POLIS UNIVERSITY
FACULTY OF RESEARCH AND DEVELOPMENT

BACHELOR IN COMPUTER SCIENCE
OCTOBER 2022 – SEPTEMBER 2025

Procedural Generation using Wave Function Collapse Algorithm

Soni Seli
Supervisor: Dr. Luca Lezzerini

TIRANË, SEPTEMBER 2025

Acknowledgements

I would like to express my heartfelt gratitude to my mentor and supervisor, Dr. Luca Lezzerini, for his valuable guidance, encouragement, and trust throughout this journey. His support and insightful advice have been instrumental in shaping this thesis, and I am truly grateful for the opportunity to work under his supervision. I would also like to thank my family and friends for their unwavering support, patience, and motivation, which have sustained me not only during the completion of this thesis but throughout my academic studies. Their belief in me has been a constant source of strength and inspiration.

Abstract

To address the "content bottleneck" in modern game development, this thesis details the design, implementation, and evaluation of a 2D tile-based map generator using the Wave Function Collapse (WFC) algorithm. Developed as a practical tool in the Unity engine, the system is grounded in a theoretical analysis of WFC as a Constraint Satisfaction Problem. The prototype employs a Tiled Model, enhanced by a 3x3 positional system and an intuitive, code-free workflow that allows designers to define complex adjacency constraints visually. Evaluation of the generator confirms its ability to produce structurally coherent and aesthetically varied maps. A quantitative performance analysis reveals the algorithm's super-linear time complexity relative to map size, clarifying the practical trade-offs between scale and computational cost. The project culminates in a functional map generator, offering valuable insights into the application and scalability of the WFC algorithm.

Abstrakti

Për të trajtuar “ngushticën e përmbajtjes” në zhvillimin bashkëkohor të lojërave, kjo tezë paraqet dizajnin, implementimin dhe vlerësimin e një gjeneruesi hartash 2D me bazë pllakat, i ndërtuar mbi algoritmin Wave Function Collapse (WFC). Sistemi është zhvilluar si një mjet praktik në softuerin Unity dhe mbështetet mbi një analizë teorike të WFC-së të konceptuar si një Problem i Përmbushjes së Kufizimeve. Prototipi aplikon Modelin e Pllakuar, i pasuruar me një sistem pozicional 3x3 dhe një rrjedhë pune intuitive, pa nevojën e programimit, e cila u mundëson dizajnerëve përcaktimin vizual të kufizimeve të ndërlikuara të afërsisë. Vlerësimi i gjeneruesit dëshmon aftësinë e tij për të prodhuar harta koherente në aspektin strukturor dhe të larmishme estetikisht. Analiza kuantitative e performancës nxjerr në pah kompleksitetin kohor super-linear të algoritmit në raport me madhësinë e hartës, duke sqaruar kompromiset praktike ndërmjet shkallëzimit dhe kostos llogaritëse. Projekti përmbyllet me zhvillimin e një gjeneruesi funksional hartash, i cili ofron kontribute të rëndësishme mbi aplikimin dhe shkallëzueshmërinë e algoritmit WFC.

List of Figures

List of Tables

Abbreviation

2D	Two-Dimensional
3D	Three-Dimensional
CSP	Constraint Satisfaction Problem
DL	Down-Left
DM	Down-Middle
DR	Down-Right
G	Grass Tile
LTS	Long-Term Support
ML	Middle-Left
MM	Middle-Middle
MR	Middle-Right
MRV	Minimum Remaining Values
PCG	Procedural Content Generation
UL	Up-Left
UM	Up-Middle
UR	Up-Right
WFC	Wave Function Collapse

Table of Contents

1 Introduction

In the landscape of modern digital entertainment, the scale and complexity of virtual worlds have reached unprecedented levels. From the expanding, continent-sized maps of open-world role-playing games to the infinitely replayable levels of independent roguelikes, the demand for vast, detailed, and engaging content is a defining feature of the industry. This appetite for content has, however, created a significant challenge for developers. The traditional paradigm of manual content creation, where artists, designers, and modelers meticulously craft every environment, object, and texture by hand, is struggling to keep pace. This approach is not only incredibly time-consuming and labor-intensive but also scales poorly, leading to ballooning development budgets and prolonged production cycles. This "content bottleneck" presents a major obstacle to innovation and creativity (Hendrikx et al., 2013).

As a direct answer to this challenge, developers are turning to Procedural Content Generation (PCG). PCG flips the traditional model on its head: instead of artists hand-crafting every asset, they design intelligent systems that do the creating for them. These systems use algorithms, following a mix of rules and randomness, to generate content on their own (Shaker et al., 2016). This shift in focus, from making the thing to making the thing-maker, allows teams to generate a staggering volume and variety of content that would otherwise be out of reach. More than just a production tool, PCG is a creative force, enabling games with near-infinite replayability, emergent gameplay, and journeys that feel truly personal.

1.1 Motivation and Problem Statement

There are plenty of good reasons why developers are turning to PCG. On the business side, it helps control the ballooning costs of creating game assets. For designers, it's a powerful tool that allows for rapid prototyping and testing, making it easier to refine gameplay ideas across countless different environments. Most importantly, for the player, it creates a sense of near-infinite novelty. It ensures no two playthroughs are ever the same, a magic ingredient in beloved titles like The Binding of Isaac and vast universes like No Man's Sky (Shaker et al., 2016).

Figure 1: Minimaps of the full layout of a level in The Binding of Isaac [1].

But procedural generation isn't a silver bullet. Its biggest challenge is that it often produces content that feels generic, chaotic, or just plain broken (Hendrikx et al., 2013). A simple noise function might spit out a jagged, ugly landscape, while a purely random dungeon generator could create a nonsensical maze of disconnected rooms. The real goal, then, isn’t just to generate content,it’s to generate high-quality content that feels intentional and follows a clear set of structural and aesthetic rules.

This is where the Wave Function Collapse (WFC) algorithm comes in as a truly innovative solution. First introduced by Maxim Gumin in 2016, WFC is a clever, constraint-based approach inspired by concepts in quantum mechanics. At its heart, WFC works by making sure that every little piece of the output fits perfectly with its neighbors, according to a predefined set of patterns. It starts with a canvas of pure possibility and, step-by-step, "collapses" those possibilities into a single, concrete result, with each decision sending ripples outward to constrain the choices of its neighbors. The outcome is remarkable: WFC can generate large, complex structures that look incredibly coherent and plausible. This makes it perfect for creating detailed tile-based maps, intricate textures, and believable architecture.

However, for all its power, WFC isn't a simple "plug-and-play" solution. To use it effectively, you need a solid grasp of its core principles and how to adapt them to the task at hand. That's the core problem this thesis sets out to solve: to design, build, and apply a robust and accessible WFC tool for generating coherent and varied 2D tile-based maps in a typical game development workflow.

1.2 Research Objectives

To solve this problem, I've set out three clear goals for this thesis. The plan is to build from the ground up: starting with a strong theoretical understanding, then creating a practical tool, and finally, taking a hard look at the results.

1. Get a Deep Understanding of the Wave Function Collapse Algorithm. My first objective is to go far beyond a surface-level description of WFC. I'll connect the dots between the original concept, academic research, and community-built examples to create a complete picture. This deep dive will clarify how WFC is formally related to Constraint Satisfaction Problems (CSPs), translate the confusing "quantum mechanics" analogy into practical, step-by-step logic, and carefully examine how it uses entropy to make its decisions.

2. To Design and Implement a Functional Prototype for 2D Map Generation. The theoretical understanding will be translated into a practical tool within the Unity game engine, using the C# programming language. This objective encompasses key architectural decisions, including the design of efficient and intuitive data structures for representing tilesets and adjacency constraints. The goal is to create a system that is not only algorithmically sound but also integrates smoothly into a designer's workflow.

3. To Evaluate the Artifacts and Performance of the Implemented System. The final objective is to critically assess the outcome of the project. The quality of the generated maps will be evaluated against qualitative criteria such as structural coherence, visual plausibility, and the capacity for producing variety from a single set of rules. Concurrently, the system's performance will be quantitatively measured to understand the computational trade-offs between key variables like map dimensions, tileset complexity, and generation time, ensuring its viability as a practical design tool.

1.3 Scope and Limitations

To ensure a focused and achievable research outcome, the scope of this thesis is deliberately centered on the design and implementation of a 2D tile-based map generator intended for use as an editor-time tool within the Unity engine. The system operates on manually defined constraints, granting the designer direct artistic control, rather than automatically extracting rules from an example image. Key limitations are acknowledged to define the project's boundaries: the implemented algorithm is a greedy, non-backtracking solver, meaning it will fail upon encountering a contradiction; its application is strictly confined to two dimensions; and the final output is a functional proof-of-concept prototype rather than a commercially polished or exhaustively optimized tool.

1.4 Structure of the Thesis

This thesis is organized into six chapters to guide the reader logically from theory to conclusion. Chapter 2 establishes the theoretical groundwork by reviewing Procedural Content Generation, Constraint Satisfaction Problems, and the Wave Function Collapse algorithm in detail. Following this, Chapter 3 presents the conceptual design and architecture of the map generation system, which is then documented in its concrete implementation within the Unity engine in Chapter 4. Chapter 5 offers a critical evaluation of the project, analyzing the quality of the generated maps and the performance of the system. The thesis concludes with Chapter 6, which summarizes the contributions of this work and proposes potential avenues for future research.

2 Theoretical Foundations and Related Work

To really appreciate what makes the Wave Function Collapse algorithm so special, we first need to understand the world it comes from: automated content creation. This chapter will lay all the theoretical groundwork. We'll start by exploring the broader field of Procedural Content Generation (PCG). Then, we'll move on to Constraint Satisfaction Problems (CSPs), which are the fundamental logic that WFC is built on. Once that's covered, we'll take a deep dive into WFC itself, examining how it works, where the ideas came from, and how it's used in practice to create content.

2.1 Procedural Content Generation (PCG)

Procedural Content Generation (PCG) is all about teaching a computer how to create digital content on its own, rather than making everything by hand. At its heart, PCG is a shift in thinking: instead of creating the content directly, a developer creates the process that generates the content (Shaker et al., 2016). This process, often called a "generator," uses a combination of rules, settings, and a bit of randomness to produce a huge sometimes endless variety of content. The ultimate goal isn't just to make something new every time, but to ensure that what's created actually makes sense.

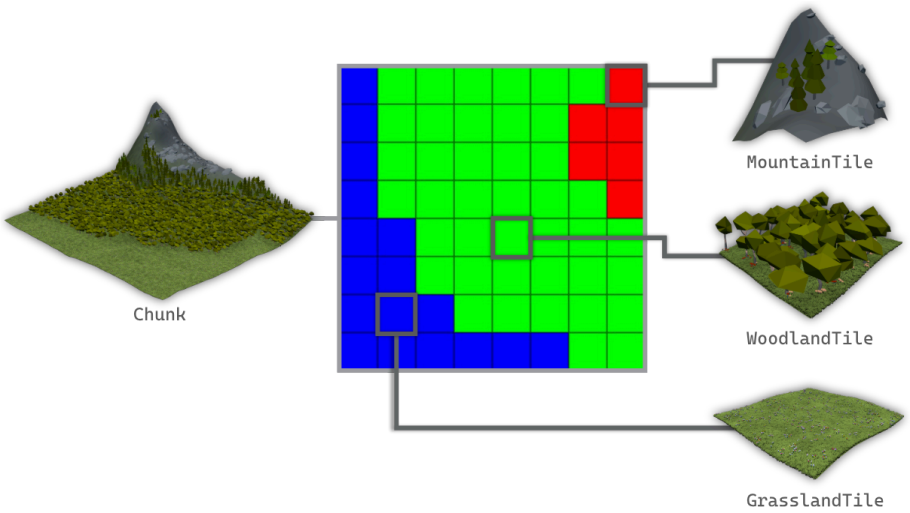


Figure 2: Procedurally generated terrain based on tiles and biome chunks [2]

2.1.1 Overview and Taxonomy

The main reason PCG is so popular is its ability to solve the "content bottleneck" ,the constant growing demand for new game assets that puts a strain on development teams (Hendrikx et al., 2013). By automating how content is made, PCG can dramatically cut down on the time and money spent on manual work. It empowers smaller teams to build vast worlds and makes it much easier to quickly prototype and test new ideas. Beyond that, PCG is a powerful creative tool in its own right, capable of generating surprising gameplay moments and offering immense replay value by making every player's journey unique.

As you might imagine, "procedural generation" isn't a single technique; it's a broad field filled with many different methods. To help organize this variety, researchers have developed ways to compare and classify them. A widely used approach, described by Togelius et al. (2011) and Shaker et al. (2016), is to categorize any PCG system based on a few key characteristics:

- Online vs. Offline PCG: This is the most critical distinction, defining when the generation occurs. Offline PCG takes place during the development process. A designer might use a procedural tool to generate a city, a forest, or a set of textures, and then manually select and integrate the best results into the final game. In contrast, Online PCG occurs at runtime, as the user is playing the game. This approach is used to generate content on-the-fly, creating seemingly infinite and unpredictable worlds, as seen in the planetary systems of No Man's Sky or the dungeon layouts of traditional roguelikes.
- Stochastic vs. Deterministic PCG: This axis describes the role of randomness. A stochastic generator will produce a different output every time it is run, even with identical input parameters. A deterministic generator, given the same initial "seed" (typically a number), will always produce the exact same output. This is crucial for experiences where players may want to share a specific world with others, as famously implemented in Minecraft, where a world's seed can be used to perfectly replicate its terrain.
- Constructive vs. Generate-and-Test: This describes the core generative strategy. Constructive algorithms build the content directly and incrementally, without significant revision. The Wave Function Collapse algorithm, for example, is constructive; it places a tile and does not go back to change it. A generate-and-test approach, conversely, generates a complete piece of content (or a part of it) and then evaluates it against a set of quality criteria (a "fitness function"). If the content fails the test, it is discarded, and the process repeats until a satisfactory result is produced.

- Degree of Control: This spectrum describes the extent to which a human designer can guide the generative process. At one end, a generator might only take a single random seed as input, offering no further control. At the other end, a "mixed-initiative" or "experience-driven" PCG system may involve a sophisticated set of parameters, rules, and even real-time feedback loops that allow a designer to steer the output towards a desired outcome (Yannakakis & Togelius, 2011).

2.1.2 Applications in Game Development

The history of PCG in game development is nearly as long as the history of video games themselves. From the earliest days of limited memory, developers have used procedural techniques to create content that would have been impossible to store directly.

One of the earliest and most influential examples is the 1980 game Rogue (Fig. 3), which used PCG to generate a new dungeon layout for every playthrough. This foundational use of online, constructive generation defined an entire genre, the "roguelike", and established replayability as a core design pillar. Shortly after, the space-trading game Elite (1984) famously used deterministic PCG to generate an entire galaxy of eight thousand star systems from a few mathematical formulas, a feat of immense scale that was impossible to achieve with manual creation on the hardware of the era.

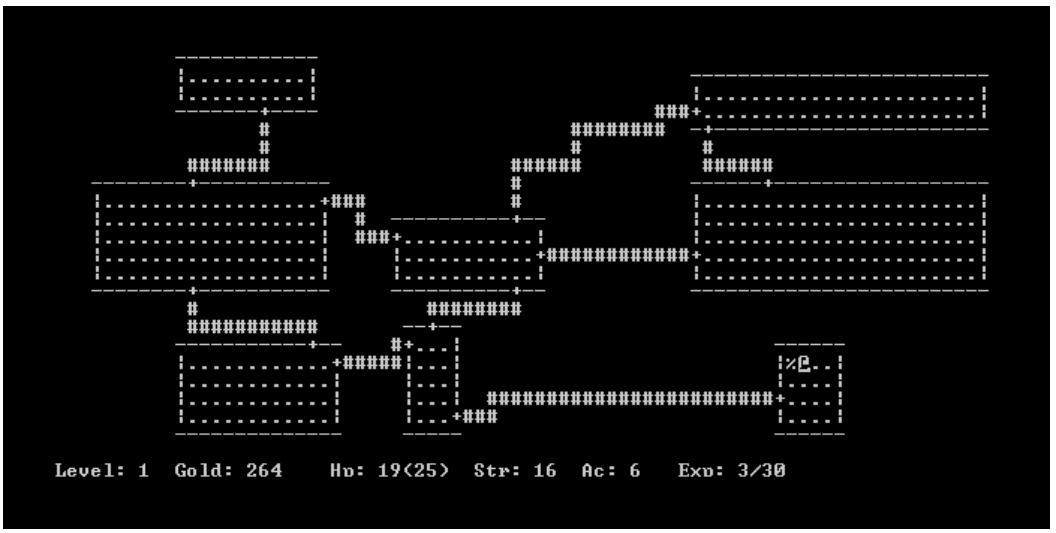


Figure 3: A procedurally generated dungeon map in the video game Rogue [3]

In the modern era, the applications of PCG have diversified significantly:

Game Worlds and Levels: This remains the most prominent use of PCG. Minecraft generates vast, open-ended voxel worlds using Perlin noise and a deterministic algorithm (Fig.4). The Diablo series uses PCG to create semi-randomized levels and enemy placements, ensuring replayability while maintaining a curated gameplay arc. Indie titles like Spelunky use constructive methods to assemble platforming levels from a set of predefined room templates, ensuring each level is unique but guaranteed to be solvable.

Figure 4: Minecraft voxel terrain generated based on PCG and Perlin noise [4]

Game Assets and Items: PCG is also widely used to create individual game assets. The Borderlands franchise is famous for its procedural weapon system, which combines a variety of parts, materials, and statistical modifiers to generate "bazillions" of unique guns. L-systems (Lindenmayer systems) are often used in offline tools to generate realistic and varied foliage, such as trees and plants (Prusinkiewicz & Lindenmayer, 1990). Similarly, noise functions like Perlin or Simplex noise are fundamental to generating natural-looking textures for terrain, water, and other materials.

Narrative and Quests: While more complex, PCG is also being applied to dynamic storytelling. Games like Wildermyth procedurally generate character backstories, rivalries, and overarching narratives that are unique to each playthrough, creating

a deeply personal experience. Even simpler systems in larger games can procedurally generate "radiant quests," combining a quest template (e.g., "fetch item X from location Y") with specific characters and locations to create a near-endless supply of side content.

2.2 Constraint Satisfaction Problems (CSP)

While many Procedural Content Generation (PCG) techniques rely on direct, constructive processes like noise functions or grammatical expansion, a powerful subset of generative methods is based on a more declarative approach. Instead of specifying how to build the content step-by-step, a designer specifies a set of rules or constraints that the final content must obey. A separate, general-purpose solver is then used to find a solution that satisfies all of these constraints. This paradigm is formally known in computer science as a Constraint Satisfaction Problem (CSP). The Wave Function Collapse algorithm is, at its core, a specialized and heuristic-driven solver for a specific type of CSP (Karth & Smith, 2017). Understanding the principles of CSPs is therefore crucial to understanding how WFC operates.

2.2.1 Formal Definition

A Constraint Satisfaction Problem can be formally defined as a triplet consisting of a set of variables, their corresponding domains of possible values, and a set of constraints that restrict the values these variables can take (Russell & Norvig, 2021).

Variables (X): This is a finite set of variables, $X = \{X_1, X_2, ..., X_n\}$, which represent the entities for which we need to assign values.
Domains (D): For each variable X_i, there is a corresponding domain D_i which is a finite set of all possible values that X_i can be assigned.
Constraints (C): This is a set of constraints, $C = \{C_1, C_2, ..., C_m\}$, that specify the allowable combinations of values for some subset of the variables.

A solution to a CSP is a complete and consistent assignment of a value to every variable, where "complete" means every variable has a value, and "consistent" means this assignment does not violate any of the constraints. The goal of a CSP solver is to find one such solution.

2.2.2 Solvers and Heuristics

To make the abstract definition of a CSP and the process of solving it concrete, consider the familiar puzzle of Sudoku. The provided image illustrates this process perfectly on a small 3x3 subgrid.

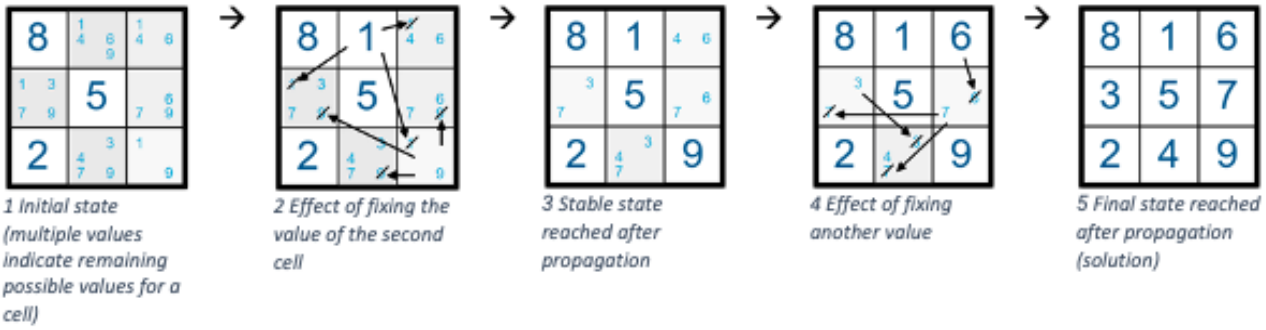


Figure 5: A 5-step diagram showing the process of solving a 3x3 Sudoku-like grid using constraint propagation [5]

We can map this Sudoku problem directly to our formal CSP definition:

- Variables: The 9 cells in the grid are the variables.
- Domains: For the unassigned (gray) cells, the initial domain is the set of all possible numbers that could go there based on the starting state. For example, in Panel 1, the top-middle cell has an initial domain of {1, 4, 6, 9}.
- Constraints: The primary constraint in this 3x3 example is an "all-different" constraint: no two cells in the same row or column can have the same number.

The process of solving this puzzle, as shown in Figure 5, demonstrates the core concepts of CSP solvers:

Panel 1: Initial State
This represents the initial setup of the problem. Some variables (cells) are already assigned (8, 5, 2), while others are unassigned. The small numbers in the gray cells represent the current domains for those variables, the full set of possibilities. This is analogous to the initial state of a WFC grid, where every cell is in a "superposition" of all possible tiles.

Panel 2 & 3: Assignment and Propagation
In Panel 2, a decision is made: the top-right cell is assigned the value 1. This is the assignment step. Immediately, constraint propagation begins. Because the top-right cell is now 1, the "all-different" constraint is enforced. The value 1 is removed from the domain of every other cell in the top row and the rightmost column. This is shown by the arrows and the crossed-out numbers. For instance, the domain of the cell below it, which was {6, 9}, becomes just {6, 9} (a typo in the original image, it should have been {6,7,9} becoming {6,7,9}), and the domain of the cell to its left, {1, 4, 6, 9}, becomes {4, 6, 9}. This propagation continues until the system reaches a new stable state (Panel 3), where no more values can be eliminated based on the current assignments. This process of using constraints to prune the domains of neighboring variables is known as enforcing arc consistency.

Panel 4 & 5: Iterative Solving
The process repeats. In Panel 4, another cell is assigned a value (6). This new assignment triggers another wave of constraint propagation. The value 6 is removed from the domains of its peers, which in turn causes the domain of the top-middle cell to shrink from {4, 6} to just {4}. This iterative cycle of assigning a value (or "collapsing a cell") and then propagating constraints is the fundamental loop of many CSP solvers, including WFC. This continues until every variable's domain has been reduced to a single value, resulting in the final, consistent solution shown in Panel 5.

The key takeaway is that WFC operates on this exact principle. Each cell in its grid is a variable, the tileset is the domain, and the adjacency rules are the constraints. At each step, it chooses a cell to "collapse" (assign a tile to) and then propagates the constraints of that choice to its neighbors, reducing their possible tile domains until a complete and coherent map is formed.

Heuristics for Efficiency
Because the search space for CSPs can be enormous, heuristics are used to guide the search process intelligently. Two key heuristics are directly applicable to WFC:

Variable Ordering (Which variable to assign next?): The Minimum Remaining Values (MRV) heuristic is a common and effective strategy. It advises selecting the variable that has the fewest legal values remaining in its domain (Haralick & Elliott, 1980). The intuition is that this choice is the most constrained and is most likely to cause a failure early, thus pruning large parts of the search space. This is

the single most important heuristic in WFC. The algorithm's concept of selecting the cell with the "lowest entropy" is a direct implementation of the MRV heuristic.

Value Ordering (Which value to assign first?): Once a variable is selected, the Least Constraining Value heuristic suggests choosing the value that rules out the fewest choices for the neighboring variables. This tries to keep options open for the future. WFC typically uses a simpler, stochastic approach here: it makes a weighted random choice from the remaining legal values, where the weights are often based on the frequency of that value (or pattern) in an input example. This randomness is what allows WFC to produce different valid outputs each time it is run.

2.3 The Wave Function Collapse (WFC) Algorithm

Now that we've covered the basics of Procedural Content Generation and the logic of Constraint Satisfaction Problems, we can zoom in on the main event: the Wave Function Collapse (WFC) algorithm. WFC is a powerful PCG method that's especially good at making sure every small piece of its output fits perfectly with its neighbors, all based on a given set of rules.
It's important to know that this isn't machine learning; it doesn't "learn" from looking at data. Instead, think of it as a clever puzzle solver. It uses a fascinating analogy from quantum mechanics to describe its process: it starts with a state of pure uncertainty where anything is possible, and step-by-step, it collapses those possibilities down to a single, concrete result.

2.3.1 Conceptual Origins: From Quantum Mechanics to Constraint Solving

The name "Wave Function Collapse" sounds complicated, but it's really just a helpful story borrowed from quantum physics. Let's be clear: this algorithm has nothing to do with actual quantum mechanics. There are no complex numbers, no Schrödinger's equation, none of the heavy physics. Instead, it just borrows the language and ideas from quantum measurement to give us an intuitive way to think about what's really going on. This whole analogy is built on three key ideas:

The Wave Function (*Superposition***)**: In quantum mechanics, a particle's state (e.g., its position or spin) is described by a wave function, which represents a probability distribution of all its possible states. Until a measurement is made, the particle is said to be in a superposition of all these states simultaneously. In WFC, each cell in the output grid has its own "wave function." This is simply the list of

all possible tiles that the cell could become, corresponding directly to the domain of a variable in a CSP. As long as a cell's domain contains more than one possible tile, it is considered to be in a state of superposition; it is, conceptually, all of those tiles at once. For each possible tile, the system maintains a state (typically a boolean true, indicating it is still a possibility) which can be thought of as a "coefficient" in the wave function.

Observation (*Measurement*): In quantum mechanics, the act of observing or measuring a particle forces its wave function to "collapse" into a single, definite state. The particle instantly resolves from a superposition of possibilities into one concrete reality. In WFC, the observation is an algorithmic decision. At each step, the algorithm selects one cell from the grid (based on a heuristic described later) and collapses its wave function. This collapse is the act of choosing a single tile from that cell's domain and discarding all other possibilities. This is the assignment step in a CSP. The cell transitions from a state of uncertainty (e.g., its domain is {'grass', 'sand', 'water'}) to a state of certainty (e.g., its domain is now just {'water'}).

Entanglement (*Propagation*): In quantum mechanics, entangled particles are linked in such a way that the state of one instantly influences the state of the other, regardless of the distance between them. WFC uses a local, classical version of this concept in its propagation phase. When a cell's wave function collapses, its state becomes certain. This certainty has immediate implications for its neighbors. The constraints (adjacency rules) that connect the cells act as the medium of this "entanglement." The collapse of one cell forces its neighbors to update their own wave functions, removing possibilities that are no longer valid. This chain reaction, where one collapse triggers changes in neighboring cells, which in turn may trigger changes in their neighbors, is the propagation of information through the system.

2.3.2 The Core Loop: An Iterative Process of Observation and Propagation

The power of the Wave Function Collapse algorithm lies in its elegant and efficient core loop. Once the tileset and adjacency constraints are defined, the generator executes a cycle that systematically reduces the uncertainty across the grid, moving from a state of total possibility to a single, coherent result. This process consists of three main stages: a one-time initialization, followed by a repeating cycle of Observation and Propagation that continues until a final state is reached.

Step 1: Initialization
Before the main loop begins, the system must be initialized. An output grid of the desired dimensions is created. At this initial moment, the system is in a state of maximum uncertainty. Each cell in this grid is initialized to be in a superposition of all possible tiles defined in the tileset. This means the "wave function," or domain, of every single cell contains a complete list of every available tile. Conceptually, every cell is simultaneously every tile it could possibly be. This state represents the highest possible entropy for the system; no information has yet been used to constrain the possibilities.

Step 2: The Iterative Loop
Once initialized, the algorithm enters its core iterative loop, which will repeat until the grid is fully generated or a contradiction is found. Each iteration of this loop consists of two distinct phases: Observation and Propagation.

A. Observation: Collapsing Uncertainty via Minimum Entropy

The first and most critical phase of the loop is the Observation. The algorithm must intelligently select one cell from the entire grid to "collapse", that is, to force it out of its superposition and assign it a single, definite tile. A naive approach might be to pick a cell at random, but this is highly inefficient. Instead, WFC employs a powerful heuristic to make the most informative choice possible. This is where the concept of entropy becomes central.
In the context of WFC, entropy is a measure of a cell's uncertainty. It is derived from the formal concept of Shannon entropy from information theory, which quantifies the amount of "surprise" or information in a variable's possible outcomes. For our purposes, we can understand it more simply:

- High Entropy: A cell with many possible tiles remaining in its domain is in a state of high uncertainty and thus has high entropy.
- Low Entropy: A cell with very few possible tiles remaining is in a state of low uncertainty and has low entropy.
- Zero Entropy: A cell that has already been collapsed and has only one possible tile has no uncertainty and an entropy of zero.

The guiding heuristic of the Observation phase is to find the cell with the minimum non-zero entropy. This is a direct implementation of the Minimum Remaining Values (MRV) heuristic from the study of Constraint Satisfaction Problems. The intuition behind this choice is that the cell with the fewest

remaining options is the most constrained; it is the "tightest spot" in the puzzle. Making a decision at this point is the most efficient course of action for two reasons:
It has the highest chance of revealing a contradiction early if the system is heading towards an impossible state.
It will have the most significant and immediate constraining effect on its neighbors, leading to the most effective pruning of the overall search space.

Once this lowest-entropy cell is identified, the final step of Observation is to collapse it. A single tile is chosen from its small remaining domain. This choice is typically a weighted random selection, where the weights are based on the predefined frequencies of the tiles in the tileset, allowing for artistic control over how often certain tiles appear. With this choice, the cell's state becomes definite.

B. Propagation: Enforcing Local Consistency

The immediate consequence of an Observation is Propagation. The collapse of one cell provides new, concrete information that must be broadcast to its neighbors, forcing them to update their own states to remain consistent. This phase is a practical application of the arc consistency algorithm.

The propagation process works as a cascading chain reaction:

When a cell C is collapsed to a specific tile T, its immediate neighbors are flagged for an update.
The algorithm then examines a flagged neighbor, cell N. It looks at the adjacency rules for tile T in cell C.
For every tile in N's current domain, the algorithm checks: "Is this tile allowed to be adjacent to tile T?"
Any tile in N's domain that violates the adjacency rule is removed. This act of removing possibilities reduces cell N's entropy.
If N's domain was changed during this check (i.e., if at least one tile was removed), it means new information has been established. Therefore, all of N's other neighbors are now also flagged for an update, as this change might affect them.
This process continues, rippling outwards from the site of the original collapse, until no more domains can be reduced. The system then settles into a new, stable state, ready for the next Observation.

Step 3: Termination

The core loop of Observation and Propagation repeats until one of two termination conditions is met:

Success: Every cell in the grid has been collapsed to a single tile (i.e., all cells have an entropy of zero). The result is a complete, fully generated map that is guaranteed to be locally consistent according to all the defined adjacency constraints.
Failure: During the Propagation phase, the domain of a cell is reduced to zero possibilities. This means there is no valid tile that can be placed in that cell, creating a contradiction. The constraints are unsolvable from the current state. The algorithm terminates and reports a failure.

2.3.3 The Role of Entropy in WFC

At every step of its execution, the Wave Function Collapse algorithm is faced with a critical decision: which of the many cells still in superposition should be collapsed next? The choice is fundamental to the algorithm's efficiency and success. A random choice would be aimless, often leading to trivial areas being solved first while complex, constrained areas are left until the end, increasing the likelihood of contradictions. To solve this, WFC employs a powerful and intelligent heuristic to guide its search: it always chooses the cell with the minimum entropy. This principle is the strategic heart of the algorithm, ensuring that it always makes the most informed and impactful decision possible at every step.

In this context, entropy is a direct application of Shannon entropy, a foundational concept from information theory used to measure the uncertainty or "information content" of a probability distribution (Shannon, 1948). The formal equation for Shannon entropy is:

$$entropy = -\Sigma p_i log(p_i)$$

To understand its role in WFC, we can break down this formula in the context of a single cell's wave function:
$\mathbf{p_i}$ (The Probability of a Tile): This term represents the probability of a specific tile i being chosen from the cell's current domain of possibilities. In WFC, this probability is determined by user-defined weights. If a tile i has a weight w_i, and

the sum of the weights of all currently possible tiles in the cell's domain is Σw, then $p_i = w_i / \Sigma w$. A tile with a higher weight is more likely to be chosen.
$\log(p_i)$ (The "Surprisal"): This term quantifies the amount of information or "surprise" associated with an outcome. A very likely outcome (high p_i) has low surprisal, while a very unlikely outcome (low p_i) has high surprisal. The logarithm captures this inverse relationship.
-Σ (The Summation): The formula sums the probability-weighted surprisal over all possible tiles i in the cell's domain. The negative sign ensures the final entropy value is non-negative. The result, entropy, is a single number that represents the total average uncertainty for that cell. A high value means the cell is highly uncertain (many likely options remain), while a low value means it is close to being resolved.

From Theory to Practice: A Crucial Simplification

While the full Shannon entropy formula provides the most accurate measure of a cell's uncertainty, many practical implementations of WFC, including the one described in the source provided (Boris the Brave, 2020), use a highly effective simplification.

Consider the case where all tiles have a uniform weight (e.g., a weight of 1). If a cell has N possible tiles remaining in its domain, the probability for each is $p_i = 1/N$. In this scenario, the Shannon entropy formula simplifies to log(N). Because the logarithm is a monotonically increasing function, finding the cell with the minimum log(N) is mathematically identical to finding the cell with the minimum N.

Therefore, in its most common implementation, the "minimum entropy" heuristic is simplified to the Minimum Remaining Values (MRV) heuristic: at each step, the algorithm simply finds the cell with the lowest number of possible tiles remaining in its domain. This approach is computationally cheaper than calculating the full entropy for every cell, yet it perfectly preserves the core strategy of tackling the most constrained part of the problem first. The full Shannon calculation only becomes truly advantageous in systems with highly variable tile weights, as it can more accurately distinguish the uncertainty between, for example, a cell with two equally likely options and a cell with one very likely and one very unlikely option.

Ultimately, the role of the entropy heuristic is to make the greedy, non-backtracking nature of WFC viable. By always collapsing the cell where the choice is most restricted, the algorithm rapidly prunes the search space and

propagates constraints in the most effective way possible. It ensures that the path taken through the vast space of possibilities is not random but is instead a focused, intelligent, and efficient convergence toward a valid, coherent solution.

2.4 Implementation Models of Wave Function Collapse

The Wave Function Collapse algorithm is a powerful idea for generating content, but it's useless without a rulebook. The key question is: where do those rules come from? The algorithm needs to know which tiles are allowed to be placed next to each other.

Essentially, there are two main ways to create this rulebook. These two approaches, the Tiled Model and the Overlapping Model, represent a fundamental choice that shapes the entire process. This decision affects your workflow, how much creative control you have, and the final look and feel of what you generate. Choosing between them is one of the first and most important decisions you'll make when building a WFC generator.

2.4.1 The Tiled Model

The Tiled Model, also referred to as the Simple Tiled Model, operates on a principle of explicit, manual definition. In this approach, the developer provides the generator with a discrete, predefined set of tiles and a meticulously crafted set of adjacency rules. This is analogous to designing a puzzle; the developer provides not only the unique puzzle pieces but also the complete instruction manual that dictates precisely how they are allowed to connect. The generator’s task is to find a valid assembly of these pieces according to the provided rules. This model grants the designer the highest possible degree of control over the final output, as every potential interaction between tiles is explicitly permitted or forbidden.

The process is clearly illustrated in the provided example of a road-tile system. The set of seven unique tile types shown in part (a) (Fig.6) represents the complete tileset, the fundamental building blocks of the world. The core of the model is detailed in part (b) (Fig.6), the adjacency table. For each individual tile type, this table explicitly lists every other tile type that is permitted to be its neighbor in each of the four cardinal directions. This rule set is absolute and unambiguous. For example, a straight road piece is only allowed to have other straight road pieces or specific corner pieces adjacent to it in the direction of the

road. Finally, part (c) (Fig.6) demonstrates the generative process in action, where the algorithm iteratively places tiles that satisfy these local constraints, gradually resolving the grid from a state of total possibility into a single, coherent layout. The primary advantage of this model is its unparalleled control, making it ideal for generating content with strict functional or logical requirements, such as solvable puzzle levels (Salcedo et al., 2024), well-formed building interiors, or circuit-like patterns. Its main disadvantage, however, is the labor-intensive nature of creating the rule set, which can become exponentially more complex and prone to human error as the size of the tileset increases.

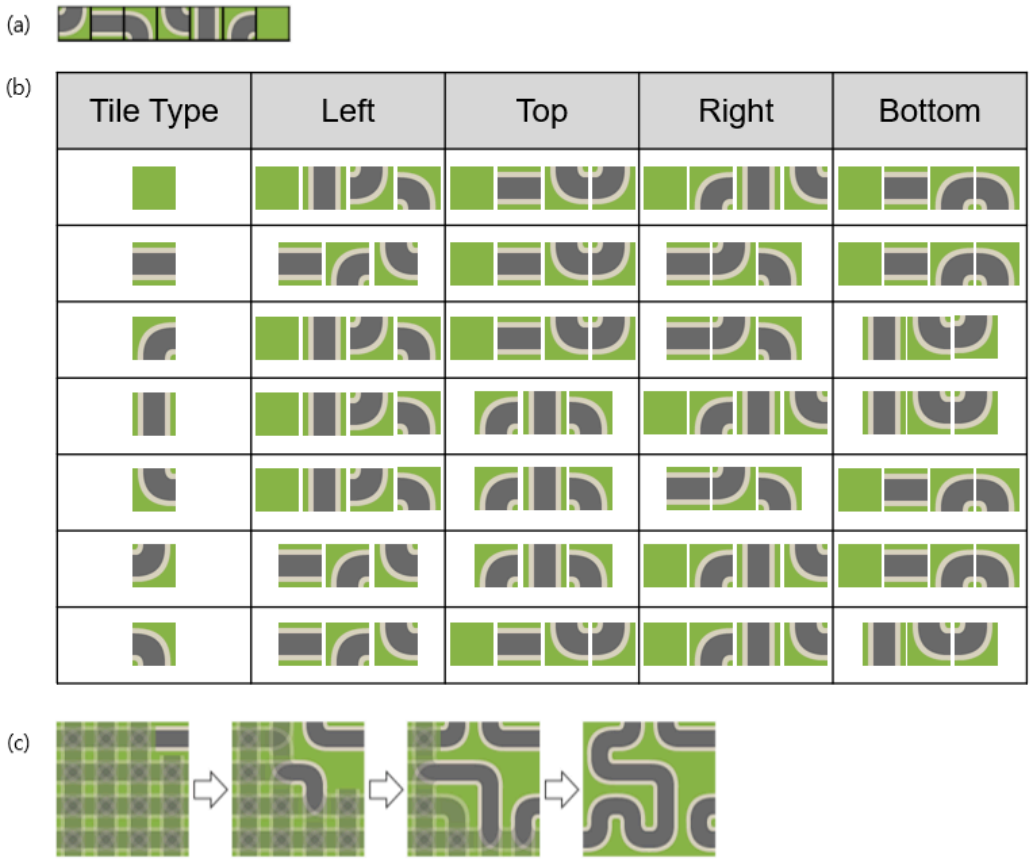


Figure 6: WFC simple tiled model example. (a) Seven types of tiles, (b) The connectivity of the tiles. Each row for individual tile types shows which tiles can be connected to the left/above/right/bottom of each tile. (c) Resulting image that gradually becomes clearer when tiles are placed [6].

2.4.2 The Overlapping Model

In contrast to the explicit nature of the Tiled Model, the Overlapping Model operates on a principle of implicit, learned constraints. Instead of manually defining a set of rules, the developer provides the algorithm with an example input, such as an image or an existing tilemap. The algorithm then analyzes this sample and automatically derives the tiles (referred to as patterns) and their adjacency rules. This shifts the developer's role from that of a rule architect to a curator of examples. The core constraint it learns is that any small, local arrangement of patterns in the output must also have existed somewhere in the input.

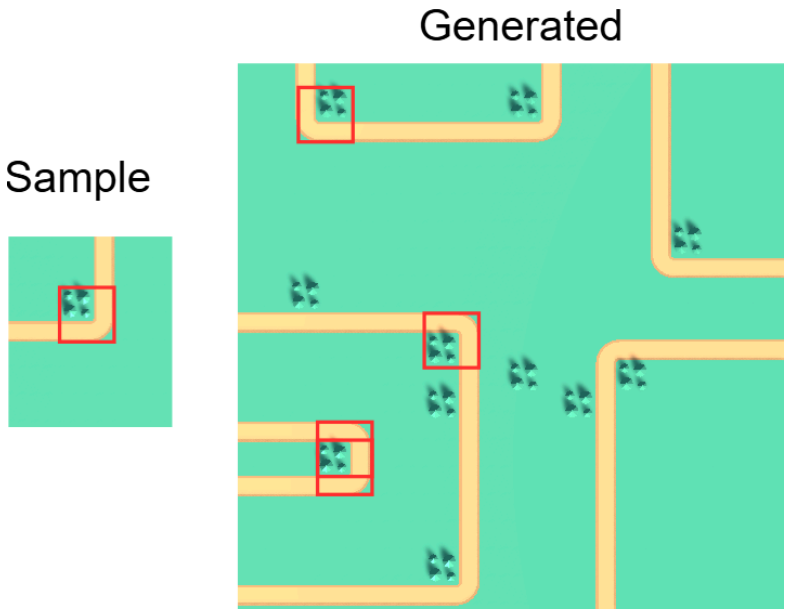


Figure 7: On the left, a small 6x7 map of tiles has been specified as the sample, and a larger map has been made from it. Every 2x2 rectangle in the output comes from somewhere in the input, with the red box showing one particular patch [7].

This process is exemplified in the provided image of a forest path (Fig. 7). The small map on the left serves as the input sample. The algorithm analyzes this sample by sliding a small N×N window over it, cataloging every unique pattern it observes. The larger map on the right is the generated output. The fundamental rule it enforces is that every N×N pattern of tiles in the output must be a direct copy of a pattern found in the input. The red boxes illustrate this perfectly: the 2x2 pattern of a corner path highlighted in the output is valid precisely because that exact 2x2 pattern can be found in the sample. The size of this sliding window,

N, is the most critical parameter in this model. A small N (e.g., 2) captures only very simple, low-level features and adjacencies, leading to more varied and potentially chaotic outputs. A larger N captures more complex and specific arrangements, resulting in an output that is more structured and repetitive, closely resembling larger chunks of the input sample. The advantage of the Overlapping Model is its ability to effortlessly capture the complex, subtle, and organic relationships present in a sample that would be nearly impossible to define by hand (Boris the Brave, n.d.). Its main drawback is the corresponding lack of direct control; the generator will faithfully reproduce any undesirable artifacts present in the sample, and the only way to forbid a specific adjacency is to meticulously edit the source material to remove all instances of it.

2.4.3 A Comparative Analysis: Tiled vs. Overlapping Models

While the primary distinction between the Tiled and Overlapping models is often framed by their workflow, explicit rules versus learned examples, a more fundamental structural difference lies in how they define the relationships, or adjacencies, between the slots in the output grid. This conceptual difference is clearly visualized in the provided diagram (Fig. 8), revealing that the true divergence between the models is the density and complexity of their underlying constraint graph. The choice between them is not merely about control versus automation, but about selecting the very nature of the relationships that will define the generated world.

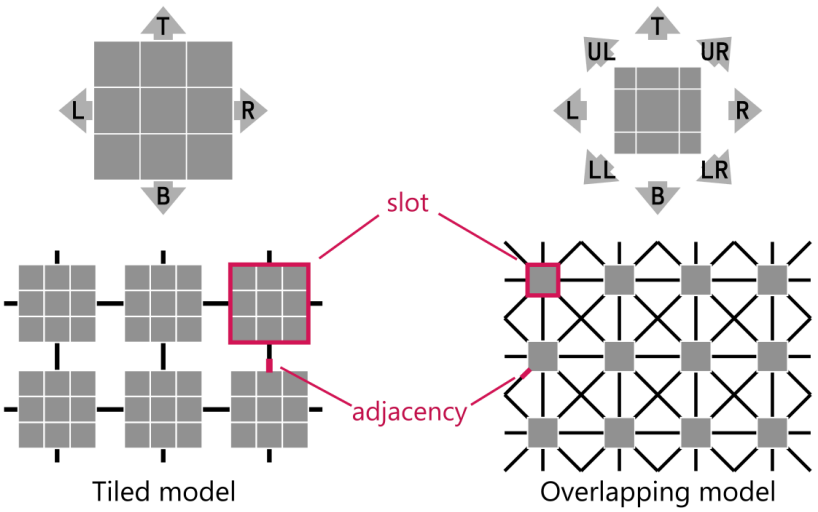


Figure 8: Diagram showcasing the difference between the two models [8]

The Tiled model operates on a simple and intuitive grid of constraints. As depicted, each slot has a direct relationship with only its four cardinal neighbors: top, left, right, and bottom. The constraints are binary and local; the validity of a tile in one slot is determined by a simple lookup against the tiles in its immediately adjacent slots. This results in a sparse and predictable graph of dependencies, where the rules are straightforward for a designer to define and reason about. The adjacencies are clean, one-to-one connections between the edges of discrete tiles, making the system highly controllable and well-suited for creating content with clear, grid-aligned structures.

The Overlapping model, on the other hand, is a different beast entirely. It creates a much denser and more complex web of rules. Instead of looking at one tile at a time, its basic building block is a small pattern of tiles, like a 2x2 square. This means that for a spot to be valid, it has to consider a much larger group of neighbors.

As the diagram shows for a 2x2 pattern, each spot isn't just checked against its four immediate neighbors, but all eight of them, including the diagonals. This happens because the patterns themselves overlap. The "connection" isn't just a single edge; it's a shared region where all the surrounding patterns must agree.

This tangled web of connections is exactly what gives the Overlapping model its power. It can pick up on the subtle, organic details in a source image, like diagonal lines, complex textures, and other arrangements that the simpler Tiled model would never see. And the bigger the pattern you use, the more complex these rules become, with each spot's decision influencing an even wider area.

This incredible expressiveness comes at a price, though: it's more computationally expensive to get started. Ultimately, the choice comes down to this: The Tiled model offers simplicity, direct control, and predictable results through its straightforward rules. In contrast, the Overlapping model provides powerful automation and surprising complexity by figuring out a dense, intricate web of relationships all on its own from an example.

3 System Design and Concept

Now that we've covered the theory behind PCG and Constraint Satisfaction Problems, it's time to shift from the abstract to the concrete. This chapter dives into the actual design of the 2D map generator I built for this thesis. My main goal was to create a powerful yet intuitive tool in Unity for making coherent, tile-based maps, all powered by the Wave Function Collapse (WFC) algorithm, a technique that can build complex worlds from a simple set of rules.

In this chapter, I'll walk you through the system's architecture. We'll start with a bird's-eye view of how the generator is structured and how it works, explaining how all the pieces, the tiles, the grid, and the WFC solver fit together. After that, we'll get into the details of the data, looking at how the tiles and their connection rules are defined. Finally, we'll see how the WFC algorithm itself was adapted for this project, including how the grid is set up, how the step-by-step generation cycle works, and how map boundaries are handled. Throughout the design, my focus was on making a system that is not only clever on the inside but also practical and easy for a game designer to use.

3.1 Overview of the 2D Map Generation System

At its core, the 2D map generation system is a practical implementation of the Tiled Model of the Wave Function Collapse algorithm, tailored for the Unity engine. The system is designed to transform a simple set of user-defined rules into complex and varied game maps, shifting the creative burden from manual placement of every tile to the design of the tiles' interaction rules. This approach empowers a designer to think in terms of local possibilities, which tiles can appear next to each other, and allows the algorithm to explore the vast combinatorial space of those possibilities to produce a coherent global outcome.

The system is built upon three fundamental components:

Tiles: These are the basic building blocks of the generated map. In the context of the Unity project, each tile is a Prefab containing not only its visual representation (a sprite) but also the crucial adjacency information. This is managed by a Tile.cs script, which holds references to all possible neighbors in the four cardinal directions (up, right, down, and left).

The Grid: The generation process takes place on a two-dimensional grid, where each position is represented by a Cell. Each Cell object is managed by the Cell.cs script acts as a variable in a Constraint Satisfaction Problem. Initially, each cell is in a state of superposition, meaning it holds a list of all possible tiles it could become.

The WaveFunction Engine: This is the central orchestrator of the algorithm, implemented in the WaveFunction.cs script. It initializes the grid, and then iteratively performs the core WFC loop: it identifies the cell with the lowest entropy (the fewest remaining tile possibilities), "collapses" it by choosing one tile from its list, and then propagates the consequences of this choice to all neighboring cells, reducing their possibilities to maintain local consistency. This cycle repeats until every cell on the grid has been collapsed into a single, definite tile, resulting in a complete map (Fig. 9).

The entire process is designed to be executed within the Unity editor, providing an efficient workflow for level design and content creation. The designer provides the system with a collection of tile prefabs and, after configuring their adjacency rules, can generate a complete and valid map with a single command. The output is a visually realized grid of instantiated tile prefabs, ready for use in a game.

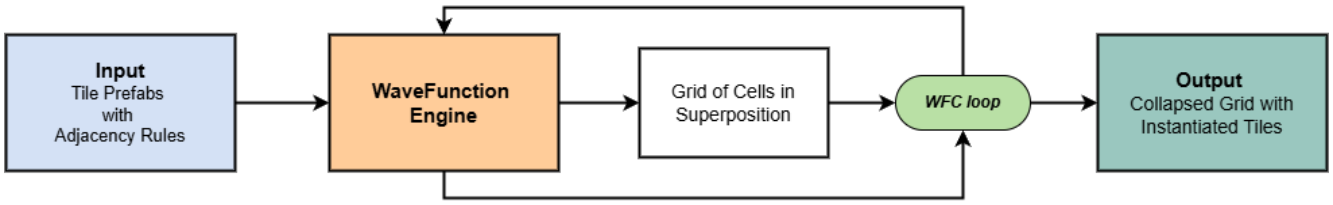


Figure 9: Diagram representing a simplified version of the main system

3.2 Tile-Based Data Representation

The effectiveness of the Wave Function Collapse algorithm is fundamentally dependent on the clarity and efficiency of its underlying data structures. The representation of tiles and their corresponding adjacency rules dictates the expressive power of the generator, the level of control afforded to the designer, and the computational performance of the system. The design for this project employs a two-tiered approach to data representation: a high-level conceptual framework for organizing tiles into logical groups, and a low-level, direct-authoring system for defining the specific adjacency constraints.

3.2.1 Tile Categorization and the 3x3 Positional System

A simple, flat list of tiles with arbitrary connections is often insufficient for generating structured and recognizable features like lakes, castles, or forests. To create these larger, coherent "biomes," a higher-level organizational principle is required. This project introduces a positional naming convention that embeds a tile's structural role directly into its identity. Tiles are first categorized into thematic sets (e.g., Lake, Mountain, Castle) and then assigned a position within a conceptual 3x3 grid layout.

This name_position system uses two-letter codes to denote a tile's placement within its 3x3 block (Fig. 10):

UL, UM, UR:
Up-Left, Up-Middle, Up-Right

ML, MM, MR:
Middle-Left, Middle-Middle, Middle-Right

DL, DM, DR:
Down-Left, Down-Middle, Down-Right

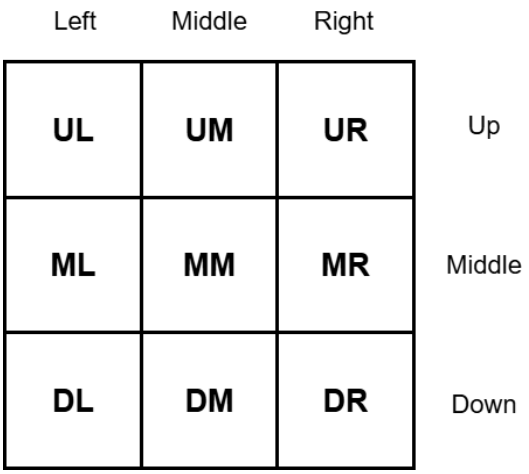


Figure 10: 3x3 position system of tile placement

For example, a tile named Lake_UL is not merely a generic water tile; it is specifically the top-left corner of a lake structure. Similarly, Lake_MM represents the center, and Lake_DR represents the bottom-right corner. This methodology enriches the information content of each tile, transforming them from simple atomic units into components of a larger, predefined pattern. By defining adjacencies between these positional tiles, a designer can enforce the creation of complete 3x3 structures. For instance, the rules will dictate that a Lake_UM tile must have a Lake_MM tile below it, ensuring the lake structure maintains its integrity.

This system also allows for deliberate exceptions to create more varied and natural-looking features. As noted in the project's connectivity rules, the MiddleMiddle (MM) tile for the Mountain and Castle sets is a Grass tile. This design choice carves out open spaces within these large structures, preventing them from appearing as unnatural, solid blocks and allowing for more organic integration with the surrounding landscape.

3.2.2 Authoring Adjacency Constraints via Neighbor Lists

The low-level implementation of the adjacency rules is the practical core of this system's Tiled Model approach. The constraints are authored manually and explicitly by the designer, providing unambiguous control over the generator's output. This is achieved through a C# script, Tile.cs, which is attached as a component to every tile prefab in the Unity project.
The Tile.cs script contains four public lists of Tile objects, one for each cardinal direction:

upNeighbours[] - tile objects for the upper neighbours
rightNeighbours[] - tile objects for the right neighbours
downNeighbours[] - tile objects for the down neighbours
leftNeighbours[] - tile objects for the left neighbours

The workflow for defining constraints is both direct and intuitive. Within the Unity Inspector, a designer can select a tile prefab (e.g., Lake_UM) and physically drag-and-drop other tile prefabs into its respective neighbor lists. For example, to enforce the rule that a Lake_UM tile can have a Lake_UL to its left or another Lake_UM, the designer would drag the Lake_UL and Lake_UM prefabs into the leftNeighbours list of the Lake_UM prefab.
This method translates the abstract connectivity tables, such as those provided for the Lake and Mountain tiles, into concrete, machine-readable data. The row for

"UM" in the Lake tile table, which allows "UL, UM" as valid left neighbors, corresponds directly to the contents of the leftNeighbours list on the Lake_UM prefab. This component-based approach makes the system highly modular and easy to iterate upon. Rules can be modified, and new tiles can be integrated, simply by updating these lists in the Unity Inspector (Fig. 11), after which the generator can be re-run to immediately observe the impact of the changes.

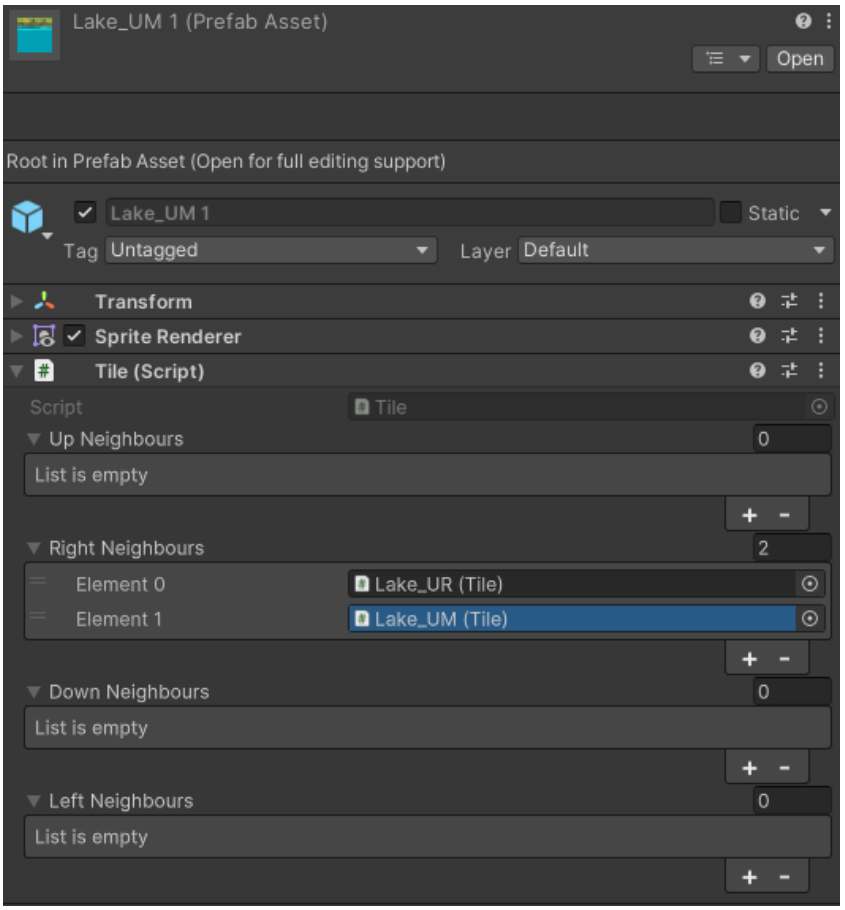


Figure 11: Unity Inspector view for a "Lake_UM" tile prefab with the drag and drop option selected on the right neighbours

3.2.3 Tile Combinations

While the Tile.cs script provides the technical framework for storing adjacency rules, the strategic logic that governs the formation of coherent structures is best understood by examining a concrete tile combination table. These tables serve as the design blueprint for the generator, explicitly defining the local relationships that, when combined, produce predictable global patterns. The Lake tile set provides a clear and comprehensive example of this principle in action.

The connectivity table for Lake tiles is as follows:

Tile/Position	Top	Right	Down	Left
UL	G	UM, UR	ML, DL	G
UM	G	UR, UM	MM, DM	UL, UM
UR	G	G	MR, DR	UM, UL
ML	ML, UL	MM, MR	ML, DL	G
MM	MM, UM	MM, MR	MM, DM	MM, ML
MR	MR, UR	G	MR, DR	MM, ML
DL	ML, UL	DM, DR	G	G
DM	MM, UM	DM, DR	G	DM, DL
DR	MR, UR	G	G	DM, DL

Table 1: Connectivity table for Lake tiles Note: G stands for grass tiles

This table (Tbl. 1) dictates every valid placement for a Lake tile relative to its neighbors. Each row represents a specific tile (e.g., Lake_UM), and the columns specify the list of tiles that are permitted to be placed in the adjacent cell in that direction. Clarification that the G letter on the table stands for grass tiles.

Figure 12: Lake tiles with their corresponding tile position

Examples of these tiles, followed by illustrations

To illustrate how these rules function, we can analyze a few key examples:

Example 1: A Corner Tile (Lake_UL)
The UL (Up-Left) tile represents the top-left corner of a lake formation. Its rules are highly constrained to enforce this role.
Top: "G" and Left: "G": The only valid neighbor above or to the left of a Lake_UL tile is a Grass tile. This is a critical boundary rule; it ensures that the lake's corner correctly terminates against the surrounding landscape, preventing other lake tiles from being placed out of sequence.
Right: "UM, UR": The tile to the right must be either a Lake_UM (the top-middle edge) or another Lake_UR. This rule forces the top edge of the lake to continue correctly.
Down: "ML, DL": The tile below must be either a Lake_ML (the middle-left edge) or a Lake_DL (the bottom-left corner), ensuring the left edge of the lake continues downwards.

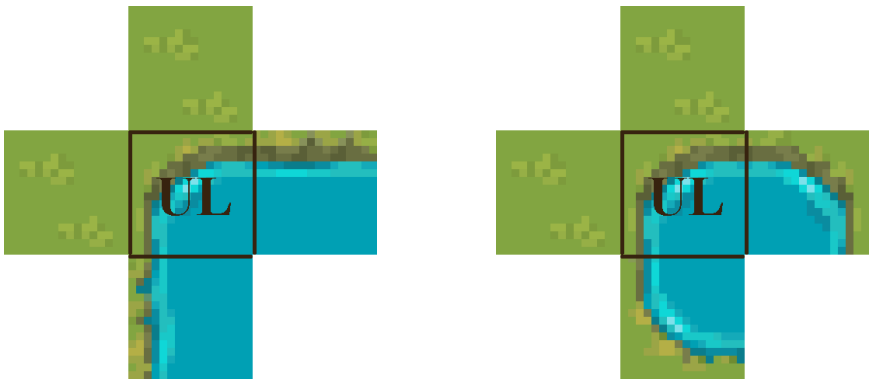


Figure 13: Visual representation of two examples of Lake tile UL combinations

Example 2: A Central Tile (Lake_MM)
The MM (Middle-Middle) tile is the most flexible, designed to form the main body of the lake.
Top: "MM, UM": It can have another Lake_MM above it (allowing the lake's center to expand vertically) or a Lake_UM (allowing it to connect to the top edge).
Right: "MM, MR": Similarly, it can connect to another Lake_MM to expand horizontally or to a Lake_MR to connect to the right edge.

The same logic applies to its Down ("MM, DM") and Left ("MM, ML") neighbors, giving it the necessary flexibility to form lakes of varying sizes and shapes.

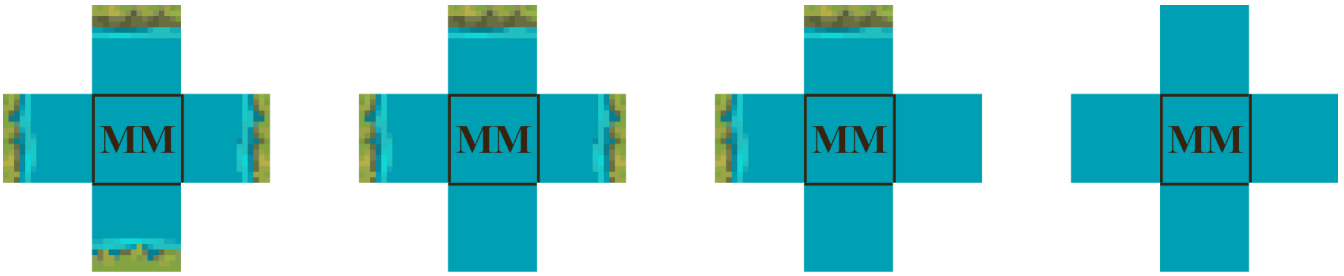


Figure 14: Visually representation of four examples of Lake tile MM combinations

Example 3: An Edge Tile (Lake_DM)
The DM (Down-Middle) tile represents the bottom-middle edge of the lake.
Top: "MM, UM": It must connect upwards to the main body (Lake_MM) or the upper edge (Lake_UM) of the lake.
Down: "G": The only valid neighbor below it is Grass, enforcing the bottom boundary of the lake.
Left/Right: "DM, DR" and "DM, DL": It can connect to other bottom-edge pieces, allowing the bottom edge of the lake to extend horizontally.
Through these meticulously defined, local constraints, the system guarantees that a coherent global structure will emerge. A Lake_UL tile can never be placed in the middle of a field on its own, because its rules demand that it be connected to other specific lake tiles. The Wave Function Collapse algorithm's role is to find a valid arrangement of tiles that satisfies every one of these local rules simultaneously across the entire grid, effectively "solving" the puzzle defined by this combination table.

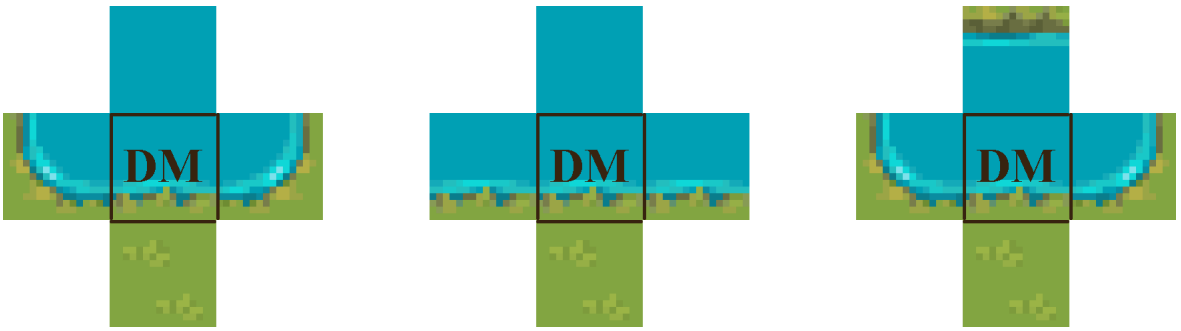


Figure 15: Visually representation of three examples of Lake tile DM combinations

3.3 The Wave Function Collapse Algorithm in a Grid

With the data representation for tiles and their constraints defined, the focus shifts to the dynamic process of generation itself. The Wave Function Collapse algorithm operates on a grid of cells, systematically reducing uncertainty at each step until a fully determined and coherent map is formed. This section details the adaptation and implementation of the WFC algorithm's core mechanics, from the initial state of total possibility to the final, collapsed output.

3.3.1 Initializing the Grid with Superpositions

Before the generation process can begin, the system must establish its initial state. This is a state of maximum uncertainty, where no decisions have been made and every outcome is still possible. The WaveFunction.cs script begins by instantiating a two-dimensional grid of Cell objects, corresponding to the desired dimensions of the output map.

Each Cell in this grid is initialized to be in a "superposition" of all possible tiles. In practical terms, the tileOptions list within each Cell.cs instance is populated with a complete copy of every tile prefab provided to the generator. This initial state is analogous to the setup of a Constraint Satisfaction Problem where the domain of every variable is the full set of all possible values. This represents the highest possible entropy for the system; every cell is simultaneously every possible tile, and no constraints have yet been applied. This foundational step creates the "blank canvas" of possibilities upon which the algorithm will work.

3.3.2 The Iterative Observation and Propagation Cycle

Once the grid is initialized, the WaveFunction engine enters its core iterative loop, which continues until every cell has been assigned a single tile. Each iteration of this loop consists of two distinct phases: Observation and Propagation. This cycle is the heart of the WFC algorithm, driving the system from a state of high entropy to a single, stable, and locally consistent solution.

A. Observation: Collapsing Uncertainty via Minimum Entropy

The first phase of each iteration is to make a decision. The algorithm must intelligently select one cell from the entire grid to "collapse," forcing it out of its superposition and into a single, definite state. A naive approach might be to

choose a cell at random, but this is inefficient and risks creating unsolvable contradictions late in the process.

Instead, the system employs the Minimum Remaining Values (MRV) heuristic, referred to in WFC terminology as selecting the cell with the "minimum entropy." The WaveFunction script iterates through every cell on the grid that has not yet been collapsed and identifies the one whose tileOptions list is the smallest (but greater than one). This cell is the most constrained position on the map, the "tightest spot" in the puzzle. Choosing this cell to collapse is the most strategically sound move because it is the most likely to reveal a contradiction early and has the most significant immediate impact on its neighbors, thus pruning the search space most effectively.

Once the lowest-entropy cell is identified, the Observation is completed by choosing a single tile from its tileOptions list. This selection is typically a random choice, which provides the necessary stochasticity for generating different maps on each run. The chosen tile becomes the cell's final, collapsed state, and all other options are discarded.

B. Propagation: Enforcing Local Consistency

The immediate consequence of an Observation is Propagation. The collapse of a single cell provides new, definitive information that must be broadcast to its neighbors to maintain consistency across the grid. This phase is a practical application of arc consistency, ensuring that the domains of neighboring variables only contain values that are compatible with the newly assigned variable.

The propagation process works as a cascading chain reaction, managed by the WaveFunction script:

- When a cell is collapsed to a specific tile, its immediate neighbors (up, down, left, and right) are flagged for an update.
- The algorithm examines a flagged neighbor. For each tile remaining in this neighbor's tileOptions list, it checks against the adjacency rules of the newly collapsed tile. For example, if a cell was collapsed to Lake_UM, the algorithm checks the tileOptions of the cell to its right. It removes any tile from that list that is not present in Lake_UM's rightNeighbours list.
- This act of removing possibilities reduces the neighbor's entropy. If the neighbor's tileOptions list was changed in any way, it means new information

has been established. Therefore, all of that neighbor's own neighbors are now also flagged for an update, as this change might affect them.

- This process continues, rippling outwards from the site of the original collapse, until no more tile options can be removed from any cell on the grid. The system then settles into a new, stable state, ready for the next Observation phase.

3.3.3 Boundary Handling and Edge Constraints

A critical consideration in a grid-based system is the handling of its boundaries. Cells at the edges and corners of the map have fewer than four neighbors, and the algorithm must account for this. The current implementation handles this implicitly: the propagation logic only attempts to update neighbors that actually exist. If a cell is on the rightmost edge of the map, there is no neighbor to its right to propagate constraints to, and the algorithm simply moves on.

However, a more explicit design choice for boundary handling is embedded in the tile connectivity rules themselves. By examining the adjacency tables, it is clear that the "G" (Grass) tile serves as a universal connector and a default boundary condition. For example, the Lake_UL tile lists "G" as a valid neighbor for its Top and Left directions. This means that a lake can form cleanly at the edge of the map, terminating against a border of grass rather than being unnaturally sliced off. This design ensures that complex biomes can be generated anywhere on the map, including its edges, without creating jarring visual artifacts. The Grass tile effectively acts as a "sea" or "padding" out of which more complex structures emerge, providing a natural and controllable method for managing the map's outer constraints.

4 Implementation

Moving from the conceptual framework detailed in the previous chapter, this chapter documents the practical implementation of the Wave Function Collapse map generator. It covers the translation of the system's design and architecture into a functional prototype within the Unity game engine. The focus will be on the concrete software engineering choices, the structure of the core scripts, and the workflow established for a designer to create and configure the tile sets used by the generation algorithm. This section will bridge the gap between the theoretical model and the tangible tool, detailing the "how" of its construction and operation.

4.1 Development Environment: Unity and C#

The choice of development environment is critical as it provides the foundational tools and frameworks upon which the system is built. The prototype was developed using the Unity game engine (version 2022.3 LTS) with all scripting implemented in the C# programming language. This combination was selected for its robust feature set, rapid prototyping capabilities, and suitability for the project's specific architectural needs.

Unity was particularly well-suited for this project for several key reasons. Its component-based architecture aligns perfectly with the system's design, allowing the Tile.cs script, which contains the adjacency rules, to be directly attached as a component to each tile's data structure. The engine's Prefab system provided an ideal framework for creating reusable tile assets, where each prefab encapsulates its visual sprite, its Tile.cs component, and any other necessary data. Furthermore, the visual and intuitive nature of the Unity Editor, particularly the Inspector window, was instrumental in realizing the designer-centric workflow for authoring adjacency constraints. The ability to simply drag-and-drop tile prefabs into the neighbor lists of another tile streamlined what would otherwise be a cumbersome and error-prone data entry process.

C#, as the primary scripting language for Unity, provided a modern, object-oriented, and type-safe environment for implementing the core WFC logic. Its comprehensive standard library and features, such as the generic List<T> collection, were essential for managing the dynamic tileOptions in each Cell and the neighbours lists in each Tile. The language provided the necessary

performance and control to efficiently execute the iterative, grid-based computations at the heart of the WFC algorithm.

The visual foundation for the generated maps is built upon a freely available asset pack: the "16x16 puny world tileset," sourced from the online digital art repository OpenGameArt.org[1]. To adapt and expand upon this base tileset to fit the specific requirements of the 3x3 positional system, the dedicated pixel art editor Aseprite was utilized. Aseprite facilitated the creation of new tile variations, such as isolating corner and edge pieces from larger sprites, ensuring a consistent color palette and pixel style across all assets, and exporting the final sprites in a format ready for import into the Unity project.

4.2 Core Scripting Architecture

The functionality of the map generator is distributed across three core C# scripts: Tile.cs, Cell.cs, and WaveFunction.cs. Each script has a distinct and well-defined responsibility, working in concert to execute the WFC algorithm. This separation of concerns creates a modular and maintainable codebase where the data, the state, and the logic are managed independently.

4.2.1 Tile.cs: The Tile Prefab and Rule Definition System

The Tile.cs script is the foundational data-holding component of the system. It is a MonoBehaviour designed to be attached directly to each tile prefab. This script effectively transforms a simple visual asset into an intelligent "puzzle piece" that is aware of its own connection rules. Its primary purpose is to serve as the data container for the adjacency constraints that form the basis of the Tiled Model implementation.
The script's implementation is straightforward, containing four public fields, each a list of Tile objects:

```
public class Tile : MonoBehaviour
{
   public List<Tile> upNeighbours;
   public List<Tile> rightNeighbours;
   public List<Tile> downNeighbours;
   public List<Tile> leftNeighbours;
}
```

[1] Tile set from OpenGameArt.org. See: https://opengameart.org/content/16x16-puny-world-tileset

By declaring these lists as public, they are automatically exposed in the Unity Inspector when a tile prefab is selected. This design choice is central to the project's user-friendly workflow. It abstracts the complex logic of constraint definition into a simple, visual, drag-and-drop interface. A level designer does not need to write or modify any code to define the rules of the map; they can simply populate these lists by dragging other tile prefabs from the Project window into the appropriate slots.

This approach makes the entire rule system highly modular and scalable. Each tile prefab is a self-contained entity that defines its own valid neighbors. Adding a new tile to the system is as simple as creating a new prefab, attaching the Tile.cs script, and then updating the neighbor lists on both the new tile and any existing tiles it is intended to connect with. During the generation process, the WaveFunction.cs script reads this static, pre-configured data to determine which tiles are valid candidates during the propagation phase.

4.2.2 Cell.cs: Managing Grid State and Superposition

While Tile.cs defines the static rules of the system, the Cell.cs script manages the dynamic state of each individual position on the grid during the generation process. Each Cell object represents a single variable in the Constraint Satisfaction Problem, tracking its journey from a state of total superposition to a final, collapsed state.

The WaveFunction.cs script begins by instantiating a grid of GameObjects, each with a Cell.cs script attached. The Cell.cs script contains two key fields:

```
public class Cell : MonoBehaviour
{
   public bool collapsed;
   public List<Tile> tileOptions;
}
```

collapsed: A boolean flag that provides a quick and efficient way to check the state of the cell. It is initialized to false and is set to true only when the cell's wave function has been collapsed to a single, definite tile. This prevents the algorithm from trying to re-collapse an already decided cell.
tileOptions: This list is the practical implementation of the "wave function" for each cell. At the moment of initialization, this list is populated with every possible tile provided to the generator, representing the cell's initial state of maximum

entropy. As the Propagation phase executes, the WaveFunction script removes tiles from this list that are no longer valid based on the state of neighboring cells.

The number of elements in the tileOptions list (tileOptions.Count) is the direct measure of the cell's entropy. The WaveFunction script uses this count to identify the uncollapsed cell with the fewest remaining options, making it the core of the Minimum Remaining Values heuristic. When a cell is finally chosen for observation, one tile is selected from this list, the list is cleared, and the single chosen tile is re-added. The collapsed flag is then set to true, and the cell's state is now considered final.

4.2.3 WaveFunction.cs: The Core Generation Engine

The WaveFunction.cs script is the central nervous system of the entire procedural generation process. It is a comprehensive MonoBehaviour designed to be attached to a single controller object within the Unity scene. This script orchestrates the entire workflow, managing the grid data structure, executing the core WFC algorithm, and handling the instantiation of tile prefabs to construct the final visual map. It seamlessly integrates the static rule data from the Tile.cs components with the dynamic state of the Cell.cs objects, driving the system from a state of total uncertainty to a single, coherent outcome.

The process is initiated in Unity's Awake lifecycle method, which first calls the InitializeGrid function. This function is responsible for creating the foundational data structure of the map. It executes a nested loop corresponding to the desired dimensions of the grid, instantiating a cellObj prefab at each coordinate. A Cell.cs component is attached to each of these GameObjects, establishing the grid of variables. Immediately following this structural setup, each new cell's CreateCell method is called, which populates its tileOptions list with the complete array of tileObjects supplied to the script. This critical first step establishes the initial state of the system: a grid where every cell is in a superposition, holding all possible tiles as potential outcomes. After the grid is fully initialized, the Awake method concludes by starting the RunWFC method as a coroutine, which contains the main generative logic.

The decision to implement the main loop as a coroutine in RunWFC is a deliberate architectural choice that facilitates step-by-step visualization of the algorithm's execution. Rather than running instantaneously, the process is punctuated by a yield return new WaitForSeconds(stepDelay) at the end of each iteration. This pause allows an observer to watch the map being built piece by

piece, providing invaluable insight into the decision-making process of the algorithm.

The core of the RunWFC coroutine is a while loop that continues as long as the generationComplete flag is false. The first and most critical operation within this loop is the Observation phase, which is handled by the FindCellsWithLowestEntropy method. This method implements the minimum entropy heuristic by iterating through every cell in the gridComponents list. It checks if a cell is uncollapsed and compares the number of tiles in its tileOptions array to find the minimum value. Crucially, it does not stop at the first cell it finds with this minimum entropy; instead, it compiles a list of every cell that shares this lowest entropy value. This list represents all the "best" possible candidates for the next collapse. Once this list is returned, the RunWFC method selects one cell at random from it. This injection of randomness into the selection of equally constrained cells is a primary source of variation, ensuring that each run of the generator can produce a unique and unpredictable map layout.

Once a target cell is selected, the CollapseCell method is called. This function performs the "measurement" step, forcing the cell into a definite state. First, it sets the cell's collapsed flag to true, permanently removing it from consideration in future entropy checks. It then randomly selects a single Tile from the cell's remaining tileOptions. The cell's state is then updated by calling RecreateCell, which replaces its list of options with a new array containing only the single selected tile. A significant feature of this implementation is that the script then immediately calls Instantiate, creating a visual representation of the chosen tile at the cell's position in the game world. This means the map is built visually and incrementally as the algorithm progresses, rather than being rendered only at the end.

The collapse of a cell triggers the Propagation phase, managed by the Propagate method. This method uses a Stack data structure to handle the cascading constraint enforcement. The process is initiated by calling AddNeighborsToStack, which seeds the stack with the uncollapsed neighbors of the just-collapsed cell. The script then enters a while loop that continues as long as the stack is not empty. In each iteration, it pops a currentCell from the stack and determines its new set of valid tile options by calling the GetPossibleTilesFromNeighbors method. This function is the heart of the constraint propagation logic. It begins by creating a HashSet containing all possible tileObjects. It then iterates through each of the cell's neighbors (up, right, down, and left). For each neighbor, it aggregates all the tiles that would be valid neighbors from that direction (e.g., for the "up" neighbor,

it gathers all of its downNeighbours). It then performs an intersection (IntersectWith) between the cell's master set of possible tiles and the set of valid tiles derived from that neighbor. By repeating this for all four neighbors, the HashSet is progressively filtered down to contain only those tiles that simultaneously satisfy the adjacency constraints of every surrounding neighbor.

After GetPossibleTilesFromNeighbors returns this refined set of possibleTiles, the Propagate method checks if the count of these new options is less than the cell's previous number of options. If it is, this signifies that new information has been established and the cell has become more constrained. The cell's tileOptions are updated via RecreateCell, and crucially, its own neighbors are then added to the stack via AddNeighborsToStack. This ensures that the new constraint information from this cell is, in turn, propagated to its own surroundings. This process continues until the stack is empty, signifying that the system has reached a new stable state of local consistency. At this point, the main RunWFC loop yields for the specified delay and then begins the next cycle of Observation, Collapse, and Propagation, continuing until the entire map is complete.

4.3 Core Scripting Architecture

A primary goal of this thesis was to create not just a functional algorithm, but a practical and intuitive tool for a game designer. The workflow is designed to be self-contained within the Unity Editor, providing a tight, iterative loop of creation, configuration, and testing. This process is divided into two main stages: the initial setup and rule definition for the tile assets, and the execution and visualization of the generation process itself.

4.3.1 Creating and Configuring Tile Prefabs

The foundation of the entire system lies in the creation and meticulous configuration of the tile prefabs. This is the stage where the designer's intent is encoded into the data that the algorithm will use. The workflow is designed to be entirely visual, requiring no direct code modification.

The process for integrating a new tile into the system begins with its visual asset. A 16x16 sprite, either taken from the base tileset or modified in Aseprite, is imported into the Unity project. A new, empty GameObject is then created in the scene. This GameObject is given a SpriteRenderer component, and the corresponding sprite is assigned to it, making the tile visible.

Next, the crucial step is to attach the Tile.cs script to this GameObject. This action immediately exposes the four public neighbor lists (upNeighbours, rightNeighbours, downNeighbours, leftNeighbours) in the Unity Inspector. This is the interface for defining the tile's adjacency constraints. The designer then populates these lists by dragging other existing tile prefabs from the Project window directly into the appropriate list slots in the Inspector. For example, to establish that a Mountain_ML (Middle-Left) tile can have a Mountain_MM (Middle-Middle) tile to its right, the designer would select the Mountain_ML object, locate its rightNeighbours list in the Inspector, and drag the Mountain_MM prefab into it. For a connection to be fully valid, this relationship must be reciprocal; the designer must also select the Mountain_MM prefab and drag the Mountain_ML prefab into its leftNeighbours list (Fig. 16).

This manual, explicit linking process is repeated for all desired connections. Once a tile's rules are fully configured, the GameObject is dragged from the scene's Hierarchy view into the Project window, saving it as a self-contained prefab. This prefab now bundles its visual appearance with its complete set of adjacency rules, ready to be used by the generation engine.

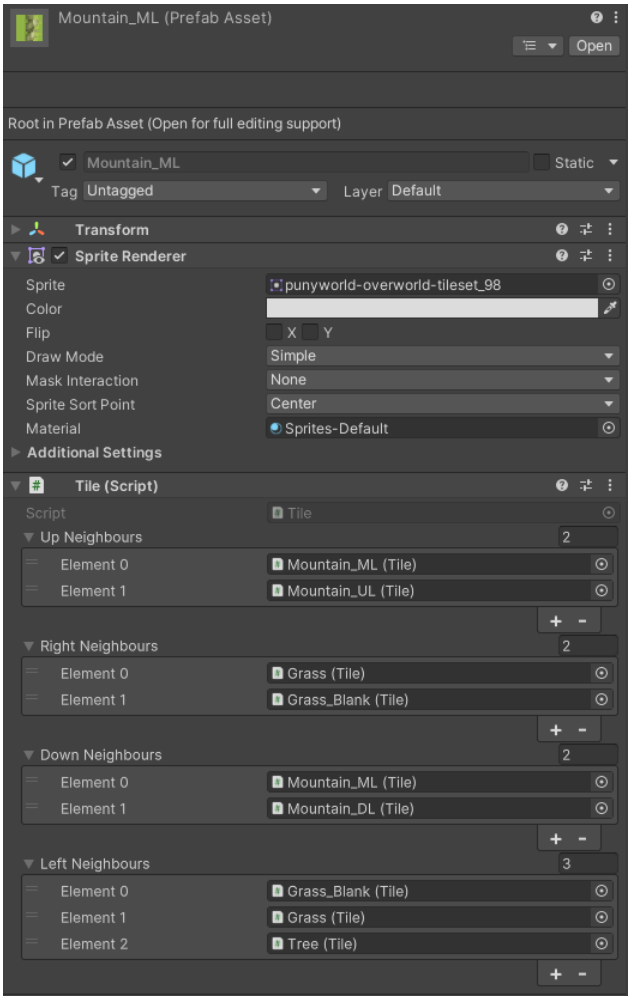


Figure 16: Mountain_ML configuration tab inside the Unity Editor

4.3.2 Executing the Generation and Visualizing the Output

Once the library of tile prefabs has been created and configured, the designer can proceed to the generation phase. This is managed by the central WaveFunction GameObject in the scene. The Inspector for this object exposes the key parameters for the generation: the dimensions of the map, the master list of tileObjects, the cellObj prefab, and the stepDelay for visualization.

The primary setup task for the designer is to populate the tileObjects array. This is done by selecting the WaveFunction object and dragging every single configured tile prefab from the Project window into this array. This provides the algorithm with its complete dictionary of "puzzle pieces."

With the parameters set, executing the generation is as simple as pressing the "Play" button in the Unity Editor. This triggers the Awake method in the WaveFunction.cs script, which initializes the grid and begins the RunWFC coroutine. Because the main loop is implemented as a coroutine that yields after each collapse, the generation process is not instantaneous. Instead, the designer can observe the map's formation in real-time in the Scene or Game view.

At each step, a single tile prefab is instantiated at the position of the newly collapsed cell. The designer can see the algorithm's decisions unfold: a tile appears, and the consequences of that choice implicitly guide the next placement. This step-by-step visualization(Fig. 17-20) is an invaluable diagnostic and design tool. It provides immediate visual feedback on the effects of the defined adjacency rules, allowing the designer to quickly assess whether the emerging patterns and structures align with their creative vision. If a lake is forming incorrectly or a mountain range appears disjointed, the designer can stop the process, adjust the neighbor lists on the relevant prefabs, and run the generation again. This tight feedback loop between rule definition and visual output makes the system a powerful and responsive tool for procedural level design. The final output is a fully constructed map, composed of instantiated tile prefabs, organized neatly within the scene hierarchy and ready for use.

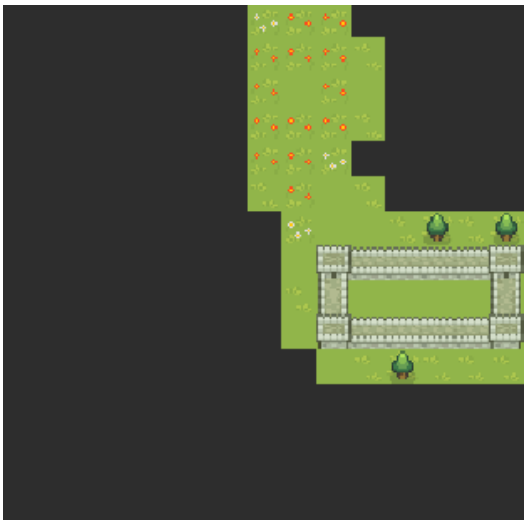
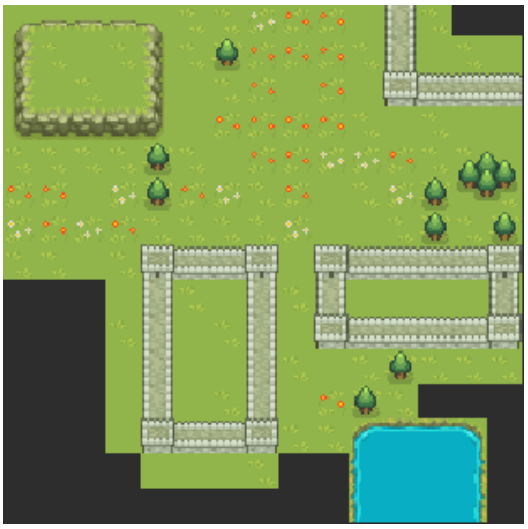


Figure 17&18: First step of the initialization and collapse of the tiles, second step when the WFC function collapses other tiles connected to it's neighbours tiles



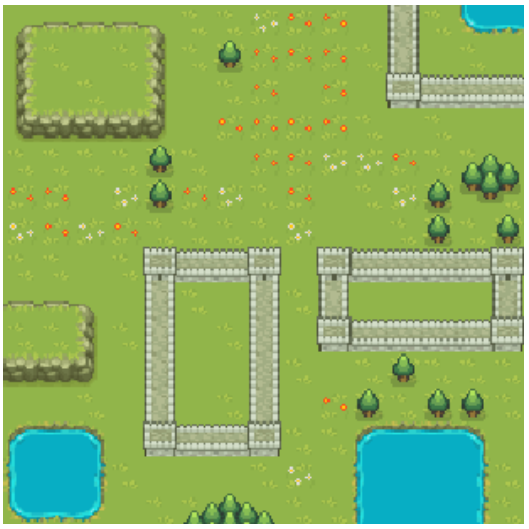


Figure 19&20: Third and final step of the grid creation and population with tiles

5 Results and Evaluation

This chapter provides a critical assessment of the implemented Wave Function Collapse map generator. The evaluation is twofold: first, a qualitative analysis of the generated maps themselves, focusing on their structural coherence, aesthetic qualities, and variety. Second, a quantitative analysis of the system's performance, measuring key metrics like generation time to understand its computational efficiency and scalability. The goal of this chapter is to objectively determine the degree to which the project met its research objectives and to identify both the strengths and inherent limitations of the implemented system.

5.1 Analysis of Generated Maps

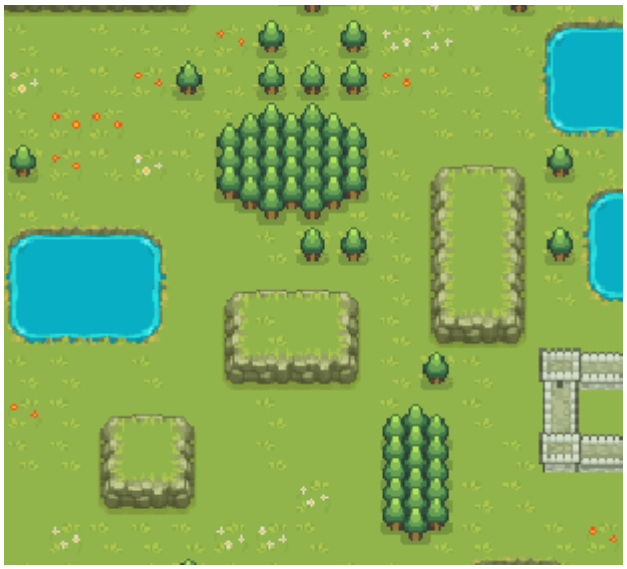
The ultimate measure of a procedural content generator's success is the quality of its output. This section evaluates the generated maps based on their visual and structural characteristics. The analysis focuses on the system's ability to produce maps that are not only technically correct according to the defined rules but also aesthetically pleasing, varied, and capable of forming the recognizable, large-scale features required for a compelling game world.

5.1.1 Coherence and Biome Formation

The primary strength of the implemented Tiled Model of WFC is its capacity to produce maps with perfect local coherence. By its very nature, the propagation algorithm guarantees that no two adjacent tiles will ever violate the predefined neighbor rules. Every tile placed on the map is guaranteed to be a valid neighbor to all of its already collapsed neighbors. This results in maps that are free from logical contradictions, such as a lake edge appearing in the middle of a forest without a proper transition or a castle wall terminating abruptly against an incompatible tile.

Beyond this fundamental local consistency, the system demonstrates a high degree of success in forming the intended large-scale biomes. The design of the 3x3 positional tile system proved to be highly effective. The strict adjacency rules linking corner (UL, UR, DL, DR), edge (UM, MR, DM, ML), and center (MM) tiles force the algorithm to assemble them into complete, recognizable structures. As a result, the generated maps feature well-defined lakes with continuous shorelines, contiguous mountain ranges with clear ridges, and expansive forests

that feel like distinct regions. The system does not merely place water tiles randomly; it constructs a "lake" as a cohesive geographical feature.



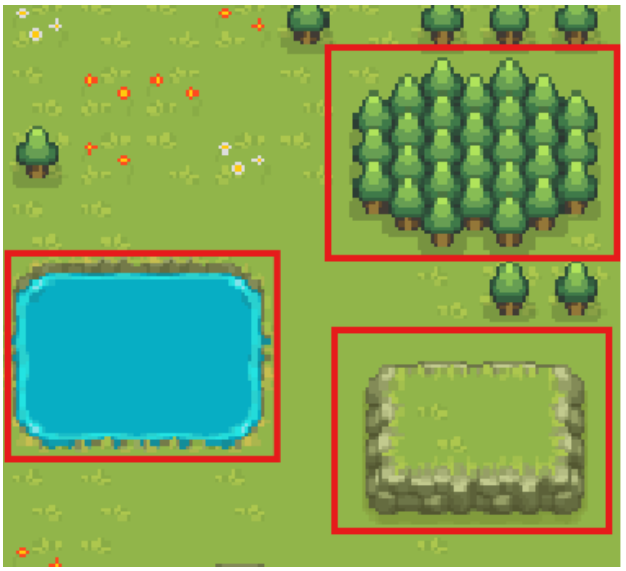


Figure 21&22: Geographically correct generations of the lake, forest and mountains formations surrounded by grass blocks

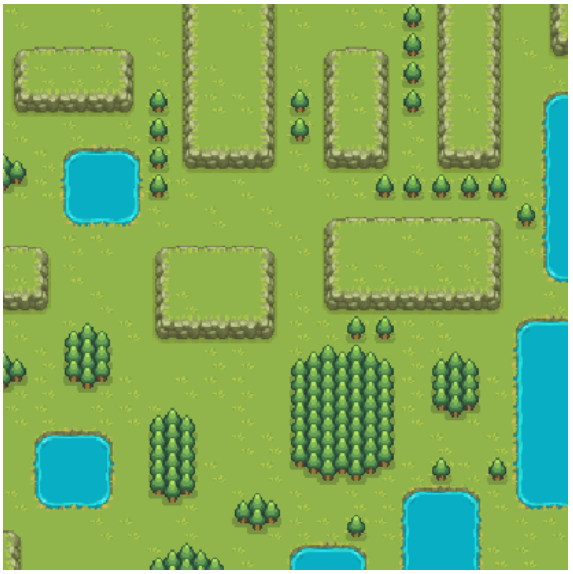
5.1.2 Variety and Unpredictability

While the system operates under a strict set of deterministic rules, it successfully produces a high degree of variety across multiple generation runs. Two key sources of randomness are embedded in the algorithm, ensuring that even with an identical set of input tiles and rules, the output is unique and unpredictable.

The first source of variation occurs during the Observation phase, in the FindCellsWithLowestEntropy method. While the algorithm always chooses from the cells with the minimum number of remaining options, there are frequently multiple cells tied for this lowest entropy. The script's decision to select one of these cells at random is a critical branching point. An early-stage decision to collapse a cell into a Grass tile versus a Mountain_UL tile can have a profound butterfly effect, setting the generation process on a completely different path and resulting in vastly different macro-structures.

The second source of randomness is in the CollapseCell method itself. Once a cell is selected, a single tile is chosen at random from its remaining tileOptions. Even if a cell has only two or three possibilities left, this choice introduces further unpredictability.

The combination of these stochastic elements ensures that the generator explores a wide range of valid solutions within the defined possibility space. Executing the generator multiple times yields maps with significant structural differences. One map might feature a large, central lake, while another might contain several smaller, scattered ponds. A mountain range might dominate the northern edge in one instance and form a diagonal chain across the map in another. This successful balance between strict rule enforcement and strategic randomness fulfills a core objective of procedural generation: the ability to create content that is always coherent but never entirely predictable.



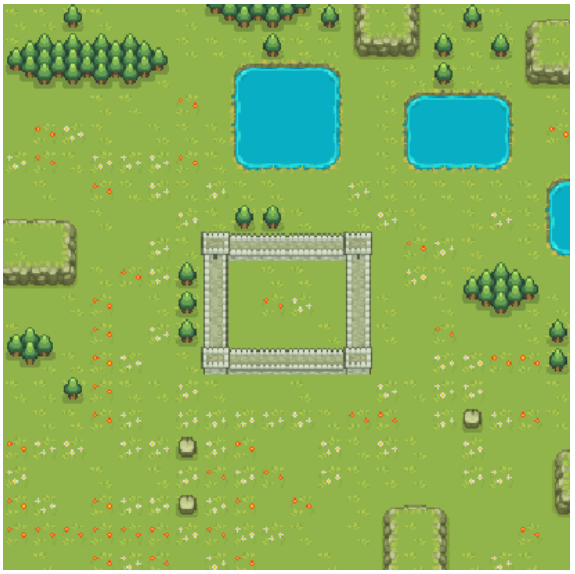


Figure 23&24: Variety of two generations of the grid map with different layout of tiles

5.2 Performance Metrics

While the qualitative assessment of the generated maps confirms the system's ability to produce high-quality content, a quantitative analysis of its performance is necessary to evaluate its viability as a practical development tool. This section examines the computational cost of the generation process, focusing on how key variables like map size and rule complexity impact the algorithm's efficiency and scalability.

5.2.1 Generation Time vs. Map Size

The most significant factor influencing the performance of the generator is the size of the output map. To measure this relationship, the generation time was recorded for several different map dimensions, while keeping the tileset constant at 41 unique tile objects. The tests were performed on a standard development machine, and the results were averaged over multiple runs to ensure consistency.

The recorded generation times were as follows:

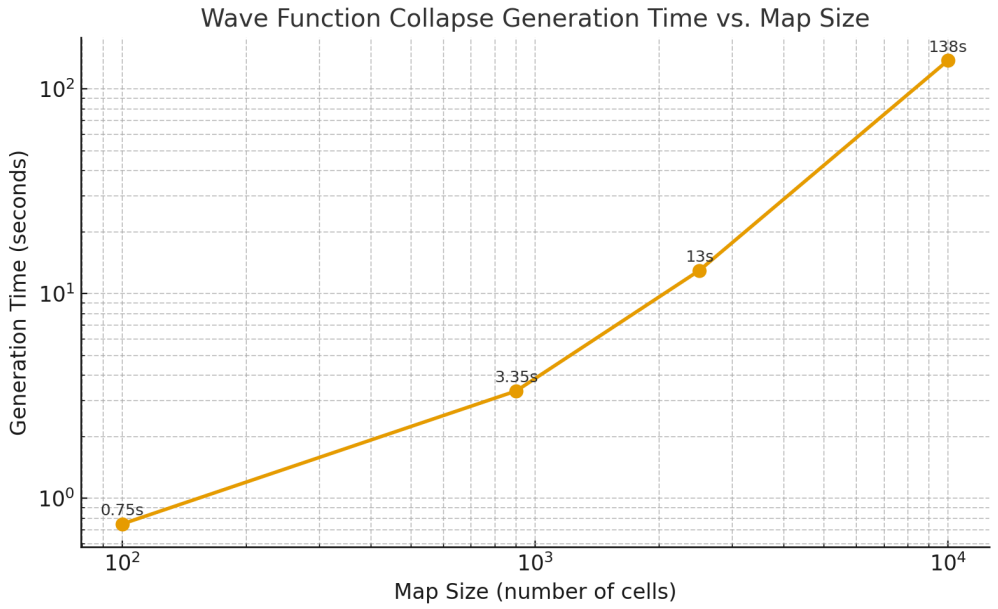


Figure 25: WFC generation time vs map size line graph

The data reveals a clear and expected trend: the generation time increases at a rate that is significantly faster than linear. Doubling the grid dimension from 50 to 100 (a 4x increase in the number of cells) results in more than a 10x increase in generation time. This super-linear, or polynomial, growth is characteristic of the WFC algorithm.

This performance curve can be attributed to the scaling of the two primary operations in the main loop. At each step, the Observation phase requires iterating through all N cells in the grid to find the lowest entropy. The Propagation phase, in the worst case, could also touch every cell in the grid. Since this loop runs N times (once for each cell to be collapsed), the complexity grows rapidly. For small to medium-sized maps (up to around 50x50), the generation time remains well within acceptable limits for an editor-time tool, allowing for rapid iteration and experimentation. However, the steep increase in time for larger maps indicates a practical limit on the dimensions that can be generated in a comfortable, interactive timeframe. Generating extremely large maps (e.g., 200x200 or more) would likely require significant optimization or a more patient workflow.

5.2.2 Impact of Tileset and Rule Complexity

The performance of the generator is also heavily influenced by the complexity of the tileset and its associated rules. The number of tiles in the initial tileObjects array, along with the density of their interconnections in the neighbor lists, defines the size and intricacy of the constraint satisfaction problem. A larger tileset increases the initial entropy of every cell, meaning more possibilities must be pruned during propagation. The algorithm's time complexity is polynomial O(N), heavily dependent on both the number of cells (N) and the number of tiles (T). Each of the N collapse steps can trigger a propagation wave that, in the worst case, must update constraints across the entire grid. The core of this update, the GetPossibleTilesFromNeighbors function, performs set intersections whose cost scales with the number of tiles. Consequently, a larger and more richly connected tileset, while enabling more varied and detailed output, invariably increases the computational load of each propagation step, leading to longer generation times even when the map dimensions remain constant. This demonstrates the fundamental trade-off between the expressive power of the rule set and the computational efficiency of the generation process.

6 Conclusion and Future Work

6.1 Summary of Contributions

This thesis has documented the complete journey of creating a 2D tile-based map generator in Unity, from design and implementation through to final evaluation. The project's primary contribution is a functional prototype that is both intuitive to use and powerful in its results. It leverages the Tiled Model of Wave Function Collapse, enhanced with a 3x3 positional system that simplifies the creation of coherent biomes. By integrating the rule-definition process directly into Unity's native architecture, this work offers a practical workflow for designers, allowing them to configure complex rules without needing to write code. The resulting system is fully capable of generating maps that are locally coherent, aesthetically diverse, and structurally sound, demonstrating that constraint-based procedural generation is a truly viable tool for game development. Furthermore, the performance analysis sheds light on the algorithm's scalability and highlights the inherent trade-offs between map size, rule complexity, and computational cost.

6.2 Avenues for Future Research

Although the implemented system achieved its objectives, it also serves as a strong foundation for future research and enhancements.
A significant next step would be to introduce a backtracking mechanism. The current greedy approach is effective but brittle; allowing the algorithm to retrace its steps would dramatically increase its reliability, especially when dealing with highly constrained rule sets. Another compelling avenue would be to implement the Overlapping Model. This would shift the creative process from manual rule-setting to curating examples, offering a powerful method for capturing subtle, organic aesthetics.

The core principles are not limited to two dimensions; extending them into 3D could lead to generators for voxel terrain or intricate architecture. Finally, to overcome the current performance limitations, future work could focus on optimization. By exploring more sophisticated data structures or parallel processing, we could mitigate the rapid growth in generation time and empower the creation of vastly larger and more complex worlds.

Bibliography

Gumin, M. (2016). WaveFunctionCollapse. GitHub. Retrieved from https://github.com/mxgmn/WaveFunctionCollapse

Haralick, R. M., & Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, 14(3), 263–313.

Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural content generation for games: A survey. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 9(1), 1–22.

Karth, I., & Smith, A. M. (2017). WaveFunctionCollapse is constraint solving in the wild. Proceedings of the 12th International Conference on the Foundations of Digital Games.

Mackworth, A. K. (1977). Consistency in networks of relations. Artificial Intelligence, 8(1), 99–118.

mr-matthew. (n.d.). 16x16 puny world tileset. OpenGameArt.org. Retrieved from https://opengameart.org/content/16x16-puny-world-tileset

Prusinkiewicz, P., & Lindenmayer, A. (1990). The algorithmic beauty of plants. Springer-Verlag.

Russell, S. J., & Norvig, P. (2021). Artificial intelligence: A modern approach (4th ed.). Pearson.

Salcedo, M. V., Eguia, A. M. G., & Fernando, A. M. (2024). Puzzle-Level Generation with Simple-tiled and Graph-based Wave Function Collapse Algorithms. 2024 IEEE 16th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment, and Management (HNICEM), 1–6.

Shaker, N., Togelius, J., & Nelson, M. J. (2016). Procedural content generation in games: A textbook and an overview of current research. Springer.

Shannon, C. E. (1948). A mathematical theory of communication. The Bell System Technical Journal, 27(3), 379–423.

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). Search-based procedural content generation. Proceedings of the 11th European Conference on Evolutionary Computation in Combinatorial Optimization, 141–152.

Yannakakis, G. N., & Togelius, J. (2011). Experience-driven procedural content generation. IEEE Transactions on Affective Computing, 2(3), 147–161.

Boris the Brave. (2020, April 13). Wave function collapse explained. Boris the Brave. https://www.boristhebrave.com/2020/04/13/wave-function-collapse-explained/

[1] Image source: https://www.researchgate.net/figure/Several-minimaps-of-the-full-layout-of-a-level-in-The-Binding-of-Isaac-Rebirth_fig1_351841420

[2] Image source: https://gpuopen.com/learn/work_graphs_mesh_nodes/work_graphs_mesh_nodes-procedural_generation/

[3] Image source: https://www.sportskeeda.com/minecraft/what-3d-biomes-minecraft-s-1-18-update

[4] Image source: https://levelup.gitconnected.com/port-an-existing-c-c-app-dungeon-crawler-rogue-to-flutter-with-dart-ffi-a701284aa289

[5] Image source: https://community.fico.com/s/blog-post/a5Q4W000001V7gdUAC/fico2486

[6] Image source: https://www.boristhebrave.com/docs/tessera/6/articles/models/overlapping.html

[7] Image source: https://www.researchgate.net/figure/WFC-simple-tiled-model-example-a-Seven-types-of-tiles-b-The-connectivity-of-the_fig2_378377175

[8] Image source: https://threadreaderapp.com/thread/1267045322116734977.html