will be with performance and improving it on lower-end hardware.

1.2 Objectives

Objective 1: Research the Wave Function Collapse algorithm

Objective 2: Research the best way to extend the Unity Inspector

Objective 3: Extend the Unity Inspector to create a basic 3D tilemap editor

Objective 4: Build a preliminary world using the WFC algorithm as proof of concept

Objective 5: Fix any issues within the WFC algorithm and the tilemap editor

Objective 6: Marry the tilemap editor and the WFC algorithm

Objective 7: Create a 3D model using the editor and the algorithm

Objective 8: Evaluate the system

Optional Objective 1: Extend the Wave Function Collapse from simple model to overlapping model

Optional Objective 2: Extend the tile editor with more functionality

1.3 Overview

The report starts with the literature review where we first discuss what the algorithm was based on. Starting with procedural content generation, followed by an explanation of texture synthesis, discrete model synthesis and finally an explanation of the algorithm itself and how it goes through different phases.

The next chapter focuses on the methodology used in the project and the reasoning behind taking a more research heavy approach when developing this solution.

Next up we will discuss the requirements of the project alongside a justification for each requirement with a discussion on the design of the project and why some design decisions are the way they are. Both the requirements and the design have been split into sprints.

Following this, an explanation of the implementation written in the same flow as the algorithm in a bottom-up approach detailing how the algorithm was implemented, the results and any issues that arose during development.

The next chapter will discuss our testing methodology and what tests have been done, along with a justification for the use of different tests, followed by all the test cases that have been done.

The penultimate chapter is a critical evaluation of whether the project followed the initial plan, with any justifications for deviations from that plan with reflections done on the overall work and lessons learned during the time of this project.

Finally, the last chapter concludes the document with a review of the work done with comparisons made to the original algorithm and a reflection on what went well and what didn't go so well.

2.Literature Review

2.1 Procedural Content Generation Overview

Procedural content generation or PCG in games, "*is the algorithmic creation of game content with limited or indirect user input"* (Togelius et al., 2011). It is also important to note how the term "content" is used in the definition as (Togelius et al., 2016) uses it to define maps, stories, items, levels etc. But makes the clear distinction that neither the game engine nor the Non-Playable Character AI are examples of PCG. And while PCG is based on AI methods it is not related to the more popular use of AI in games where the AI algorithms "learn" to play the game for example.

2.1.1 Desirable Traits of a PCG solution

Togelius et al. (2016) states that "*We can think of implementations of PCG methods as solutions to content generation problems*". The implementation of these methods containing different properties related to the problem we are trying to solve. He then explains that there are common properties that are usually desired for a PCG solution, them being:

- *Speed*
  Where speed can vary from implementation to implementation based on different factors such as whether the algorithm is running during game time or whether it is used during the game's development.
- *Reliability*
  There are cases when we need our algorithm to reliably satisfy some constraints such as ensuring that there is always an exit to our dungeon.
- *Controllability*
  In most cases of PCG we want to be able to have control over some of the variables. This allows us to control the result to some degree. What that control might look like differs from implementation to implementation.
- *Expressivity and Diversity*
  There is a need in PCG to be able to generate diverse content to avoid having all models looking too similar. How much expressivity is allowed within the system depends on the implementation and the problem it is trying to solve.
- *Creativity and Believability*
  There needs to be some level of believability in the product of some PCG algorithms, an example can be generating a forest procedurally and then being able to control some aspects of that forest such as the number of trees in specific areas, paths, clearings etc… to make a scene look believable.

2.1.2 Automatic generation versus mixed authorship

Traditionally PCG has allowed limited input from game designers with automatic generation where only few parameters can be changed. However, there has been a recent emergence of mixed authorship generation where the designer cooperates with the algorithm during the design process. WFC for example is a mixed authorship algorithm where the designer inputs a design, and the algorithm then takes that and refactors it while maintaining some similarities from the original input.

2.2 Texture Synthesis by non-parametric sampling

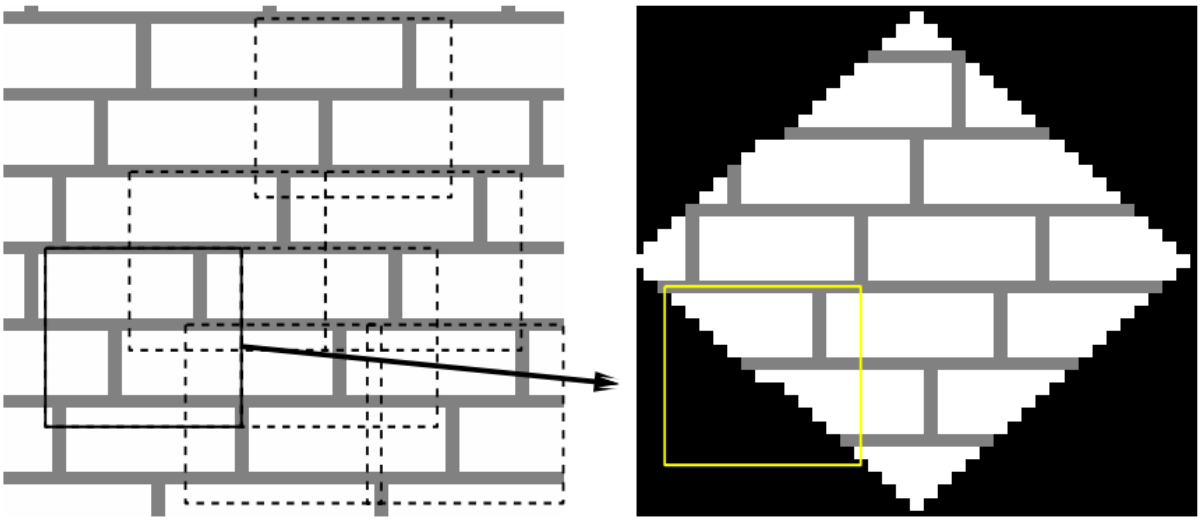
(Efros et al, 1999) describes the problem as follows; A texture is a visual pattern on an infinite 2-D plane where we want to "synthesize" multiple texture samples from some other given finite texture without any assumptions about the sample given (I.E: size of sample is large enough or texture elements scale is known).

2.2.1 The Algorithm

Given the fact that we cannot make any assumptions on a given sample, there must be a way to capture all the information about the texture. The method used by (Efros et al, 1999) has us querying each sample image and then creating a histogram of the distribution of *p*. *p* being the unit of synthesis and equal to one single pixel (size of *p* is adjustable by the user). The algorithm then grows the texture pixel by pixel outwards from an initial seed as shown in Figure 2.1, with all the previously synthesized pixels in a square window used as context.

Figure 2.1

Synthesis of texture from sample



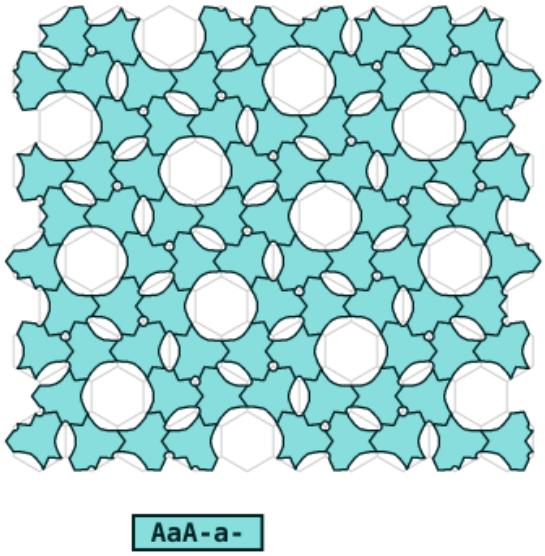
Note. This model was produced by Efros et al in 1999, showing how the algorithm produces a texture from a sample. From "Texture synthesis by non-parametric sampling", by A. Efros and T. K. Leung, 1999, IEEE International Conference on Computer Vision, Corfu, Greece. Copyright 1999 IEEE

2.2.2 Declarative Texture Synthesis

(Harrison, 2005) defines declarative texture synthesis as designing textures using rules. These rules help specify which elements in a pattern may be adjacent to one another. Elements are typically square or hexagonal, each having their unique edges that are compatible with other element's edges. The goal then is to take these elements(tiles) and populate them with specific rules(constraints) that allow us to connect shapes together like a jigsaw puzzle to form a cohesive 2D pattern as show in Figure 2.2.

Figure 2.2

Pattern formed by the algorithm



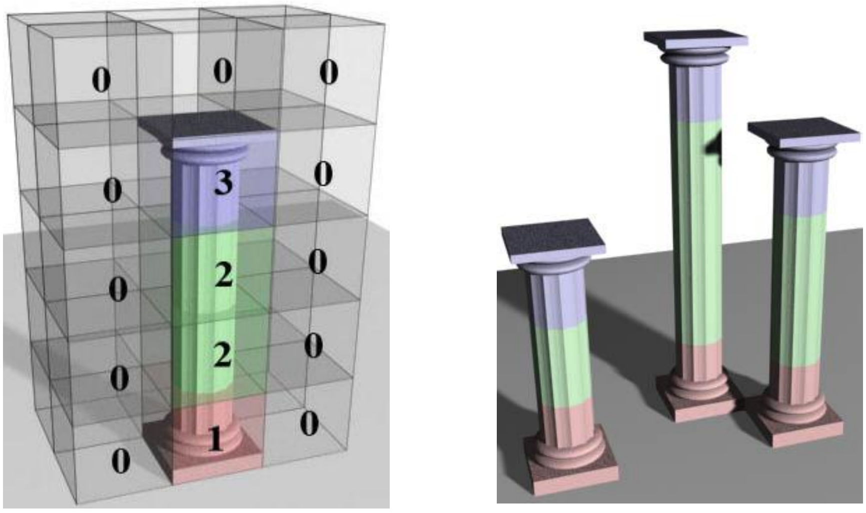
Note. This model was produced by Harrison in 2005, showing the pattern produced by the algorithm. From "Image Texture Tools", by P. F. Harrison, 2005, Clayton School of Information technology, Monash University. Copyright 2005 P. F. Harrison

2.3 Discrete Model Synthesis

(Merrell, 2009)'s thesis focuses on one main issue; Generating a new 3D model that resembles a given input model. Using discrete model synthesis, the user divides an input model into discrete building blocks(labels). The goal then is for the algorithm to generate a new model where each pair of adjacent labels in the output resembles a pair of labels in the input. This is called "the adjacency constraint". Whereby forcing this rule on the algorithm, we can create seamless models that resemble the input as shown in Figure 2.3.

Figure 2.3

Input and output models



Note. This model was produced by Merrell in 2009, showing how the user divides a given model and the output produced by the algorithm using "adjacency constraint". From "Model Synthesis", by P. C. Merrell, 2009, Chapel Hill, University of North Carolina. Copyright 2009 P. C. Merrell

2.4 Wave Function Collapse Algorithm (WFC)

WFC is a constraint solving algorithm that can generate output bitmaps locally similar to the input. (Gumin, 2016) gives two requirements for local similarity:

- "The output should contain only those NxN patterns of pixels that are present in the input." (Strong Requirement)
- "Distribution of NxN patterns in the input should be similar to the distribution of NxN patterns over a sufficiently large number of outputs. In other words, probability to meet a particular pattern in the output should be close to the density of such patterns in the input." (Weak Requirement)

Before going any further with the explanation, we need to define a few key components for the WFC algorithm:

Wave: The output grid that will contain the solved bitmap. In this implementation we can think of the wave as a chess board with all the positions on the board filled with all the chess pieces available. See Figure 2.4 for further clarification.

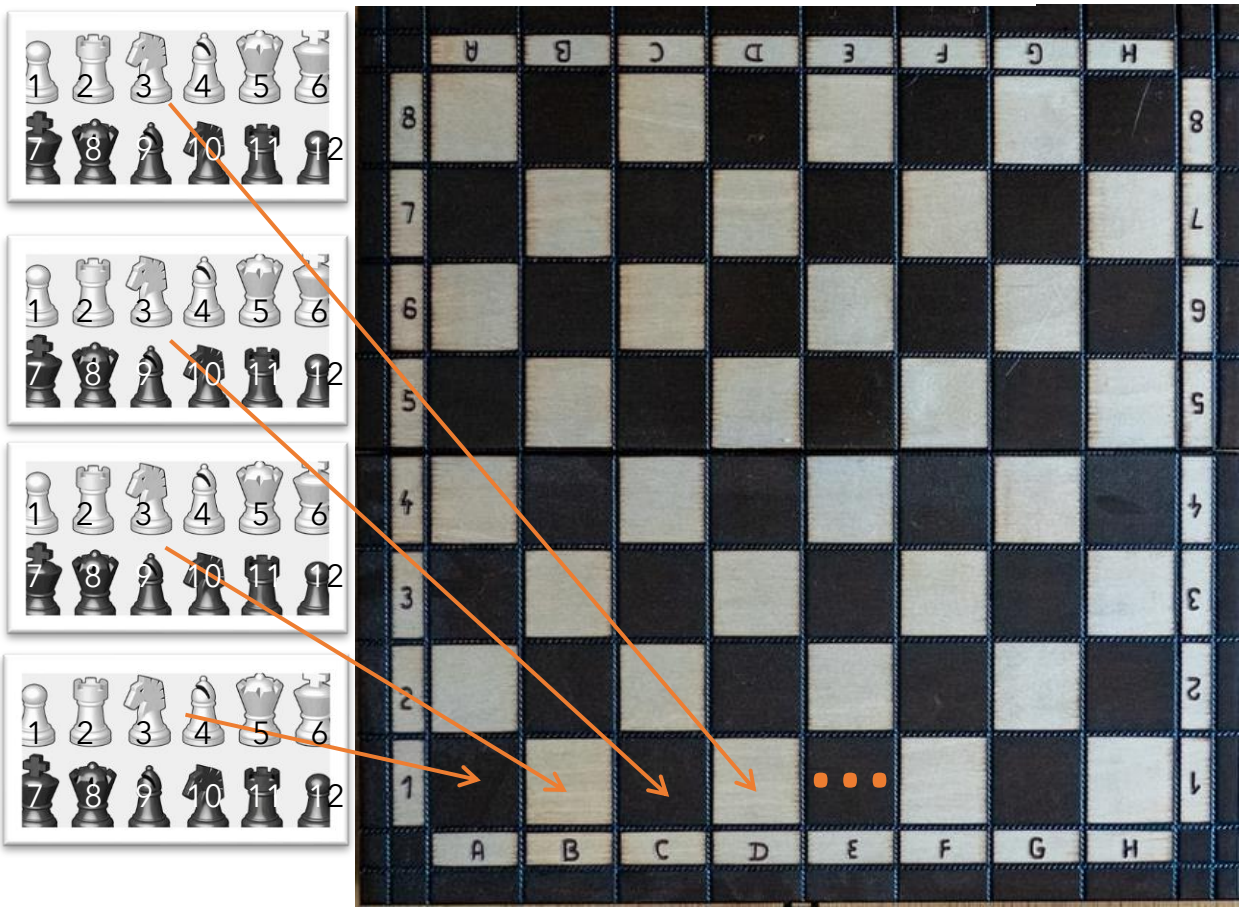
Tiles: Basic building blocks for any output/input image, they are individual images represented by discrete variables that when placed together can represent a specific image or 3D model. In our chess analogy we can think of tiles as the chess board squares.

Patterns/Neighbours: Local constraints that consists of some tiles next to each other. Later in the algorithm, these patterns are used to generate a new output image.

Edges/Adjacencies: They are the lines in between each tile and will be where the patterns lie.

Figure 2.4

Chessboard analogy



Note. This is an abstract representation of WFC initialization phase where the wave data structure(chessboard) is filled with a list of all the discrete tiles (chess pieces) at each index.

There are two main variations of the algorithm, the Simple Tiled Model, and the Overlap Model. Where the simple tiled model only considers the adjacencies between each tile, the overlap model breaks the input into NxN chunks of tiles and considers adjacencies between those NxN chunks of tiles.

We will focus on the Overlap Model as it will be the easiest to design input for, since the Simple Tiled Model requires adjacencies to be specified before runtime, whereas the Overlap Model can extract its own adjacency data. For an output image to be generated from an input image, the algorithm goes through a few modular substages as defined by (Karth et al, 2021).

2.4.1 Adjacencies

The first step of the Overlap model is to find and define adjacencies, this is where we take an input image, convert it to tiles then assign each unique tile an integer. These integers will represent our tiles later in the algorithm (we do not use the tiles themselves for performance reasons). The algorithm then takes these discrete values (integers) and forms them into NxN patterns according to the input adjacencies. This step is crucial to the Constraint solver process of the algorithm as we need clearly defined constraints in the form of a pattern data matrix that contains all valid patterns.

2.4.2 Constraint Solver

This is the core of the WFC algorithm. And where we use the existing patterns to generate a usually larger and varied set of patterns. In (Gumin, 2016)'s implementation the solver can be split into 4 main stages and are described below.

2.4.2.1 Initialization

During this phase of the algorithm, we start initializing all the data-structures of the algorithm as follows:

- *Wave (State) data structure*: A matrix (2d array) representing the current state of the solver, the first index representing each location on a grid with the second representing all valid patterns, in a later stage given the width and height this can be translated into x and y coordinates. Each index being initialized with all true values, true representing the fact that there are valid patterns available.
- *Adjacency data structure:* A matrix (3d array) representing which patterns are valid at each grid location, with the first index representing the location on the grid, the second one representing the pattern, and the third one representing each cardinal direction of that pattern.
- *Propagation Stack:* A stack that will be used during Observation and Propagation to remove patterns, contains tuples (int, int), the first representing the location on grid, and the second representing the pattern that is to be removed.
- *Entropy calculations:* We calculate the entropy of each cell location on the grid using Shannon's entropy and make decisions based on that.

2.4.2.2 Banning

This is not a phase but rather an operation that is used by both the Observation and Propagation phases of the algorithm. It takes a grid location and a pattern, then sets the Wave matrix to false at that location for that pattern. It then adds all

the 4 cardinal directions of the pattern to the propagation stack and updates the entropy calculations.

2.4.2.3 Observation

This phase is used to select a cell and decide its pattern. It does so by using a heuristic that chooses the most constrained cell, meaning that it searches for the cell with the lowest non-zero entropy. It then chooses a pattern by using weighted random sampling where the weight is derived from the frequency of the appearance of each pattern in the input. Finally, it bans all other patterns by making use of the Ban function described above.

2.4.2.4 Propagation

This step involves banning all the neighboring cells of a removed pattern, given that our propagation stack is bigger than 0, meaning that we have observed and banned a pattern already. Looping through banning and finding neighboring cells until all the patterns follow the constraints given. This step is the costliest one and should be the first one looked at if improvements are wanted to be made to the speed of the algorithm.

2.4.3 Rendering

This is the final step in the process, where the solved adjacency matrix is translated into x and y coordinates and then the result is drawn onto the screen using the tile's assigned integers.

2.4.4 Enhancing the WFC

There are many different implementations of this algorithm and as such we can take some of these different ideas and apply them to the "vanilla" version of the algorithm, below are some such features that would be useful for our use case of the algorithm.

2.4.4.1 3D WFC

This will be the main "upgrade" from the original 2D WFC to pursue and it has been implemented several times, due to the nature of this algorithm it already supports 3 dimensions meaning all we need to do is give it another dimension when it comes to the data structures and the input as shown by (Gumin, 2016)'s implementation.

2.4.4.2 Design-level Constraints

Ideally, the designer should be very involved in the generation process and thus the algorithm could have a variation where the designer can specify what kind or

type of tiles (eg 4 different types of trees) we can place in an x and y position and just how many total tiles to have in the solution.

Such an example can be seen in (Cheng et al, 2020)'s experiments where he made use of global minimum and maximum constraints. Global minimum being the minimum number of tiles to have in the solution and maximum referring to the maximum number of tiles.

3. Methodology

Developing a 3D version of WFC is not hard, you could simply take (Gumin, 2016)'s implementation and treat it like a black box where you can just build on top of the output and the core of the algorithm remains untouched. However, improving and adapting the WFC for specific solutions requires an understanding of how the algorithm operates on a low level. Therefore, a research-heavy approach must be taken with clearly defined methods of measuring improvement.

(Gumin, 2016) references several defining papers in his implementation's docs, and as such we have a way to work backwards to understand the motivations and influences behind building the algorithm. Once we understand how the "vanilla" version of the algorithm works, we can move on to research that focuses on improving or substituting some aspects of the algorithm to have a baseline of what generally works and what doesn't.

Having this baseline allows us to form some research questions and hypotheses that represent the essence of the objectives of this project:

RQ1: Can Unity be extended to allow for 3D tile placement in conjunction with a 3D WFC?

RQ2: To achieve better speed and reliability, what processes should be changed in the WFC?

RQ3: What level of controllability achieves the best qualitative results?

H1: Unity's inspector and tools are very flexible, there is an example of WFC working in conjunction with Unity's Tilemaps already, as for a 3D version of the WFC, Gumin describes the use of higher dimensions the same as dimension 2, with the drawback of performance becoming an issue.

H2: The constraint solver portion of the algorithm should be the first to be changed as there lies most of our time complexity and reliability is directly tied into this complexity as there is a hard limit on the number of loops the algorithm can make before giving up on finding a result.

H3: The best-looking outputs will come from abstract controls, such as allowing the user some control over what group of tiles may be placed in a specific area.

Giving the user too much control, specifically in 3D can overwhelm designers as each tile may have tens or even hundreds of options to choose from.

Finally, to answer these questions for certain, a series of quantitative tests will be done on the "vanilla" version of the algorithm to form a baseline of speed, reliability, and controllability. When any features are to be added or changed, we will have a way to measure any differences between the two and from there we can deduct whether whatever we did was an improvement or not.

The way we will measure these 3 features of the algorithm are as follows:

- *Speed:* The easiest feature to measure, can be done by taking the time at the start of the algorithm and at the end.
- *Reliability:* In Gumin's implementation we have no way to backtrack and as such the algorithm will sometimes reach a conflict that results in no output; Therefore, we can measure how many times this algorithm fails vs how many times it generates an output.
- *Controllability:* We will want to allow the designer to be able to control some aspects of tile placement with different levels of control ranging from 1-4. 1 being least controllable with the user just inputting training data, 4 being most controllable allowing the user to choose which tiles to collapse.

4. Requirements, Specification and Design

4.1 Requirements

The requirements in the first sprint were developed with my understanding of the algorithm then. And as a result, were not a real reflection of the actual requirements of the software solution that has been developed, the main error I had made was to write specific requirements during my first sprint, most of which have changed or don't exist anymore (eg. Req 4). The abstract requirements however have remained the same (eg. Req 2).

During the second sprint, requirements became more specific and less abstract as I gotten a better understanding of what the software needed to function. The second sprint requirements also changed along as I got along further in the software's development.

4.1.1 Sprint 1

Req 1. [3D Tilemap Editor] Shall be an extension of Unity's editor

Unity encourages the extension of its classes to fit any kind of development work on its platform and since it is such a niche idea to have 3D tilemaps we can extend

the editor to fit that idea and allow the user to edit tilemaps without having to run the application.

Req 2. [3D Tilemap Editor] Shall allow the user to place/erase prefabs

The user should be able to draw the 3D tilemaps as you would a 2D tilemap, by just choosing a prefab that they wish drawn and using a brush to draw in an area. Should a mistake happen, the user should be able to choose the eraser and delete the prefabs in the area.

Req 3. [3D Tilemap Editor] Shall allow the user to change grid settings

The user should be able to define the width and height of the area that they are going to be drawing in.

Req 4. [WFC] Shall take an xml document to extract its constraints

Due to being a constrain-solving algorithm, WFC should be able to take an xml document that represents the constraints of the output model and create a data structure that essentially tells the algorithm which pattern is allowed to reside next to another.

Req 5. [WFC] Shall have 3 main functions that allow it to run (Initiation, Observation, Propagation)

During my time dissecting (Gumin, 2016)'s algorithm, I have noticed that the core of the algorithm revolves around three main functions:

- Initialization, where all the data structures are initialized to their default values.
- Observation, where a cell is chosen to be "observed", meaning that we look inside the array that resides in that specific cell and choose one pattern while also banning that cell from being observed again.
- Propagation, where we remove any illegal patterns according to our constraints from any cells in our data structure.

Req 6. [WFC] User shall be able to define the output model's size

The user should be able to define what width and height the output will be, with the main feature being the fact that we can have a smaller input than our output.

Req 7. [WFC] Observation phase shall be tied to each cell's entropy

When choosing which cell to observe next, rather than choosing at random, we will have a higher success rate with the algorithm if we choose the cell with the least information available (i.e. we choose a cell with only 2 patterns left inside of it rather than a cell with 10).

4.1.2 Sprint 2

Req 8. [3D Tilemap Editor] Shall use the already available gameobject brush in conjunction with Unity's tile palette

Since Unity already has a brush that allows us to draw gameobjects on a tilemap we do not need to code a custom-brush.

Req 9. [3D Tilemap Editor] Shall use the already available Unity 2D tilemaps

Unity already has a 2D tilemap editor which we can extend to allow for 3D tilemap editing by stacking several tilemaps on top of each other.

Req 10. [3D Tilemap Editor] Shall allow the user to change cell size

Since the size of prefabs can vary from artist to artist, the user should be able to change the general size of all cells.

Req 11. [WFC] Shall extract its own constraints rather than use an xml document

It is much easier for the user to draw an input rather than having to write a lengthy document describing how each pattern is constrained by another and therefore this requirement replaces Req. 4 from Sprint 1.

Req 12. [WFC] Shall be able to read input tilemaps and translate its prefabs into discrete values on a 3D grid

For the algorithm to perform and to simplify its design, rather than working on the prefabs themselves we can simply take each unique prefab, assign it an integer, and then save the prefab name along with its value in a dictionary that we can access later and then take those integers and fill a grid with them in the same relative positions that their prefabs were in.

Req 13. [WFC] Shall save an nxnxn square portion of the grid (referred to as a pattern) and assign a discrete value to it

Similarly, how we have assigned an integer to each prefab and saved it into a dictionary, we can take a square group of integers from that grid, assign them into an array with the same relative positions and assign to each unique array an integer.

Req 14. [WFC] Shall be able to compare patterns for equality

We should be able to loop through the array of integers comparing each integer with each other to check for equality between patterns.

Req 15. [WFC] Shall be able to check if a pattern can continue another in any direction of the original pattern's faces

This can be achieved by checking for equality between the faces of each pattern with another.

Req 16. [WFC] Shall have a data structure for each result from requirements 11, 12, 14

Req. 11 will return a data structure that we can use within the core of our algorithm for constraint solving, while Req. 12's data structure is used as a representation of the layout of our prefabs and helps with the generation of Req. 13's data structure that generates prefabs.

Req 17. [WFC] Shall fill in an nxnxn grid with all available patterns at each position of the grid

This is the beginning state of our solver, each cell of our grid is filled with an array of all the patterns discrete values that we have extracted in Req. 13.

Req 18. [WFC] Shall choose a cell from the filled in grid based on the cell with the lowest entropy

This is a more concrete version of Req. 7. During the observation phase of the solver, we want to use the minimal entropy heuristic to limit the amount of information we must propagate throughout the grid.

Req 19. [WFC] Shall propagate the information throughout the grid

This step is crucial and is where we use the constraints extracted in Req. 11, we take whatever information was gained in the Observation step, and we propagate to all the relevant cells. Meaning that we remove any incompatible patterns from the cells of our grid.

4.2 Design

The design of the software has been based on (Gumin, 2016)'s design and (Sunny Valley Studio)'s implementation of the algorithm. The first sprint focused on (Sunny Valley Studio)'s design which was found to be too complex with over thirty abstractions and classes whereas (Gumin, 2016)'s implementation lacked first party documentation. And as stated by (Karth et al, 2021), (Gumin, 2016)'s project is not large, less than a thousand lines of C# code. However, the ideas of the algorithm are hard to interpret by just reading the code. Therefore, my design attempted to be a middle ground between (Gumin, 2016)'s simple yet hard to understand design and (Sunny Valley Studio)'s abstracted yet somewhat bloated design.

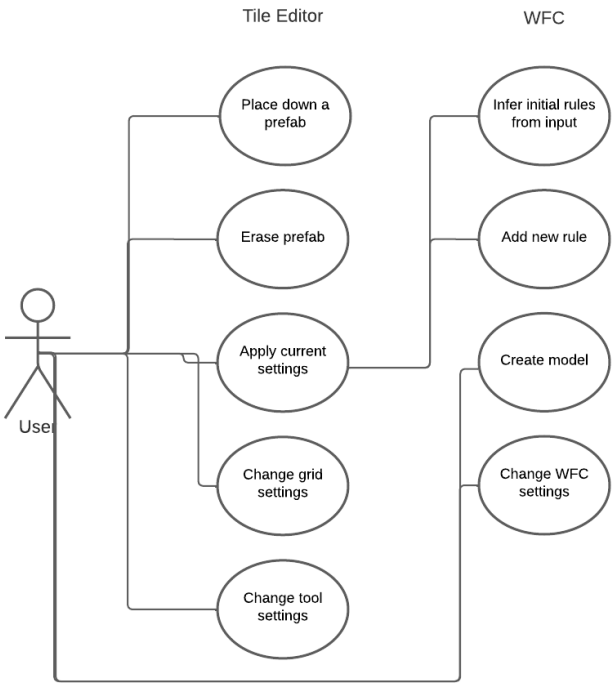
4.2.1 Sprint 1

During this sprint the main approach that was taken was a naïve one since I had still not understood the algorithm to the point of being able to design software around it. And thus, the focus of this sprint was simply getting a good idea of how the algorithm was designed and functioned to begin with along some variations of the original design. This step helped me come up with a more concrete design of the final system during the second sprint.

Design-wise, sprint 1 was used to create an abstract idea of how the software should look and behave, and how the two main components, the tile builder and the algorithm will interact. There was also some thought put into use cases with a use case diagram drawn up(Figure 4.1).

Figure 4.1

Early use case diagram of the solution



Note. The use cases were split into two logical parts and shows how the user interacted with the systems and how the systems interacted with each other at a highly abstract level.

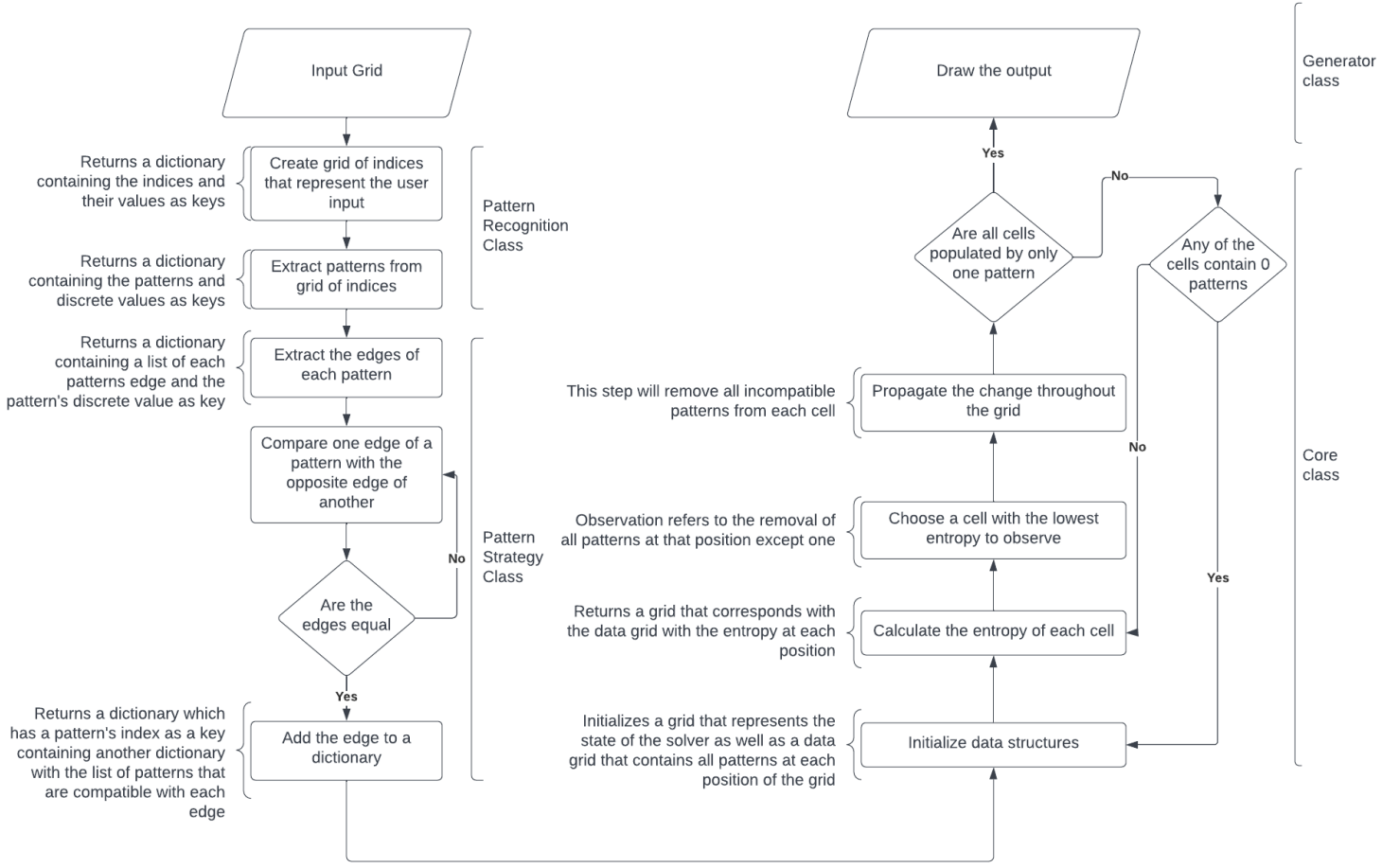
4.2.2 Sprint 2

During my second sprint I started to understand the idea behind the algorithm and how it was a pipeline of data transformations involving adjacency learning and constraint solving stages as stated by (Karth et al, 2021). Therefore, I worked on modelling the software around this idea. Each class represents a part of these stages as detailed in Figure 4.2 & Figure 4.3 below.

You can think of the algorithm to be split into two main parts, a data transformation pipeline whose main goal is to extract adjacencies between the user drawn tiles, and the core of the WFC algorithm itself whose goal is to create a model given the constraints found in the first part and an empty grid.

Figure 4.2

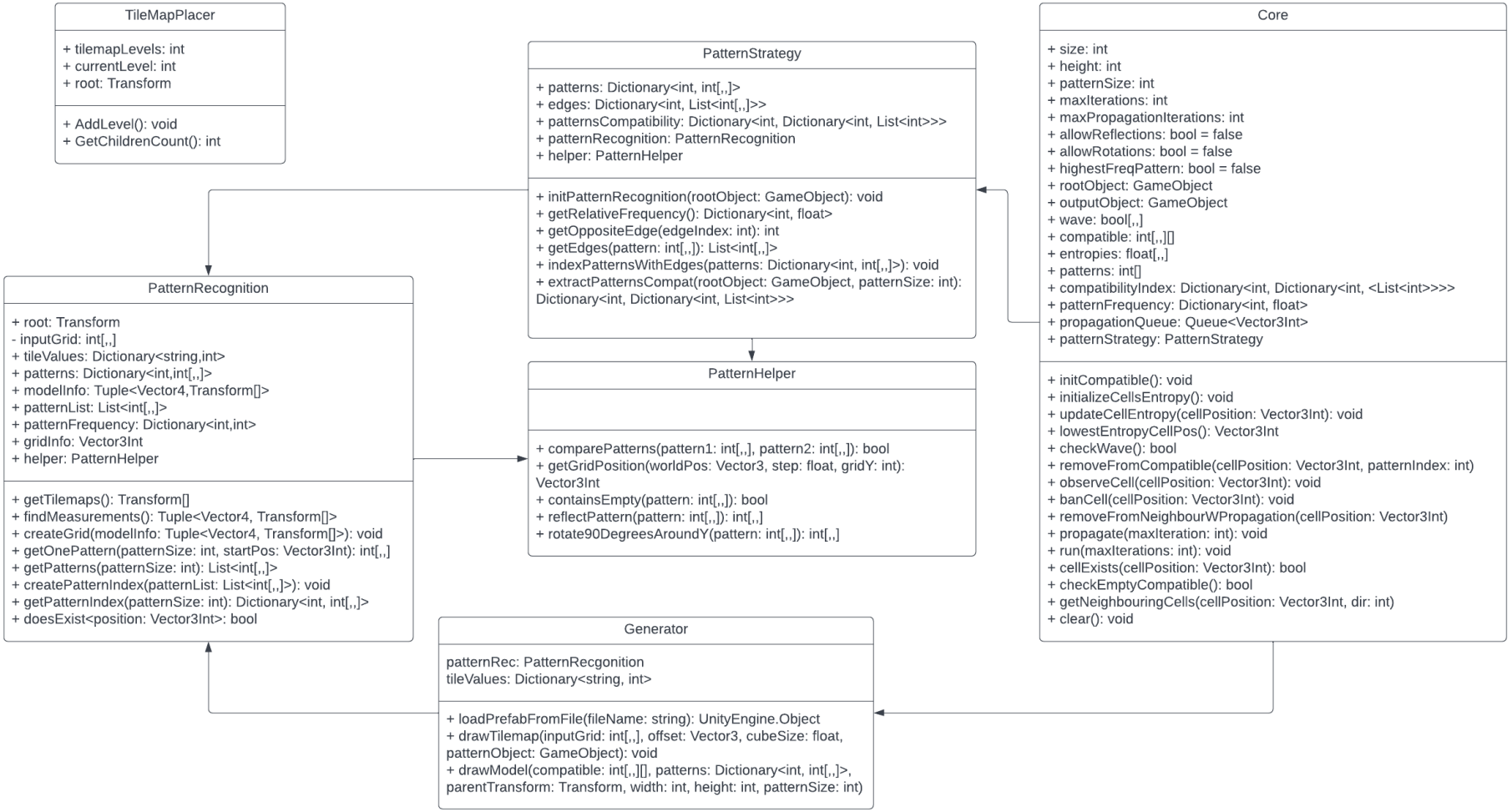
Flow diagram of the solution



Note. This flowchart represents the data transformations and data structures that are returned/initialized at each step of the program (left side), and the flow of the constraint solving WFC (right side).

Figure 4.3

Class Diagram of the Solution



Note. This is a representation of the classes in my solution and while quite simple in design the unidirectional associations are required due to the nature of having to transform the data gathered in each step of the software and the fact that the next class in the association chain only needs to know about the data from the last class.

4.2.2.1 Tile Map Placer class

This class satisfies two of our main requirements (Req. 8 & 9) by allowing the user to create a model using specific 3D tiles. It is detached from the rest of the software and only serves to help give structure to our Tilemaps.

Since Unity does not support 3D Tilemaps by default, we circumvent that by extending the functionality of the 2D Tilemaps already available.

4.2.2.2 Pattern Recognition class

To satisfy several of our requirements (Req. 11, 12 & 13) a class was designed to handle all our input and transform the data accordingly. This contains the first steps in our data transformation journey, the first one being just the reading of our data (input prefabs) and transforming them into discrete variables on a 3D grid. The next step being the gathering of patterns that are then used within the next class.

4.2.2.3 Pattern Strategy class

Within this class we have a few distinct functions that all run off the patterns that we have gathered in the last class and essentially serves us by returning a very important data structure that satisfies one of our main requirements (Req. 15). The main data structure that is created within this class is passed on to the Core class to help constrain our output to some specific parameter (i.e., pattern1 may only reside to the left of pattern2).

4.2.2.4 Pattern Helper class

The main function of this class is to be a toolkit that we can use within the last two classes to operate on our patterns. This class was added on much later to help clean up the last two classes and increase readability.

4.2.2.5 Core class

This is where our main algorithm resides, it satisfies (Req. 17, 18 & 19). It also contains all of the user-given variables that allow for control over the algorithm (i.e. maximum iterations, size of our output, pattern size and more). This class utilizes the main data structure that is returned in the pattern strategy class to constrain our output.

4.2.2.6 Generator class

This is class serves to take the solved grid from the Core class and draw a model of it in our application with a few extension functions that help with debugging, since when you are able to visualize data it is a lot easier to spot errors.

4.3 Summary

In this chapter the requirements to build a working 3D WFC collapse were discussed alongside a justification for each requirement. Next, was a discussion on the design of the software with a representation of the system by use case diagram, flow diagram and uml class diagram, followed by an explanation of the design of each class alongside justification of how each class satisfies some requirements.

5. Development and Implementation

Development of the application started in the middle of the first sprint, however quite a bit of that work was not used in the final application since a naïve approach was taken towards development, where most of the first sprint was focused on just understanding the algorithm and how it functions in detail with small scale tests done. Whereas in the second sprint actual development of the application had started using the knowledge gained in the first sprint.

5.1 Sprint 1

The first sprint was mainly focused on just attempting to build something following along tutorials that explained in depth how the algorithm functions and their own take on it. The other focus was creating a Tilemap builder that would function as a base to draw the input in.

5.1.1 Tile map builder

In this sprint, the tilemap builder was made as an extension of Unity's own tilemaps, this was done by creating a new editor window and allowing the user to programmatically add tilemaps to a specific game object called "root". The user has the option to add a single level or multiple ones in the root object with the option to change cell size.

Listing 5.1

Code responsible for the addition of a single tilemap level

```
public void AddLevel(Transform root)
{
      GameObject tileLevel = new GameObject("TilemapLv" +
      GetChildrenCount());

      tileLevel.AddComponent<Tilemap>();
      tileLevel.transform.parent = root;
      tileLevel.AddComponent<Grid>();
      tileLevel.GetComponent<Grid>().cellSwizzle =
      GridLayout.CellSwizzle.XZY;

      tileLevel.GetComponent<Grid>().cellSize = new
      Vector3(cellSize,cellSize,cellSize);

      tileLevel.transform.position = new Vector3(0, cellSize*currentLevel,
      0);
}
```

TilemapWindow.cs, Line 81

5.1.2 WFC Algorithm

In the middle of this sprint I ran into a few issues to do with serialization of classes in C# and how the algorithm was becoming more and more complex as I was getting further in the development, so in order to achieve the MVP that was promised at the end of this sprint(a 2D working algorithm) and to get a better understanding of how other people tackled this problem, I followed along a few tutorials of different implementations of the algorithm and settled on using (Sunny Valley Studio, 2019)'s implementation.

This implementation went through four defined phases:

- Reading of Input data
- Finding of patterns
- Application of the WFC algorithm
- Output of results

5.1.2.1 Reading of Input data

The first step of the implementation was to read tilemaps and extract each tile with a limit on only rectangular tilemaps with no empty spaces inside of it. Each tile is then saved in a value container that saves the position and value of the tiles. Each unique value container has a unique index. All these tiles are then translated into a grid which contains the indices of the tile value containers themselves.

5.1.2.2 Finding Patterns

Before we can start finding any patterns, the values grid has an offset added to it at each edge to avoid having patterns that can only fit in the corners or at the edges of the grid. These offsets are comprised of the opposite edges of the value grid, so an assumption is made that the input is a seamless texture. The bigger the pattern size the more cells we add to each edge for the offset.

Next up a new grid is created to represent the patterns, for a pattern size of 1, a unique index is created for each unique tile, for pattern sizes of 2+, groups of tiles are taken and then assigned a unique integer where the patterns represent a grid of values from the value grid. The patterns are then added to the pattern grid as they appear in the values grid.

The next step in the process is finding the neighbors of each pattern from the pattern grid, the way this is done differs based on the pattern size that is being used.

Finding neighbors for pattern size 1

In the case of pattern size of 1 each cardinal direction is looked at in respect to the pattern and the patterns that appear in these cardinal directions are saved in a data structure.

Finding neighbors for pattern size 2+

In this case the opposite edges of different patterns are overlapped, if they are the same, the matching patterns are saved in a data structure similarly to how patterns of size 1 are saved

5.1.2.3 Application of WFC algorithm

At the start of the algorithm, an output grid is created that stores all the available patterns in each cell. The algorithm then iterates through three main defined phases.

Cell collapse

This is where we choose a cell based on the entropy of that cell and we “collapse” it, meaning that we choose a pattern inside of that cell based on its relative frequency.

Propagation

After the collapsing of a cell, the output grid is checked for values that cannot exist next to the collapsed cell based on the constraints gathered in the last step.

Then the modified cells’ neighbors are checked for impossible and so on until we have no impossible patterns inside the cells.

5.1.2.4 Output of result

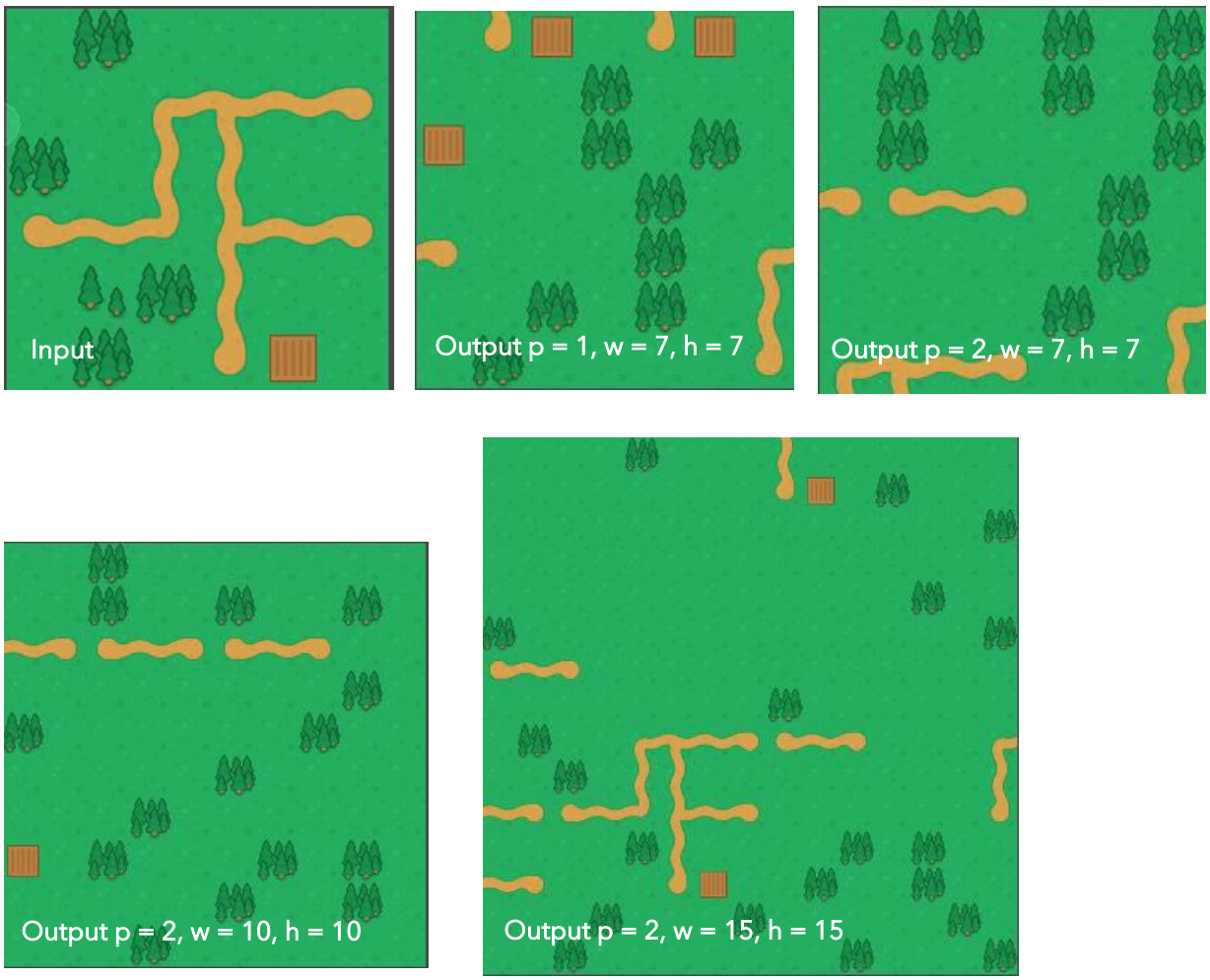
The final step is to take the output grid and translate it back into a values grid, for pattern size of 1 this is simple. Each pattern is taken and checked for its indexed value, however for pattern size of 2+ an offset needs to be set for the next pattern in the grid.

5.1.3 Results

The resulting software can create outputs similar to the input, with the main caveat that it does not work very well with a size of bigger than 15-20 tiles meaning that it is not as big as the input.

Figure 5.1

Input vs several outputs



Note. Above you can see the results of different pattern sizes with different output widths and heights, anything above a 15x15 output does not run or takes too long to run (3mins+).

5.1.4 Issues

This sprint, the main issues that cropped up during development were due to not having a full understanding of the algorithm.

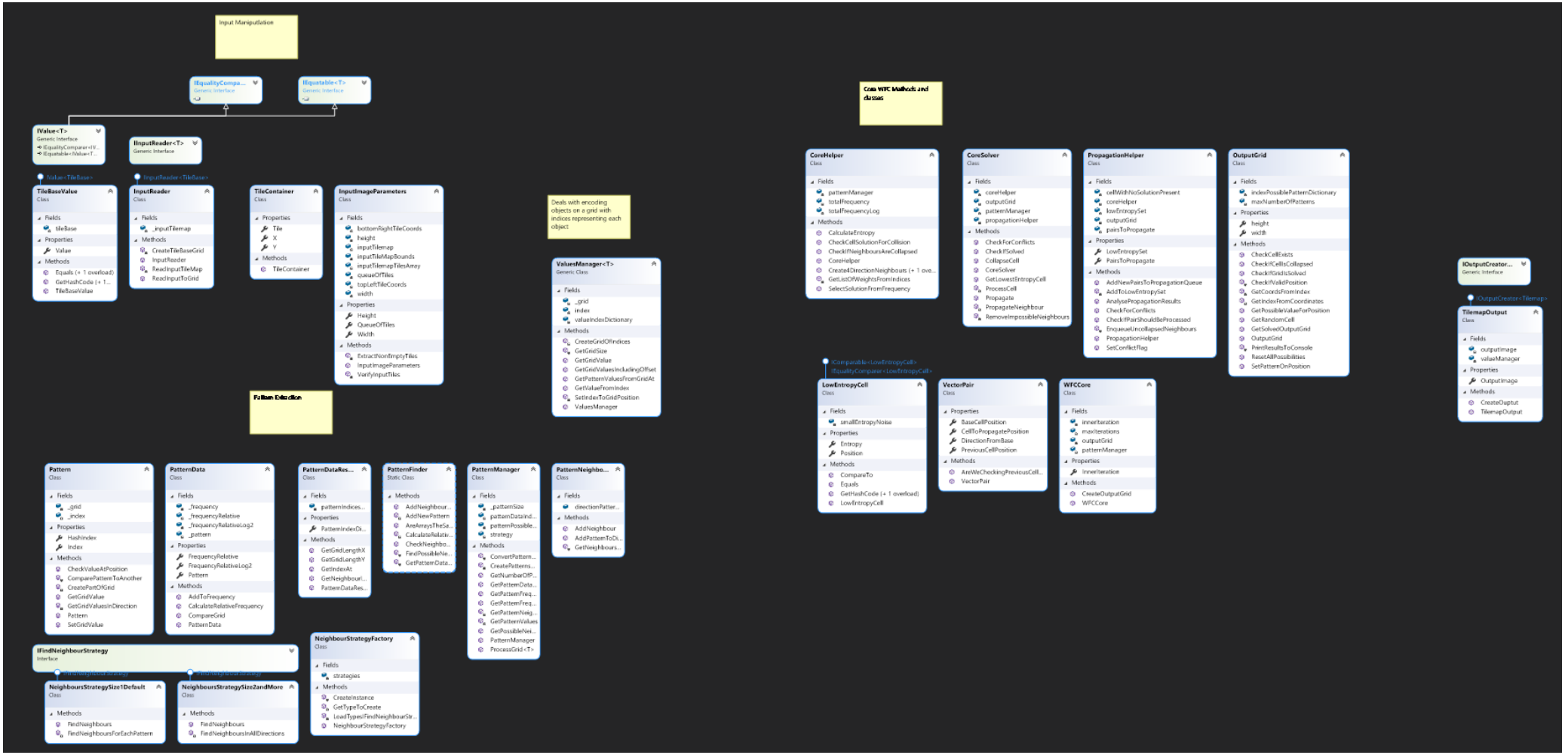
The first issue was the fact that serialization of the constraints was attempted which wasted quite a bit of time and added unneeded complexity to the project due to the need to use classes to represent every bit of information that would go into the constraints, after doing the tutorial I realized that a non-persistent data structure would be more fit for this kind of project rather than have persistent data.

The second issue had to do with the performance of the tutorial algorithm, the best results would only output around 3-4x the size of the input and that would take 1min+ to run. The plan was to expand this implementation to include a third dimension which would've added another layer of complexity to the algorithm. Given that it was already struggling to create outputs bigger than the inputs it was reasonable to assume that it would struggle to create outputs with another layer of complexity.

The final issue and reason why this implementation was scrapped had to do with the complexity of the classes. There were several layers of abstraction behind each class and piece of functionality, ideally the implementation that was needed had to be much simpler. (Figure 5.2)

Figure 5.2

Class diagram of the tutorial solution



Note. Class diagram of the solution, the ideas behind the algorithm were sound, however it became too complex for my use case.

5.2 Sprint 2

During this sprint, the implementation was split into four main deliverables based on the knowledge gained from the last sprint:

- Translation of input into a grid.
- Analysis of the translation and extraction of "patterns".
- Creation of constraints for our software using the patterns.
- WFC algorithm.

5.2.1 Input Translation & Pattern Recognition

To translate a user's input into a 3D grid we must go through a few steps beforehand. The first step being finding the size of the input grid that we are working with. The next step is to translate that size into whole numbers that we can use as the size for an array that will be our grid representation later. The final step is to take each prefab and assign it a value that we can use to fill the array with.

Calculating the size of the original input

To implement the first step, we take all the Tilemaps (levels) that we have, and iterate over each one finding the minimum x, z and maximum x, z positions for the prefabs inside those Tilemaps. To find the minimum and maximum positions I have used .NET's LINQ to simplify the code and try to keep it as concise as possible (Listing 5.2).

Listing 5.2

LINQ finding min and max position values for each Tilemap and saving them to an array.

```
if(tiles.Select(tile => tile).ToList().Count > 0)
{
      minXA[i] = tiles.Select(tile => tile.localPosition.x).Min();
      maxXA[i] = tiles.Select(tile => tile.localPosition.x).Max();
      minZA[i] = tiles.Select(tile => tile.localPosition.z).Min();
      maxZA[i] = tiles.Select(tile => tile.localPosition.z).Max();
      y += step;
}
```

PatternRecognition.cs, Line 139

We take the min and max values for each level, save them, and then compare them with the other levels to find the lowest and highest values. Finally, we then remove the max from the min and add the width of our cubes to get the width and the length of our model. For the height, the calculation is much simpler, we take the number of Tilemaps (levels) that we have and we times that by the width of our cubes.

All of this is then saved to a tuple that's formatted as a <Vector4, Tilemaps[]>. This is to be used in the next step of our process to create a grid of indices that represents our user input. This function serves a secondary purpose, which is to create a Dictionary that will function as a "decoder" for the indices that will be representing each tile.

Calculating the size of the representational grid

In order to create the 3D grid that represents our input we must now take the Tuple that has been created in the last function and create an array that matches the relative size of the original model. Eg. The original model had a size of 5.5 width and 6.5 length with a tile size of 0.5 then our grid representation will be equal 5.5/0.5 = 11 width & 6.5/0.5 = 13 length. By having the tile size saved in our Tuple we assure that we will always end up with a whole number for our array.

Finding patterns

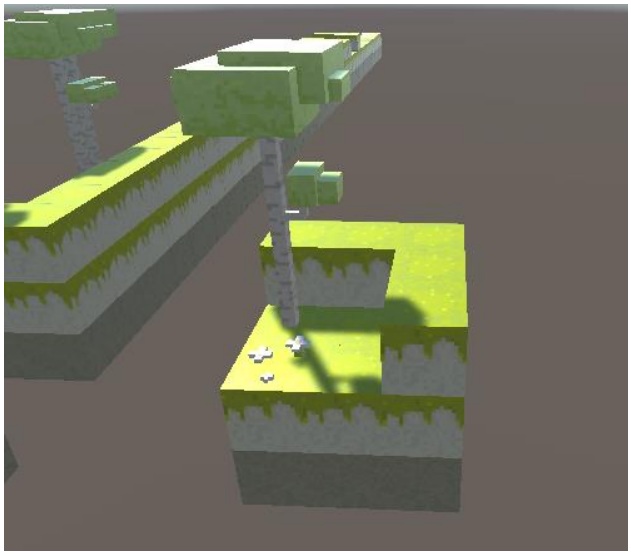
The next step is to iterate over each tile in each Tilemap, assign it an integer, save that to our decoder Dictionary and during this iteration we also find its relative position to our representational grid and finally assign that specific integer value to the correct x, y, z position of our representational grid (Listing 5.3).

To recognize patterns, the representational grid is crucial as we will be extracting all our pattern data from there. To do so we need to loop over each position on the that grid and save a user specified n x n x n portion of it, that is what we refer to as a pattern.

To save all the patterns available on the grid we make use of two main functions, formatted this way for clarity's sake. The first function will take the size of the pattern and a Vector3Int that represents our position on the main grid, it will then save each cell up to the pattern size's limit in a new array that will be our data structure holding the representation of a pattern (Figure 5.3 & Listing 5.4).

Figure 5.3

Visual representation of a pattern



Note. This is a reconstruction of a pattern extracted in the previous step of the software. Each tile in the nxnxn cube shown was represented by an integer inside an array and then drawn using the "decoder" dictionary explained above.

Listing 5.3

Taking each tile and assigning it an integer value saving it to a dictionary and positioning the integer in a 3D array representation of our input graph

```
foreach(Transform tile in tiles)
{
	Vector3 tilePosition = tile.localPosition;
	if(!tileValues.ContainsKey(tile.name))
	{
		tileValues.Add(tile.name, currentValue);
		currentValue++;
	}
	int valueToAdd = tileValues[tile.name];

	Vector3Int gridPosition = helper.getGridPosition(tilePosition,
	measurements.w, gridY);

	inputGrid[gridPosition.x, gridPosition.y, gridPosition.z] =
	valueToAdd;
}
```

PatternRecognition.cs, Line 95

Listing 5.4

Extraction of one pattern from the representational grid

```
public int[,,] getOnePattern(int patternSize, Vector3Int startPos)
{
	int[,,] pattern = new int[patternSize,patternSize,patternSize];
	int patternX = 0;
	int patternY = 0;
	int patternZ = 0;

	for (int x = startPos.x; x < startPos.x + patternSize; x++)
	{
		for(int z = startPos.z; z < startPos.z + patternSize; z++)
		{
			for(int y = startPos.y; y < startPos.y + patternSize; y++)
			{
				if(doesExist(new Vector3Int(x,y,z)))
				{
					pattern[patternX, patternY, patternZ] =
					inputGrid[x, y, z];

					patternY++;
				}
			}
			patternZ++;
			patternY = 0;
		}
		patternX++;
		patternZ = 0;
	 }
	 return pattern;
}
```

PatternRecognition.cs, Line 247

Finally, after being able to extract one single pattern, we can get to the second function which allows us to loop over the whole grid only needing to change the start position in the first function and adding all patterns extracted to a single list. In this secondary function we can also change our input data, such as reflecting and rotating the patterns to get more varied data for the output model.

Indexing patterns & finding their frequencies

The final step in the whole pattern recognition process is to index these patterns with a unique integer, similar to how we indexed each prefab and had a dictionary "decoder". During this process we also cull any patterns that might be duplicates while saving the frequency of each pattern for use later in the software.

We do so by iterating over each pattern in the list that was extracted in the last step, and we assign it a unique integer and add it to a dictionary, we also create a frequency list for each unique pattern. We then check if any future patterns that are to be added to this dictionary are duplicates, if they are then we do not add it to our dictionary and we simply just add 1 to its frequency (Listing 5.5).

Listing 5.5

Creation of our pattern index and frequency list

```
public void createPatternIndex(List<int[,,]> patternList)
{
	for (int i = 0; i < patternList.Count; i++)
	{
		if(patterns.Count == 0)
		{
			patterns.Add(i, patternList[i]);
			patternFrequency.Add(i, 1);
		}

		bool patternIndexed = false;

		foreach (KeyValuePair<int, int[,,]> currentPattern in patterns)
		{
			patternIndexed = helper.comparePatterns(patternList[i],
			currentPattern.Value);

			if (patternIndexed)
			{
			//add 1 to frequency list every time we see the same pattern
				patternFrequency[currentPattern.Key]++;
				break;
			}
		}
		if(!patternIndexed)
		{
			patterns.Add(i,patternList[i]);
		  //if pattern is not indexed add it to the frequency list with a value
of 1
			patternFrequency.Add(i, 1);
		}
	}
}
```

PatternRecognition.cs, Line 290

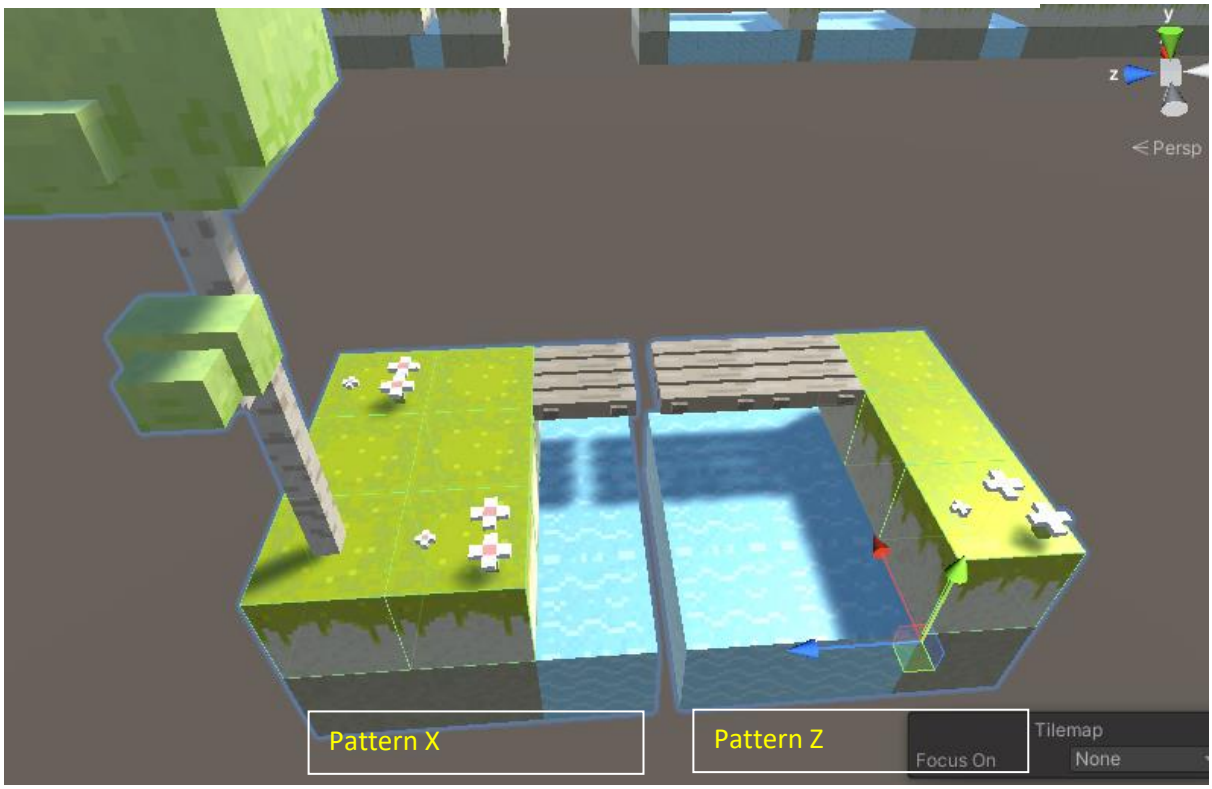
After extracting all the patterns and indexing them, we are left with two main data structures, a dictionary containing an index that points to a 3D array (pattern) [Dictionary<int, int[,,]>] and a dictionary that contains the patterns index and its frequency [Dictionary<int, int>]. The first data structure is to be used in the next part of the software for finding constraints while the next data structure will be used to find the relative frequency of each pattern.

5.2.2 Pattern Constraints

In this level of the software, we work on extracting the constraints that our "solver" will be working under. The idea behind the constraints is that if a pattern's edge can overlap another pattern's opposite edge, then they can be tiled, therefore we can say that pattern y may reside next to pattern x on the right edge and vice versa (Figure 5.4).

Figure 5.4

Two patterns that can be tiled



Note. The left right edge of pattern x is the same as the patterns z left edge, therefore they can "continue" each other and can be tiled.

Extracting pattern edges

The idea of comparing groups of patterns together based on their edges allows us to create a data structure that serves as our constraints for the next part of the software which will be the algorithm itself.

To compare each edge of each pattern, we first need to extract the edges by themselves and assign them an index that will represent which side the edge resides in (eg. 0 = front edge, 1 = back edge etc..). The extraction is done similarly to how we first extracted patterns, we take each pattern and since we know it is a cuboid shape, we know we only have precisely 6 edges to extract. We can also find out where each edge starts and ends by taking the length of the cube and applying it to the correct coordination. For example, we can extract the front edge of a pattern by iterating over the x and y coordinates while keeping the z coordinates at 0 (Listing 5.6 & Figure 5.5).

Listing 5.6

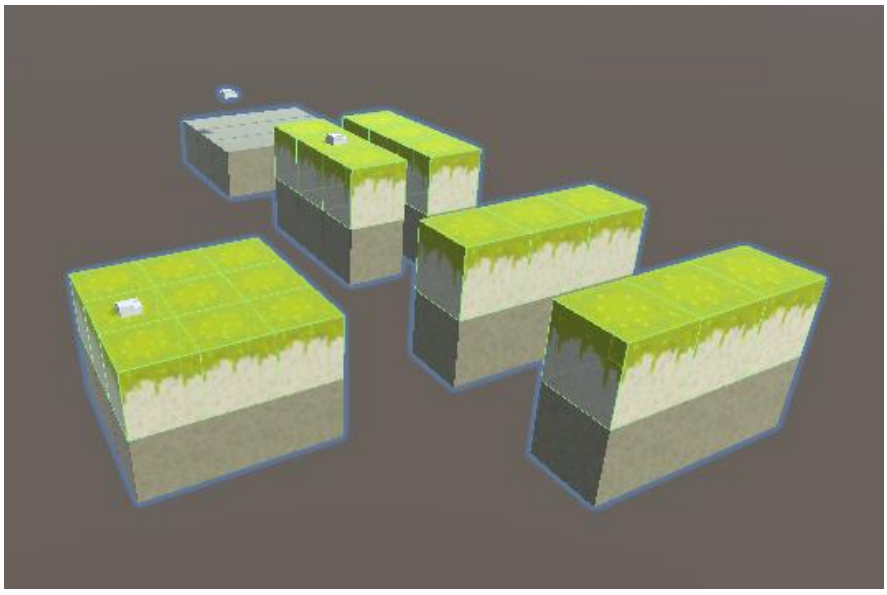
Extraction of the front edge (z = 0) of a pattern

```
int[,,] frontEdge = new int[edgeSize, edgeSize, edgeSize];

for (int y = 0; y < length; y++)
{
      for(int x = 0; x < length; x++)
      {
            frontEdge[x, y, 0] = pattern[x, y, 0];
      }
}
patternEdges.Add(frontEdge);
```

PatternStrat.cs, Line 95

Figure 5.5

A pattern(left) alongside its extracted edges(right)



Note. Each edge has been extracted in the following order: front, back, left right, above, and below so that each edge can have its own numerical index.

Comparing pattern edges

After we have extracted all edges, we can begin their comparison. This is done by iterating over all patterns, then iterating over each of its edges and comparing it to all other pattern's opposite edges, if they are equal, then they can be added to a data structure that will be our constraints (Listing 5.7).

This data structure is comprised of a Dictionary whose key is the index of our main pattern and its value is another dictionary whose key is the index of the side of the pattern which contains a list that has the index of all allowed patterns. [Dictionary<patternIndex, Dictionary<edgeIndex, List<allowedPatternIndices>>>]

This allows us to query the dictionary during the runtime of the algorithm to check whether a pattern is allowed to reside in the cell it currently is in based on neighboring patterns.

Listing 5.7

Comparison of pattern edges and creation of our constraint dictionary

```
foreach (KeyValuePair<int, List<int[,]>> patternEdges1 in edges) //first pattern that
we will compare
{
	for (int i = 0; i < patternEdges1.Value.Count; i++) // each of the pattern's
edges
	{
		foreach (KeyValuePair<int, List<int[,]>> patternEdges2 in edges) // the
	pattern edges that we compare the first one to
		{
			if (helper.comparePatterns(patternEdges1.Value[i],
			patternEdges2.Value[getOppositeEdge(i)]) && patternEdges1.Key !=
			patternEdges2.Key)
			{
			    compatiblePatterns.Add(patternEdges2.Key);
			}
		}
		compatibilityList.Add(i, compatiblePatterns);
		compatiblePatterns = new List<int>();
	}
	patternsCompatibility.Add(patternEdges1.Key, compatibilityList);
	compatibilityList = new Dictionary<int, List<int>>();
}
```

PatternStrat.cs, Line 39

5.2.3 WFC Algorithm

The solver part of the software runs by taking the constraints extracted above and applying them to a grid that contains all the patterns possible in each cell. It has three distinct phases that it goes through:

- *Initialization*

It's where we initialize our three main data structures, our output grid that holds patterns at each cell, a grid that represents the state of our output grid, and a grid that represents the current entropy of each cell.
This step can end up being done more than once in the runtime of the algorithm as the solver works without any backtracking, so if it finds an impossible situation where a cell has no patterns inside, it will simply discard whatever it has done and start again.

- *Collapsing of a cell to one single pattern randomly*
  If we aren't propagating information, then we can choose a cell with the lowest entropy from our output grid to collapse to one pattern. We use the low entropy heuristic because it gives us a few advantages over a high entropy or a random heuristic. Such an advantage is that when picking a low entropy cell to collapse, it is likely for the cell to be near other cells with just one pattern filled in. This allows for less conflicts when trying to join up different areas.
- *Propagation of information throughout the grid*
  When a cell has a pattern removed, we use the constraints that we extracted earlier to check the affected cells and remove any patterns that violate them.

Cell collapse, propagation & entropy calculations

The algorithm relies on a queue of positions to iterate through the cells, whenever any cell has had any changes made to it (patterns have been removed), the cell is added to a queue which signals that this cell's neighbors need to be checked for illegal patterns. Therefore, if we have no positions in our queue, we need to choose a cell with the lowest entropy to collapse to a single pattern.

The entropy of each cell is calculated by taking the relational frequency of each pattern found in the last step of the software and applying Shannon's entropy calculation, the result is then stored in a grid at the same location as the original cell.

Once a cell has been collapsed, we "ban" that cell from being accessed again, which achieves 2 main things during the runtime of the algorithm; it updates the state grid of the algorithm as solved, meaning that we will not need to access that cell anymore and it will start populating the propagation queue again.

To propagate and change the information throughout the grid, we deque the next position, access that position in our output grid and take all the patterns inside it. We then take each pattern from our original cell and compare it against the patterns inside the neighboring cells. If it is contained within our constraint dictionary at the same positional index as the neighboring cell, then we can add that pattern to a new list. The new list will serve as the updated information inside the neighbor cell allowing us to discard any patterns that do not satisfy our constraints. Finally, we check that the cell has had changes made to it by

comparing our old list with the new one, and if their sizes differ, we re-calculate the cell's entropy and add the neighboring cell to the propagation queue to have its neighbors checked (Listing 5.8). Finally, after the propagation step is over, we check whether we've reached our termination case; all cells only containing one pattern inside (Listing 5.9).

Listing 5.8

Removing of illegal patterns, addition to propagation queue and update of entropy

```
List<int> allowedPatterns = new List<int>();
foreach (int currentPattern in currentPatterns)
{
	foreach (int comparingPattern in neighbourPatterns)
	{
		if
	(patternsCompatibility[currentPattern][i].Contains(comparingPattern))
		{
			if (!allowedPatterns.Contains(comparingPattern))
			{
				allowedPatterns.Add(comparingPattern);
			}
		}
	}
}
if (allowedPatterns.Count == 1)
{
	banCell(neighbourCell);
}
compatible[neighbourCell.x, neighbourCell.y, neighbourCell.z] =
allowedPatterns.ToArray();

if(allowedPatterns.Count < neighbourPatterns.Length)
{
	propagationQueue.Enqueue(new Vector3Int(neighbourCell.x,
	neighbourCell.y, neighbourCell.z));

	updateCellEntropy(neighbourCell);
}
```

Core.cs, Line 424

Listing 5.9

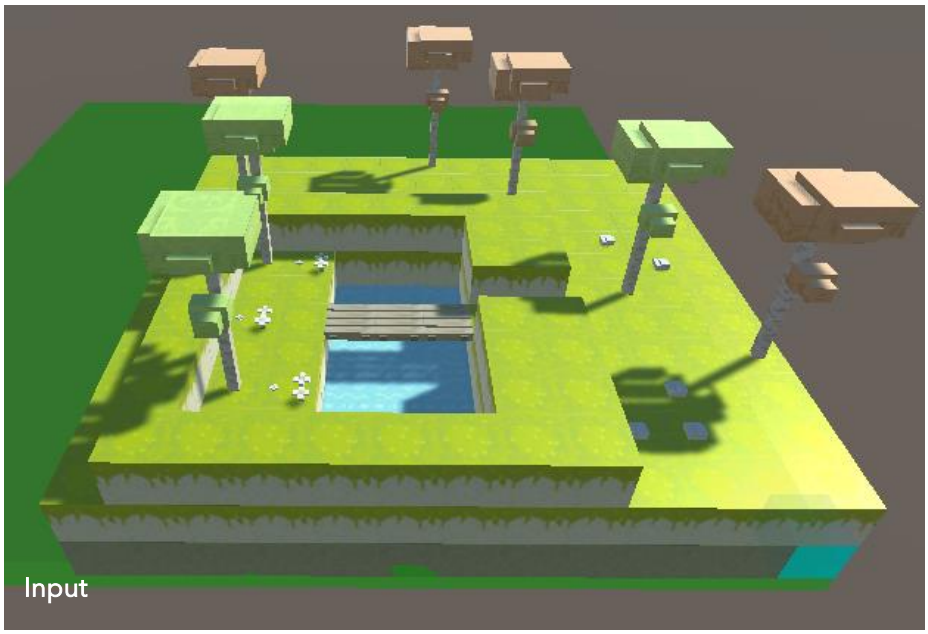
The outer loop of the algorithm that collapses cells & propagates the changes throughout the grid

```
public void run(int maxIterations)
{
      int i = 0;
      while(!checkWave() && i < maxIterations)
        {
            Vector3Int lowEntropyCellPos = lowestEntropyCellPos();
            if (lowEntropyCellPos.x == -1)
            {
                  lowEntropyCellPos = new Vector3Int(0, 0, 0);
            }
            observeCell(lowEntropyCellPos);
            propagate(maxPropagationIterations);

            if (checkEmptyCompatible())
            {
                  Clear();
            }
            i++;
      }
}
```

Core.cs, Line 303

Figure 5.6

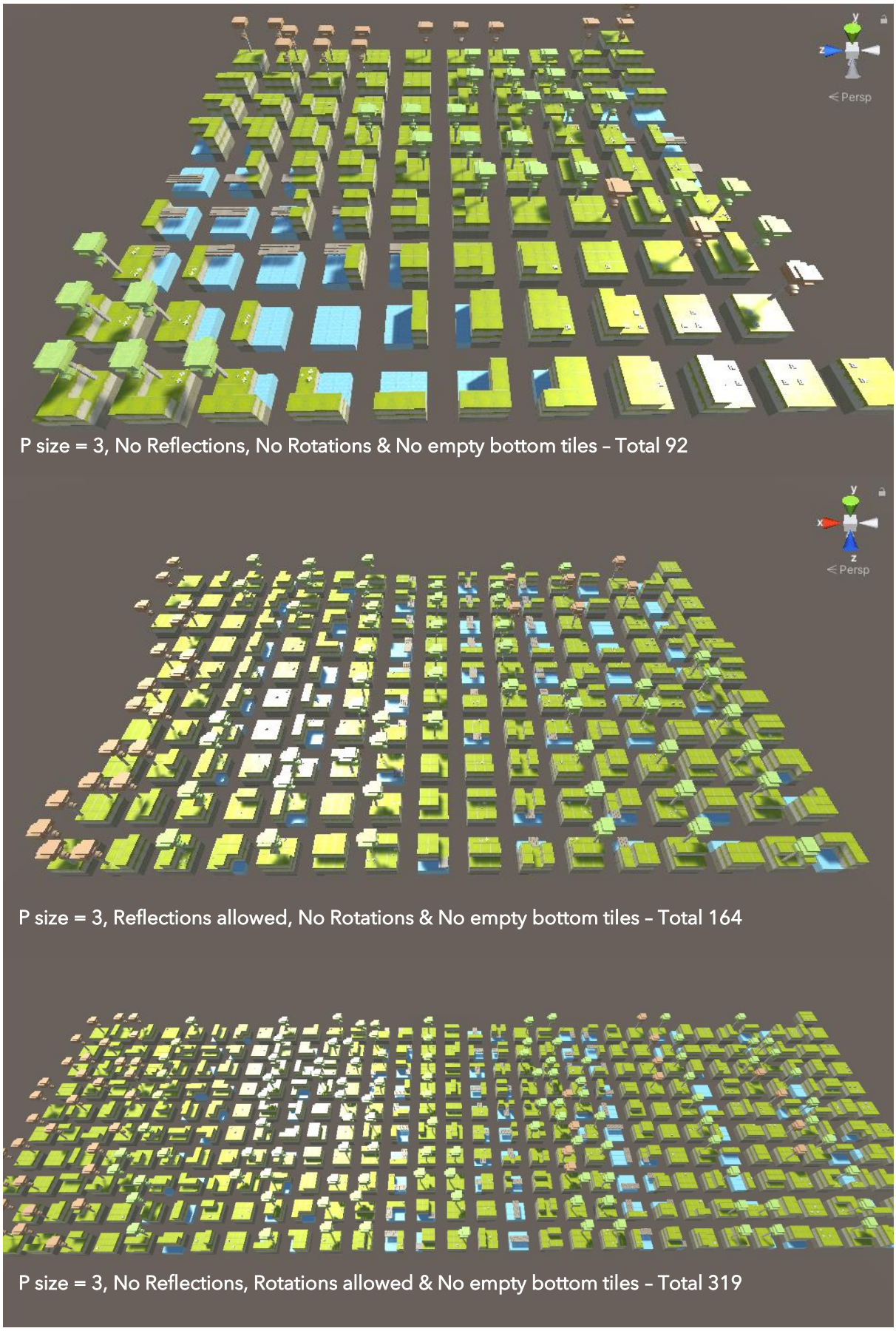
Input for the algorithm



Note. The input has a width of 13 blocks and a length of 12 blocks with a height of 3.

Figure 5.7

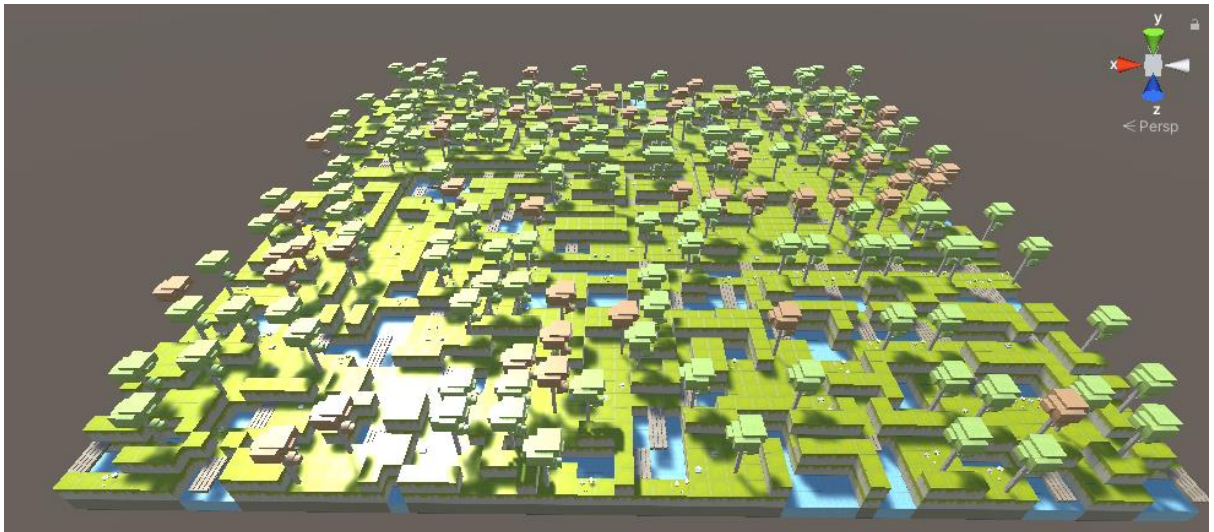
Patterns extracted by the algorithm



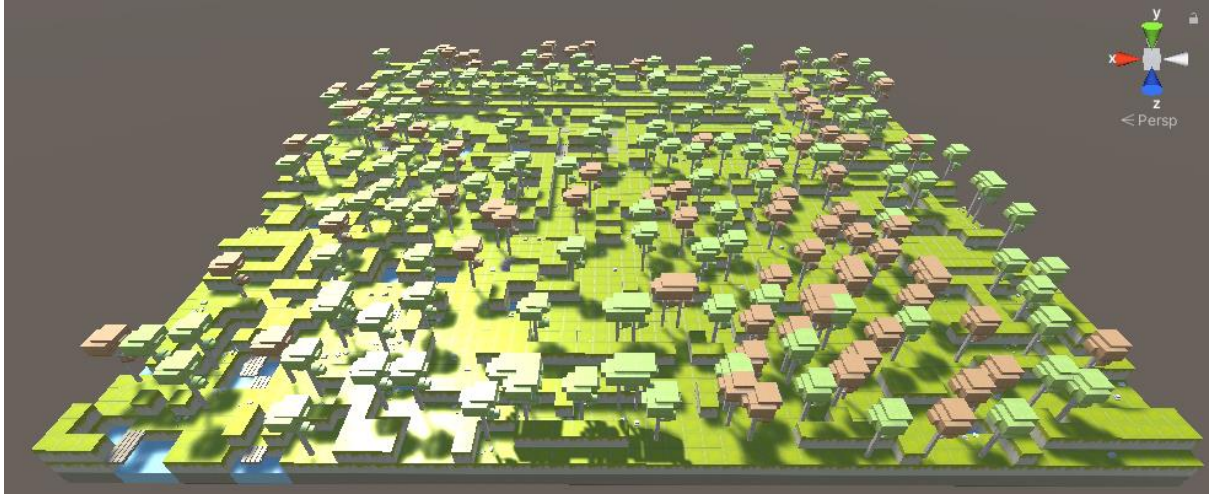
Note. This is a reconstruction of the information contained in any pattern data structure.

Figure 5.8

Outputs of the algorithm



Output size = 20 P, P size = 3, No Reflections, No Rotations & No empty bottom tiles

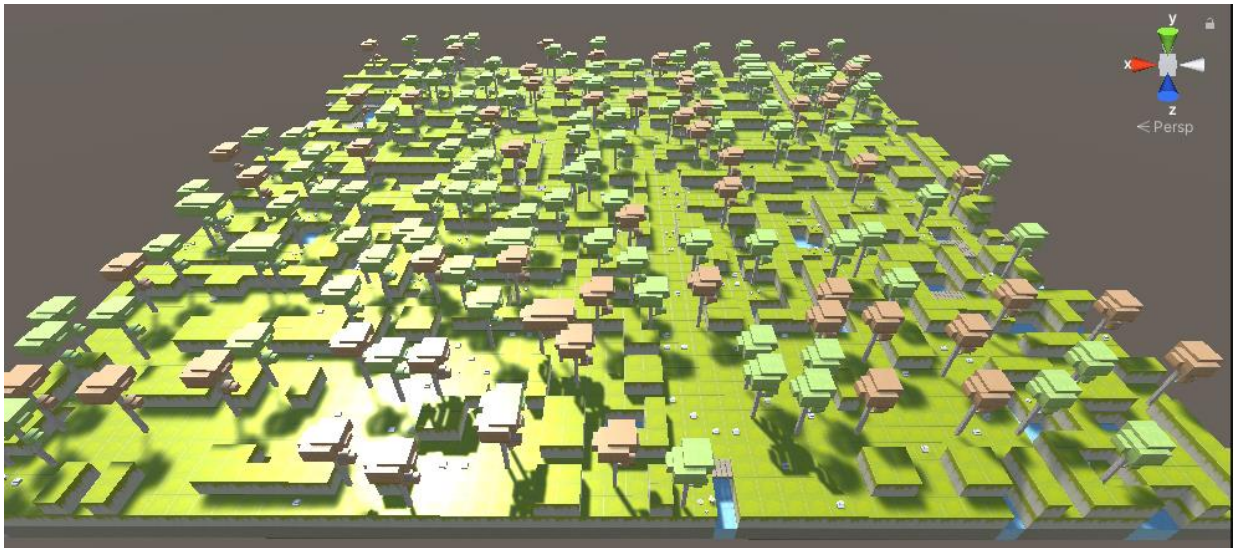


Output size = 20 P, P size = 3, Reflections allowed, No Rotations & No empty bottom tiles

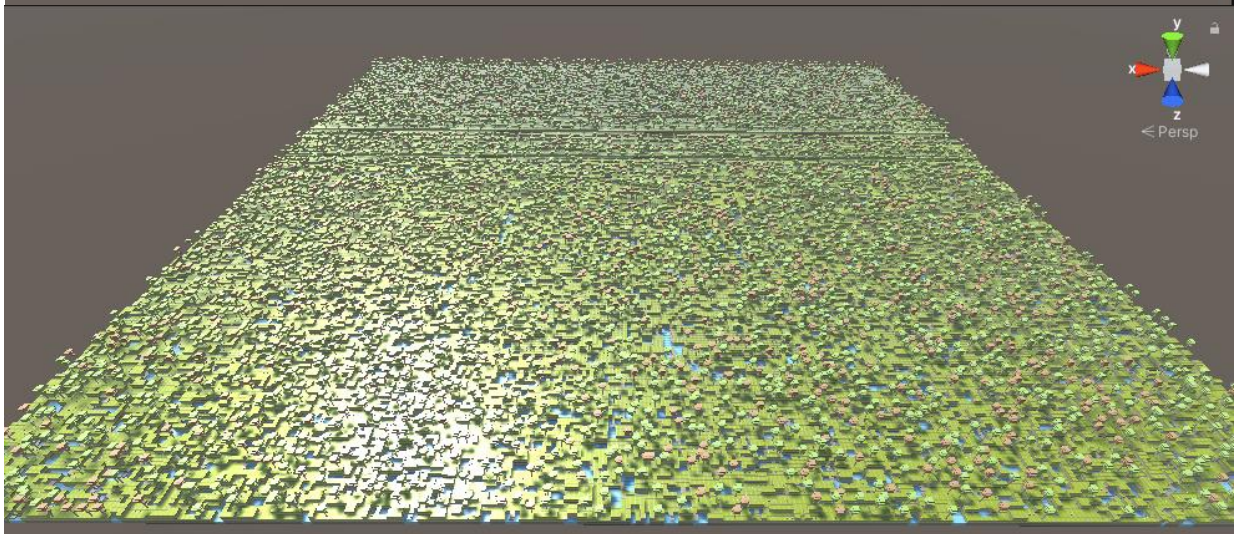
Output size = 20 P, P size = 3, No Reflections, Rotations allowed & No empty bottom tiles

Figure 5.8 – Continued

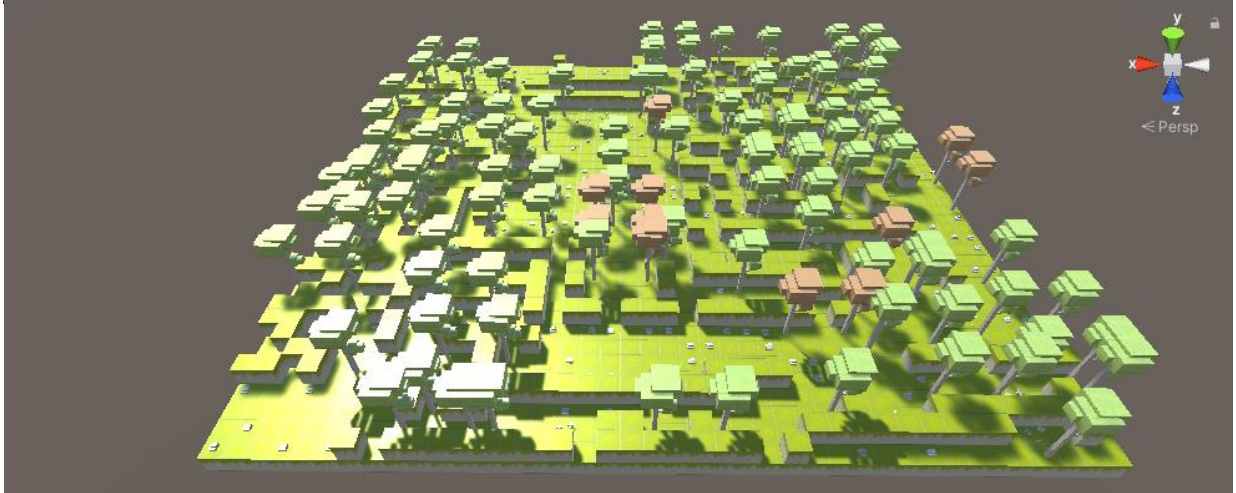
Outputs of the algorithm



Output size = 20 P, P size = 3, Reflections allowed, Rotations allowed & No empty bottom tiles



Output size = 100 P, P size = 3, Reflections allowed, No Rotations & No empty bottom tiles



Output size = 20 P, P size = 2, Reflections allowed, No Rotations & No empty bottom tiles

Note. Making changes to the availability of input dataset (patterns) changes the overall look of result.

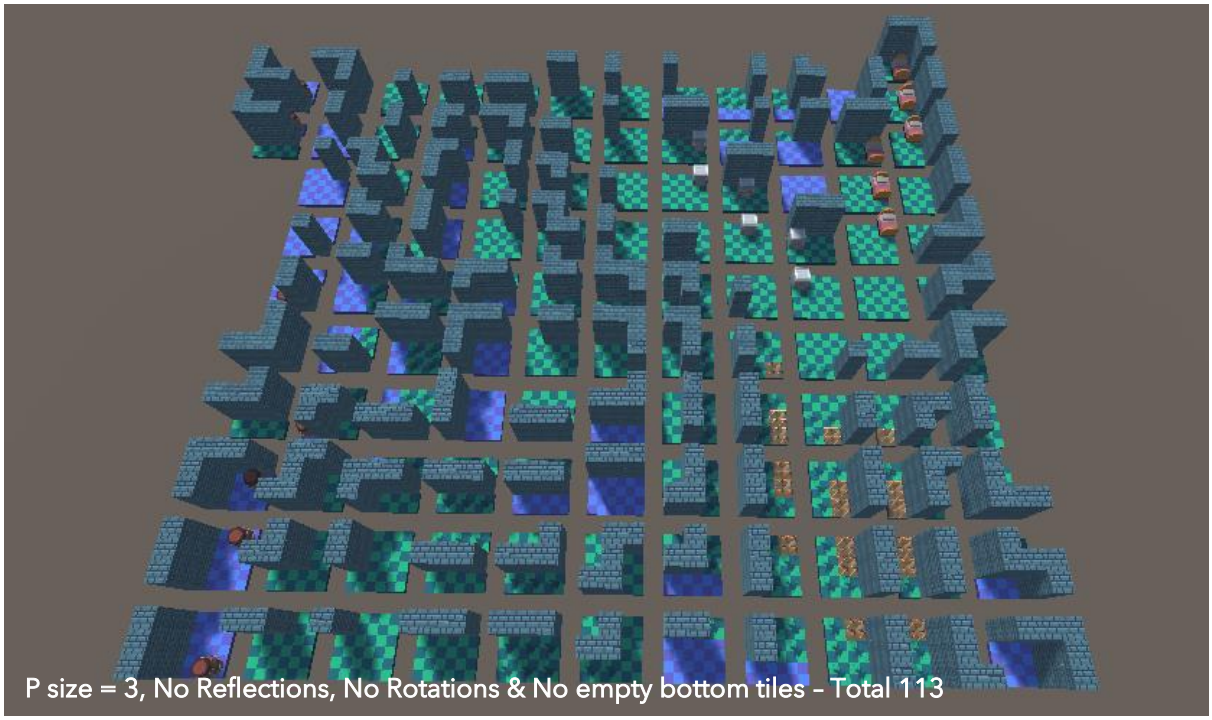
Figure 5.9

Different tileset as input for the algorithm

Note. A different input dataset for the algorithm to work out.

Figure 5.10

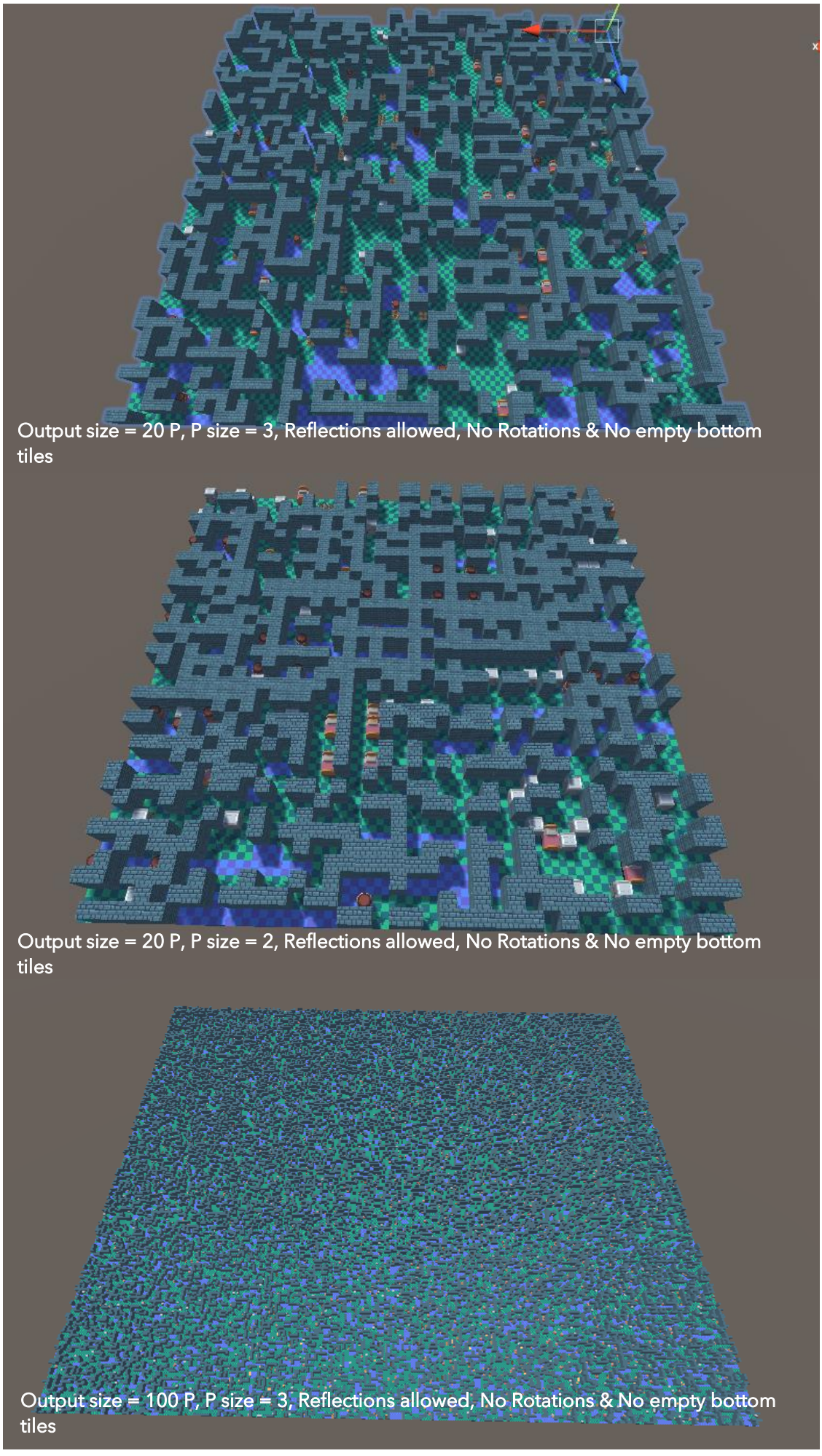
Patterns extracted



Note. A different group of patterns are extracted based on the input.

Figure 5.11

Outputs of the algorithm



Note. A group of different outputs with a different dataset.

5.2.5 Issues

Infinite propagation loop

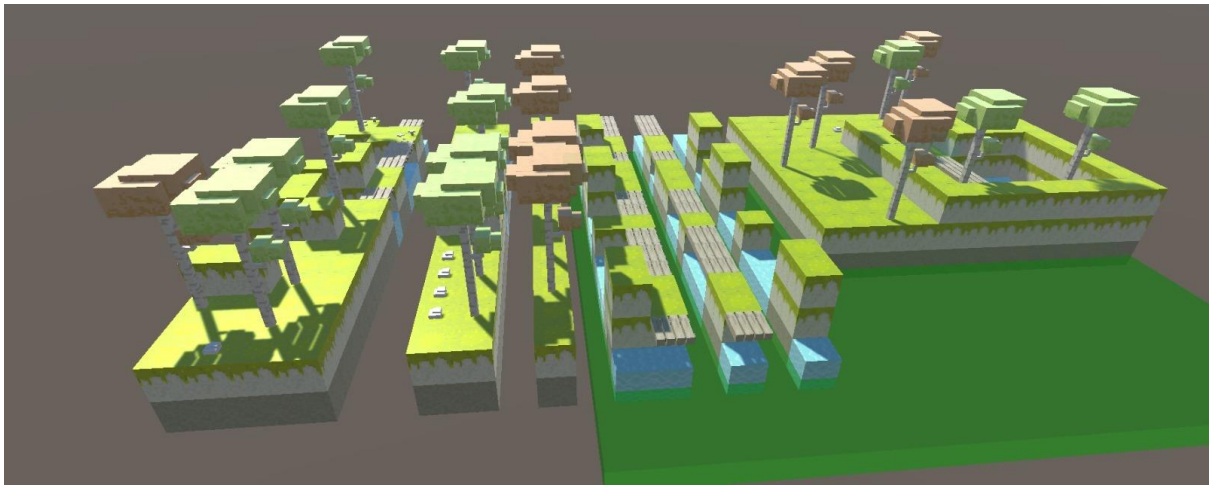
The first main issue that came up during development and caused the algorithm to fail was due to an error that was made in the propagation phase. After a cell has been collapsed (just one pattern is in that cell), the position of the cell is added to a queue. When propagating, the position is de-queued and then each neighbor of that position is checked, if an illegal pattern is found, it is removed and then the cell with the violation is added to the queue. The issue came from not checking whether a pattern has been removed from that cell. This caused the algorithm to continually add cells to the queue causing it to enter an infinite loop. This was fixed by checking if the cell's pattern size was changed during propagation and adding it to the queue only if it has changed. As an intermediary fix earlier during development the algorithm would do a full passthrough of all the cells after each cell is collapsed, this caused performance issues and would cause the output to be limited to a size of 5x5 patterns.

Losing data in release mode

During the development of the application, Unity's Debug mode was continually turned on, which would cause C# to not optimize the code written. This caused the bug to go unnoticed until late into development. When switched to release mode, the output would lose patterns on the X axis (Figure 5.12). After a few unrelated bugfixes this issue seems to have disappeared and the algorithm runs as expected under release mode.

Figure 5.12

Algorithm output under release mode



Note. The cause of this bug is still unknown.

Slow algorithm due to propagation issues

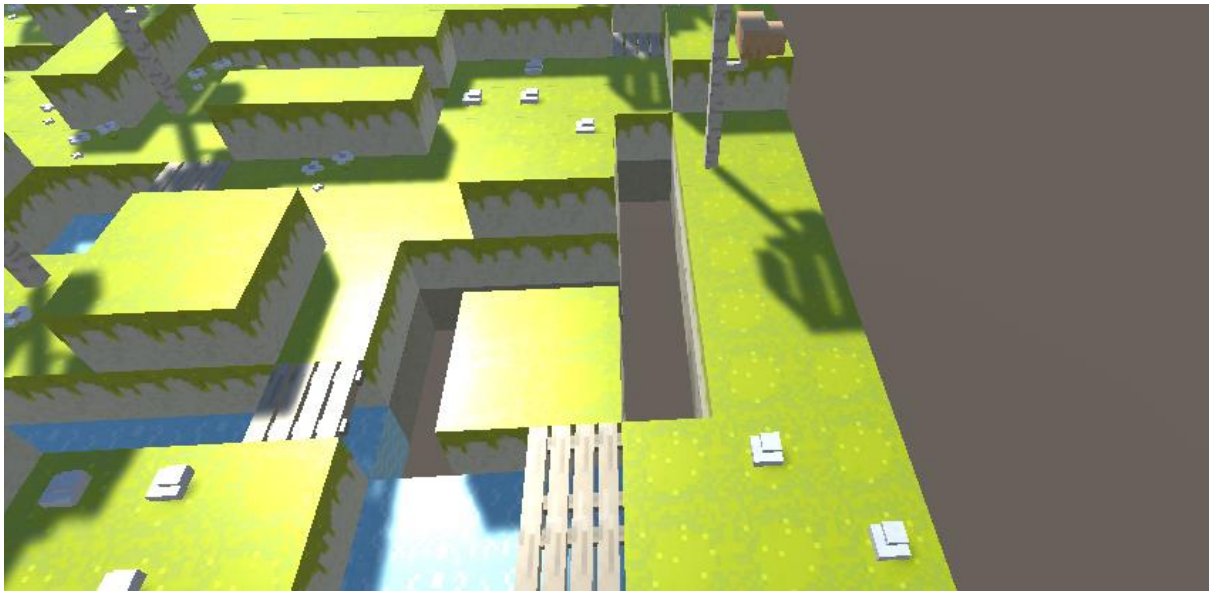
The algorithm would struggle to produce outputs bigger than 10x10 and would take 2mins+ to output, this was due to an inefficiency in the propagation step. This algorithm has no backtracking and as a result once it reaches and illegal state (0 patterns in a cell) it should terminate and start over with a new output grid. The cause of this issue was that the check for the termination case was made too late and as a result the algorithm would still try to solve the grid even though an illegal state was present in a cell. This was fixed by having a check done in the propagation phase, where we remove illegal patterns, if a cell has been checked for illegal patterns and was found to have 0 available patterns, then a termination case would be hit, and the algorithm would break out of the propagation loop and would start over. Fixing this issue gave the algorithm nearly a 1000% boost in performance, where before the max output size would be 10x10 it can now do a maximum of 100x100 output size.

Empty spaces in the lowest rows

Early in the development of the software, it was decided that the algorithm would accept patterns with empty spaces, as a side-effect, it would accept patterns with empty tiles in the bottom row causing the output to contain holes in some areas (Figure 5.13). Later in development, a check was added in the pattern extraction phase where any patterns with empty tiles in the bottom row are to be discarded thus constraining the dataset to fully filled patterns only, resolving this issue.

Figure 5.13

Output with empty spaces in the bottom row



Note. Fixing this issue also had the side-effect of boosting performance slightly as the algorithm has less patterns to go through.

Height higher than 1 does not work

An issue that hasn't been resolved is when trying to create an output higher than 1. Any output higher than 1 causes the levels to be sandwiched together creating an incohesive output. This most probably has to do with an error when comparing the bottom and top edges of patterns causing it to incorrectly add it to our constraint dictionary, and while the core of the algorithm is technically working correctly, the comparison of the top and bottom edges is not.

Figure 5.14

An output with a height of 2

Note. This is an issue to do with incorrectly comparing the edges of a pattern not the core of the algorithm.

Output grid array smaller than it should be

This issue went unnoticed until the unit test of the function, when extracting the input and translating it into a grid, we had to translate the size of the input into a relative size for the array. In some cases, depending on the size of the input, some of the axes of the grid were being translated as one size too small causing us to lose some information on the edges of the input. This was due to a cast from float to int and was fixed by using .NET's Convert.ToInt32 giving us a full representation of the input.

Using rotated patterns slows down the algorithm

Allowing the algorithm to use rotated patterns causes a 3x increase in patterns in the data set, causing it to run much slower albeit with more varied outputs. It also limits the output to a size of 20x20 due to the exponential increase in the patterns

dictionary, since for each pattern that exists, 3 more are created. This issue most likely has to do with the data structure that was used and is a limitation of the fact that a dictionary inside a dictionary was used to represent the constraints which are accessed at each iteration of the propagation phase.

5.3 Summary

This chapter tackles the implementation in detail and the different phases the application went through during its development. It is first discussed what was implemented in the first sprint to achieve the MVP and any issues regarding that.

Then there is a discussion on the implementation from the second sprint with a breakdown of how each functionality was implemented and finally any issues that arose during the development of the final application.

6.Testing and Analysis

The application has been tested on several levels of granularity, with different types of tests for each class and functionality. Due to the nature of how this application was built with a lot of functionality dependent on each other, the test plan was made to include all the classes but the types of tests themselves do not.

Testing has been done with a bottom-up approach starting with unit tests on the simplest of classes and functions for which data could be easily created and checked, followed by integration tests where functions that incorporated multiple units were tested and finally a system test was done on the main core class where the system was tested for any issues in functionality and output (Table 1).

The performance of the algorithm was then analyzed under different constraints and with 3 key aspects being looked at:

- Speed
- Reliability
- Controllability

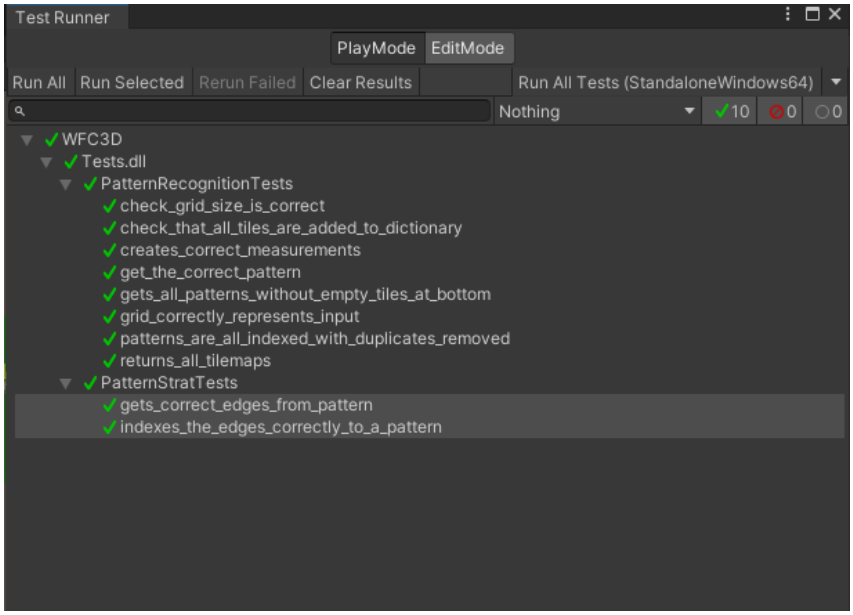
6.1 Unit tests

The unit tests were performed with .NET's NUnit testing framework in conjunction with Unity's test runner (Figure 22 & Listing 10), these tests were mainly done on the PatternRecognition class where data was being extracted or transformed from the input. Each test had an expected output that was created, which was then tested against the actual output of the function using simplified mock data.

Testing the simpler functions was crucial as any errors in the data gathering process would lead to errors in the data set itself which would be very hard to catch later in development.

Figure 6.1

Unity's test runner with all the unit tests



Note. All the unit tests created have passed.

Listing 6.1

Test function that checks if the input measurements are correctly extracted

```
public void creates_correct_measurements()
{
// will create a tuple that measures the width, height and length of the
model based
//on the minimum and maximum positions of the children inside the tilemaps

      GameObject root = makeRootObject();
      PatternRecognition patternTest = new PatternRecognition(root, true
                                                                    2);

      Vector4 expectedMeasurements = new Vector4(2.5f, 1, 3.5f, 0.5f);
      Tuple<Vector4, Transform[]> measurements =
      patternTest.findMeasurments();

      String measurementsError = "Measurements are incorrect, expected : "
      + expectedMeasurements + "Got instead : " + measurements.Item1;
      Assert.That(measurements.Item1 == expectedMeasurements,
      measurementsError);
}
```

PatternRecognitionTests.cs, Line 78

6.2 Integration tests

Integration tests were done on both the PatternRecognition class and the PatternStrategy class. The tests were done by dry running the application with a very simple input and adding breakpoints at the end of the modules that were connecting or using multiple other modules and then checking the relevant data structures for any errors.

The purpose of these tests was to see if the data was flowing as it should and if there were any defects when combining multiple modules.

6.3 System tests

System tests were done on the application with the purpose of testing both basic and any extra functionalities of the application. These were done by giving the software an input and seeing if it behaved as expected in comparison to other inputs (eg. giving the software an output size of 10 and then an output size of 100 should yield a size difference of 10x).

6.4 Performance analysis

To test the performance of the algorithm, it has been dry run 50 times under different changing constraints, some metrics have then been taken during each run and saved. Finally, these metrics have then been averaged and compared against each other.

The metrics that have been taken are as follows:

- Speed, the time it takes for the algorithm to generate an output.
- Observation and Propagation iterations, how many times the algorithm had to collapse a cell and how many times it has propagated information to other cells.

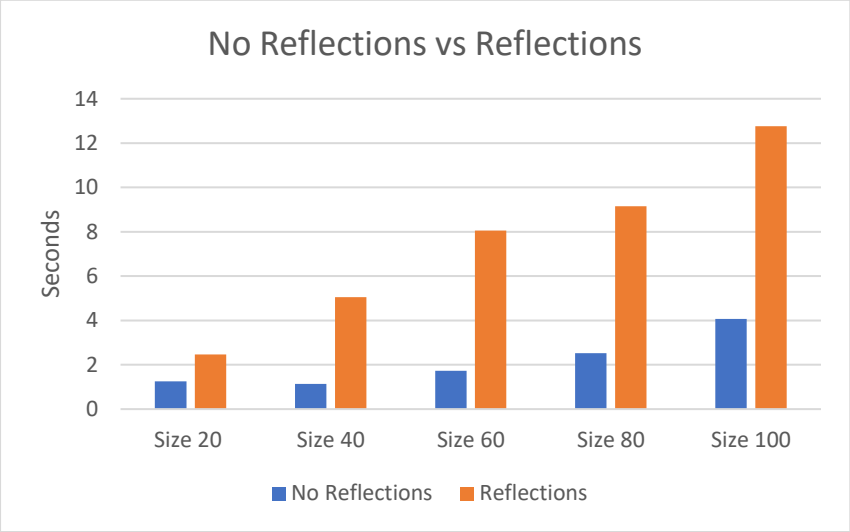
Analysis of speed – Different Sizes

In this section, it is discussed how long it took the algorithm to get to an output on average with three different main constraints changing.

- Output size
- Reflections in the patterns data set
- Rotations in the patterns data set

Figure 6.2

Time it took the algorithm to complete using no reflections vs using reflections



Note. The algorithm was run 50 times with the average speed in seconds reflected above.

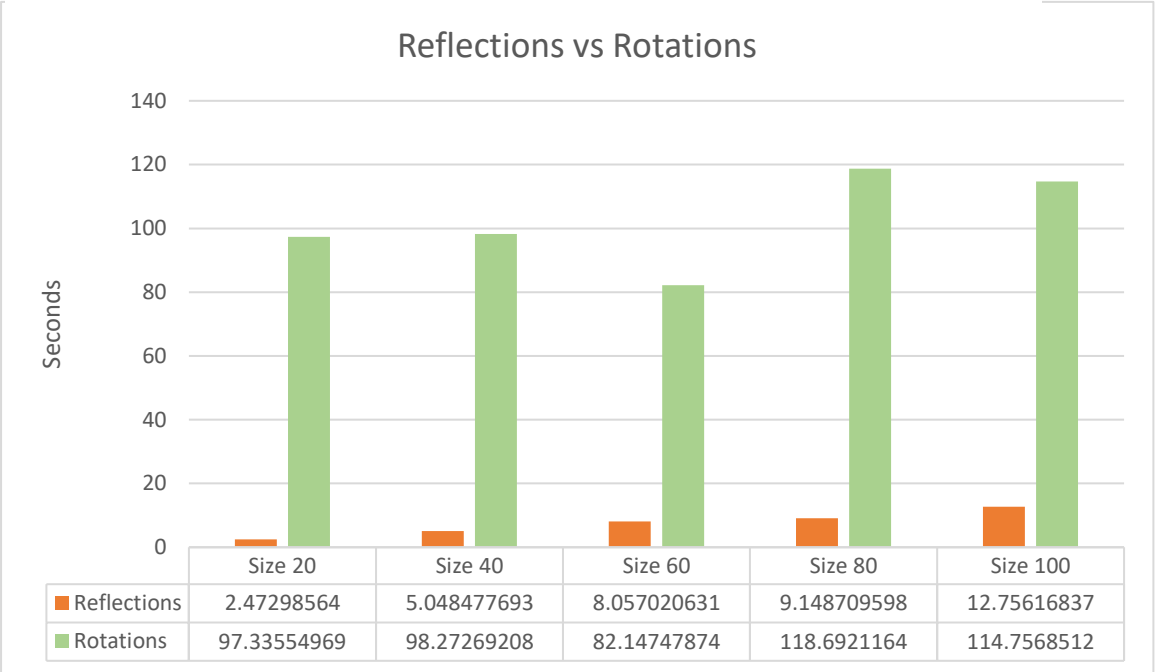
Adding reflections to the dataset causes the algorithm to jump 300% on average in time to completion (Figure 6.2), giving insight into one of the main bottlenecks of the algorithm, data set size. By increasing the size of the dataset, we increase the options available to the algorithm which in turn increases the time to completion.

This is even more apparent when adding rotations to the data set (Figure 6.3), this increases our time to completion even further by almost 4000% on average. This is due to the dataset increasing 3.5x in size.

Another trend that was observed in the outputs with rotated patterns was that output size does not seem to affect the time it takes to completion although this could be due to limited information when testing as the rotated outputs were only tested 10 times with the averages taken.

Figure 6.3

Time it took the algorithm to complete using Reflections vs using Rotations



	Size 20	Size 40	Size 60	Size 80	Size 100
Reflections	2.47298564	5.048477693	8.057020631	9.148709598	12.75616837
Rotations	97.33554969	98.27269208	82.14747874	118.6921164	114.7568512

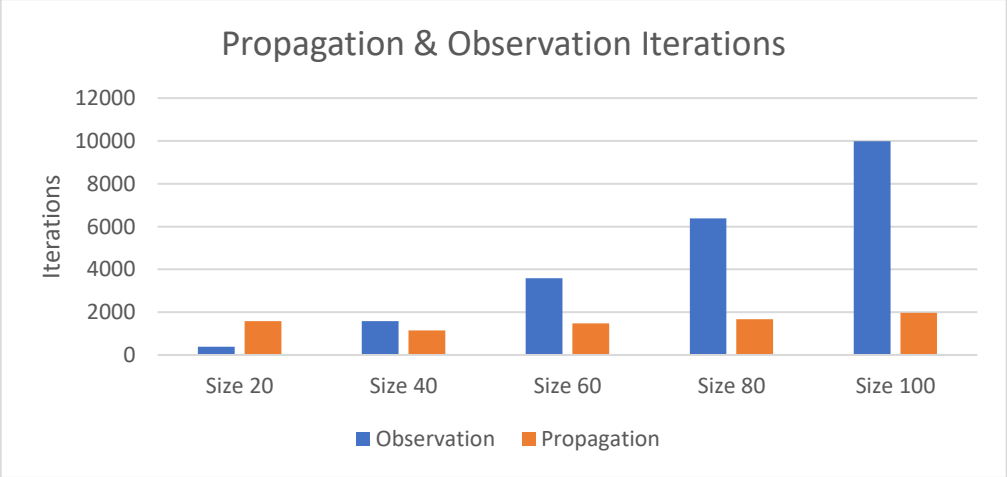
Note. The algorithm was run 10 times in the case of rotated outputs with the average speed in seconds reflected above.

Analysis of iterations

Iterations can tell us how many times the algorithm had to repeat the same time-consuming actions over its course. The observation iterations (inside which the choice for the lowest entropy cell is made) seems to be directly linked to the size of the output (Figure 6.4), meaning that the bigger the output, the more iterations & time the algorithm will take. Propagation iterations seem to be unaffected by the output size at smaller datasets such as the no rotations or reflections pattern data set. However, once we reach bigger dataset such as the rotations one, we see a very high number of iterations at smaller output sizes and then a downtrend as the output becomes larger (Figure 6.5). This could be mainly due to having a smaller grid with less options causing the algorithm to have to remove more options from each cell in the propagation iteration.

Figure 6.4

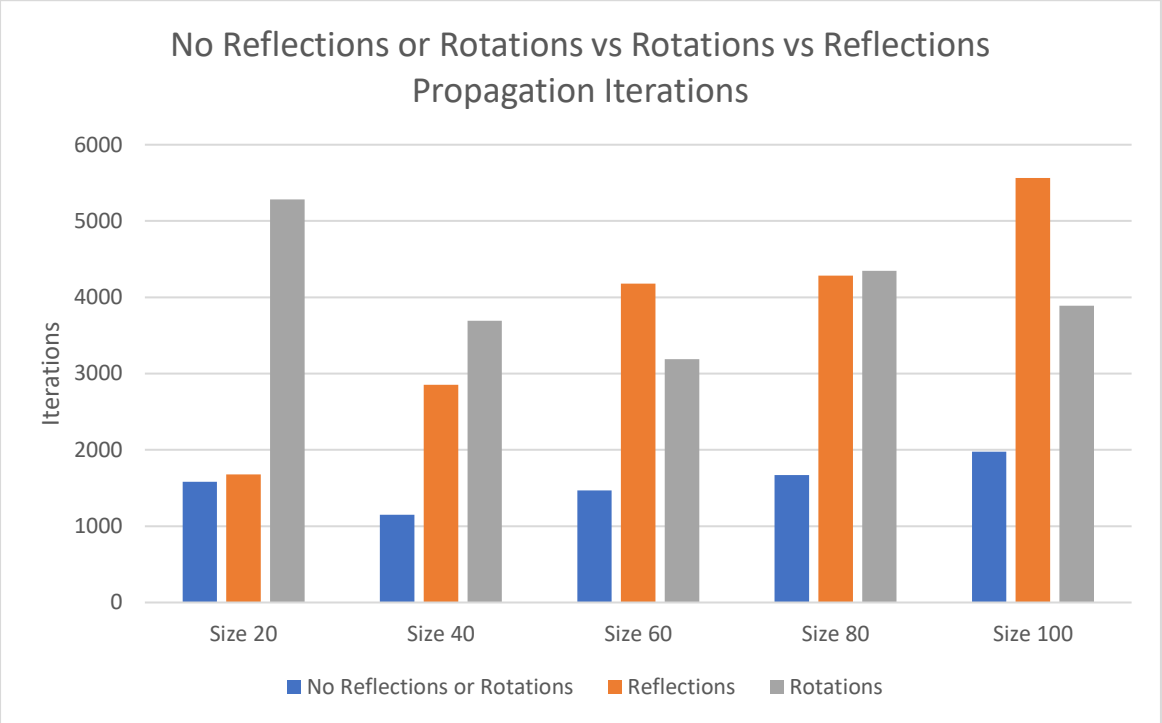
Average observation and propagation iterations it took the algorithm to reach an output



Note. The algorithm was run with no reflections and no rotations 50 times and average is reflected above.

Figure 6.5

Average propagation iterations with different constraints at different sizes



Note. The algorithm was run 50 times except for rotations where it was run 10 times with the averages reflected above.

5.2.5 Summary

This chapter deals with the testing of our software using three different methods: unit, integration, and system tests. Each phase of the testing process is explained in detail with a table representing all the test cases(Table 6.1). An analysis of the software is then made using different constraints to get an idea of the speed at which the software operates and what constraints seems to negatively affect the operation the most.

Table 6.1

Test cases

ID	Test Type	Objective	Description	Data Structure(s) tested	Dependencies	Expected Results	Actual Results
1	Unit	Check that all tilemaps are extracted from the input data.	Testing method can be found in PatternRecognitionTests.cs > returns_all_tilemaps()	Transform root (PatternRecognition.cs)	-	Two tilemaps are returned	As expected.
2	Unit	Check that the measurements of the model are correct.	Testing method can be found in PatternRecognitionTests.cs >creates_correct_measurements()	Tuple<Vector4, Transform[]> modelInfo (PatternRecognition.cs)	-	A vector4 containing the width = 2.5, height = 1, length = 3.5 and the size of the tiles = 0.5	As expected.
3	Unit	Check that the function to translate the model measurements into array measurements returns the correct values.	Testing method can be found in PatternRecognitionTests.cs >check_grid_size_is_correct()	Vector3Int gridInfo (PatternRecognition.cs)	-	A Vector3Int with the x = 5, y = 2, and z = 7	As expected.
4	Unit	Check that all the input tiles are added to a dictionary.	Testing method can be found in PatternRecognitionTests.cs >check_that_all_tiles_are_added_ to_dictionary()	Dictionary<String, int> tileValues (PatternRecognition.cs)	-	A dictionary containing 7 unique strings and values from 1-7	As expected.

ID	Test Type	Objective	Description	Data Structure(s) tested	Dependencies	Expected Results	Actual Results
5	Unit	Check that the pattern is extracted correctly with no missing data.	Testing method can be found in PatternRecognitionTests.cs >get_the_correct_pattern()	int[,,] pattern (PatternRecognition.cs)	-	A pattern containing the int 2 at position (0,0,0)	As expected.
6	Unit	Check that the patterns that are extracted contain no empty tiles at their bottom row.	Testing method can be found in PatternRecognitionTests.cs >gets_all_patterns_without_empty_tiles_at_bottom()	List<int[,,]> pList (PatternRecognition.cs)	-	A pattern containing the int 3,4,5,6 as their bottom tiles	As expected.
7	Unit	Check that the edges are extracted correctly from a pattern.	Testing method can be found in PatternStratTests.cs >gets_correct_edges_from_patterns()	List<int[,,]> patternEdges (PatternStrat.cs)	-	All 6 edges are extracted with the left, bottom, and front edge containing int 1 at position (0,0,0)	As expected.
8	Integration	Verify that all unique tiles are indexed with no duplicate tiles starting from index 1.	After the execution of the createGrid(), a breakpoint is added. (PatternRecognition.cs)	Dictionary<String, int> tileValues (PatternRecognition.cs)	Transform root getTilemaps() findMeasurements()	Dictionary containing 3 Strings which are the name of the models used in the parody data as Keys and 3 values which represent the ID's	As expected.

ID	Test Type	Objective	Description	Data Structure(s) tested	Dependencies	Expected Results	Actual Results
9	Integration	Verify that the grid is correctly created and is a relative representation of the input data.	After the execution of the createGrid(), a breakpoint is added. (PatternRecognition.cs)	int[,,] inputGrid (PatternRecognition.cs)	Transform root getTilemaps() findMeasurements()	Input grid contains tile 1 at positions (0,0,0), (0,0,1), (1,0,0), (1,0,1) & contains tile 2 at positions (2,0,0), (2,0,1), (3,0,0), (3,0,1)	As expected.
10	Integration	Verify that all the patterns are being extracted and that there are no duplicates.	After the execution of createPatternIndex(), a breakpoint is added. (PatternRecognition.cs)	Dictionary<int, int[,,]> patterns (PatternRecognition.cs)	Transform root getTilemaps() findMeasurements() createGrid() getOnePattern() getPatterns()	The data structure contains 3 patterns from keyed from 0-2, each containing a unique 3D array of integers.	As expected.
11	Integration	Verify that all patterns are indexed with their edges as values.	After the execution of indexPatternsWithEdges(), a breakpoint is added. (PatternStrat.cs)	Dictionary<int, List<int[,,]>> edges (PatternStrat.cs)	PatternRecognition class > Dictionary<int, int[,,]> patterns getEdges()	Dictionary contains three entries, each with a list of 6 edges for each face of the pattern.	As expected.

ID	Test Type	Objective	Description	Data Structure(s) tested	Dependencies	Expected Results	Actual Results
12	Integration	Verify that the compatibility index contains patterns that can overlap each other	After the execution of extractPatternsCompat(), a breakpoint is added. (PatternStrat.cs)	Dictionary<int, Dictionary<int, List<int>>> compatiblePatterns (PatternStrat.cs)	PatternRecognition class > Dictionary<int, int[,,]> patterns indexPatternsWithEdges()	Each entry in the dictionary will contain 6 other dictionaries (one for each face of the pattern) that will have a list of patterns that are compatible with that face.	With the test dataset we have 6 dictionaries for each entry but no data in the lists inside. With real data this problem does not occur.
13	System	Verify that the algorithm can create a model given input data.	In this case the input data is a a tilemap with 3 levels with each level containing some tiles. The input is a full rectangle, meaning that there are no empty spaces at the lowest level with empty spaces on higher levels.	-	-	A model is created next to our input data from a dataset of 92 patterns.	As expected.
14	System	Verify that the algorithm can create a model 10x larger than the input data.	After it was made, the algorithm would struggle with the creation of larger models, some bugfixes enabled it to perform much better. The size has been set to 40 patterns per width and length with a pattern size of 3. There are no reflections or rotations in the pattern dataset.	-	-	A model roughly 10x the size of the input is created next to our input model.	As expected.

ID	Test Type	Objective	Description	Data Structure(s) tested	Dependencies	Expected Results	Actual Results
15	System	Verify that the algorithm can create a model if allowed to use reflected patterns.	In this case the dataset given to the algorithm is much larger, and as such should give us a model that makes use of patterns that are reflected as well as normal patterns.	-	-	A model is created from a dataset of 164 patterns.	As expected.
16	System	Verify that the algorithm can create a model if allowed to use rotated patterns.	In this case the dataset should be a bit under 3x larger than without rotated patterns. A smaller output size(10) has been chosen since it takes exponentially longer for the algorithm to run.	-	-	A mode is created from a dataset of 319 patterns.	As expected but it takes the algorithm nearly 80x as longer to run.
17	System	Verify that the algorithm can create a model if allowed to use reflected and rotated patterns.	The dataset will consist of the input, reflected and rotated data and as such an output size of 10 has been chosen.	-	-	A model is created from a dataset of 348 patterns.	As expected.
18	System	Verify that the algorithm can create a model given a pattern size smaller than the original input data height.	The algorithm will be tested with a pattern size of 2 and then a pattern size of 1 with an output size of 20.	-	-	Pattern Size 2: A model is created with some cohesiveness. Pattern Size 1: A model is created with no cohesiveness.	As expected.

19	System	Verify that the algorithm can create a model given a height bigger than 1.	The algorithm is tested with a height of 2 and then a height of 3 with an output size of 20.	-	-	A model is created that has a bigger height than the original input.	Not working, the algorithm cannot create models with a height spanning more than one pattern. The result for a height of 2 is 2 cohesive models sandwiched together.

7.Critical Evaluation

7.1 Review of the Objectives

7.1.1 Objective 1 & Objective 2: Research WFC and best way to extend the Unity Inspector

I would consider the research phase of the project a success as I have managed to understand the algorithm and how it works well enough to have my own implementation of it. It was also found that the best way to extend the inspector in Unity for my use case was to extend the Tilemaps class for additional 3D functionality, although there were quite a few more options that would have gone more in depth when it comes to the extension, the simplest solution was chosen due to the time constraint given in this project.

7.1.2 Objective 3: Extend the Unity Inspector to create a basic 3D tilemap editor

This objective was achieved by having a Tile editor class that would extend the already available 2D Tilemaps class in Unity for 3D design. The tiles themselves are drawn with yet another already available component, the GameObject brush.

7.1.3 Objective 4: Build a preliminary world using the WFC algorithm as proof of concept

The first preliminary model that was built was in 2D, and it served as proof of concept, the algorithm that was used was (Sunny Valley, 2019)'s implementation of it. The performance was not fully there, and as a result the output models of that version of the algorithm would be constrained to 2x-3x the size of the original input.

7.1.4 Objective 5: Fix any issues within the WFC algorithm and the tilemap editor

This objective should have been ranked much lower and more towards the end of development as issues kept reoccurring, specifically with the re-designed algorithm that was done in sprint 2. The objective has not been fully achieved due to some issues with the final algorithm (specifically when trying to create a model of height 2+).

7.1.5 Objective 6: Marry the tilemap editor and the WFC algorithm

This objective was achieved by bringing the algorithm and the editor together in the same project, logically however, they are not coupled in any way. The algorithm depends on the tilemaps created by the editor to directly read them; it

only needs the tilemaps and as a result is not dependent on any of the editor's classes.

7.1.6 Objective 7: Create a 3D model using the editor and the algorithm

This objective has been achieved using 2 different tile sets. Due to time constraints further tilesets have not been found/made. The software can take an input tilemap and then create a procedurally generated model that loosely resembles the input. There are different levels of controls over the input data set that can change the look of the output with different rectangular sizes available for the output.

7.1.7 Objective 8: Evaluate the system

An evaluation of the system has been done in the form of tests (unit, integration, and system) with a performance analysis also done on the algorithm. This can tell us how fast the algorithm can run while allowing for any changes that are made in the future to be re-tested using the test cases.

7.1.8 Optional Objective 1: Extend the Wave Function Collapse from simple model to overlapping model

This objective has been reached quite early in the beginning of sprint 2 with the PatternStrategy class. The original plan was to create the software to use the simple model strategy which would've looked at which patterns reside next to which and used that to create a constraint dictionary, however during sprint 2 it was found that the dataset needed more variation for the output to be more diverse and thus the overlapping model strategy was implemented instead where we just compare which patterns edges can be overlapped with which and add that to our constraint dictionary.

7.1.9 Optional Objective 2: Extend the tile editor with more functionality

This objective was not met, the main reason for that was an overestimation of how important the tile editor was going to be and an underestimation of how much work the algorithm would take to get up and running. With the algorithm taking about 95% of coding time while the editor was finalized towards the end of sprint 1.

7.2 Review of the Plan

The original plan was to have three sprints with a research phase overlapping the first two. The first sprint would be dedicated to building a 2D version of the WFC with the second dedicated to extending that version to a 3D one. The third sprint was to be used for extending the algorithm to overlapping model, bugfixes & QOL features.

The first sprint ended up being mostly research with an implementation of a 2D WFC done by the end of it and ended on time. The second sprint it was decided that the current implementation of the algorithm was too slow and complex to extend to 3D and as such a new implementation was worked on that would work with 3D tiles out of the box. The implementation that was planned in the second sprint would use the overlapping model for finding pattern constraints as it was deemed easier to work with. Bugfixes had to be done while coding each unit of the implementation as the output of the algorithm is highly dependent on the extraction and comparison process of patterns. Implementing the algorithm ended up taking a lot more time than planned since every step of the implementation process had to be rigorously tested to find defects as early as possible.

Due to the above, the third sprint was scrapped and the only objective of it not met was the QOL features that were planned for the tilemap builder as the other objectives ended up being achieved much earlier within sprint two. Researching also took a lot longer and was more intense than expected resulting in the research phase eating up a lot of the first sprint's time as I could not start working on an implementation without understanding the inner workings of the algorithm.

If I had to plan another similar project again, I would allocate more time and more focus to just the research phase and drop bugfixing as a task since this will come naturally with the development of the product. I would also try not to depend too much on the outcome of one sprint to be able to move on to the next.

7.3 Evaluation of the Product

Most of the objectives have been achieved in the finalized product and the algorithm can take a small sized input and create a varied much larger output. It can take different sizes and shapes of tiles with the drawback that it will always draw the tiles as a square in relation to the cell size given by the user in the input.

Earlier in the development, the algorithm could only do about 2x the size of the original input, after some refactoring and fixes, it could do 10x the size of the original input in the same timeframe with its only limit being how long the user wants to wait for an output. The dataset that the algorithm extracted could also be changed to allow for reflected patterns and rotated patterns too, although this increased the time complexity.

One of the main issues with the finalized product is the fact that it cannot create outputs higher than the original, this is mostly to do with some error in the logic of the comparison of patterns but due to the time constraint that this project has, the issue has not been fixed. Another one of the main issues is the fact that the output will always be based off cuboid patterns, meaning that more interesting shapes cannot be achieved using this algorithm. It also cannot logically connect structures using the overlapping model. If time would've allowed it, the simple model would have been implemented to fix this issue.

7.4 Lessons Learnt

The main lesson learned was that time is very important when it comes to a larger project. I have wasted much time on code that has not been used anywhere in the final product, and while even the unused code gave me some experience regarding how the algorithm functions, the time spent coding would have been better spent researching.

Another important lesson was the fact that planning and organization play a very important role in any project, and while I started off with a solid plan, it quickly derailed at the start of sprint two when the requirements changed as my knowledge of the algorithm changed, as such I should have planned things with more flexibility in mind.

Finally, most of the issues that I have faced throughout the development of the application was because I did not have a clear picture of the end product in my mind. In future projects, I need to make sure that I more rigorously plan and extract abstract requirements rather than focusing only on what the next step will be, as I feel like that would give me a clearer idea towards what the end goal is.

7.5 Summary

This chapter is an evaluation of the project. Starting with the objectives and how they were achieved/not achieved. With a review of the original plan and justifications for any deviations made. An evaluation of the final product follows with a discussion on what went well and didn't go so well. Finally, a discussion on the lessons that I have learnt throughout the course of this project.

8.Conclusions

This work set out to create a Unity extension that can procedurally generate models using an input model as a base. The implementation of the algorithm is based off the WFC algorithm developed by (Gumin, 2019) using ideas from constraint programming and texture synthesis.

The project can successfully create outputs given an input model by using a data structure that holds nxnxn portions of the original input, comparing them to each other and then creating a constraint dictionary, which is finally applied to an output grid that needs solving (each cell contains all patterns available).

The algorithm solves the grid through three main phases, observation where the lowest entropy cell is “collapsed” (one pattern is chosen for that cell), propagation where any illegal patterns are removed from neighbouring cells, and finally banning where any cells that only contain one pattern are added to a propagation queue and their states are changed to reflect that they are collapsed.

In conclusion, the software is a group of methods from different areas of programming working together to create an output that is procedurally generated.

References

Togelius, J., Kastbjerg, E., Schedl. D., & Yannakakis, G., N. (2011). *What is Procedural Content Generation? Mario on the borderline*.[Conference Paper]. The 2010 Mario AI Championship: Level Generation Track. https://doi.org/10.1145/2000919.2000922

Togelius, J., Shaker, N., Nelson, M.J. (2016), Introduction. In Togelius, J., Shaker, N., Nelson, M.J. (Eds.) *Procedural Content Generation in Games*(pp. 1-14). Springer. https://doi.org/10.1007/978-3-319-42716-4

Efros, A., & Leung, T.K. (1999). *Texture Synthesis by Non-parametric Sampling*[Conference Paper]. IEEE International Conference on Computer Vision, Corfu, Greece. https://doi.org/10.1109/ICCV.1999.790383

Harrison, P. (2005). *Image Texture Tools: Texture Synthesis, Texture Transfer, and Plausible Restoration* [Doctoral Dissertation, Monash University]. https://logarithmic.net/pfh-files/thesis/dissertation.pdf

Merrell, P.C. (2009). *Model Synthesis* [Doctoral Dissertation, University of North Carolina]. https://paulmerrell.org/thesis.pdf

Gumin, M. (2016). Wave Function Collapse Algorithm (Version 1.0) [Computer software]. https://github.com/mxgmn/WaveFunctionCollapse

Karth, I, & Smith, A M. (2021). *WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning*[Conference Paper]. IEEE Transactions on Games. https://doi.org/10.1109/TG.2021.3076368.

Sunny Valley Studio. (2019). Wave Function Collapse Unity Tilemap Tutorial (Version 1.0) [Computer software]. https://github.com/SunnyValleyStudio/WaveFunctionCollapseUnityTilemapTutorial

Appendices

Appendix A

Project Proposal

Project Proposal – 3D Tilemap editor with procedural generation aided by the Wave Function Collapse algorithm

Student: Robert Popovici @00600031
Supervisor: Dr. Norman Murray

Aims of the project

The main aim of this project will be to create a flexible 3D Tilemap editor that extends the inspector to aid in procedurally generating worlds for games in Unity. It will make development of procedurally generated games much easier and faster for game developers by having an intuitive GUI where no programming is needed on their end to achieve a procedural world.

The user will be able to choose whichever modular assets they wish to use with the software and while traditionally you would have a different algorithm for different areas of the map, in this project, the wave function collapse algorithm will be mainly used for procedural generation.

What is the Wave Function Collapse algorithm and why use it?

The wave function collapse is a constraint solving algorithm, meaning it will take some rules & patterns given about our model chunks and apply it to a grid-based map causing new patterns to emerge and create a visually engaging image, while this algorithm is mainly based for 2D images it can be altered to create 3D models. This gives us an opportunity to procedurally generate any kind of environment space using the same algorithm but given different constraints.

The users will have the flexibility to adjust multiple functions that will directly impact what the result will look like, for example they will be able to adjust the scale of the world (as to how far it should extend). They will also be able to specify what assets each different object has (as we will be using modular assets), and the placement of such objects within the created world.

Objectives

- Research the Wave Function Collapse algorithm
- Research the best way to extend the Unity Inspector
- Build a preliminary world using the wave function collapse algorithm as proof of concept
- Extend the Unity Inspector to create a basic 3D tilemap editor
- Fix any issues within the WFC algorithm/3D tilemap editor (MUST BE DONE BEFORE THE NEXT OBJECTIVE)
- Marry the editor with the WFC
- Create a 3D world using the editor and WFC
- Evaluate the system (get user feedback)

Additional Objectives

- Extend the Wave Function Collapse algorithm from simple model to overlapping model
- Extend the tile editor with more functionality, QOL features (eg algorithms that can place objects within the game world)

Who is this for?

This project is aimed at anyone working on a game with Unity, while traditionally, they would have to implement their own algorithms, or use Unity's terrain editor to create terrain, by the end of development, the software solution will be an intuitive add-on to Unity that will allow the user to both create and procedurally generate different worlds, structures, cities, landscapes, all based on different requirements without the need to implement any algorithms.

The main drawback of using Unity with modular assets is the fact that you do not have any functions that deal with tile-mapping, for example, creating vast areas can be quite a pain when you must drag and drop prefabs and copy and paste everything. With any 3D tile map application there is always the option to draw on your prefabs as if you were using a brush, making the design process a lot easier.

Another big point is the fact that Unity's asset store seems to have very few "3D Level Editors", or "3D Tilemap Editors", which will allow me to give developers more option when choosing what tools to use to build their games with.

Why this project?

The main reason for undertaking the project is the fact that I found the idea behind the WFC algorithm interesting as it can procedurally generate seemingly complex worlds from a basic set of constraints or even just a sample image. Another great pro is the fact that I get to familiarize myself with Unity even further by being able to engage with the Editor and Engine more deeply.

While it may seem like a challenging algorithm, the implementation shouldn't be too complex even for 3D worlds and the main foreseeable issue will be with performance and improving it on lower-end hardware.

Software, Hardware. And Methodology

The main software that will be used in building this project will be Unity with Visual Studio, the reason for using Unity being the fact that more than 1.5million developers use it worldwide, with a big chunk of that userbase using it for game development. Another very important reason is the fact that Unity allows you to easily add new functionalities to your inspector even allowing for scripts to be ran from within.

The Wave Function Collapse algorithm with the simple tiled model will be used initially, with hopes of "upgrading" the algorithm to overlapping model later during development (essentially allowing the user to draw a representation of what they wish the algorithm to copy vs the simple model where the user must specify the constraints under which each block should be placed).

Having our project based off the WFC allows us to control the scale of the model being drawn and so the hardware required will depend on the scale of the developer's world. However, this project will mainly be built on:

- 16gb of DDR4 RAM running at 3200mhz
- Nvidia GTX 960 with 2GB of GDDR5 RAM
- AMD Ryzen 7 3700X

I have also planned to use the Agile methodology as it will allow me to best focus on this project and split it into 3 more sizeable chunks. There will be a planning phase during each sprint where I will look at the objectives and extract requirements and use cases, followed by a design phase (if needed). Following coding and testing, there will be a Demo where some requirements of the sprint can be showed off and tested.

During the first sprint, I aim to further research on how to extend the Unity Inspector and the WFC algorithm at the end of which I will deliver the MVP which will be split into two main deliverables:

- The 3D tile placer (using unity's Editor)
- A world built using the WFC out of basic assets (different colored cubes to represent different areas using unity's Engine)

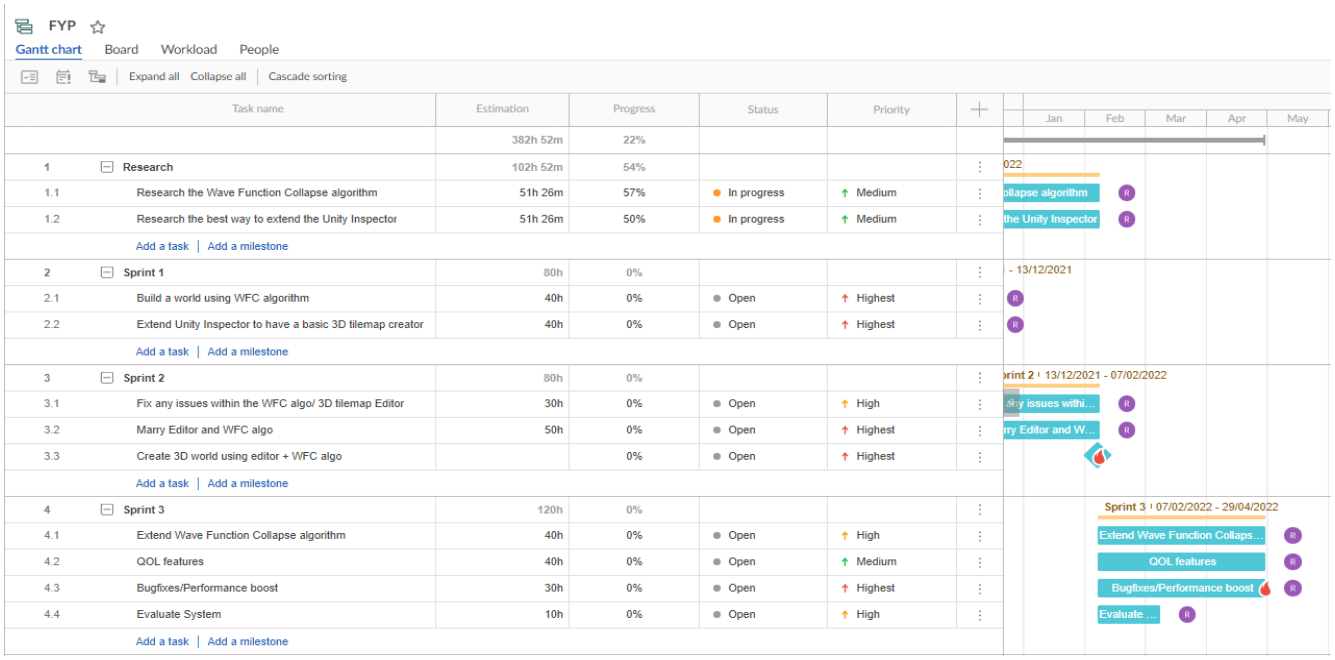
During the second sprint I will continue with the research, while also integrating the WFC algorithm with the 3D Tile Editor and since I foresee some issues creeping up during integration, I will also be dedicating some time to Testing/Bugfixing to ensure that the integrated software delivered at the end of Sprint 2 is as bug-free as possible, I have also planned to find some assets to create a 3D world using the software for the demo at the end of the sprint.

Finally, the third sprint will be mostly dedicated to adding the first additional objective we have as to allow the users to simply draw an example object for the algorithm to create a pattern out of vs having the users specify adjacencies for each model, with some time also dedicated to getting some user feedback as to be able to spot any issues during this sprint. I will also be taking some time to add some lower-priority "Quality of Life" features(given that progress on the overlapping model is going well), where I will be attempting to do the following:

- Extend the 3D Tile Editor to include 3D grid-lines + have objects collapse on those lines(for ease of drawing)
- Add a mesh generator to help keep the world from being too blocky
- Placement algorithms for forests, buildings, etc.

With some time during the sprint dedicated to bugfixing and testing yet again since we're implementing new features.

Gantt Chart



FYP

Gantt chart | Board | Workload | People

Expand all | Collapse all | Cascade sorting

	Task name	Estimation	Progress	Status	Priority
		382h 52m	22%		
1	**Research**	102h 52m	54%		
1.1	Research the Wave Function Collapse algorithm	51h 26m	57%	In progress	Medium
1.2	Research the best way to extend the Unity Inspector	51h 26m	50%	In progress	Medium
	Add a task | Add a milestone				
2	**Sprint 1**	80h	0%		
2.1	Build a world using WFC algorithm	40h	0%	Open	Highest
2.2	Extend Unity Inspector to have a basic 3D tilemap creator	40h	0%	Open	Highest
	Add a task | Add a milestone				
3	**Sprint 2**	80h	0%		
3.1	Fix any issues within the WFC algo/ 3D tilemap Editor	30h	0%	Open	High
3.2	Marry Editor and WFC algo	50h	0%	Open	Highest
3.3	Create 3D world using editor + WFC algo		0%	Open	Highest
	Add a task | Add a milestone				
4	**Sprint 3**	120h	0%		
4.1	Extend Wave Function Collapse algorithm	40h	0%	Open	High
4.2	QOL features	40h	0%	Open	Medium
4.3	Bugfixes/Performance boost	30h	0%	Open	Highest
4.4	Evaluate System	10h	0%	Open	High
	Add a task | Add a milestone				

Appendix B

Project Logbook

Journal

27/09/2021

Kick-off (2-3 hrs)

Project kick-off lecture, did some preliminary research into what kind of project to do and have decided to create something based on the Wave Function Collapse algorithm as I found its applications quite interesting.

Simple explanation of the algorithm – (https://www.youtube.com/watch?v=2SuvO4Gi7uY/)

04/10/2021

Getting Started (1-2 hrs)

As said in the lecture today, will need to keep this up to date with everything done regarding the project, from research all the way to testing.

Will be using LuaTeX to write up the dissertation(maybe?).

06/10/2021

Starting Project Proposal (1-2 hrs)

Decided on name for the project, "3D tilemap editor for procedurally generated worlds".

Will try to use more than just one algorithm in the project, perhaps some algorithms to help with placement of objects in the 3D world.

Initial project proposal draft has been started. Will need to expand the headlines further.

07/10/2021

Continuing with the Project proposal (1-2 hrs)

Have started researching more on the WFC algo, have found a useful link that details how to build the WFC algo in Unity and explains it at the same time. (https://www.procjam.com/tutorials/wfc) For the moment, I will only go through the explanation.

The paper linked above mentions that the overlap model of WFC is similar to a 2D Markov chain (NOTE: this paper is focused on 2D WFC).

(https://setosa.io/ev/markov-chains/ - Explanation of Markov chains).

I have found one of the earliest implementations of this algorithm which detail the algorithm in some very easy step by step methods. It also contains some very interesting papers that this was based off, and some notable ports forks and spinoffs. (https://github.com/mxgmn/WaveFunctionCollapse)

Having researched this allowed me to write a new heading for the project proposal(What is the wfc algo and why use it?)

11/10/2021

Project lecture – Project development methods (4-5hrs)

Having a method is important as it should help maximize my productivity and minimize my risk while building the project.

Thinking about having three agile increments in my project following the lecture's discussion and aiming to be deploying a bit of both the WFC algorithm, and the 3D tile editor at the end of the first increment.

Have started writing out some of the requirements based off my objectives.

And have started researching ways to edit the Unity Inspector and came across these links :

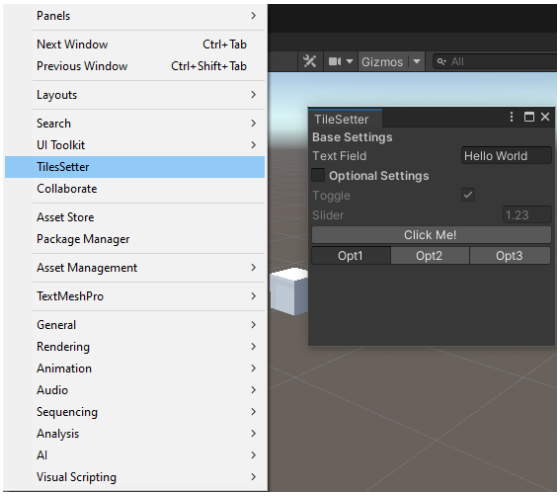
https://answers.unity.com/questions/26446/tracking-the-mouse-position-in-the-editor.html

https://blog.theknightsofunity.com/executing-custom-script-menu/

https://docs.unity3d.com/ScriptReference/GUILayout.Toolbar.html

My attempt at creating a custom window in unity

```
Unity Script | 1 reference
public class TileSetter : EditorWindow
{
    string myString = "Hello World";
    bool groupEnabled;
    bool myBool = true;
    float myFloat = 1.23f;
    int currentToolbar = 0;
    string[] toolbarStrings = { "Opt1", "Opt2", "Opt3" };

    [MenuItem ("Window/TilesSetter")]

    0 references
    public static void ShowWindow ()
    {
        EditorWindow.GetWindow(typeof(TileSetter));
    }

    Unity Message | 0 references
    private void OnGUI()
    {
        GUILayout.Label("Base Settings", EditorStyles.boldLabel);
        myString = EditorGUILayout.TextField("Text Field", myString);

        groupEnabled = EditorGUILayout.BeginToggleGroup("Optional Settings", groupEnabled);
            myBool = EditorGUILayout.Toggle("Toggle", myBool);
            myFloat = EditorGUILayout.Slider("Slider", myFloat, 3, -3);
        EditorGUILayout.EndToggleGroup();
        if (GUILayout.Button("Click Me!"))  // can also take a texture
        {

        };

        currentToolbar = GUILayout.Toolbar(currentToolbar, toolbarStrings);


    }
}
```



14/10/2021

Project proposal writing(2 – 3 hrs)

I have started to expand on some points that I had already written down and changed some headings, have also started to look for some more research papers and have come across one with the exact same idea I have, the implementation is very different, it does show however, that you can create vast areas using this algorithm.

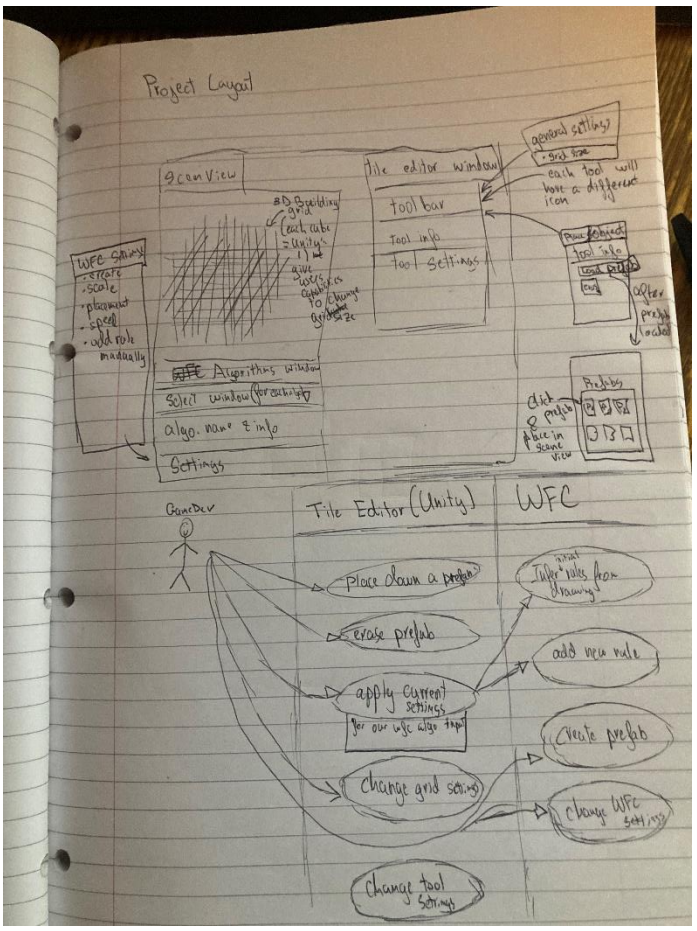
jstage.jst.go.jp/article/transinf/E103.D/8/E103.D_2019EDP7295/_pdf/-char/en

Have now finished the main points for the project proposal with a high chance I will need to expand further.

18/10/2021

Research Methods lecture (5-6hrs)

Following today's lecture and project meeting, this week I should be looking to complete my project proposal, models, requirements and get a start on literature review as I won't be able to get much coding done without. I will be also looking at some implementations of the WFC as examples to understand it better.



I have also changed the gantt chart so that my research extends over 2/3 of my sprints rather than just the first sprint now that I have realized I'll need to put in a few more research hours.

After some initial thinking and a look at Unity I have done a basic pen & paper layout for how the windows will look and going off that I have also made an initial use case diagram.

I will be adding a new heading to my journal called Implementation ideas, as I have been continuing with research on the WFC algo and procedural generation in Unity. And as a result, have come up with new ideas which I would like to document and have quick access to, they will always follow my normal journal entries.

Implementation Ideas

- Mesh generator with noise functions to create the general terrain (eg. grass, hills) as it will help with performance and will keep the world from being too "blocky". As the wfc has its limits and all the assets need to be in modular form it would be nice to be able to take some of the load off the user having to find assets.
  Below are some good starting off points for mesh generation, maybe expand on the idea to include functions for textures if the user wishes to use

a custom one to design different areas(eg desert instead of hills where it would just require a texture change)

- Tutorial for mesh generation(https://www.youtube.com/watch?v=eJEpeUH1EMg)
- Tutorial for procedural mesh generation w noise functions(https://www.youtube.com/watch?v=64NblGkAabk)
- Tutorial for procedural planets, maybe I will be able to repurpose some of the noise functions used here to give the user the ability to adjust smoothness of noise, amount, placement. (https://www.youtube.com/watch?v=lctXaT9pxA0)

21/10/2021

Reading up on the basis of the algorithm (3-4 hrs)

Upon further research and with improved method, I have found a very well cited article that this algorithm is based on(https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/papers/efros-iccv99.pdf).

I have also found an even more interesting implementation of the idea in 2D(https://selfsame.itch.io/unitywfc).

Chapter 7 of this dissertation also deals with tiling models and procedural generation, and provides some interesting insight in "backtracking" when assembling(https://logarithmic.net/pfh-files/thesis/dissertation.pdf)

After reading the beginning of this dissertation, specifically the "Thesis Goals" portion, I have realized that I should probably include some more analytical goal such as the "To analyze the strengths and limitations of such algorithms including their time and memory requirements" goal that this dissertation has(https://paulmerrell.org//thesis.pdf)

"One way to greatly simplify the continuous problem(computing the catalog of adjacencies etc) is to assume that the faces of the output lie on a set of planes parallel to the input"

In depth explanation that I can almost understand (https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/)

I have also started working on a WFC 2D implementation as it is quite slow to get started with for me and below I will create a new title called Programming, where I will keep all my programming logs.

Programming (WFC)

Created 3 scripts so far, one that deals with the WFC algorithm, another that deals with checking which side is which for our gameObjects and a final one that will contain all our constraints for each object(to be made in xml in the future)

22/10/2021

Final tweaks to the project proposal(3-4hrs)

I have added in another objective and detailed it on my gantt chart where I will be evaluating the system by gathering user feedback and have expanded a bit more.

25/10/2021

Project Requirements Lecture (started 2-3hrs)

Having listened to the lecture I realized that I might need to flesh out my requirements and models a little bit more.

I have also spent some time seeing as many implementations of the WFC and have started to think about how the class diagram should look like.

26/10/2021

Questions to ask for wkorkshop

- Best way to connect a script (that would control a prefab etc..) with the unity editor script(ie connect the unity engine to unity editor)
- Should I be concerned about performance when it comes to the WFC algorithm from now or should I just focus on releasing something that works and that could be changed later on
- Better way of finding adjacencies
- Best way to parse & write xml and how to connect that to our prefab to serve as the constraints
- Need to iron out hardcoded variables

27/10/2021

Programming the WFC(8-10hrs)

Have spent some time looking at the best way to represent our prefab constraints and after much trial and error with different data types in different arrangements(think Dictionary<key, Dictionary<key,List<string>) I have found a solution which involves creating a class that will represent our prefab constraints, creating a list that will hold all instances of that class and serialize that list into xml. You can also de-serialize this information meaning you'll be able to access the class instances in the list.

02/11/2021

Literature Review(3-4hrs)

Following the lecture I had on Monday I decided to start creating a mind-map of everything that is connected to the WFC algorithm starting from big resources such as texture-synthesis algorithms(since the WFC is derived from a TS algo) and getting more focused on just the WFC, starting from bigger resources such as Articles that describe the WFC to more specific resources(such as Maxim Gumin's Github page detailing the specifics of the algorithm since he was the one to develop this algo from a texture-synthesis algo)

After the mind map I will start reading those topics and try to extract as much information as I can, this will be very helpful as I will have some form of "cheat sheet" that I can use when writing the literature review.

Mind-Map Links

Consistency in Networks of Relations

Texture Synthesis by Non-parametric Sampling

Image Texture Tools

Model Synthesis

↓

Wave Function Collapse Algorithm

↓

Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm

Enhancing wave function collapse with design-level constraints

Expanding wave function collapse with growing grids for procedural map generation

Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm

Following this, I have created my cheat sheet in order to make referencing things easier.

09/11/2021

Literature Review(3-5hrs)

As I need a deeper understanding of the WFC algorithm to implement it, I have been looking at a few papers that explain it quite clearly.

Here is one paper detailing the algorithm and the design choices when implementing it:

WFC: Content Generation via Constraint Solving and Machine Learning

And here is another paper, a dissertation that expands on the WFC algorithm for complex 3D game generation, although for the moment I am only interested in pp 11-16 which detail the current implementation of the WFC:

Modifying WFC for more Complex Use in Game Generation and Design

Following the reading, I have created a simple Algorithm breakdown so that I can wrap my head around it more easily.

Continuing with the literature review, I have realized that I will need a few more papers, so I managed to find a textbook that details Markov random fields(as the WFC algorithm functions similarly) called "Markov Random Fields for Vision and Image Processing", I have also found a textbook detailing just Markov chains called "An introduction to Markov chains" and working my way back from there I have also found "Procedural Content Generation in Games".

Have now updated my mind-map to reflect these new additions.

I have also started structuring my literature review with a top down approach.

13/11/2021

Literature Review – Structure (2-3hrs)

Have looked at all the sources I have gathered and structured them in a top down approach starting from the broader subjects and as you head down from heading to heading you get more focused towards the WFC until you reach ways to expand the WFC.

Having done that and counted the number of headings made I have realized that I have over 30 headings with only a 3k word limit, I will need to cut some of them out or incorporate some into one.

15/11/2021

Refactoring of the WFC Code(6-8hrs)

Having now read more about the algorithm and different implementations there are 3 main functions that I need within my "Model" class[Initialization, Observation, Propagation].

Looking at different implementations, I have found a tutorial on Implementing maxim Gumin's WFC that highly abstracts different classes and makes everything very modular, I have started following this tutorial today and will look into finishing it, and then changing the necessary classes to provide 3D WFC. [The Tutorial by Sunny Valley Studio]. There is also a Github Repo but I am making a point of implementing it step by step with the tutorial as I would like to 100% understand what the code is doing as I will be refactoring it for 3D later down the line.

16/11/2021

WFC Implementation from tutorial(8-10hrs)

I have now watched the rest of tutorial and implemented the algorithm. The next steps will be to fully comment everything out make a UML diagram, after that is done I can start looking at implementing some optimizations in order to have this run as fast as possible, since the current implementation is quite slow even on a 2D grid. Ideally, I would like to optimize it well enough so that when I have to implement it in 3D I can have the fastest version of the algo. Currently it uses quite a lot of Dictionaries which I know can be optimized so maybe I will start from there.

07/02/2021

Sprint 2 Planning& Algorithm Re-Write(30-40hrs)

The past week has been spent on planning the next sprint and thinking about how to update the algorithm to work in 3D. I have taken the past implementation of the algorithm and have started to re-factor it in a new project to hopefully simplify it a bit as the current 2D version is very complex and quite slow.

A rudimentary version of the tile placer has also been finished which allows for the user to place tiles in x,z and y positions.

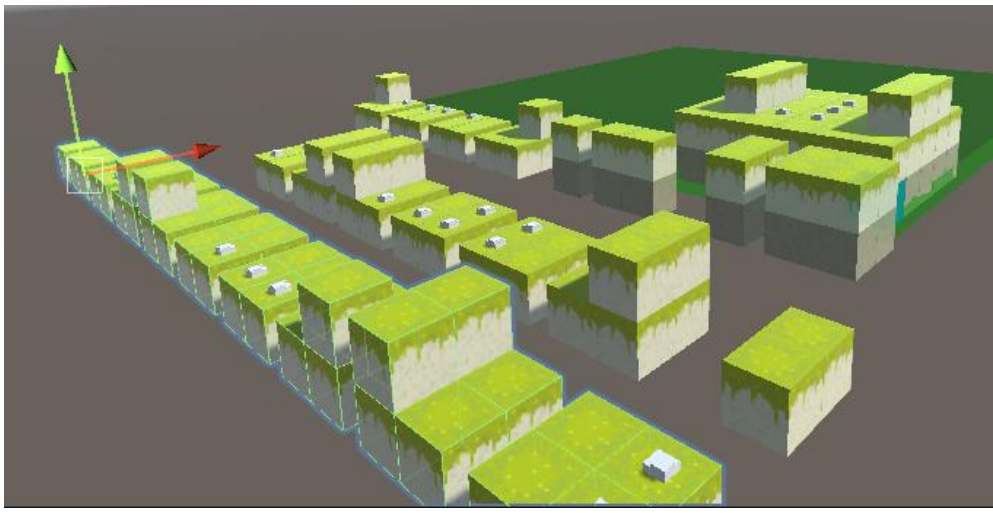
Currently the algorithm that I am making can take the tiles drawn by the user and use discrete variables(int representing a unique tile) and place them in a multi-dimensional array which is a representation of our 3D space.

Next step would be to take that array and find patterns(NxNxN areas of the array), and assign each one a unique index.

16/02/2021

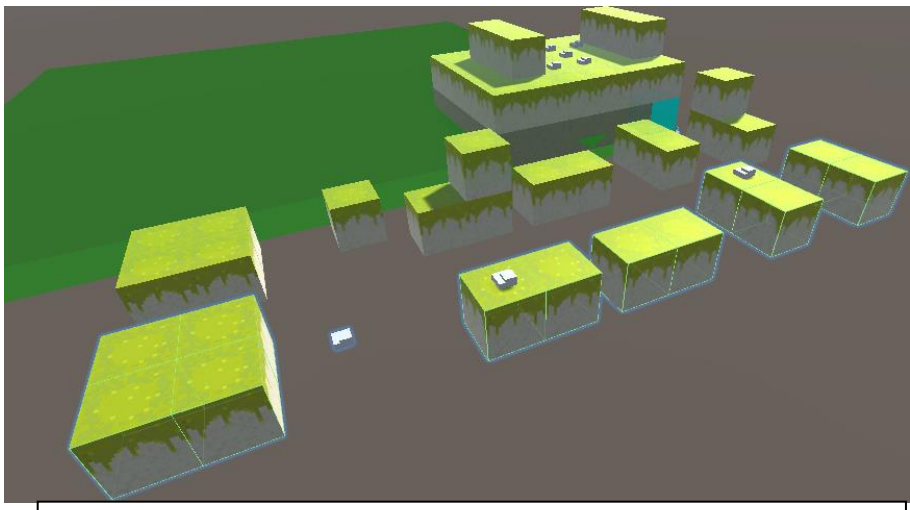
Algorithm re-write(30-40hrs)

This week, I have finished a major part of the software, we can now find patterns and compare their edges to check if the patterns can continue each other, this is all saved in a dictionary of dictionaries that holds the pattern indexes, indexes of the pattern's edges and the indexes of the patterns that can go at each edge. Using this, I will be able to query the dictionary as to which patterns are allowed next to each other in a 3D space during the algorithms runtime.



All the unique patterns that have been found in our input.

Currently, I am on the final step. I will need to implement the algorithm with the data structure that I have just created. To implement this algorithm, I will need to create another nxnxn grid that will represent our output, create another nxnxnxn grid that will represent the available patterns at each cell, next step afterwards would be to programmatically choose and then remove patterns that can no longer exist at a specific cell until a solution is found.



2 Different patterns that have been disassembled into each of their edges ready for comparison.

28/02/2021

Algorithm re-write(50-60hrs)

The last week has been spent on finishing the algorithm itself, I have created a 3D matrix that represents the state of our current algorithm(which cells have

"collapsed" meaning which pattern we have decided on in any given cell), we also have a 4D jagged array with integers at each position that give us the information on the patterns that we have chosen at the relevant positions. The way the algorithm works is as follows:

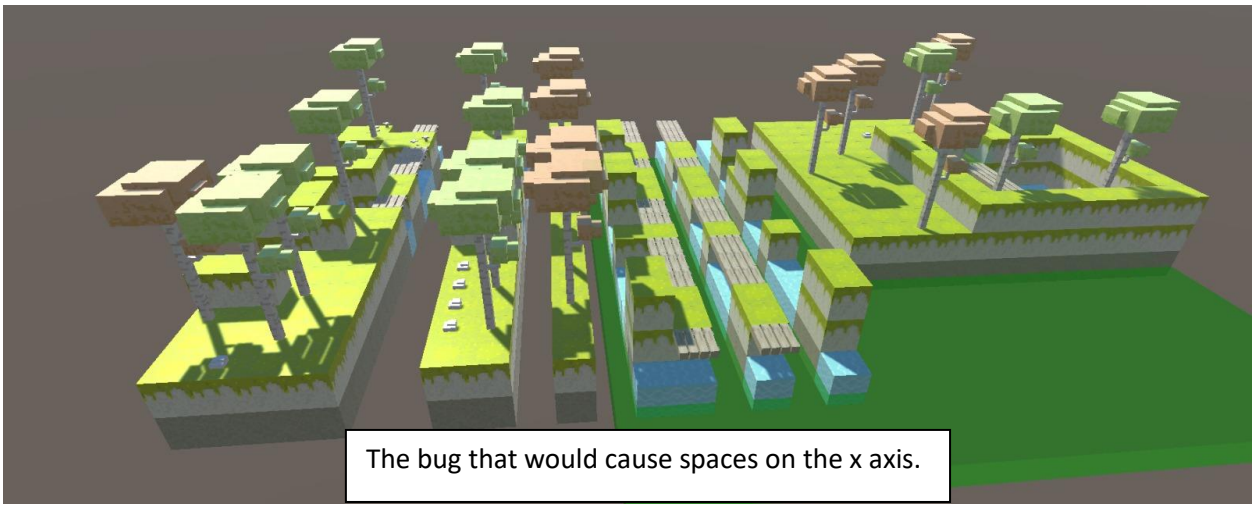
- Get the compatibility data structure that we have made last week.
- Get the pattern indices.
- Initialize our state matrix (this will help us decide much faster whether we must reiterate over a specific cell in the data matrix).
- Initialize an entropy matrix (this helps us decide which cell to choose next during the runtime of the algorithm).
- Initialize our data matrix (this holds the available patterns at each cell).
- Get the entropy of each cell.
- Start a while loop that is lower than the allowed number of max iterations and that will only run if our state matrix still contains cells that need to be filled in.
- Choose a cell with the lowest entropy (meaning that we will choose the cell with the lowest number of patterns available).
- Once a cell is chosen, choose a pattern at random (choosing a pattern based on frequency is one of the non-functional requirements that I will be aiming for).
- Once a pattern is chosen for the cell, ban that specific cell, meaning that we will set the state matrix to true (aka. the cell is filled in).
- Propagate the change to all the grid, meaning we will take out any patterns that are illegal based on our compatibility data structure (a better method of doing this is only propagating the changes to the neighbours of the cells that have been affected, either after we choose the cell, or after we propagate a change to a specific cell).
- If we have reached a resolution, meaning all cells are populated by only one pattern, then we can draw the model, otherwise we throw away whatever we have done so far and start over.

14/03/2021

<u>Algorithm bug fix(20-30hrs)</u>

There have been a few bugs that caused me to make some ineffective decisions when designing the algorithm, one of the main bugs that I have encountered is an infinite loop when trying to propagate to the neighbours of each cell, rather than the grid as a whole, this is my main focus for now as improving that aspect of the algorithm will reduce its time complexity by a lot.

Another bug that I have encountered last week was where the algorithm would generate patterns but lose some of the data in the process, specifically on the x axis causing it generate a model with spaces on the x axis for some reason. I have found that the main issue with that had nothing to do with the algorithm. It was due to Unity needing to have its external script editor set to Visual Studio.



The bug that would cause spaces on the x axis.

17/03/2021

Algorithm tweaks/Code cleanup, Dissertation start(6-10hrs)

The past 2 days have been spent cleaning up the code by commenting the last class out and creating an additional class for some of our helper functions to have a more logical structure to the software. I have also made it so that the user can enable/disable reflections and rotations, although it seems that rotations seem to have a worse effect on the quality of the output with a high time complexity while reflections seem to benefit the output with a moderate time complexity.

The last bugfix has been made, the bug in question would cause the algorithm to enter an infinite loop during the propagation phase if we attempted to propagate any information using a queue, the fix as it turned out, was to simply have an if statement that would only add a cell to the propagation queue if we had made changes to the number of patterns in that cell.

```
compatible[neighbourCell.x, neighbourCell.y, neighbourCell.z] = allowedPatterns.ToArray();
if(allowedPatterns.Count < neighbourPatterns.Length)
{
    propagationQueue.Enqueue(new Vector3Int(neighbourCell.x, neighbourCell.y, neighbourCell.z));
}
updateCellEntropy(neig
```

The code that fixes the infinite loop

I have also made a start on the dissertation by creating a skeleton of the document(headings that we will have).

04/04/2021

Bug Fix, Dissertation continuation(20hrs)

On 14/03 I talked about a bug that would cause the output to lose some of its data, and theorized that the main issue was the fact that Unity needed to have its external script editor set to VScode, that was incorrect and in reality the whole issue was due to the fact that if set to release mode, VScode would break the algorithm somewhere causing the spaces on the x axis. The fix was simple, Unity needs to be set to Debug mode prior to running the algorithm.

After making the skeleton of the dissertation, I have extracted all the requirements, created a flow diagram of the algorithm, and created a UML diagram. I have also starting writing up the first draft of Requirements, Specifications and Design.

11/04/2021

Dissertation Writing – Implementation & bugfixes(20hrs)

This week I have started writing the implementation part of the dissertation, I do so by stepping through the code with a bottom-up approach so that the flow of the writing matches the flow of the algorithm. I have also added some figures and used the open document format to insert relevant code into the body of the dissertation. One major issue I seem to face is the fact that most of my first sprint was focused on research and as such, there isn't much to talk about regarding design and development.

Writing about the code has forced me to think in more depth about the code I've written and as such I found a few small bugs that were not breaking the software but were causing issues. For example, casting a 4.0 float to an int would cause the value to go down by one so instead now we are converting it using a native c# function which gives us the correct values.

20/04/2021

Dissertation Writing – Testing & bugfixes(20hrs)

This week my focus was to create a test plan and test cases, I have made about 10 unit test to test some of my classes, I have also added some integration tests and system test. The PatternRecognition class has the most unit tests since it is the most decoupled, while PatternStrategy has more integration tests since it needs to be communicating with PatternRecognition. The core class serves as our system test base since it brings the two other classes together with the WFC algorithm. One of the main issues I found was that trying to create mock data for the Core class to do integration or unit tests on it proved to be quite difficult and as such I decided to only do system tests regarding that. I still need to do some

performance tests and finish writing the testing and analysis part of the dissertation.

While doing unit tests I found some bugs that would cause performance to drop massively, the main one was to do with our propagation loop. When the algorithm finds that a cell has 0 patterns what it should do is terminate the process and start over, however instead it would continue to solve the other cells, causing it to work through impossible models. Once this was fixed, performance increased by 200-300%.

```
if (allowedPatterns.Count == 0)
{
    terminate = true;
    break;
```

The code that fixes the performance issues

29/04/2021

Dissertation Writing (20hrs) – END

This past week has been spent finishing the dissertation off, adding all the other required components (Lit review, methodology, table of contents, logbook) and numbering all the headings.