Procedural Content Generation in 3 Dimensions using Wave Function Collapse in Minecraft

Gabriel Manners

Abstract

This investigation into the effectiveness of Wave Function Collapse as a Procedural Content Generation Technique in Minecraft sets out to determine whether this method can be used easily by players and game designers to generate content that mimics the original content. We also set out to determine whether this technique can be implemented by game designers or community modders easily enough to improve the default generation of settlements in Minecraft. We grade the effectiveness of our output using metrics provided by the Generative Design in Minecraft Competition in order to test whether generated content is effective. Tests were conducted on terrain that was taken from an existing Minecraft world, and featured a mixture of structures ranging from simple to complex in design meant to simulate structures that players would build near the beginning of the game. Unfortunately, our conclusion is that in it's most basic form, Wave Function Collapse is unsuited as a Procedural Content Generation tool for Minecraft. During the course of our testing, we found that the run times for simple algorithms were too long to be effective, and the algorithm fails to generate content for many of the test cases regularly. In order to make it more suitable, a number of improvements are suggested including global constraints, weight balancing, and layering PCG methods. Overall, this approach has potential, but requires more work before it is a suitable replacement to current PCG methods for Minecraft settlement generation.

Acknowledgments

Thank you Dr. Britton Horn for being my advisor, Drew Bullinger and Dax Henson for images.

Procedural Content Generation in 3 Dimensions using Wave Function Collapse in Minecraft

Gabriel Manners

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation
with departmental honors.

April 14, 2023

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Procedural Content Generation in 3 Dimensions using Wave Function Collapse in Minecraft

Gabriel Manners

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

This thesis seeks to explore whether Wave Function Collapse in its most basic form can be an effective tool for procedural content generation of structures and terrain within Minecraft.

1.2 Procedural Content Generation

"Procedural Content Generation for Games (PCG-G) is the application of computers to generate game content, distinguish interesting instances among the ones generated, and select entertaining instances on behalf of the players." [12] Procedural Content Generation (PCG-G or PCG) is a method of algorithm design that typically uses predefined sets of content and assembles them into new ways to create the objects in a game that the player experiences. It has several uses outside of games, but is widely seen within games for the purposes mentioned above. There are several different ways that PCG can be done, including using aritifical intelligence, mix-and-matching predefined blocks of content, or

even generating content on the fly using several different algorithms. Many games use this, including Borderlands 2, where it used to create weapons for the player to pickup that draw from a database of parts to create entirely new games for each play through [22],

Figure 1.1: Procedurally generated gun from Borderlands 2

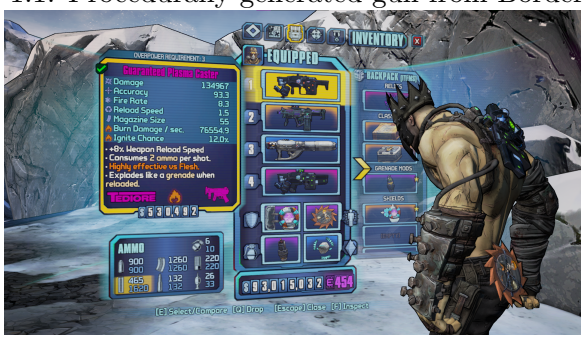


Figure 1.2: Procedurally generated gun from Borderlands 2

No Man's Sky where PCG techniques follow a unique algorithm in order to create each and every planet, start system, generated creature and generated plant that the player seems in the game [5], to levels the player walks through that challenge them in different ways depending on their skill level or items they have picked up. At it's core, PCG is the process of using rules and components to create environments, assets, stories, and levels in order to create unique, dynamic experiences for the player to increase the replay ability of a game.

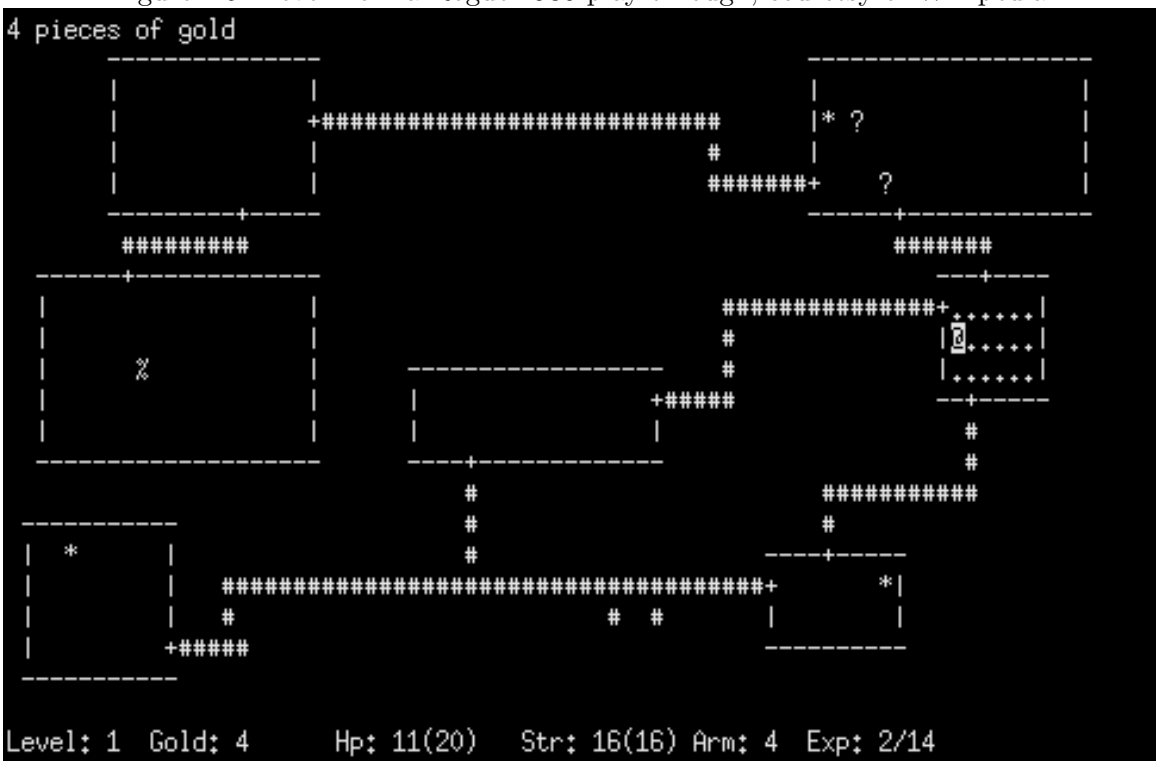
Figure 1.3: Procedurally generated planet from No Man's Sky - Courtesy of Dax Henson

Figure 1.4: Procedurally generated flora and fauna from No Man's Sky - Courtesy of Dax Henson

PCG as a game development tool is widely accredited as having first appeared in a game called Rogue [3] which used PCG techniques to generate the levels that the players moved through. This system avoided the entrapment that was possible with purely random level generation, and ensured a walkable path.

Other games like Spelunky [24] and Risk of Rain [6] have followed in Rogue's steps as a part of a genre called Rogue-likes, or games where each play-through would feature different experiences for the player either in level design, enemy design, or story-wise. More of interest to this paper is how PCG has been used to construct individual assets for games such as in No Man's Sky [5], or Dwarf Fortress [4], where PCG is used to generate specific subsets of the world in combination with an existing landscape (whether that landscape be

Figure 1.5: Level from a Rogue 1980 play-through, courtesy of Wikipedia



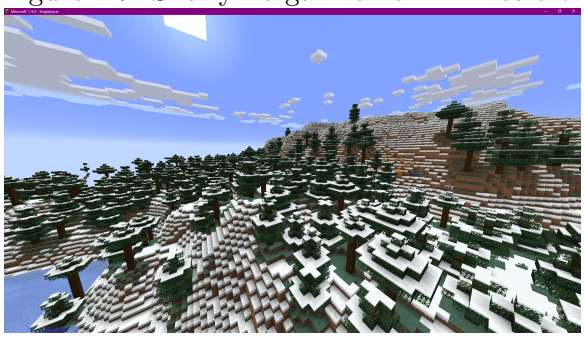
generated by PCG or not). In No Man's Sky [5], the use of guided PCG is present when generating the animals that roam the planets. Following a algorithm [7] the developers are able to create living and moving assets that look (for the most part) realistic, but never before seen. In this way, each planet has different flora and fauna, none of which the developer has to create by hand. The plants are also generated by PCG, similar to the way that Borderlands 2 guns are generated by combining properties together to form unique systems [15]. As such, PCG has seen a widespread use in video games to create content for the players to experience that is unique at the time of playing the game, allowing for practically infinite play through without duplicates.

1.3 What is Minecraft?

Minecraft is a game created in 2009 by Markus Persson and Jens Bergensten that features procedurally generated worlds, simple voxel units, and two simple core game loops [18]. In Minecraft, players explore the world that is generated as they go, mining the blocks the world is made up of with various tools. The world is separated into biomes, which represent a different set of block used to create the world that give it a different feel. Some examples of biomes are Deserts, Snowy Taiga, and Plains.

Figure 1.6: Desert Biome in Minecraft

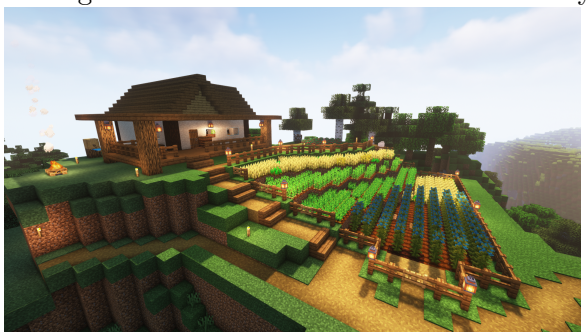
Figure 1.7: Snowy Taiga Biome in Minecraft.



During the course of game play, players can also be expected to craft the mined blocks into tools to enable them to mine faster, defend themselves, and progress. While exploring,

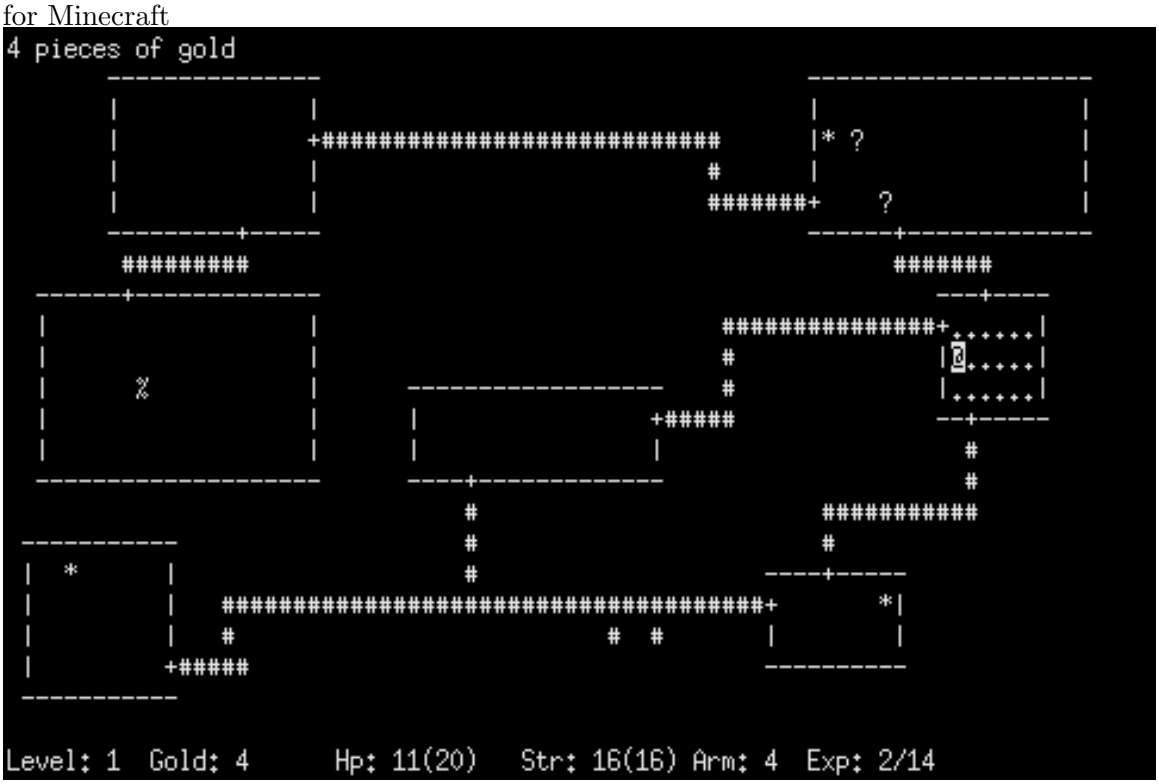
players may encounter structures such as Villages, Jungle Temples, or Water Temples that contain treasure for the players, NPC's roaming the world, and bosses. A key feature in Minecraft is it's capabilities as a sandbox game. Players are able to use the blocks that they mine to construct buildings, machines, traps, and farms in their world to personalize it and share it with friends on multiplayer. Given it's simplistic nature, the number of vast towns and constructs seen on forums is staggering (especially after the advent of creative mode allowing players to give themselves any and every block in the game).

Figure 1.8: Small farming house in modded Minecraft - Courtesy of Drew Bullinger



Minecraft also has a large modding community. Mod or modifications are alterations or additions to the games code that adds content to the game for one purpose or another. Some mods may add in a comprehensive story line and role playing elements, turning Minecraft into something more akin to a tabletop role playing game. Others may add on machines and factory elements, enabling players to delve into science-fiction or steampunk and build large contraptions to help them achieve a set of goals (also set by the mod). Many mods will alter the way that the world generates, modifying the terrain to get new biomes or structures for the player to explore.

Figure 1.9: Screenshot of a biome generated by Biomes O' Plenty, a biome generation mod for Minecraft



1.4 PCG In Minecraft

Minecraft uses PCG techniques to generate the vast world that the player experiences. The world that the player experiences is generated based off of a randomly generated seed made at the beginning of the game that is used to control the randomness of some of the layers of PCG, picking from one of 18,446,744,073,709,551,616 possible seeds if one is not defined at world creation.

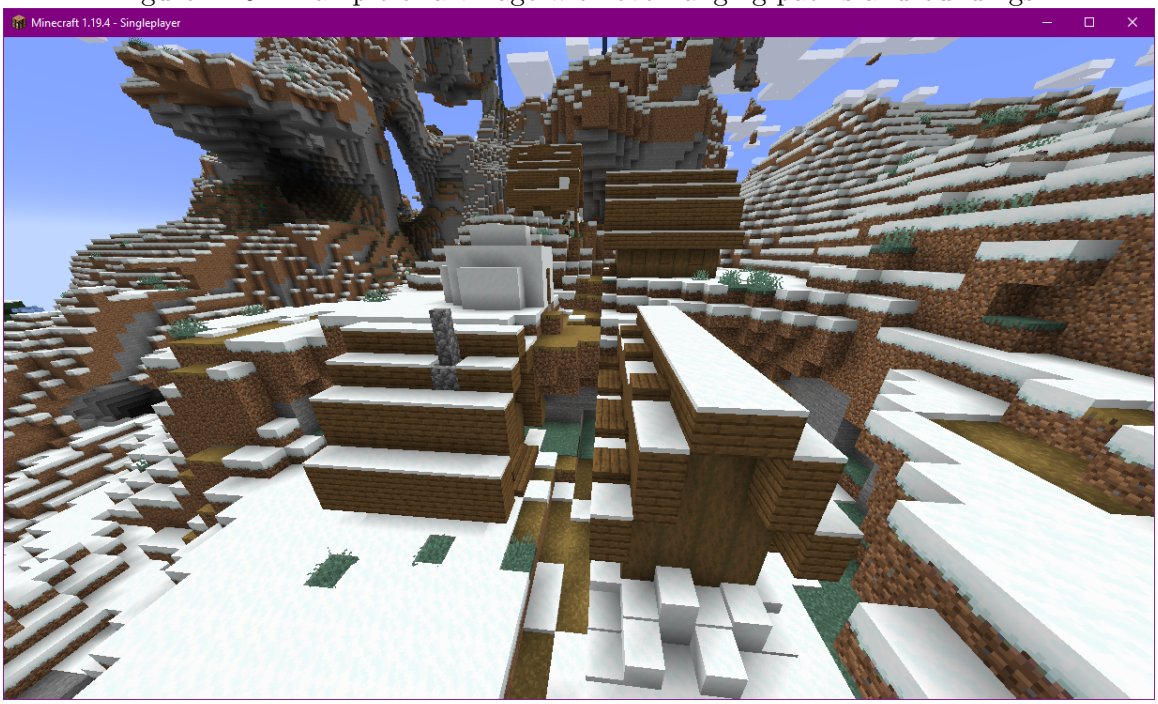
The structures, biomes, cave and ore generation, and the overall terrain of the game are generated in Minecraft using several layers of PCG techniques, beginning with the shape

of the world, and ending by populating it with structures. The first step in generation is to generate the overall shape of the world using a variety of noise maps. A noise map is a method of generating a 2D map of numbers between 0-1. One of the common uses for a noise map is for terrain height-mapping, height-mapping referring to the height of the terrain at a specific location [17]. Minecraft uses several height-maps, referring to them as low and high noise functions in order to generate the shape of the world. The game uses low noise functions to outline the shape of the terrain to the world generator and high noise when the original height-map is above a certain threshold [1]. It also uses a selector noise to vary between the two functions, drastically changing the height of the resulting terrain. Minecraft also goes through a selection of biome's and temperatures to change the blocks that are placed onto the height-map. Using this several step process, Minecraft is able to procedurally generate terrain that is different, using the seed of the game as a seed for the noise maps used to create the world maps.

After the basic layer of terrain has been generated, structures are generated within the game. These structures can range from Strongholds - large underground stone fortresses often generating in caves - to Villages - overground clusters of structures typically made from wood and stone. Generating similar structures to villages is the goal of this paper, elaboration on how current villages are created is necessary.

Villages exist around a center point that is placed in a valid spot above ground (valid spot means in a valid biome with sight lines to the sky) [1]. From here, the village buildings are spawned in a radius around the village center, being placed arbitrarily on the highest block. These buildings include terrain underneath them, which overwrites the currently existing terrain. This can result in buildings that overhang, or are carved partially into a cave. This generation can often look out of place in the world, but does work for the most part.

Figure 1.10: Example of a village with overhanging path's and buildings.



After the buildings are generated, paths are constructed that link the buildings to one another. These path's follow the current terrain exactly, placing the blocks on the highest possible block in the column it wishes to generate. This can result in paths that are unpathable by entities in the world, and visual anomalies in a region.

Many such modders have attempted to fix these problems with village generation within Minecraft, either by tailoring a modification or data-pack to create a new algorithm, or running another pass over villages after they are generated to address these issues.

The Minecraft Generative Design Competition (MGDC) - a competition created by Christopher Salge and Julius Togelius et. al [9] - is a Procedural Content Design competition aimed at inviting community members to build their own PCG algorithms in order

Figure 1.11: Example of a village with proper generation.

to improve upon village generation. Some examples of areas of improvement are layout changes, historical additions (books and texts placed in the game to tell a story), or anything that makes the resulting village fit better into the world, and have a more holistic or comprehensive design to it than existing village methods. The generated settlements are judged for organics based off of their Adaptability (cohesion with environment), Functionality (effectiveness as a living space), Evocative Narrative (stories about the people in the settlement), and Aesthetics (appropriate design for mimicking reality) [20]. These provide good metrics for evaluating content outputted by a PCG algorithm.

1.5 Wave Function Collapse

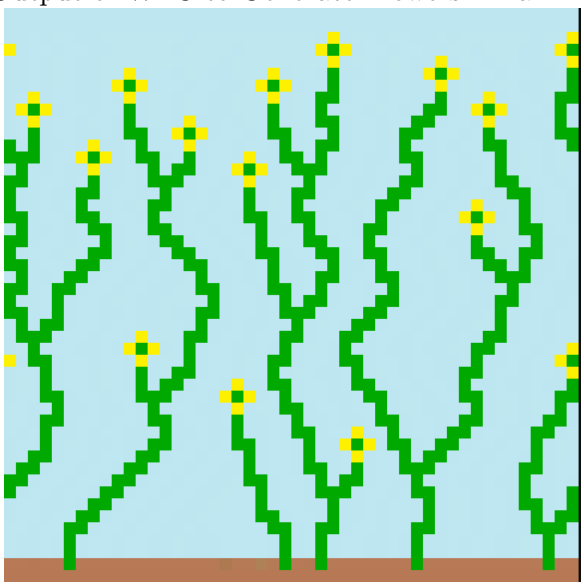
The Wave Function Collapse (WFC) algorithm solves problems by analyzing the adjacencies of input data and solving for an output space. It does this by using the adjacencies of the input as a constraint, limiting and collapsing the output space step by step until an output is found. It is used to collapse the superposition of possibilities of states inside of the output grid into a definite state that can be used for output (The superposition being the possibilities of every tile that could occupy a position based on the surrounding adjacencies). In the application we are using it for, WFC revolves around using adjacencies of input blocks in order to create output matrix's that mimics the original input, but can potentially infinitely expand upon the amount of output able to be generated (within hardware limits). While it will be expanded upon later, the fundamentals of WFC involve reading in content to an adjacency matrix, and then procedurally collapsing an output grid based on those adjacencies from super-positions of every possibility to single values that can then be definite.

This method has proved promising for procedural content generation in 2 dimensions and 3 dimensions given fixed inputs with a number of simple implementations being created with packages for popular game engines such as Unity and Unreal Engine 4 as seen in this GitHub repository [11]. It also has been used within 3 dimensions, able to generate simple structures and patterns based on a series of set inputs.

1.6 Purpose

The purpose of this research is to see whether it is possible to design an implementation of WFC for easy use within Minecraft that lets users choose their own input, as well as being an option for other mods to use as a new method of generation. We also ask the question

Figure 1.12: 2D Output of WFC to Generate Flowers - Maxim Gumin @ GitHub



of WFC limitations for use inside of the Minecraft engine, and the future methods that can be implemented to avoid or overcome the limitations of the base implementation. Overall, we aim to add to the answer of the question of WFC as a procedural content generation mechanism within more complex environments as a tool to game developers to generate content and a tool for users to generate their own content based off of environments they have built, as it is a relatively new method of procedural content generation.

1.7 Methodology

A brief overview of the Methodology is given here on the process of implementation the Wave Function Collapse Method inside of Minecraft. Firstly, the input adjacencies must be obtained. To do this, a matrix of blocks in the input grid is read in and converted to integers. Then, they are converted to chunks. Finally, the chunks adjacencies matrix are

computed and saved.

Afterwards, an output matrix must be established of an arbitrary size. From there, it is populated with possibilities, and then those super positions are collapsed sequentially until each position in the grid contains a single value.

Then, those values are read in and spawned into the Minecraft world.

In order to test the algorithm, a variety of structures and patterns were designed to test the limits of the implementation, and also verify it's functionality. A "functional" run is defined by the capacity of the algorithm to output something when possible, and to successfully inform the user if the generation fails. These outputs were manually verified when possible to ensure that larger scale generations would succeed. The eventual goal for the algorithm is to improve upon it beyond the bounds of this paper, and enter it into the GDMC for an entry during the summer of 2023 (June). The competition will judge the output of the algorithm, and rank it on the scales mentioned above.

1.8 Significance

Wave Function Collapse as an effective PCG system can be an important and effective tool in the game designer's toolkit in order to create new content for games, as well as enabling the user to create their own content for use within the pathway. This particular implementation of WFC allows the user to implement their own content to generate assets that are specific to a user, rather than being ingrained into the game system itself. This does raise some ethical concerns however.

Voxel's seem to be a straightforward method of defining chunks for the WFC algorithm to input from, however there are many games that have user-content creation that do not use a voxel system. One future challenge would be adapting this system to a non-voxel game.

Another expansion to the algorithm would be to separate the generation into several layers, that enable the algorithm to perform better under specifically village or city generation environments.

This papers aim is to, using Wave Function Collapse, see if there exists an algorithm that addresses the areas of focus of development in the MGDC and results in a more life-like village being generated within the Minecraft world. We also aim to address whether or not dynamically inputting user content can be an effective method for generating adjacencies for WFC, or if it is better to pre-generate the content that is able to be used.

Chapter 2

Literature Review

The field of Procedural Content Generation is by no means a new field, although it has been developing rapidly in the recent years. As such, there is much past and current research on the different methods of PCG and how they're used in games, as well as some applications outside of games. This previous research is helpful when determining the successes and limitations of WFC as a PCG technique, as well as provides some context for how the output of that data can be analyzed.

Secondly, in the recent years WFC as a PCG technique has been developing, and examining the other uses for this algorithm, the specifications of other designs, and the modifications that have to be made in order for WFC to function effectively as a PCG algorithm provides useful insight on the methods that can be used to improve the algorithm in the future.

Finally, examining other methods of PCG within Minecraft can help narrow the focus of the PCG algorithm to tailor it to more accurately function within the context of a Settlement or terrain generator. We can use other examples to contextualize the performance of the current algorithm, as well as guidance for how it can be used in the future.

2.1 Procedural Content Generation

Many of the questions revolving around PCG methodologies are how to ascertain whether or not the content matches up with the original in such a way that it can be considered the same. This question is important to consider as our implementation uses user input as it's foundation to generate the final content. Any content generated that does not accurately represent the original is in short, faulty content. Summerville talks about methods for evaluating this effectiveness in his 2018 paper. He discusses two methods of selection criteria that can be used to determine the effectiveness of the generated content of a PCG algorithm (effectiveness here referring to the resemblance to the original material) [23, p. 5-6]. These two methods are the Plagiarism method and the Metric Distance method. Both of these methods are ways of determining the effectiveness of the PCG content. These metrics serve as good foundations for determining whether an algorithm was successful, but are not tailored to specific algorithms in such a way that is sought after. Such a specific level of detail takes place at an individual level, which Summerville says is "all that matters" [23, p. 4]. Thus, the analysis of that individual pieces of content are more important than the role that the PCG is able to play in the experience. This can be a valuable conclusion when attempting to perform isolated generations, but in the case of generation that is part of a large whole (as opposed to when generation is the larger whole), it can be more valuable to use other methods to ascertain the success of a PCG algorithm.

We can look outside of the sphere of games to look at some of the other attempts to validate PCG algorithms, specifically in how the PCG is constrained. Risi and Togelius et. al implement a Search-Based PCG algorithm in order to specifically search (or generate) content based on some evaluation function [19, p. 5]. Their approach goes on to talk about the use of machine learning within PCG algorithms (PCG-ML) in order to develop

a learning algorithm for a specific challenge based on limited information [19, p. 8]. The concept of developing PCG-ML for a specific task is useful to consider, as that specific task can be content based off of video games. As a tool for designers, the application of this has potential in some cases, but for an algorithm such as the one presented in this paper where it is supposed to accept user input as a foundation for the output, training a machine learning model is often computationally expensive and many users would not have access to such compute power.

There are many ways to enhance the performance of a PCG algorithm for a specific task. An example of this is with Gravina and Khalifa et. al's approach to Quality Diversity in PCG. Here, there algorithm searches for "the largest possible set of diverse and high-quality solutions" [p. 1][8]. while their definition of "high-quality" varies from the definition sought after by a PCG method designed to construct settlements, the elaboration that Gravina and Khalifa provide on how to construct an algorithm to match those definitions provides good guidelines for future work of the algorithm. The Quality Diversity measures that are proposed are also useful in developing "many diverse artifacts in one run" [8, p. 7]. Diverse artifacts in one run are especially useful when attempting to create variances off of one or perhaps two sample sets (in the case of users wanting to create settlements out of simple structures or patterns).

2.2 Wave Function Collapse as a PCG

Modifications of PCG algorithms are often essential to obtain specific content for a specific purpose. There are several methods of obtaining this level of specification for WFC as a PCG. One of those, spoken about in Sandhu and Arunpreet et. al. is the use of design-level constraints [21]. These constraints, which can be placed at several levels of the WFC

algorithm in order to modify the behavior, are capable to tailoring the output of a WFC algorithm to better produce valid output. Some of these methods, such as weight recalculation [21, p.5] and non-local constraints [21, p.4-5] are examples of pre-processing methods that modify the content produced by a WFC algorithm. As it shall be seen later, the weight modification can have a large impact on the type of content produced, and smart weight recalculation during an algorithms run can be used to great effect on WFC.

Detouring slightly, WFC can also be used unmodified to great effect, as seen in Maxim Gumin's implementation of WFC C# [11]. Here, there are several implementations of fixed input WFC algorithms that are capable of generating varied outputs based on fixed set inputs. Also on this repository are linked several other implementations in other languages, as well as packages for implementations inside of popular game design engines Unity and Unreal Engine 4. These projects generally follow the same theme, and none are specifically tailored to Minecraft, but they do have useful attributes and explain the algorithm and a fundamental level well. Many of the implementations are tailored for 2 dimensions, but some have been shown to work in 3 dimensions as well.

If we can think of a settlement in Minecraft as a game level, then Cheng and Han et. al. helps to address some issues that they found that are specific to game level creation [2]. They address 4 primary areas that the WFC algorithm can be improved to help with game level creation, specifically automatic rule systems, global control, distance constraints, and multi-layer generation [2, p.1]. In particularly, the concept of multi-layer generation is key to future work that could be done on this algorithm. The process of separating different steps of the algorithm in order to only produce some of the output with WFC may help with some of the difficulties that can be foreseen when WFC is used for a holistic approach.

2.3 Minecraft and Procedural Content Generation

As WFC is a PCG algorithm, and this paper seeks to develop it inside of Minecraft, it is fitting to explore the current space of PCG algorithms within Minecraft. One of the more prominent areas of PCG is designing alternate methods of generating settlements, with a competition known as the Generative Design in Minecraft (GDMC) being run in the last few years [9]. This contest focuses on community implementations of AI in order to generate structures and settlements in Minecraft. As the ultimate goal of the algorithm implemented here is to assess the feasibility of using it to generate settlements, the outlines and standards for judging content provided by the GDMC is a good metric for judging the algorithm's performance.

Before this competition, there have been several precursors to algorithms designed to make settlements, including Green and Salge's organic building generation in Minecraft using cellular automata [10]. In this, they focus on developing floor plans for buildings using a variety of different stages such as room placement, room growth, door placement, and external wall generation [10, p. 2]. They relied on a subjective analysis of the buildings to determine the success of the algorithm [10, p. 4], which was most likely then refined into the criteria used in the GDMC for analyzing algorithms performance as Salge and Green are both on the founders team for the GDMC. This form of PCG within Minecraft provides a good ruleset for basic structures, and could be used in an eventual multi-layer approach to the WFC algorithm.

Other implementations for use in the GDMC include AgentCraft, which is more advanced settlement generator that took second place at the competition that relies on social agents to create a village starting at a center point [14]. These agents result in what Iramanesh calls "organic design" [14, p.5], or more simply design that mimics what we would

expect to see given real-life settlements. This is an alternative method of developing content, as it relies on rules for agents to create the settlement themselves instead of creating rules for a settlement based off of previous settlements (as our algorithm intends to do).

As other algorithms for Minecraft settlement generation have evolved, so too has the competition itself. The competition serves as a good foundation for judging content for Minecraft settlement PCG, as in Salge and Green et. al. (2020), they elaborate on the results of the previous year's competition and address the specifics of judging, founding scores on the concepts of Adaptation, Functionality, Evocative Narrative, and Aesthetics [20, p. 2]. These areas of focus for a settlement generator are key in developing an algorithm that performs against other accepted systems. As such, assessing the validity of WFC as an effective generator means that the content generated must begin to adhere to the concepts listed above if it can be considered effective.

Chapter 3

Methodology

3.1 Research Design and Framework

The implementation of the wave function collapse was implemented inside of the Fabric 1.17 Modding toolchain for Minecraft. This is a collection of methods aimed at letting community members implementation modifications to the game including but not limited to - complete overhauls to the games story and adding RPG elements, science-fiction and fantasy mods, steampunk mods, farming mods, etc. Some of the mods created include structure generation and biome generation overhauls. This toolchain is one of two primary toolchains for modding within Minecraft, the other being Forge. Fabric 1.17 was chosen over Forge due to it's quicker update time, better community and owner support, and prior use.

Minecraft as a framework is a good theoretical choice for this sort of generation. The game has been modded by community members for years now, meaning writing another mod would have many resources at disposal. The game structure lends itself to being split up into definite, complete units. The world is generated in singleton blocks that can be

easily converted to integer space in order to improve upon runtime costs and space costs. The blocks also make it simple for another user to construct with, enabling them to create their own input's at a whim. The set blocks of Minecraft also create definite outputs that can be imaged easily on their own.

3.2 Research Methods and Techniques

As this paper revolves around the implementation and design of an algorithm, the details of that algorithm will be discussed below.

The first step in the algorithm is to extract the information from an input matrix and turn those values into integers. This information is relatively easily stored using methods from the Fabric API until chunks are considered. Minecraft refers to chunks as 16x384x16 spaces within the Minecraft world. This algorithm refers to chunks as NxNxN cubes of single blocks that are derived from the input grid. This is the definition that will be used hereafter.

Chunks in the algorithm are important to provide some context for the adjacencies. If we imagine a chunk of 1x1x1, there is no context to differentiate between two chunks in different locations. Both will be able to generate freely within the confines of the adjacency matrix. This results in randomness in the output. Therefore it is necessary to provide the adjacency matrix with some context for each chunk. We can accomplish this by overlapping the chunks, meaning that two of the same singleton block in different locations will give different adjacencies. Given any chunk size N, the chunks will overlap by n-1 blocks.

These chunks are saved as integers, and the adjacencies are computed from all directions. When an adjacency would reach the bounds of the input grid, an edge adjacency is added, and a corresponding chunk adjacency is added to the edge. Adjacencies are computed per-

chunk, resulting in a large map of integers to a list of integers representing the direction of adjacent chunks.

Once the adjacencies have been computed, the intermittent step can be assembled. This consists of a defined MxMxM output filled with lists of integers. These integers represent the possibilities of chunks that could be in that position. We then collapse each of the faces of the cube using the adjacencies from the edges. From there we take the position with the least possibilities and collapse it to a single value based on the occurrences of each possibility in the input matrix. Once collapsed, a definite value percolates it's possibilities to it's non-collapsed neighbors based on the adjacency matrix. Once all value are collapsed, the grid can be expanded into the world. If not all values are collapsed, or a position in the grid doesn't have a value it can be collapsed to, the algorithm fails.

Once the grid has been collapsed, the algorithm runs through each of the positions, generating the contents of the chunk into the world into the space specified by the user.

This algorithm relies heavily on some of the experimental aspects of this research, including chunk overlap handling, edge handling, and terrain handling.

Edge handling is important for when the algorithm runs into the side of the output grid. In some cases, wrapping to the other side of the output grid is sufficient to get correct output. However, in the case of buildings and generating into a pre-existing world, creating boundaries in the output grid and edge adjacencies reduces the amount of error and dissegmented structures within the output. Our method treats edges as a separate type of chunk, and works by mocking up walls of collapsed edges already in place when the output grid is first being formed, enabling the collapse of faces of a cube in order to initialize and kick-start the collapse matrix.

Another key part of the algorithm is chunk overlapping. With 1x1x1 chunks, there is no context to the information saved within the chunk. In order to avoid this randomness, we

increase the size of the chunks to a variable size of NxNxN. Doing so enabled us to retrieve context from the chunks by overlapping adjacent chunks while recognizing them as different chunks. By doing this we are more easily able to create chunks of data that are coherent and related to more specific neighbors rather than singleton neighbors.

Chapter 4

Results

Results were gathered by testing sets of blocks that were constructed to test certain aspects of the algorithm, or as a general test. The average time in milliseconds for each run through the algorithm was obtained over a set of 100 runs of 100 tries each, and the number of successes and failures were recorded for each as well. The figures below display the number of successes out of the number of runs and the milliseconds on average each run took. The the tests were conducted under two likely conditions, that of Fixed tests where the size of the input and output are the same, and expanded tests where the size of the input is smaller than the size of the output (The output was 2x the size of the input in all directions, additional size increases were limited by memory and time constraints). No tests were done where the size of the input was greater than the size of the output. All tests were run on a 2018 Dell Inspiron 5577 Laptop running Windows 10, with an Intel Core i7-7700HQ @ 2.80 GHZ, 16 GB of 2400 MHz RAM, and an Nvidia GeForce GTX 1050 12 GB.

The tests can be grouped into three categories, terrain, structure, and hybrid. Terrain tests feature large sets of terrain with varying amounts of height and depth in them. They are taken from examples found within the Minecraft world, and are converted into single

block types. 3 categories of terrain have been tested, with Flat set's being 0 height variance layers of blocks, Varied set's having between 1 and 2 blocks of variance, and Complex having more than 2 blocks of variance. Structure tests involve the construction of two types of structures placed on top of flat terrain composed of a layer of stone followed by a layer of grass. Hovel's are the simplest structure a door could be placed upon, composing of solid walls and a roof. Houses feature a bit more complexity, and are larger.

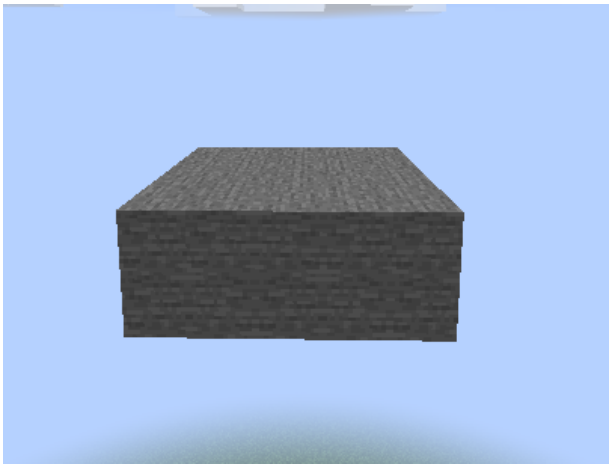
Hybrids feature a mix of both structures and terrain, combining each of the subsets into possible combinations. Hybrid's also include a mixture of alternate patterns that were tested.

Chunk sizes vary from 2x2x2 to 3x3x3. A chunk size of 2x2x2 is referred to as a "chunk size of 2", and a chunk of size 3x3x3 is referred to as "chunk size of 3" henceforth.

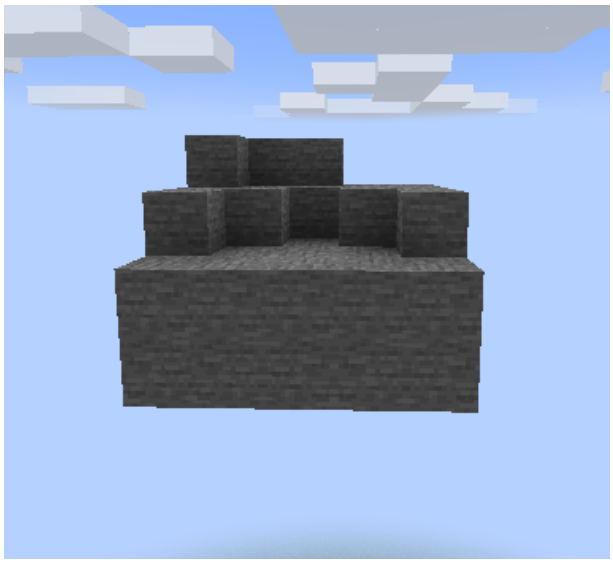
Below are listed the tests success rates under various conditions, and the average time in milliseconds of each tests, along with the images of the inputs and varying outputs acheived by the algorithm.

4.1 Terrain Based Tests

- Flat Stone



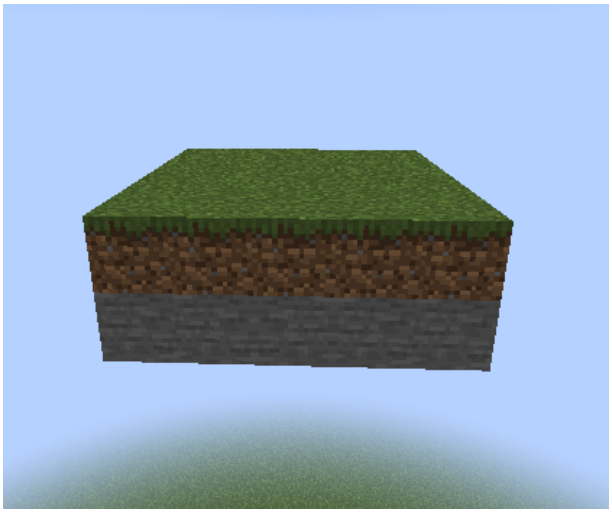
- Varied Stone



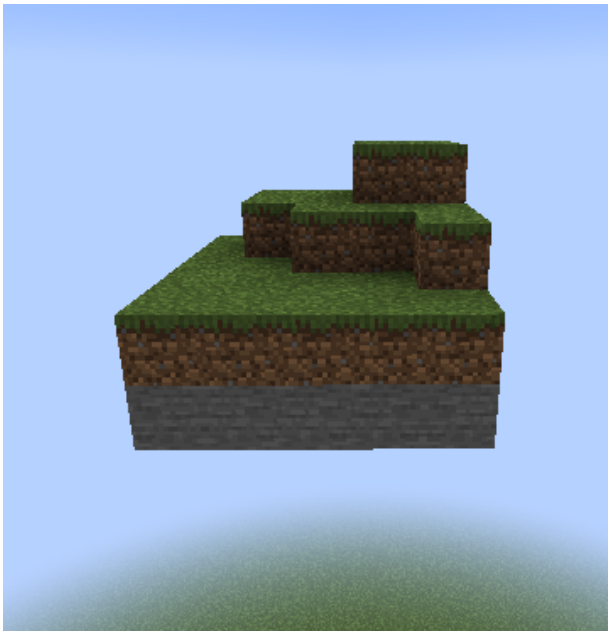
- Complex Stone

- Complex Stone (Alternate)

- Flat Grass



- Varied Grass



- Complex Grass

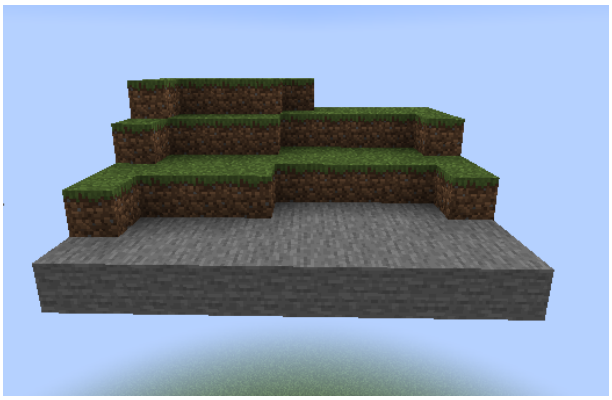


Table 4.1: Average time in milliseconds for stone terrain where chunk size is 2

	Flat Stone	Varied Stone	Complex Stone
Fixed Size	0 ms	14 ms	56 ms
Expanded Size	0 ms	8 ms	1043 ms

Table 4.2: Average successes for stone terrain where chunk size is 2

	Flat Stone	Varied Stone	Complex Stone
Fixed Size	100	2	14
Expanded Size	100	0	0

Table 4.3: Average time in milliseconds for stone terrain where chunk size is 3

	Flat Stone	Varied Stone	Complex Stone
Fixed Size	0 ms	2 ms	40 ms
Expanded Size	0 ms	72 ms	107 ms

Table 4.4: Average successes for stone terrain where chunk size is 3

	Flat Stone	Varied Stone	Complex Stone
Fixed Size	100	100	24
Expanded Size	100	100	0

Figure 4.1: Output of complex stone terrain at fixed size (using alternate complex stone)

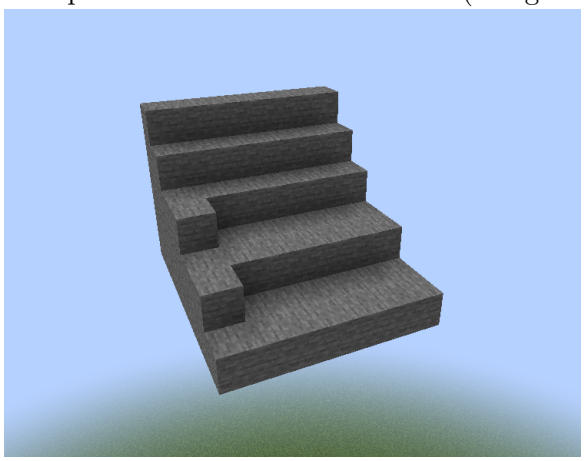


Table 4.5: Average time in milliseconds for grass terrain where chunk size is 2

	Flat Grass	Varied Grass	Complex Grass
Fixed Size	0 ms	0 ms	142 ms
Expanded Size	0 ms	191 ms	370 ms

Table 4.6: Average successes for grass terrain where chunk size is 2

	Flat Grass	Varied Grass	Complex Grass
Fixed Size	100	100	0
Expanded Size	100	14	0

Table 4.7: Average time in milliseconds for grass terrain where chunk size is 3

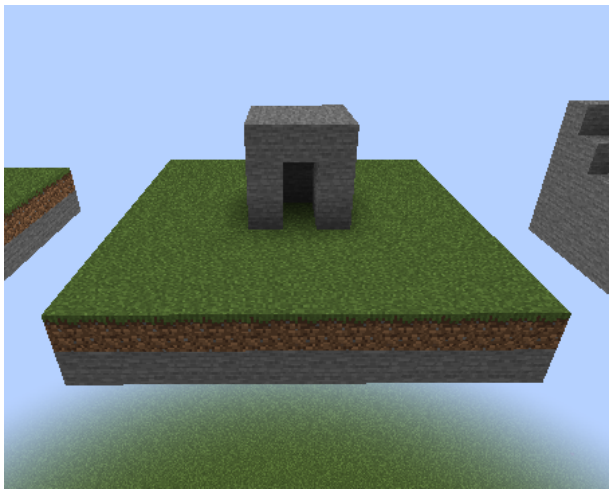
	Flat Grass	Varied Grass	Complex Grass
Fixed Size	0 ms	0 ms	65 ms
Expanded Size	0 ms	65 ms	102 ms

Table 4.8: Average successes for grass terrain where chunk size is 3

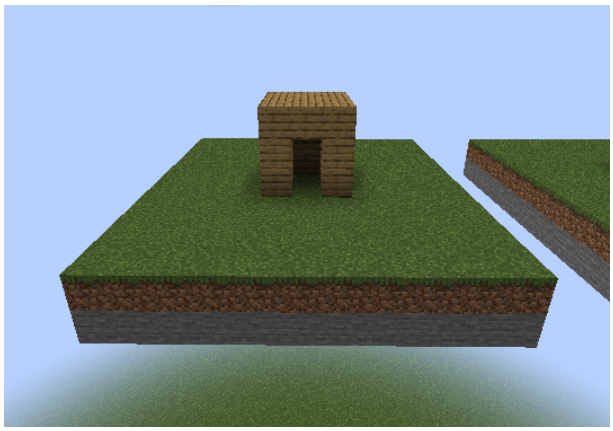
	Flat Grass	Varied Grass	Complex Grass
Fixed Size	100	100	100
Expanded Size	100	100	0

Structure Based Tests

- Stone Hovel



- Wooden Hovel



- Stone House

- Wooden House

Table 4.9: Average time in milliseconds for stone structures where chunk size is 2

	Stone Hovel	Stone House
Fixed Size	64 ms	938 ms
Expanded Size	39867 ms	> 252000 ms

Table 4.10: Average successes for stone structures where chunk size is 2

	Stone Hovel	Stone House
Fixed Size	100	100
Expanded Size	5	No Data

Table 4.11: Average time in milliseconds for stone structures where chunk size is 3

	Stone Hovel	Stone House
Fixed Size	24 ms	2944 ms
Expanded Size	17 ms	> 234000 ms

Table 4.12: Average successes for stone structures where chunk size is 3

	Stone Hovel	Stone House
Fixed Size	100	93
Expanded Size	100	No Data

Table 4.13: Average time in milliseconds for wooden structures where chunk size is 2

	Wooden Hovel	Wooden House
Fixed Size	61 ms	739 ms
Expanded Size	34054 ms	> 463000 ms

Table 4.14: Average successes for wooden structures where chunk size is 2

	Wooden Hovel	Wooden House
Fixed Size	100	100
Expanded Size	3	No Data

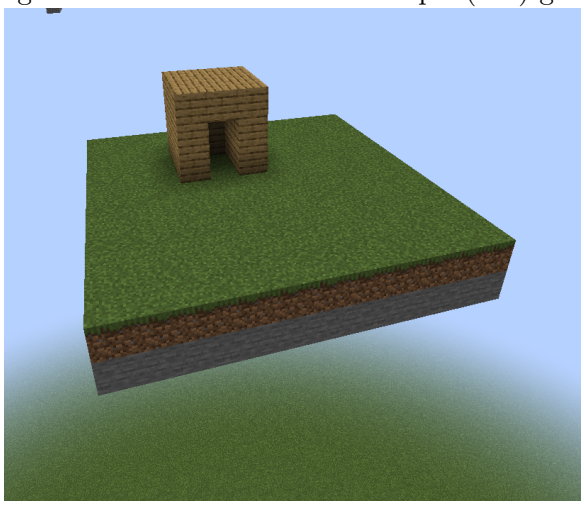
Table 4.15: Average time in milliseconds for wooden structures where chunk size is 3

	Wooden Hovel	Wooden House
Fixed Size	28 ms	599 ms
Expanded Size	50 ms	39626 ms

Table 4.16: Average successes for wooden structures where chunk size is 3

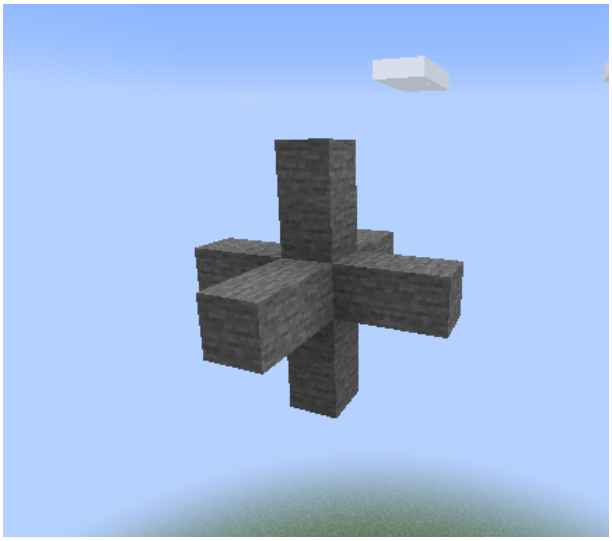
	Wooden Hovel	Wooden House
Fixed Size	100	100
Expanded Size	100	52

Figure 4.2: Wooden Hovel on simple (flat) grass



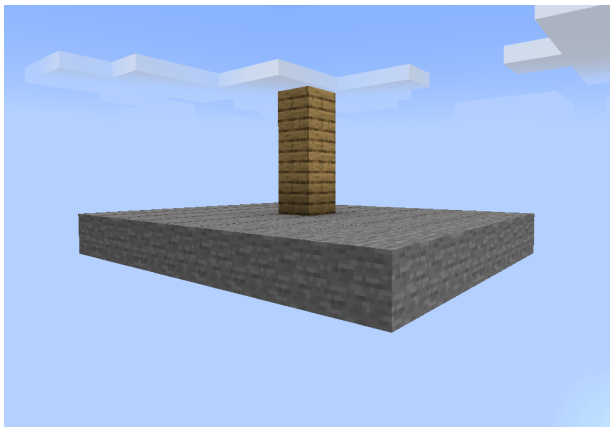
Hybrid Tests

- Stone Pipes

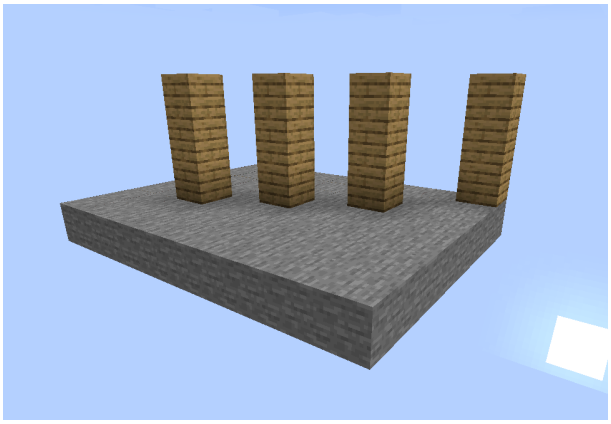


- Simple Striations

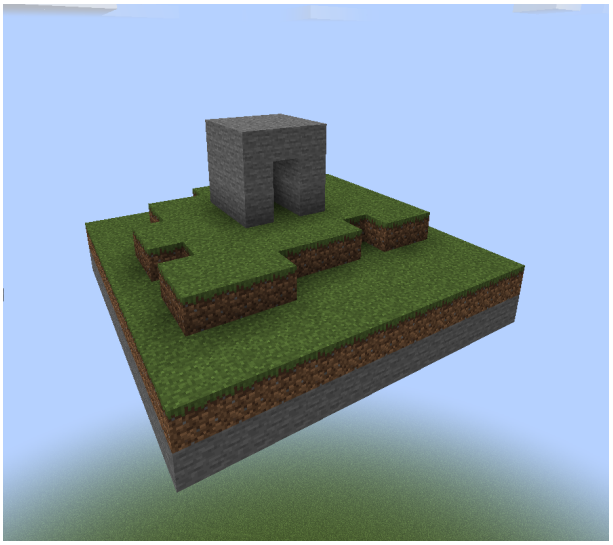
- Wooden Pillars Few



- Wooden Pillars Many



- Stone Hovel on Varied Grass



- Stone House on Varied Grass

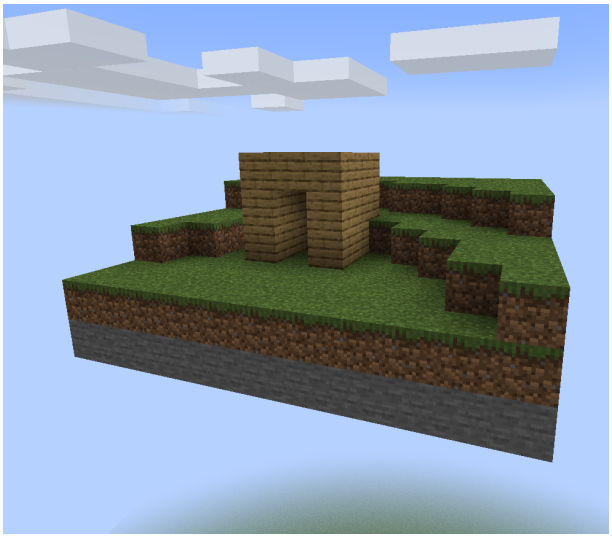
- Stone Hovel on Complex Grass

- Stone House on Complex Grass

- Wooden Hovel on Varied Grass

- Wooden House on Varied Grass

- Wooden Hovel on Complex Grass



- Wooden House on Complex Grass

Table 4.17: Average time in milliseconds for structures on varied grass where chunk size is 2

	Stone Hovel	Stone House	Wooden Hovel	Wooden House
Fixed Size	2704 ms	3309 ms	138 ms	3112 ms
Expanded Size	5796 ms	$>$ 265000 ms	5139 ms	$>$ 256000 ms

Table 4.18: Average successes for structures on varied grass where chunk size is 2

	Stone Hovel	Stone House	Wooden Hovel	Wooden House
Fixed Size	0	0	0	0
Expanded Size	0	No Data	2	No Data

Table 4.19: Average time in milliseconds for structures on varied grass where chunk size is 3

	Stone Hovel	Stone House	Wooden Hovel	Wooden House
Fixed Size	2697	9183 ms	2519 ms	2640 ms
Expanded Size	56371 ms	$>$ 670000 ms	55509 ms	2698 ms

Table 4.20: Average successes for structures on varied grass where chunk size is 3

	Stone Hovel	Stone House	Wooden Hovel	Wooden House
Fixed Size	0	0	0	0
Expanded Size	0	0	0	0

Table 4.21: Average time in milliseconds for structures on complex grass where chunk size is 2

	Stone Hovel	Stone House	Wooden Hovel	Wooden House
Fixed Size	1276 ms	3072 ms	3309 ms	3333 ms
Expanded Size	2393 ms	$>$ 368000 ms	4330 ms	$>$ 245000 ms

Table 4.22: Average successes for structures on complex grass where chunk size is 2

	Stone Hovel	Stone House	Wooden Hovel	Wooden House
Fixed Size	0	0	0	0
Expanded Size	0	No Data	2	No data

Table 4.23: Average time in milliseconds for structures on complex grass where chunk size is 3

	Stone Hovel	Stone House	Wooden Hovel	Wooden House
Fixed Size	54521 ms	0 ms	2640 ms	4982 ms
Expanded Size	0 ms	0 ms	286 ms	No data

Table 4.24: Average successes for structures on complex grass where chunk size is 3

	Stone Hovel	Stone House	Wooden Hovel	Wooden House
Fixed Size	0	0	0	0
Expanded Size	0	0	0	No data

Table 4.25: Average time in milliseconds for miscellaneous tests where chunk size is 2

	Simple Striations	Stone Pipes	Wooden Pillars Few	Wooden Pillars Many
Fixed Size	2 ms	1 ms	77 ms	720 ms
Expanded Size	311 ms	566 ms	2385 ms	1097 ms

Table 4.26: Average successes for miscellaneous tests where chunk size is 2

	Simple Striations	Stone Pipes	Wooden Pillars Few	Wooden Pillars Many
Fixed Size	100	100	21	100
Expanded Size	75	48	7	100

Table 4.27: Average time in milliseconds for miscellaneous tests where chunk size is 3

	Simple Striations	Stone Pipes	Wooden Pillars Few	Wooden Pillars Many
Fixed Size	0 ms	0 ms	2 ms	89 ms
Expanded Size	75 ms	181 ms	68 ms	1936 ms

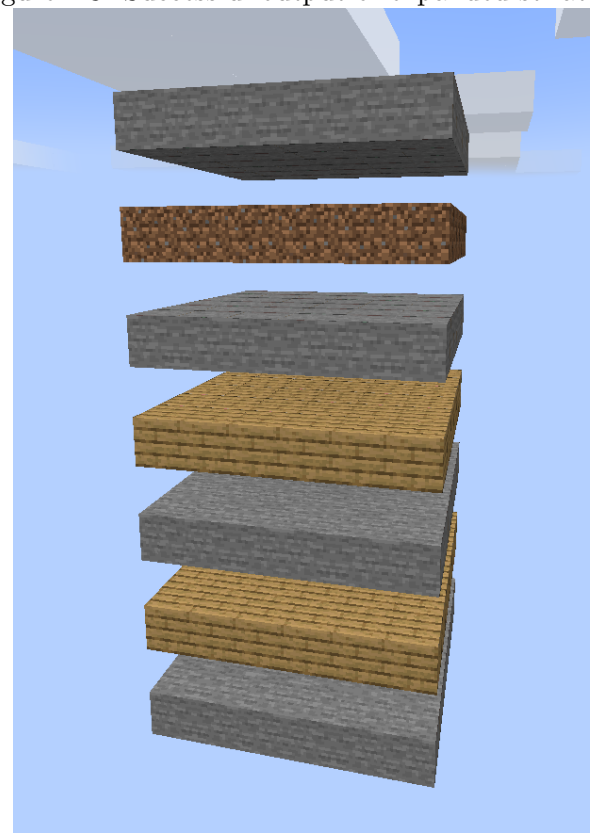
Table 4.28: Average successes for miscellaneous tests where chunk size is 2

	Simple Striations	Stone Pipes	Wooden Pillars Few	Wooden Pillars Many
Fixed Size	100	100	100	1
Expanded Size	0	0	100	0

4.2 Noteworthy Observations and Additions

Several of the successful generations were obtained only by sleeping the thread intermittently during generation to allow the render thread to catch up. These examples usually revolved around the stone structures, and include but are not limited too - Stone Hovel of Fixed size with a chunk size of 2, wooden hovel with fixed size and chunk size of 2, wooden hovel with expanded size with chunk size of 3, stone hovel of fixed size with a chunk size of 3, and the wooden hovel of fixed size with a chunk size of 3. In many of these cases, the output is

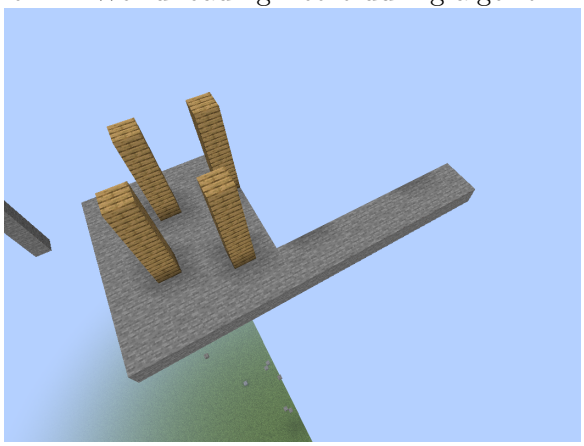
Figure 4.3: Successful output of expanded striations



generated into the world, but the client crashes, the algorithm finishes the test, and then the client crashes due to an internal render thread exception. A solution to this is to pause the generation of a successful run by 50-250 milliseconds by sleeping the thread, which allows the generation to handle the incoming content and generate the output without crashing. The limitations of this solution are discussed.

Several of the tests were unable to complete a single generation within a reasonable time frame. As discussed later, the running of the algorithm effectively pauses game play, and as such test runs exceeding 100000 ms average per run were not allowed to continue, and were deemed unsuccessful. Such tests include stone house of expanded size with a chunk

Figure 4.4: World loading freeze during algorithmic run



size of 2, wooden house of expanded size with a chunk size of 2, stone house on varied grass of expanded size with a chunk size of 2, stone house on complex grass of expanded size with a chunk size of 2, wooden house on varied grass of expanded size with a chunk size of 2, wooden house on complex grass of expanded size with a chunk size of 2.

Some tests are prone to outputting flat air as their primary output as the weighting of the algorithm encourages these flat surfaces initially, which are then propagated through the algorithm. Examples of tests where this generation pattern is common include the following, stone hovel of fixed size with chunk size of 2, stone hovel of fixed size with chunk size of 3, stone pipes of fixed size with a chunk size of 2, stone pipes of expanded size with a chunk size of 2, stone house of fixed size with a chunk size of 3, wooden hovel of a fixed size with a chunk size of 3, and few wooden pillars of fixed size with a chunk size of 3 (this result is expected unlike the others). A method of achieving output that is less consistent in success but more varied in result was found by removing the weighting function from the algorithm to allow non-specific generations to occur more frequently. This is discussed later.

Two tests caused the client to run out of memory, exceeding the 8 Gb allotted to it. The wooden house on complex grass of expanded size with a chunk size of 3, and the stone house on complex grass of expanded size with a chunk size of 3. In both cases, a single run of the algorithm was unable to be completed over approximately 50 attempts. In each case the client would crash and require a full restart.

Chapter 5

Discussion

Overall, the simplest form of WFC as a PCG algorithm inside of Minecraft is not useful for either game designers or for players/modders. There are two primary reasons for this, the time requirements of the algorithm and the preciseness of which the inputs and outputs must be setup. In order to be useful, a different approach for reading in input data is required, or some sort of tailoring of data in order to render is useful and possible to quickly and regularly generate content. Some possible approaches or modifications will be presented. Additionally, the ethical implications of this algorithm (in perhaps a more expanded form) are discussed. With the recent developments of artificial intelligence surrounding ChatGPT and Image Generation AI's, the question of intellectual property in regards to this algorithm poses some interesting answers [16] [13].

5.1 Current Work

Currently, this algorithm is unsuited to usage within Minecraft as an effective tool. This is due to two primary reasons, the time requirement for larger and large sizes, and the precision

with which input and outputs must be sized and constructed to get regular, normal looking output. Here, normal looking output is defined as output that has elements or mimics the input in a cohesive manner. In the case of the misc. tests or the simple terrains, many possibilities can be considered normal as the terrain in Minecraft can sometimes be abstract, but in most cases it should be navigable if the original input is navigable. In the case of structures, the features desired by the GMDC are taken into consideration [9].

Terrain is capable of achieving normal output in many cases, while structures are not. This relates heavily to the precision required by the algorithm in both the inputs and the outputs in order to create desired, curated content. Some limits that exist with this system are: lack of empty space or "air" almost ensures a failed generation, content close to the edges that is not air will impede the variety of generation (this is common in the terrain generations with many of the tests producing identical output at higher chunk sizes), the 1-to-1 weighting of the algorithm means that even in a successful generation flat surfaces or empty generations of "air" are common.

These are all issues because from a player perspective, they render the algorithm unusable. As a player, attempting to precisely calculate the bounds of inputs and outputs without knowing the under-laying code, and being faced with empty generations as a "successful" result would be frustrating and deter the player from attempting such a thing.

From a game designers perspective, this is not so much of an issue. There are existing implementations of WFC that use pre-defined input sets that make it much easier for outputs to be generated consistently. In this way, it would function more like an existing PCG algorithm, taking pre-existing content and combining it into new patterns.

Another problem is the time that the algorithm takes to run (even on simple tasks). In cases that are not at all uncommon - Stone House on Complex Grass Terrain for example - that is often a starter house for players. With a success rate of 0%, having to wait upwards

of 3072 ms for each generation to take place where it may or may not generate content to perform a copy and paste operation (in the case of matching size input and output) will not be worth it to the player. In the case where the input and output sizes are not matching, the player is faced with an even longer run-time, and an even less chance of success. This time scales exponentially with the size of the input. Additionally, while the algorithm is processing the Minecraft engine is unable to process commands and other world events, preventing the player from exploring the world or crafting (the two main game loops of Minecraft).

From a game designers perspective, the problem of time is less of an issue. While it is undesirable to have an inefficient algorithm, the generative step in Minecraft that this would be most commonly used in takes place in parallel to many other steps (referring to the spawning of village centers or overhauling terrain after the original height-mapping). The algorithm would still be limited however, as while exploring the world, players are like to encounter villages. When this happens, the algorithm would have to be run again in order to find valid output, putting strain on the game and potentially slowing down or crashing the clients of the players.

In general, the test runs of chunk sizes of 3 takes less time than the chunk sizes of 2. This does make sense, as the output grids are divided into slightly smaller chunks which requires less time to iterate over. The more expansive the original test (to accommodate for larger structures or larger chunks of terrain) leads to drastically increased times (observe the difference in milliseconds between the stone hovel and the stone house when generating on flat terrain: 17 ms for the stone hovel compared to greater than 234000 ms for the stone house). This increase in time is immense, and renders repeat testing of the stone house impractical.

The drawbacks of the current algorithm in both time and precision render WFC not

preferable for a PCG method within Minecraft.

5.2 Future Work

Despite the most base form of WFC not working as an effective PCG tool, there are several possibilities for improving the algorithm. These include a variety of pre-processing methods designed to isolate the content, layer the generation so that WFC is used in segments of the total product and other PCG methods are used elsewhere, and modifying the algorithm to accommodate for errors seen in generation.

Work that could be done directly on the algorithm includes modifying how the chunk size is determined. Currently, the chunk size is a constant value that the user has to change, however expecting a player to understand which chunk size would work best for their output is unfeasible. Therefore a solution to how chunk size is determined is a necessity, as the larger the chunks are the faster the algorithm is able to run over consistently larger outputs.

Addressing pre-processing methods, these are the most likely to have a significant impact at improving the algorithm for use inside of Minecraft. Some examples of pre-processing methods are to refine the chunks as the user is inputting them so that they better represent the desired output. Grouping together chunks of similar content so that more cohesive content is generated, adjusting the weight of chunks so that "air" is not the predominant output in larger sets are examples of methods that can be used to tailor the inputs so that the output is more definite and regular (generates more frequently).

Another method that is a pre-processing method is the layered or delayed generation of content during the WFC algorithm. For structures or villages, this can involve only applying the WFC algorithm to buildings, and letting other methods of PCG develop the layout of roads and other features in the city. One of those methods of PCG could be WFC,

applying the algorithm in multiple passes instead of trying to generate all the features in one go. This could also be applied to terrain, generating it in multiple layers of the WFC run to produce different aspects of the structure or village. However, reducing the amount of content that needs to be generated by WFC, or tailoring the specifics of WFC and not relying on it being a be-all-end-all algorithm would improve it's effectiveness as a method of PCG.

One of the more frequent issues seen in the results was the generation of flat pieces of land caused by the prevalence of those chunks inside of the input. Because of the preferential selection of chunks with higher weights, and the higher frequency of those chunks, often the output would not mimic the input. A potential solution to this would be something similar to the constraints placed upon the algorithm discussed in Sandhu and Arunpreet et. al [21], modifying or augmenting the weights during the generation based on previous data.

5.3 Limitations

One of the many problems that this algorithm faced was the usage of the algorithm inside of Minecraft. Problems such as crashing when generating content, the render engine failing in other ways, and the client being unable to respond to other input while the algorithm was running lead to difficulties in testing, and output causing crashes that would remove the generated content. Additionally, the nature of Minecraft means that while the algorithm is running, the game will not process additional input. The world will not load outside of the players viewpoint, and they will be unable to craft or mine blocks without significant errors occurring after the algorithm finishes. These issues combined with the long run times of the algorithm in it's present state mean that Minecraft itself poses several issues to the success of the algorithm. These issues are solvable in the context of other mods, but in the basic

form of the algorithm pose difficulties.

Another limitation that exists is running out of client memory. 2 of the larger tests that represent more realistic conditions in which the algorithm would be forced to run under were unable to complete a run due to running out of client memory. These two tests were repeated on a different computer, where it was able to complete a run after 594,000 ms with 16 Gb of memory allocated to Minecraft. These results were not included originally because any modification to Minecraft should not be using upwards of 16 Gb of memory to complete the task, as many computers only feature that much memory for the entirety of the computer. As such, the use of Java objects within the algorithm would have to be changed, or a method of writing to and from a file implemented in order to overcome the memory restriction. This might also have the added benefit of preventing crashing during generation of valid outputs as the render thread is allowed time to process.

5.4 Ethical Considerations

This algorithm, as it does take content that has been built inside of the game, poses some interesting ethical questions. Given that a user takes content built inside of the game to make the algorithm function, can the output be considered theirs? Does the generated content belong to the designers of the whatever algorithm is used. In previous methods of PCG, the content that is being selected from typically exists as property of the game designer (or whoever owns the intellectual rights to the game it exists as a part of). However, here the content could be created by the user inside of a game, or by another user separate from the one initiating the generation. There is not an answer to this question in this paper, but the implications of such a discussion are worth exploring.

Chapter 6

Conclusion

Overall, the WFC algorithm as a PCG tool inside of Minecraft in is most basic form is not suited for effective content generation based off of user input. It faces many flaws as a result of the engine itself, as well as suffers from time complexity and precision issues that players of the game will not want to endure while they are exploring this algorithm.

Despite these limitations and ineffectiveness concerning content generation, WFC is still a promising PCG tool that has potential to be further developed and improved. Optimizing the algorithm for use in Minecraft's engine, or developing other ways to include WFC as part of larger-scale PCG algorithms, or as a multi-step process could develop better methods of PCG using WFC that do not suffer form the same drawbacks as this initial implementation. The integration of WFC into dual-faceted terrain and structure/village generation could improve the game experience for players from the perspective of a game developer.

Overall, PCG is a developing field that is an exciting area of research, enabling bounding leaps in the game development industry. Further development of WFC as a PCG algorithm may lead to dynamic, engaging content that is immersive for the players. As such, it is an area that warrants continued attention and investigation to determine the full potential of

the algorithm, in the context of a tool for game developers and a tool for players to create new content to be explored.

Bibliography

[1] Noise generator.

[2] Darui Cheng, Honglei Han, and Guangzheng Fei. Automatic generation of game levels based on controllable wave function collapse algorithm. In Nuno J. Nunes, Lizhuang Ma, Meili Wang, Nuno Correia, and Zhigeng Pan, editors, *Entertainment Computing – ICEC 2020*, pages 37–50, Cham, 2020. Springer International Publishing.

[3] Epyx, Glenn Wichman, Ken Arnold, and Michael Toy. Artificial intelligence design. DOS-Systems, 1980.

[4] Bay 12 Games and Zach Adams, Tarn amd Adams. Dwarf fortress. PC, MacOS, Linux, 2006.

[5] Hello Games and Sony Interactive Entertainment. No man's sky. PC, 2016.

[6] Hopoo Games, Gearbox Software, Gearbox Publishing, Chucklefish, and PlayEverywhere. Risk of rain. PC, 2013.

[7] Johan Gielis. A generic geometric transformation that unifies a wide range of natural and abstract shapes. *American Journal of Botany*, 90(3):333–338, 2003.

[8] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. Procedural content generation through quality diversity. *CoRR*, abs/1907.04053, 2019.

[9] Michael C. Green. Generative design in minecraft.

[10] Michael Cerny Green, Christoph Salge, and Julian Togelius. Organic building generation in minecraft. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Maxim Gumin. Wavefunctioncollapse.

[12] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1), feb 2013.

[13] David Holz.

[14] Ari Iramanesh and Max Kreminski. AgentCraft: An Agent-Based minecraft settlement generator. October 2021.

[15] Lenni009.

[16] OpenAI. Introducing chatgpt.

[17] Amit J. Patel. Making maps with noise, Jan 1970.

[18] Markus Persson and Jens Bergensten. Minecraft. PC, MacOS, Linx, Andoird, iOS, Xbox, PlayStation, 2011.

[19] Sebastian Risi and Julian Togelius. Procedural content generation: From automatically generating game levels to increasing generality in machine learning. *CoRR*, abs/1911.13071, 2019.

[20] Christoph Salge, Michael Cerny Green, Rodrigo Canaan, Filip Skwarski, Rafael Fritsch, Adrian Brightmoore, Shaofang Ye, Changxing Cao, and Julian Togelius. The AI settlement generation challenge in minecraft. *KI - Künstl. Intell.*, 34(1):19–31, March 2020.

[21] Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. Enhancing wave function collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Gearbox Software and 2k Games. Borderlands 2. PC, 2009.

[23] Adam Summerville. Expanding expressive range: Evaluation methodologies for procedural content generation. In *Fourteenth artificial intelligence and interactive digital entertainment conference*, 2018.

[24] Derek Yu. Spelunky. PC, 2008.

Appendix A

Example appendix

Here is my code.