Trinity University
Digital Commons @ Trinity

Computer Science Honors Theses | Computer Science Department

5-2021

Modifying Wave Function Collapse for more Complex Use in Game Generation and Design

Quentin Edward Morris
Trinity University, quentinem13@gmail.com

Follow this and additional works at: https://digitalcommons.trinity.edu/compsci_honors

Recommended Citation

Morris, Quentin Edward, "Modifying Wave Function Collapse for more Complex Use in Game Generation and Design" (2021). *Computer Science Honors Theses*. 58.
https://digitalcommons.trinity.edu/compsci_honors/58

This Thesis open access is brought to you for free and open access by the Computer Science Department at Digital Commons @ Trinity. It has been accepted for inclusion in Computer Science Honors Theses by an authorized administrator of Digital Commons @ Trinity. For more information, please contact jcostanz@trinity.edu.

Modifying Wave Function Collapse for more Complex Use in Game Generation and Design

Quentin Morris

Abstract

Wave Function Collapse (WFC) is an image-based Procedural Content Generation (PCG) algorithm that uses constraints extracted from an input image to generate a similar, yet novel output. The goal of this thesis is to modify WFC with extra constraints and parameters that would allow a game designer to have more control over the algorithm to produce more varied and specific results. This thesis introduces the field of Procedural Content Generation (of which Wave Function Collapse is a part) and details the WFC algorithm. It then examines past work done in these two topics, along with developments in the field of content generation for the game *Super Mario Bros.* and developments of heuristics used to analyze PCG content. The thesis next explains the specifics of WFC's core algorithm, introduces the new modifications made to it, and details heuristics used to analyze its output. Finally, experiments are run using these modifications to generate content using levels from the original *Super Mario Bros.* as input, and previously introduced heuristics are used to assess the results.

Acknowledgments

I would like to thank everyone who helped to make this thesis possible. First, to my advisor who helped me find my way through my first academic paper of this scale and also to my thesis committee who offered me advice and corrections to help me improve this thesis as much as possible. Next to my friends who worked hard and successfully in their own studies, motivating me to work as hard as I could in my own. Last to my parents who encouraged me to push myself toward such an accomplishment as this in the first place. My thanks to you all for your support.

Modifying Wave Function Collapse for more Complex Use in Game Generation and Design

Quentin Morris

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

April 1, 2021

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Student Copyright Declaration: the author has selected the following copyright provision:

[X] This thesis is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License, which allows some noncommercial copying and distribution of the thesis, given proper attribution. To view a copy of this license, visit `http://creativecommons.org/licenses/` or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

[ ] This thesis is protected under the provisions of U.S. Code Title 17. Any copying of this work other than "fair use" (17 USC 107) is prohibited without the copyright holder's permission.

[ ] Other:

Distribution options for digital thesis:

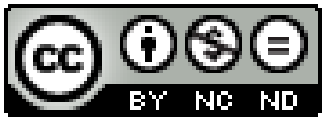
[X] Open Access (full-text discoverable via search engines)

[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via `digitalcommons.trinity.edu`)

Modifying Wave Function Collapse for more Complex Use in Game Generation and Design

Quentin Morris

This work is subject to the “Creative Commons **Recognition - NonCommercial - No Derivative Work**” license.



Contents

1 Introduction **1**

1.1 Procedural Content Generation 1

1.2 Wave Function Collapse 2

1.3 Thesis Statement 3

2 Background **4**

2.1 Procedural Content Generation 4

2.2 Mario Generation 6

2.3 Expressive Range 7

2.4 Wave Function Collapse 9

3 Methods **11**

3.1 The Wave Function Collapse Algorithm 11

3.1.1 Patterns from Sample 12

3.1.2 Build Propagator 14

3.1.3 Observe 15

3.1.4 Propagate 16

3.2 Proposed WFC Modifications 17

3.2.1 Forced Tile Placements . . . 17
3.2.2 Bounded Tile Appearances . . . 18
3.3 Parameterization for Generated Levels . . . 19
3.3.1 Forced Tile Placements . . . 20
3.3.2 Bounded Tile Appearances . . . 21
3.3.3 Snapshot Size Variations . . . 21
3.4 Analysis . . . 22
3.4.1 Density . . . 22
3.4.2 Linearity . . . 22
3.4.3 Playability . . . 23

4 Results **24**

5 Discussion **32**
5.1 Standard Wave Function Collapse Generation . . . 32
5.2 Bounded Tile Appearances . . . 33
5.2.1 Density . . . 33
5.2.2 Linearity . . . 34
5.2.3 Playability . . . 35
5.3 Forced Tile Placements . . . 36
5.3.1 Density . . . 37
5.3.2 Linearity . . . 37
5.3.3 Playability . . . 38
5.4 Altered Snapshots . . . 39
5.4.1 Density . . . 39
5.4.2 Linearity . . . 42

5.4.3 Playability 42

6 Future Work **44**

6.1 Different Heuristics 44

6.2 Use of Current Modifications 45

6.3 Further Modifications 46

6.3.1 Forced Tile Frequencies 46

6.3.2 Removal of Screen-Wrapped Adjacency Rules 47

List of Figures

3.1 An example of an input image (left) that would be taken in by WFC, and a possible output (right). Images used are taken from Karth and Smith [8]. . 12

3.2 An example of 2×2 snapshots extracted from the input image in figure 3.1. Note that these snapshots include rotated versions of extracted snapshots. Rotation of snapshots is optionally used in WFC's image-processing, but is not used for the purpose of this thesis. Image taken from Karth and Smith [8]. 13

3.3 Levels 1, 2, and 3 from the original *Super Mario Bros.* accompanied by the text file representations used as an input image for WFC. Level images from [1]. 19

3.4 A ground tile, represented by 'X' in the text file representations of levels. . 20

4.1 Candlestick graphs for density 29

4.2 Candlestick graphs for linearity 30

4.3 Candlestick graphs for failure completion percentage 31

5.1 Output 15 of the Bounded1 configuration generated from level 2 34

5.2 Output 23 of the Bounded2 configuration generated from level 1: an example output demonstrating the high walls restricting progression, seen near the left in front of Mario's starting location. 35
5.3 Output 43 of the Forced3 configuration generated from level 1 36
5.4 Output 10 of the Forced3 configuration generated from level 3 38
5.5 Output 183 of the Column Snapshot configuration generated from level 1. . 39
5.6 Output 7 of the 3x3 Snapshot configuration generated from level 1: an example of the more coherent stair-step '#'-tile structures seen when generated from level 1 with a 3×3 snapshot size. 40
5.7 Output 174 of the Column Snapshot configuration generated from level 3: An example of a more dense level-3-produced level including longer platforms. The fill of solid bricks can also be seen on the right side. 41

6.1 An example output (below) produced from level 1 (above) with standard WFC. Ground tiles ('X') are able to appear at any height. 48

Chapter 1

Introduction

1.1 Procedural Content Generation

Procedural Content Generation (PCG) is defined as the creation of content through the use of an algorithm; commonly, PCG is connotatively tied to the generation of video game content, but this is not always the case [16]. Though it is often used to generate game assets, some PCG algorithms can be used to generate images, art, or other designs. For example, Cullen and O'Sullivan [2] and Lipp *et al.* [11] use PCG to create buildings and city layouts, and Guzdial *et al.* [5] uses a PCG system to generate visual art, though their primary focus is applying this art to games.

When applied to games, PCG can reduce the amount of time spent generating content by reducing human development of background assets like natural terrain or non-playable characters [16]. It is also possible to generate more integral parts of a game, like entire levels, with PCG. As it happens, one of the first applications of PCG was the generation of dungeon levels for the game *Rogue* in 1980. Now, dungeon generating algorithms are more common and have been implemented in games like *Diablo*, *Daggerfall*, and *Daylight* [10].

Most PCG algorithms are based on using hand-crafted pieces of game content in combination with algorithms, parameters, rules and constraints to generate new content; however, it is much more uncommon that a PCG algorithm uses finished game content to extract rules and create more content [16]. This thesis focuses on one of these algorithms known as Wave Function Collapse.

1.2 Wave Function Collapse

Wave Function Collapse (WFC) is a specific PCG algorithm developed by Maxim Gumin [4] that aims to create a novel output image by considering the inherent rules and patterns of a given input image. It determines these rules and patterns by looking at small $N \times M$ subsections of the input and assessing how many times each subsection appears and which subsections appear adjacent to each other. Gumin addresses the first of these rules, concerning the frequency of the subsections, as "local similarity." This involves a strong requirement that each $N \times M$ subsection in the output should occur at least once in the input and a weak requirement that the frequency of certain subsections in the output should be similar to their frequency in the input. The second of these rules,"adjacency," describes how certain subsections can be placed near each other. It is a strong requirement that if two subsections appear adjacent to each other in the output, they must have appeared adjacent in the input [4].

Upon its release, WFC gained a fair amount of traction and popularity with online indie game developers and hobbyists. In addition to being implemented in multiple environments [4], WFC has appeared in a few officially published indie games, with *Caves of Qud* being one of the most prominent. This game is another example where designers use PCG to generate the central parts of the level, like those mentioned earlier in section 1.1 [8]. It is

this application of Wave Function Collapse to the generation of levels and design in video games that inspires the focus of this thesis. The algorithm works well for this purpose, but, by restricting its function to only local similarity and adjacency in an input image, it may prove cumbersome or limited to developers who desire more control over the process while maintaining the advantages of PCG. As such, I propose methods to modify Wave Function Collapse that will grant users more friendly and versatile control over what the algorithm produces.

1.3 Thesis Statement

The goal of this thesis is to apply Wave Function Collapse to the generation of levels for a framework based around the original *Super Mario Bros.* and modify it by adding user constraints in order to generate a usable output. I will observe changes in the density, linearity, and playability of WFC outputs through the use of these customized user constraints as well as though the altering of native WFC parameters.

Chapter 2

Background

This chapter details relevant work done in Procedural Content Generation and, as this thesis uses the game *Super Mario Bros.* as the center for its experiments, this chapter also reviews some past applications of PCG towards generation of content for said game. It also outlines Expressive Range metrics used to analyze various PCG algorithms, and, finally, reviews research and modifications already done on WFC.

2.1 Procedural Content Generation

Togelius *et al.* [18] devise a taxonomy and survey on searched-based PCG algorithms that aims to define different types of PCG algorithms. They break algorithms up into five different classifications defined by whether or not the algorithm contains a specific property. 1) Online or Offline, which specifies (respectively) whether generation of content happens while the game is running or during the game's development. 2) Necessary or Optional, which specifies whether the content created is necessary to progress in the game or if it is simply background or avoidable. 3) Random Seeds or Parameter Vectors, which is more of

a scale that specifies how many parameters may be specified by the designer before generation to offset a simple random number generator. 4) Stochastic or Deterministic, which, somewhat similarly to the previous, specifies how much randomness comes out in the generated content. 5) Constructive or Generate-and-Test, which specifies whether the algorithm tests that content is playable through specific actions while the algorithm generates it or if it generates the content and then tests playability according to some criteria. They then focus on defining a specific subset of generate-and-test algorithms called search-based PCG, where the post-generation test function evaluates the content with a number called a fitness value (as opposed to accepting or rejecting), while further generation attempts to increase this value. The paper then follows the taxonomy with a survey listing various papers that have done work on each of these classifications.

Liapis *et al.* [10] explores the use of procedural personas as a method of evaluating dungeons generated by PCG. A procedural persona is a game-playing agent that acts out a specific style of play; Liapis lists five different types of personas based on different utility functions for the game they used: killing monsters, collecting treasure, reaching the exit, performing the lowest possible number of actions, and avoiding death as much as possible. They use the monster-killing persona and the treasure-hunting persona on a dungeon-based game created for use in modeling decision making called *MiniDungeons*. They use the performance metric given by these two personas to evaluate the quality of levels that are deemed playable as a baseline, and then programmatically evolve these levels based on the metric. The paper demonstrates how the personas actively influenced the design patterns of the levels differently based on their architecture. In fact, they find that when one of the two persona types was used exclusively to evolve a level[pretty sure this is what "maps evolved towards deviations between monsters killed or treasures collected" means], it resulted in more difficult levels with significant risk/reward areas.

Summerville *et al.* [16] combines the evolving field of PCG with the many recent developments in deep learning to create Procedural Content Generation via Machine Learning, or PCGML. This technique utilizes the training of machine learning algorithms on large data sets and applies it to PCG so that algorithms may create game content through prior training on existing game content. The research focuses specifically on different ways of representing training data and applies five different training methods to these representations. For data representation, they use sequences, grids, and graphs, and they train each using back propagation, evolution, frequency counting, expectation maximization, and matrix factorization. For the use cases of PCGML, they list Autonomous Generation, which allows generation without designer input, but state that a more interesting use is co-creative design, which blends the intent of a designer with the intent of a trained artificial intelligence. Along these two main benefits, they also list other positives like content repair, analysis, and data compression.

2.2 Mario Generation

Dahlskog and Togelius [3] uses PCG in combination with software engineering design patterns in an effort to generate *Super Mario Bros.* levels. Design patterns are a language originally devised for architecture and adapted to computer programming in order to solve recurring problems. Dahlskog and Togelius applied this concept to previous research where they discovered that *Mario* levels tend to contain recurring patterns that fit into larger families. By combining these patterns with a PCG algorithm they aim to increase the variety and control with which PCG may generate levels. They conclude that this application of design patterns works particularly well with search-based solutions of PCG.

Shaker *et al.* [13] employs a subset of Genetic Programming called Grammatical Evolu-

tion to evolve *Mario* levels as they are created. Grammatical Evolution entails the combination of an evolutionary algorithm with a context free grammar that represents the possible solutions of the algorithm. They test and examine their results using expressive range, a concept which will later be touched on in section 2.3.

Jain *et al.* [7] uses autoencoders, a type of neural network that encodes input into a number of dimensions, to generate, repair, and classify level styles. The use of autoencoders in this generation serves two purposes: first, they are able to use original and existing levels as a representation of patterns understood by the autoencoders. Second, they are able to influence what kinds of levels are generated by parametrically changing how much noise goes into the autoencoder. The results show a good degree success and promise for more complex use in future work.

Summerville *et al.* [15] uses neural networks to analyze *Mario* gameplay videos from YouTube and ultimately generate levels based on different play styles. The paper generally demonstrates a feasible alternative to more traditional methods of experience-driven PCG which required in-person experiments and gameplay collection. They note that, overall, the results are satisfying but there are many areas which could be expanded upon.

2.3 Expressive Range

When generating game content, and especially game levels with PCG, it is necessary to ask how to assess the quality of a level and, by extension, the quality of the PCG algorithm used to generated it. This is a difficult problem to answer when posed generally like this as much what defines a "good" level is dependant on the specifications of the game, or even the section of the game, that the level appears in. Previously, PCG users tested the quality of a level by having a player or computer agent play the level directly within the

game itself. However, this produces only vague understandings of the qualities of a level that are entrenched in the context of the game, and doesn't help to compare different PCG generators on a more general level [6].

In order to solve this problem, Smith and Whitehead [14] proposed a set of metrics to examine the expressive range of an algorithm. They define expressive range as the classification of a PCG algorithm's ability to generate style and variety. By thus quantifying a PCG algorithm's abilities, they may be better able to compare them. They then test the expressive range of generators by developing two metrics with which they may analyze generated platformer game levels very similar to that of Super Mario.

The first of these metrics is **linearity** which is the r-squared best fit regression line taken from every platform midpoint in a level. This is measured to describe, what they call, the level's "profile." The second metric is **leniency**, which measures how difficult a level is to complete by assessing every point in a level where a player would be required to give input and giving these locations a weight based on their difficulty and possibility to lead to death. These points are then averaged and normalized to produce the leniency score.

Horn *et al.* [6] further extends this idea by reapplying Smith and Whitehead's metrics and adding three more. This thesis only implements two of these metrics, but I will list and briefly define all of them here.

Leniency is much the same as in Smith and Whitehead's paper, and they specify that the average difficulty is normalized by the level's length and how many possible paths a player could take.

Linearity is also similar to as previously defined, only, instead of taking the midpoints of a platform, they take the endpoint. A platform endpoint is defined as any position where the player must move either up or down at least one level when advancing by one position.

Density deals similarly with level elevation by finding how packed a level is with plat-

forms. For any given x position in the level, it counts the number of possible y positions a player could occupy and averages and normalizes that over the whole level.

Pattern Density counts how many meso-patterns (or generated patterns that match patterns present in the original Mario levels) and averages and normalizes that over the level length.

Pattern Variation is the same as pattern density only it counts unique meso-patterns as opposed to all meso-patterns.

This paper does not compare Wave Function Collapse to other PCG algorithms directly, as is the original intent of Expressive range, but it still uses two of these heuristics to analyze the differing effects of my proposed WFC modifications and to better understand the content that is generated from them.

2.4 Wave Function Collapse

Kim *et al.* [9] summarize WFC first then extend its application from grid-based input images to graphs. Grids are simply a subset of graphs where nodes are restricted to a fixed number of neighbors. The extension to a graph-structure with a variable amount of neighbors allows them to apply the concept to nav-meshes and more 3D spaces.

Karth and Smith [8] simulate the ideas of WFC with Answer-Set Programming in order to examine and critique WFC. Through this, they concluded that WFC is very useful for a wide variety of content (from poems to games). They also conclude, through their ASP simulations, that the idea of no in-progress backtracking in WFC is nice, but not necessary for further iterations of the algorithm, and they note that some other papers are currently working on implementing runtime backtracking.

Sandhu *et al.* [12] focus on a similar concept to this thesis by adding design-friendly

constraints to WFC. The first modification they add is weighted choice which factors the weight of WFC's tiles into the choices it makes in the output. The second modification they add is the separation of different types of tiles by adding non-local constraints. In essence, if a certain type of tile gets placed on the board, then these non-local constraints can ensure that a certain number of other tiles get placed within a certain vicinity. The third modification allows a WFC user to reconfigure certain weights within the whole output or within local areas. The final modification allows a "mini-wfc" to be run on certain areas of the map which eliminates certain tiles when others are placed. I.e. if a lamp tiles is placed, only tiles that look lit up will appear.

Chapter 3

Methods

3.1 The Wave Function Collapse Algorithm

Wave Function Collapse (WFC) uses constraint solving to generate an output image based on an example image or a set of rules. Constraint solving entails the algorithmic solution of Constraint Satisfaction Problems where a number of continuous or discrete decision variables can take on any number of values. When the value of a decision variable is determined, it affects the possible values that other variables may take based on specified constraints [8].

WFC begins by examining an input image and extracting local patterns (which will be referred to as "snapshots") and rules dictating which of those snapshots may appear adjacent to another. It next fills each section of the output space with each possible snapshot, and, on each cycle of the function, chooses to eliminate snapshots from one section of the output. Then, it iterates through the entire output space, eliminating snapshots from adjacent stacks in accordance with the rules collected. This process relates to the original concept of Wave Function Collapse in quantum physics where a wave function (a mathematical function

representing the degrees of freedom to which an observable object may take a certain state) is directly observed and collapsed into a single state.

3.1.1 Patterns from Sample

The first step of Wave Function Collapse is only run once and involves extracting snapshots (or "patterns") from an input image (or "sample") and determining adjacency rules for each of these snapshots. Some implementations of the algorithm allow for snapshots and adjacency rules to be specified and given to the algorithm directly, but, for the purpose of this paper, I will explore the input-parsing functionality of the algorithm.

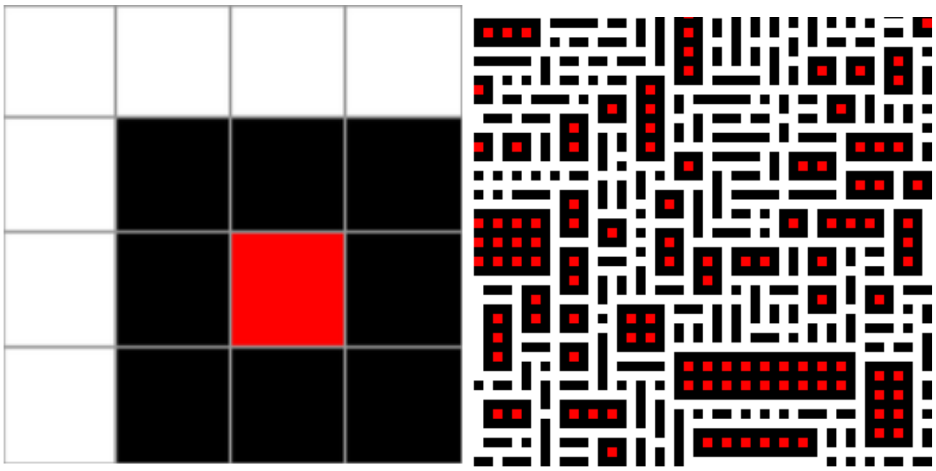


Figure 3.1: An example of an input image (left) that would be taken in by WFC, and a possible output (right). Images used are taken from Karth and Smith [8].

The input image analysis phase of WFC consists of two main operations. The first is to collect snapshots from the image, and one important parameter of Wave Function Collapse is the snapshot's dimensions. Typically, a snapshot is a square with equal x and y dimensions, but it is possible to specify rectangular snapshots as well. When taking snapshots from the

input image, WFC will iterate through the input space, starting at the top left, and store a snapshot of the image that is as large as the specified snapshot dimensions. By the end of the parsing, it will generate a snapshot of every possible $x \times y$ section.

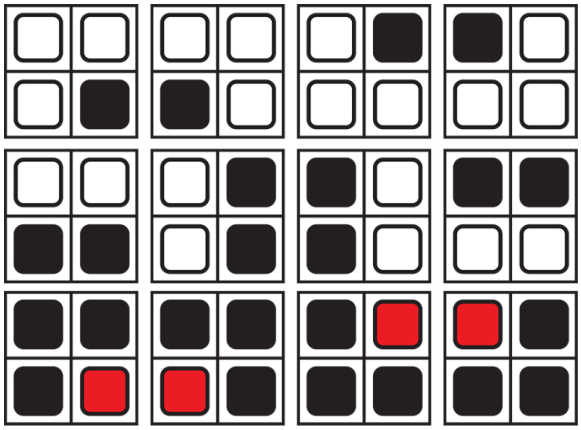


Figure 3.2: An example of 2×2 snapshots extracted from the input image in figure 3.1. Note that these snapshots include rotated versions of extracted snapshots. Rotation of snapshots is optionally used in WFC's image-processing, but is not used for the purpose of this thesis. Image taken from Karth and Smith [8].

The second operation is to collect adjacency rules to pair with every snapshot, which dictate which snapshots appear next to others in the input image. One crucial concept to note is that every snapshot, regardless of its x and y dimensions, is centered, as it were, on its upper-left corner. This is because adjacent snapshots are not defined as snapshots that appear on the current snapshot's outermost boundary, but as snapshots that have a "center" one position off from the current snapshot's. So, for instance, the snapshot at location $(1, 1)$ has adjacent snapshots at location up: $(1, 0)$, right: $(2, 1)$, down: $(1, 2)$, and left: $(0, 1)$. Regardless of a snapshot's size, each location of the output image naturally contains one tile from the input, so it is necessary to take one tile from a snapshot and use the rest as guidelines for what tiles may be adjacent to it. These adjacency rules are

coupled with each snapshot and stored for later use in the generation of the output.

An additional piece of information this function records in tandem with snapshots and adjacency rules is snapshot frequency. After the function has finished collecting snapshots, it must collapse any duplicate snapshots into one entry and aggregate all of the corresponding adjacency rules so they may be stored together. In doing this, it counts the number of times a snapshot appeared in the input image and stores that as its frequency. The frequency of each snapshot is a vital statistic during the next step of the algorithm.

3.1.2 Build Propagator

When the algorithm finishes its analysis of the input image, a function called BuildPropagator organizes all of the information it has just collected into a format that the algorithm can use. Among the data collections this function handles, the most important of these is called the wave, (which is loosely derived from the quantum physics concept of a quantum wave function). The wave is a multi-dimensional boolean array representing all possibilities in the output space. Each position in the output image is represented in the wave, and each of these positions is initialized with an array of booleans of the same length as the number of snapshots previously collected. Sandhu *et al.* [12] proposes a helpful way to think about this concept: the wave is a checkerboard, and on each square sits a stack of chips representing each of the possible snapshots that the square may eventually be occupied with. As the algorithm continues, chips are removed from these stacks, and, once it terminates, each square will be left with only one chip. Thus, I will be referring to single sections of the wave as stacks of snapshots henceforth.

In addition to data collections, BuildPropagator also initializes a few heuristic collections that aid later steps. When deciding which snapshots to eliminate, WFC relies on a heuristic called entropy, a number that summarizes the probability distribution of snapshots in a stack

based on the frequency of each snapshot. Therefore, at the start, every stack begins with the same entropy, and, when a snapshot is eliminated from a stack, the stack's entropy is updated based on that snapshot's frequency.

3.1.3 Observe

After the wave has been initialized, WFC begins its search-based observe-and-propagate loop in which it eliminates snapshots from the the possibilities in the wave and arrives at a constraint-satisfying solution. The purpose of the observe function, then, is to analyze the wave and select a stack to begin eliminating snapshots from. It makes this selection by finding the stack with the smallest nonzero entropy. This corresponds with the common constraint solving principle of selection of the most constrained variable [8].

Once it has taken a stack from the wave, it randomly selects a snapshot out of this stack and prepares to eliminate all others. This concept of taking a possibility space and reducing it to one reality is the essence of "observing." This random selection is weighted based on each individual remaining snapshot's frequency, so the selection is biased toward snapshots that appear more often in the input image. This helps to satisfy the weak requirement mentioned previously that snapshot frequency in the output should be similar to their frequency in the input.

The crux of removing a snapshot of a stack lies in maintaining the consistency of the adjacent stacks. Therefore, a Ban function exists to handle the nuances of this operation. When a certain snapshot is called to be banned from a stack, it removes the snapshot from the stack, then accesses the four stacks directly adjacent and removes the adjacency snapshots corresponding to the specified snapshot. This action implies the further banning of snapshots of adjacent stacks, but, in fact, the ban function only does the job of priming those snapshots for removal from the wave in a later step. Therefore, it also adds the position

of the banned snapshot to a push-and-pop stack data-collection that dictates order of areas of consideration in a later step. The function also handles the updating of all entropy and frequency-related variables for the stack for reference by the greater WFC function.

One final important distinction of the observation phase is how it handles failure during an iteration of the algorithm. Gumin [4] implemented WFC to function without runtime backtracking, which means that, if the algorithm ends up in an impossible state that violates the input constraints, it can not rectify its mistakes and can only give up and quit. The observe step achieves this by checking if any of the snapshot stacks have been completely emptied of snapshots. If this is the case, it means that choices were made elsewhere in the wave that led to a situation in which no tile could possibly exist in the given empty stack according to the adjacency rules. This signifies a failure for the algorithm, and it terminates.

3.1.4 Propagate

The final function in the WFC algorithm, called Propagate, simulates the collapsing behavior of the wave. Where the Ban function only considered the adjacency rules one layer out from a removed snapshot, Propagate travels through the entire wave and fully explores the chaining consequences of a single snapshot's removal. It begins popping the top value off of the stack of snapshots that were primed by Ban and checks the four stacks adjacent, removing all of the adjacent snapshots that had been primed. Ergo, Propagate must call Ban for each of these snapshots, which adds more removed snapshots to the stack to be considered later. Until the stack is empty, Propagate will continue to spread through the wave and collapse the possibilities in the output space.

If the wave still contains non-collapsed stacks by the end of propagation, the cycle will begin again, and the observe function will identify the most constrained of the remaining

stacks and collapse it. This cycle ends either when all stacks in the wave are collapsed or when the observe function finds that the wave has entered an impossible state.

3.2 Proposed WFC Modifications

The primary goal of this paper is not simply to examine the results of WFC-generated *Mario* levels but to modify WFC in ways that may improve certain aspects of these outputs. As a result, two simple but versatile additions were created and added to the basic algorithm. I use these, alongside variation in snapshot sizes, in an attempt to influence the output in a way that might benefit a designer using WFC in designing a game.

3.2.1 Forced Tile Placements

The first modification, rather simply, forces any specified tile in the input to appear at a specific location in the output.This is done by passing a list of rules into the WFC function, each of which consist of a tile and a set of x, y coordinates. The tile must exist in the input image, and the coordinates must reside within the space of the output image. This modification allows the user fine-tuned control over specific tiles in the output in the event that certain tiles should or must appear in certain locations. Furthermore, by taking into consideration WFC's use of adjacency rules, the user can influence what tiles may be placed in certain areas of the output by placing only a single tile somewhere directly.

The function is run after the WFC initialization steps and before the observation-propagation loop. It finds the snapshot stack at the appropriate position in the wave and searches through the stack for any that match the specified tile, calling the Ban() function on any snapshots that do not match. This reduces the stack to all existing snapshots containing the specified tile at their top left and ensures that adjacency consistency

is maintained by properly handling propagation for every elimination from the stack.

This paper uses three similar configurations of the forced tile placements modification which will be listed further below.

3.2.2 Bounded Tile Appearances

The second modification, as opposed to a forced specification from the user, is more of a restriction of the capabilities of the algorithm while generating the output. It confines certain tile placements to certain areas of the output image while keeping them from appearing anywhere else. The input is the same as for the forced tiles (a tile and an x, y tuple); however, in this case, the x and y values each define a boundary in the output space past which the specified tile can not appear. The function is configured so that a positive x or y value will restrict the specified tile to appearing in positions greater than that value, and a negative x or y will restrict it to appearing in spaces less than. WFC does not naturally consider the original absolute x, y location of tiles and snapshots during generation, so chunks of ground and other features are able to appear wherever they please in the output, provided they obey the adjacency rules. This modification, therefore, allows the user to more strongly enforce tile locality in WFC-generated outputs.

The implementation of this function is similar to the forced tile placements modification; however, instead of searching for one snapshot stack in the wave, it searches for all stacks outside of the specified x and y range and bans any snapshots that match the specified tile from each. This paper uses two configurations of this modification which are also listed below.

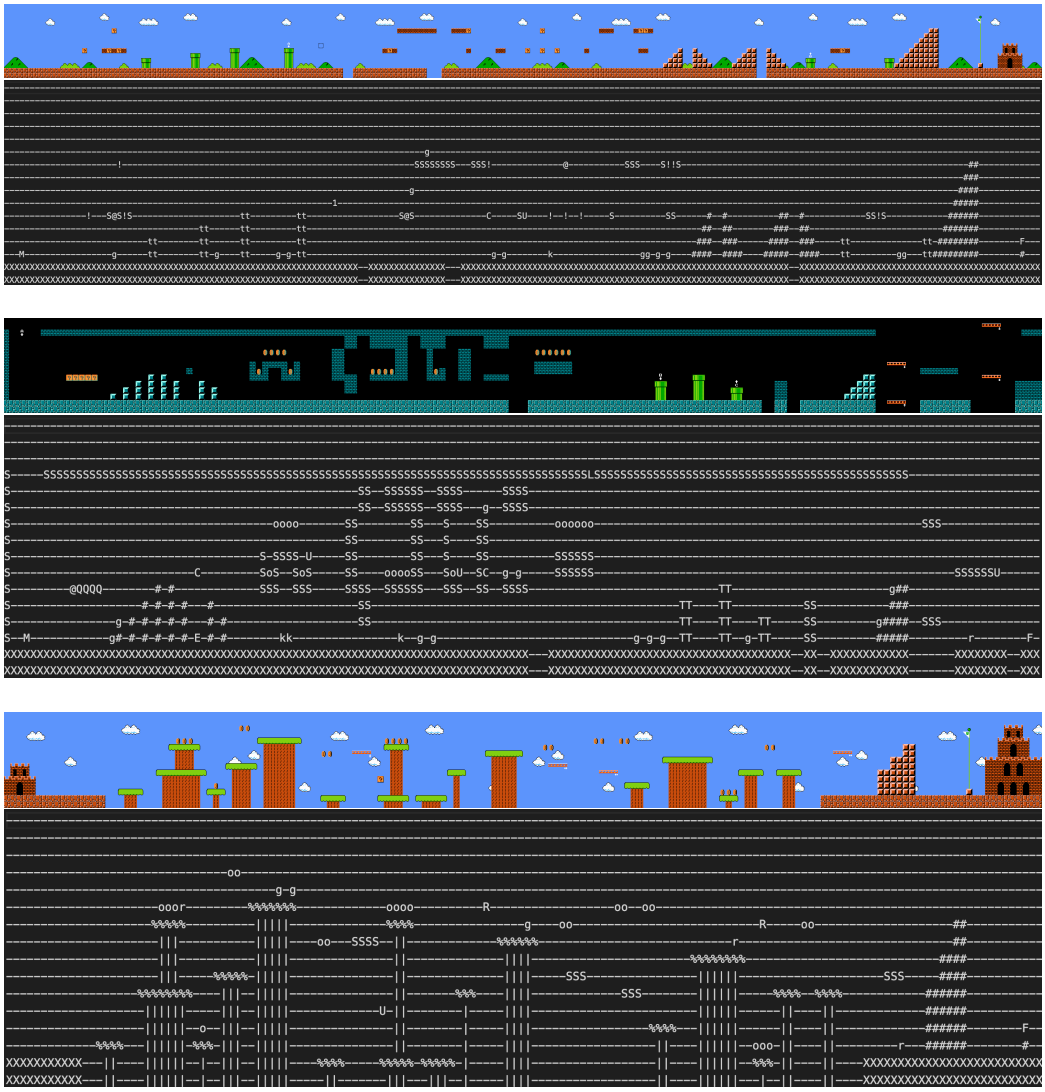


Figure 3.3: Levels 1, 2, and 3 from the original *Super Mario Bros.* accompanied by the text file representations used as an input image for WFC. Level images from [1].

3.3 Parameterization for Generated Levels

In order to test the effectiveness of WFC and its proposed modifications, I used three separate levels from the original *Super Mario Bros.* The levels chosen were the very first three

as they appeared both visually distinct with differing terrain layout and varied heuristically when analyzed with the density and linearity functions I used to later assess the outputs.

I used each of these three levels as the input image for the WFC algorithm to generate 1000 output levels per trial. Eight trials were performed on each level, the first being a standard run of WFC with no modifications and a standard 2x2 snapshot size. The remaining seven involved the use of one altered parameter from the modifications listed below. The output was kept at a consistent size across all three levels of $x = 202$ and $y = 16$ which, out of pure choice, matches the size of level 1.

3.3.1 Forced Tile Placements

Three trials of WFC were run for each level using three configurations of the forced tile placement modification. In the first, I forced a single ground tile (figure 3.4) near the lower left edge of the output at $x = 3$ and $y = 15$. The second was the same as the first with an added forced ground tile at the lower right edge of the output at $x = 198$ and $y = 15$. The third was the same as the second with a final ground tile forced near the lower middle at $x = 101$ and $y = 15$.

Figure 3.4: A ground tile, represented by 'X' in the text file representations of levels.

As a common issue with level generation using unmodified WFC is a lack of platforms for Mario to stand on, the goal of these configurations was to influence the algorithm into creating a consistent plane of ground near the bottom of the level so that Mario had more of a chance of making it through. Ground tiles typically occur adjacent to each other in the

input levels, so forcing one into an area typically encouraged the algorithm to place more next to it.

It should be noted that this modification is in some sense a glorified paintbrush, allowing the user to place tiles nearly wherever they chose. As such, I aimed to be adequately frugal in forcing tiles so as not to take too much of the work away from WFC in its generation and, in the process, undermine the goal of generating levels with PCG.

3.3.2 Bounded Tile Appearances

Two further trials of WFC were run for each level using two configurations of the bounded tile appearance modification. In the first, ground tiles were allowed only to appear below $y = 10$ of the output image. In the second, that limit was pushed further down to $y = 13$. For level 3 in particular, the most common ground tile was different than in levels 1 and 2; platforms were made predominantly of a tile represented by a '%' character, so that tile was restricted by the bounds instead.

Similarly to the forced tile placements, these configurations focused on influencing the ground beneath Mario's feet. In many unmodified level generations, floating platforms tended to appear sporadically at any height in the output. As such, these configurations aimed to push those floating platforms further towards the bottom of the level to keep Mario's footing more smooth and to keep potentially important tiles (like the goal flag) within reach. As there would be less extra platforms near the top of the level, I also considered that these modifications may increase overall linearity of the output.

3.3.3 Snapshot Size Variations

The final two trials were run, not with modifications to WFC, but instead with variations to snapshot size and dimensions. In the first, I extended the size of the snapshots from 2×2

to 3×3 in order to capture more information within one snapshot. For the second, I used snapshot dimensions of 2×16 to alter the shape of the snapshots into a vertical slice of the level as opposed to small square.

The aim of these configurations was to look for differences in output when WFC captured more or different information from the input image. I believed that more information given to each snapshot could increase the playability and create outputs that look more visually like a canonical *Mario* level.

3.4 Analysis

After generating 1000 levels for eight configurations on three levels, I ran the resulting levels through two of the expressive range heuristics previously mentioned along with a game-playing agent to assess playability. I chose density and linearity for the heuristics and programmed an evaluation function

3.4.1 Density

Density was the first heuristic I chose to analyze the level, and I created an analysis function based on the Horn *et al.* [6] specification mentioned previously. However, the method with which to normalize the count of different possible y positions in the level was somewhat unclear, so I settled on using the total area of the level. For all outputs, this would remain constant as all outputs were specified with a size of 202×16.

3.4.2 Linearity

I chose linearity as the second heuristic and created an analysis function based on the methods previously specified. Between Smith and Whitehead's [14] method of using platform

midpoints, and Horn *et al.*'s [6] method of using platform endpoints, I sided with platform endpoints. Due to the focus in previous sections on ground tiles, I would like to note that the density function does not consider only the 'X' ground tiles in its calculation. There are many other tiles that Mario is capable of standing on (like question tiles or breakable bricks) which were necessary to factor in and affected the output of this heuristic.

3.4.3 Playability

The final heuristic used to evaluate levels did not come from expressive range; instead, a Java-based framework for generating and playing *Super Mario Bros.* levels [17] ran an A* game-playing agent on all of the levels generated by the WFC algorithm. Each run was given a 20-second timer (the standard specification of the framework), and upon completion, would specify if the agent completed the level, died, or ran out of time. The framework also reported many other stats concerning the agent's actions and performance during playtime. From these stats, I included completion percentage, signifying the amount of the level the agent made it through regardless of the result, (with 100% signifying a completed level), in the final data aggregations. The goal of collecting these stats from this agent was to asses how completable levels generated by a given WFC configuration and to assess, to some extent, how difficult (or otherwise how feasible) it was to complete these levels on average.

Chapter 4

Results

After the algorithm generated the levels and the heuristics were processed, I averaged and totaled the values of the statistics collected and produced one table for each level (tables 4.1, 4.2, **??**). Each row is a different heuristic, and each column is a different configuration of the modifications on WFC. The first two statistics are the average density and linearity. Following that are the A* agent's total successes, failures, and time-outs when run on the 1000 levels produced. Following these is a percentage that is simply the total number of successes divided by 1000 to better contextualize the agent's rate of success. Finally, Average Completion Percentage is the average of the level's completion percentage (as mentioned in section 3.4.3) reported by the agent for each level, and Failure Completion Percentage is the average of the agent's reported completion percentage only for levels that were failed or timed-out.

The names used for each of the different WFC configurations are defined below:

Standard - WFC with a 2×2 snapshot size and no modifications used

Bounded1 - ground blocks allowed only to appear below $y = 10$

Bounded2 - ground blocks allowed only to appear below $y = 13$

Forced1 - One ground block forced at $(x, y) = (3, 15)$

Forced2 - Ground blocks forced at $(x, y) = (3, 15)$ and $(195, 15)$

Forced3 - Ground blocks forced at $(x, y) = (3, 15)$, $(101, 15)$, and $(195, 15)$

3x3 Snap - snapshot dimensions changed to 3×3

Column Snap - snapshot dimensions changed to 2×16

As seen in these tables, level 2 on average tends to produce the most dense of the levels while level 3 tends to produce the least dense across most configurations. This matches the density of the original input levels when run through the density function (level 1 = 0.074, level 2 = 0.126, level 3 = 0.055). In terms of Linearity, however, The levels are much tighter in terms of average linearity; though, for the Bounded and Forced configurations, level 2 is consistently the least linear with either level 1 or 3 being the most (for reference, the linearity of the original levels is: level 1 = 0.002, level 2 = 0.001, level 3 = 0.0005). For playability, the number of victories almost always decreases from level 1 to level 3, with the exception of the Bounded2 configuration where level 3 sees more victories than either of the other two levels.

Of note is the Column Snapshot configuration, which tends to be an exception to many of the previously stated rules and leads to heavy increases in playability statistics. For levels 1 and 2, this configuration also has a profound effect on linearity, leading to a substantial increase compared to all other trials.

In addition to the aggregate charts for each level are candlestick graphs to present a better understanding of the distribution of relevant values across all 1000 outputs of each WFC configuration (figures 4.1, 4.2, 4.3). For each of the three levels, there is a graph showing the distributions of density, linearity, and failure completion percentage. These help to further visualize some of the trends previously discussed and to easier compare trends across configurations of the same level.

	Standard	Bounded1	Bounded2	Forced1	Forced2	Forced3	3x3 Snap	Column Snap
Avg. Density	0.060	0.047	0.041	0.061	0.062	0.068	0.045	0.059
Avg. Linearity	0.037	0.032	0.033	0.039	0.037	0.039	0.049	0.104
No. Success	168	82	30	133	149	161	157	706
No. Failures	803	785	681	862	849	829	816	275
No. Time-Outs	29	133	289	5	2	10	27	19
Success %	16.80%	8.20%	3.00%	13.30%	14.90%	16.10%	15.70%	70.60%
Average Completion %	43.18%	37.79%	29.14%	47.85%	40.83%	45.57%	39.51%	83.04%
Failure Completion %	31.70%	32.23%	26.95%	39.85%	30.47%	35.13%	28.25%	42.30%

Table 4.1: Aggregated stats from all trials with Level 1 as input.

	Standard	Bounded1	Bounded2	Forced1	Forced2	Forced3	3x3 Snap	Column Snap
Avg. Density	0.083	0.069	0.067	0.089	0.089	0.094	0.087	0.089
Avg. Linearity	0.023	0.024	0.021	0.021	0.022	0.020	0.050	0.083
No. Success	92	29	24	108	98	129	112	555
No. Failures	875	939	941	881	885	845	867	429
No. Time-Outs	33	32	35	11	17	26	21	16
Success %	9.20%	2.90%	2.40%	10.80%	9.80%	12.90%	11.20%	55.50%
Average Completion %	36.44%	28.21%	26.12%	40.63%	35.86%	36.31%	36.15%	76.89%
Failure Completion %	30.20%	26.07%	24.30%	34.04%	29.32%	27.09%	28.24%	48.01%

Table 4.2: Aggregated stats from all trials with Level 2 as input.

	Standard	Bounded1	Bounded2	Forced1	Forced2	Forced3	3x3 Snap	Column Snap
Avg. Density	0.025	0.019	0.015	0.024	0.024	0.024	0.025	0.106
Avg. Linearity	0.031	0.042	0.051	0.023	0.027	0.024	0.052	0.029
No. Success	20	29	41	15	6	9	6	127
No. Failures	978	967	956	985	994	991	994	870
No. Time-Outs	2	4	3	0	0	0	0	3
Success %	16.80%	8.20%	3.00%	13.30%	14.90%	16.10%	15.70%	70.60%
Average Completion %	17.32%	21.15%	21.95%	13.90%	12.32%	12.70%	16.02%	29.76%
Failure Completion %	15.71%	18.87%	18.69%	12.77%	11.80%	12.00%	15.51%	19.94%

Table 4.3: Aggregated stats from all trials with Level 3 as input.

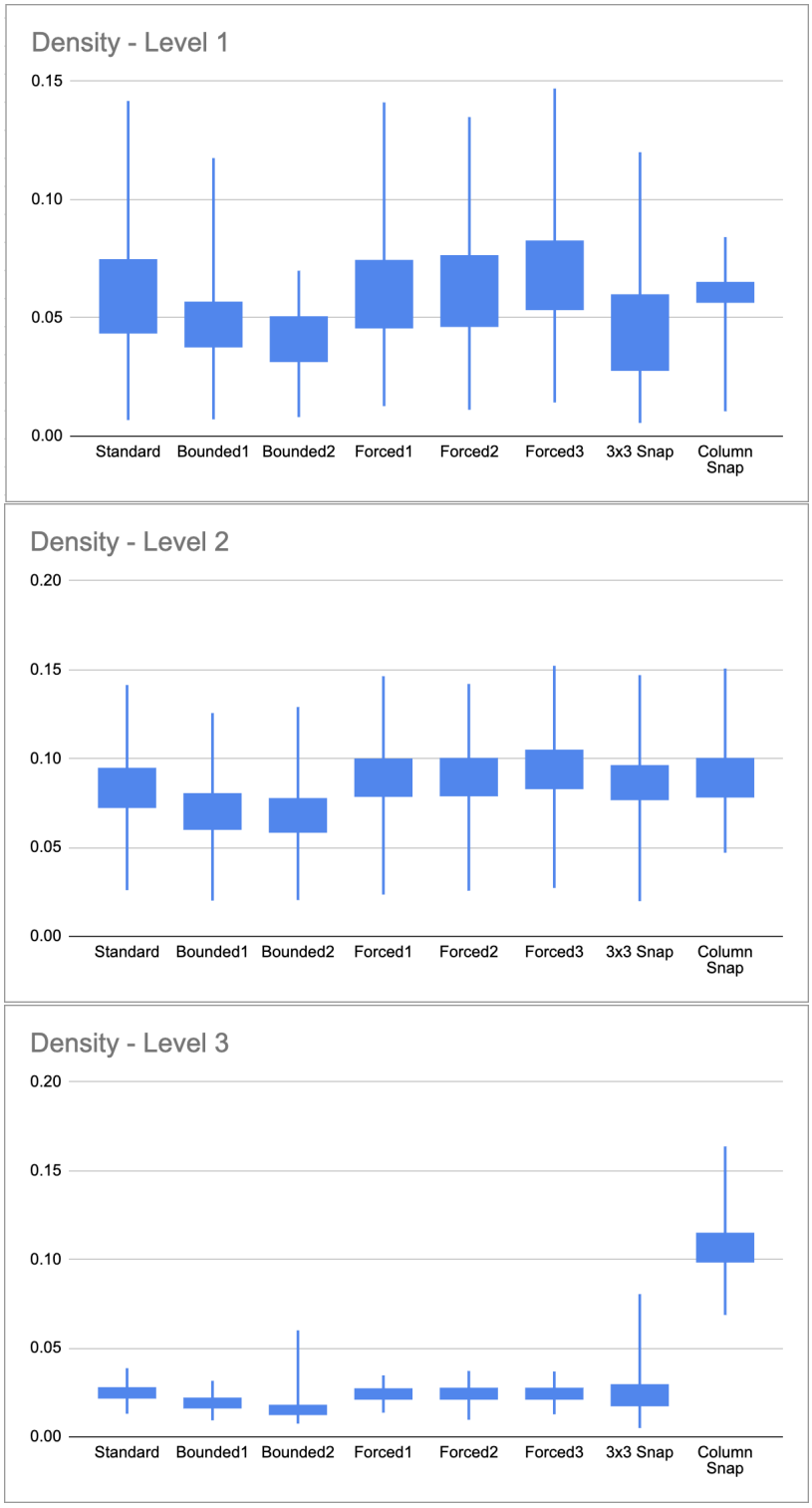


Figure 4.1: Candlestick graphs for density

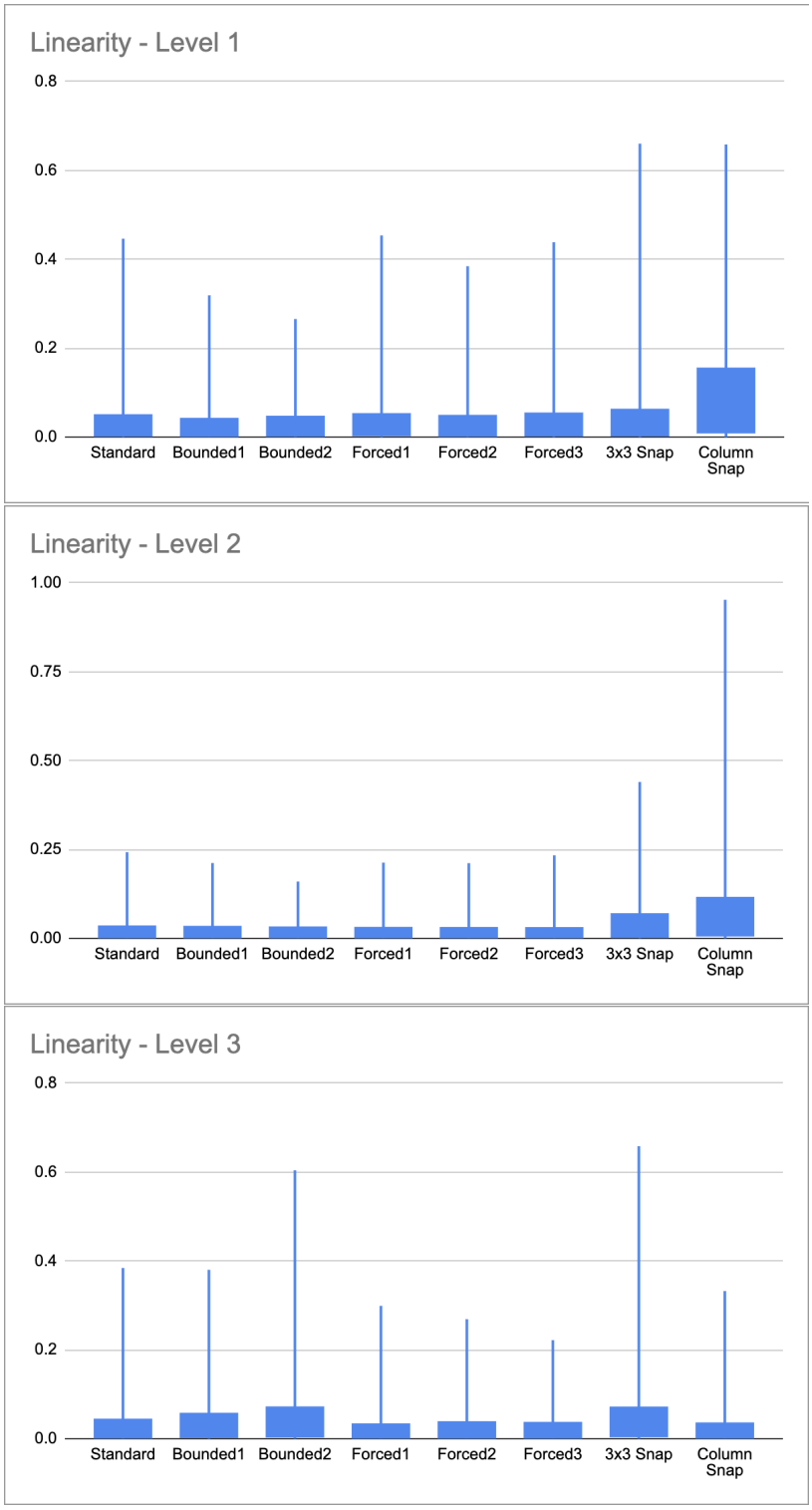


Figure 4.2: Candlestick graphs for linearity

Figure 4.3: Candlestick graphs for failure completion percentage

Chapter 5

Discussion

5.1 Standard Wave Function Collapse Generation

Wave Function Collapse, when unmodified, generates serviceable, yet rather unrealistic levels. Many are not completable, and even those that are contain many floating platforms and out-of reach locations. From the three levels used as input for WFC, the resulting outputs show a variation in many of the heuristics used to evaluate them. Level 2 is shown to produce the most dense levels on average while level 3 produces the least dense. Conversely, average linearity remains approximately constant across the output of the three levels. Finally, playability, concerning both the raw number of levels completed and completion percentage, decreases as the levels progress, with a significant drop for levels produced by level 3. This is understandable as the game-playing agent used was unable to complete even the original level 3.

In following discussion of the outputs of modified WFC, the results and observations of these standard WFC trials will serve as a baseline to compare against.

5.2 Bounded Tile Appearances

This section examines the results of of both applications of the bounded tile appearances modification (Bounded1 and Bounded2 as defined earlier in chapter 4) and examines their affect on each heuristic in turn.

5.2.1 Density

Through both uses of bounded tile placements, WFC produces levels with more air-space near the top and more structures near the bottom. This does, however, vary by level. Level 1 is, for the most part, a straight line with structures attached to the ground near the bottom; any floating block structures are smaller and more sparse. As a result, restricting ground tiles near the bottom results in more empty space near the top (as seen in figure 5.2), and this is reflected in a decrease in average density compared to standard WFC, along with a tightening of the total ranges of density, shown in the candlestick graph. Level-2-produced levels also drop in average density, but the tightening of the total range of densities was not as substantial as level 1. Level 2 originally has many more structures hanging in the air made of different tiles that were not restricted by the bounding modification; this allows the algorithm the freedom to generate levels with less empty space towards the top and more platforms (as seen in figure 5.1), leading to some levels with higher density. Levels generated from level 3 act much the same as levels generated from level 2 when concerning density, though, of all three levels, it has the lowest average decrease in this heuristic compared to unmodified level-3-produced levels.

Figure 5.1: Output 15 of the Bounded1 configuration generated from level 2

5.2.2 Linearity

I predicted previously that the use of the bounded tile appearances modification would lead to an increase in linearity of the produced levels. However, for levels 1 and 2, this prediction is not true. Outputs produced by these two levels remain similar to unmodified WFC concerning average density, with only a small increase by tenths of a percent. The candlestick graphs for these two levels also do show a tightening in the overall range of linearity values, but it was not as harsh as for density for level 1. Levels generated from level 3, however, did display the predicted increase in linearity when bounded, and, furthermore, the y-depth to which it was bounded positively affected the amount it increased.

This phenomenon is likely due to a point that was previously touched on when describing density. Both level 1 and 2 (though to varying degrees) contain a number of floating structures, so bounding the ground tiles toward the bottom of the level still allowed for other structures and platforms to appear higher up. Conversely, the vast majority of floating platforms in level 3 are the ones made up of the level's primary ground tile (the '%' block), so, when this tile is bounded by the modification, WFC is not motivated to populate the higher areas of the level with many solid blocks. This leads to a smoother level with less height variation as signified by a higher linearity.

5.2.3 Playability

Somewhat counterproductive to the intent of these configurations, for levels 1 and 2, bounding appears to have decreased the overall playability of the resulting level. The number of successes of the agent playing the level dropped drastically, along with the total average completion percentage. However, there is a key difference between these two levels; the level-1-produced levels saw a heavy spike in the number of time-outs in which the agent ran out of time to complete a level. This is likely due to the fact that many of the levels produced by level 1 with bounding generated high walls which Mario was unable to progress past. Level 2-produced levels, however, do not see the same spike; instead the total number of standard losses increases with the addition of bounding while time-outs remain relatively consistent. For level 2, then, it appears that keeping the main ground area restricted to lower y levels is counterproductive to creating a playable level.

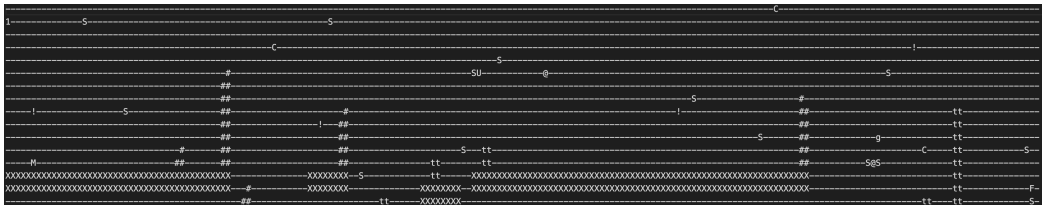


Figure 5.2: Output 23 of the Bounded2 configuration generated from level 1: an example output demonstrating the high walls restricting progression, seen near the left in front of Mario's starting location.

Level-3-produced levels, however, do experience an increase in both victories and average completion percentage. In fact, even among the levels not successfully completed, the agent is able to make it further in the level, demonstrated by the average failed completion percentage. Once again, the absence of solid tiles that were not restricted by the bounded tile appearance modification in the original level leads to less variation in the output overall

when the solid blocks that do exist are bounded. The implications this has on playability are that the agent is less likely to encounter a level where it is hindered by sporadic placement of other solid blocks or kept from reaching a flag that appears too high for it to reach.

5.3 Forced Tile Placements

This section examines the results of of the three applications of the forced tile placements modification (Forced1, Forced2, and Forced3 as defined earlier in chapter 4) and examines their affect on each heuristic in turn. As previously stated, one of the primary goals of the three forced tile placement configurations was to create a coherent plane of ground that may help Mario to make it further through the level. Many of the created levels demonstrate successful production of this effect (as seen in figure 5.3), especially in the third configuration, Forced3. However, it should be noted that these three configurations are next to useless when applied to level 3; the ground 'X' tiles forced in the first two levels are sparse in the third level, so long platforms at the bottom almost never appear. This, in part, helps to demonstrate the way WFC takes snapshot frequency into consideration, and the effects of this will be discussed in the following.

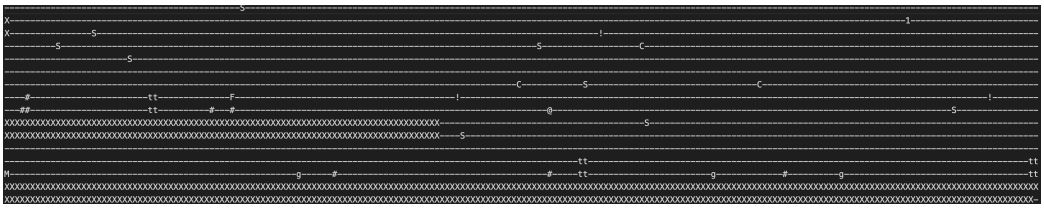


Figure 5.3: Output 43 of the Forced3 configuration generated from level 1

5.3.1 Density

While this configuration places ground tiles near the bottom of the level similar to what the bounded configurations achieved, it does not restrict them from appearing higher up. Thus, contrary to bounded configurations, average density tended to increase compared to levels generated by standard WFC. We can see this effect most clearly in levels 1 and 2, with a greater increase as more tiles were forced in from forced1 to forced3. The candlestick charts demonstrate this as well with higher centers but none of the shrinking of value ranges as seen previously in bounded tile appearances.

Level 3, in contrast, remains constant in density, a fact shown in both the the averages and candlestick chart. Adding ground blocks near the bottom does not influence WFC to add entire platforms, resulting in only small clusters near the areas where tiles were forced. Because fewer blocks were added, overall density of the levels saw very little change.

5.3.2 Linearity

For level 1, linearity saw a very slight increase compared to standard WFC levels, though only by a few tenths of a percent. An increase of this size is not entirely meaningful, but it stands to reason that, when entire platforms are generated along the bottom of a level (as is especially common from Forced3), it would create a slightly more linear level. This differs from the bounded configurations which forced ground tiles lower in the level, but did not grantee that they would appear in a straight line.

In contrast, levels generated from levels 2 and 3 with these configurations suffered a decrease in linearity. For level 2 the decrease was slight, but 3 suffered a more noticeable decrease. This is likely due to the fact that these three configurations did nothing to restrict or influence the main ground tile in level 3 (the '%' tile) and simply add a few scattered

platforms near the bottom (seen in figure 5.4), creating a less linear level. The concept is similar for level 2, though, in this case, the forced tiles do serve as the original level's ground; thus they appear more often and allow the configurations to take more of an effect. However, the frequent presence of different types of tiles commonly appearing in the level leads to more variation throughout the output, so forcing blocks to appear near the bottom only serves to make the level more linearly erratic.

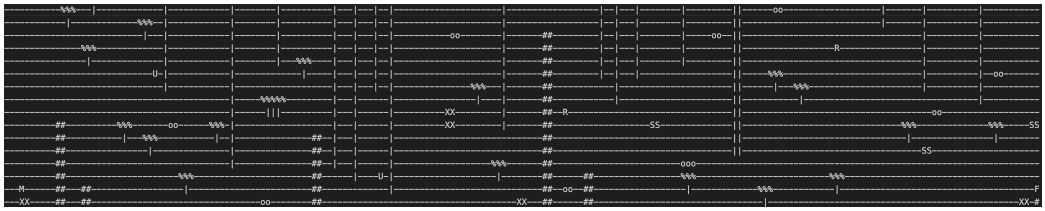


Figure 5.4: Output 10 of the Forced3 configuration generated from level 3

5.3.3 Playability

For level-1-produced levels, the number of total levels completed by the agent decreased for all configurations of the forced tile placement modification when compared to standard WFC. Despite this, the average level completion percentage for the agent's failed attempts generally increased. As discussed previously concerning linearity and density, the levels tended to see more of a linear plane of ground near the bottom, while still experiencing a high degree of variation in higher areas of the level. This likely tends to allow the agent to travel a greater distance in the level, even if the level is eventually failed. Level-2-produced levels, in contrast, saw a slight increase in overall levels completed as well as an increase in average completion percentages; we saw an increase in average density with level 2, so a longer flat platform from the beginning of the level may have aided the agent in orienting itself before navigating its way to the goal in other varied parts of the level.

Finally, levels produced from level 3 saw a sizeable dip in overall completion. This is likely because the forced configuration would typically create a small platform at the bottom left of a level which, due to the low frequency of ground blocks in the input level, would not extend very far. The framework would then use this platform as Mario's starting location, and, because the rest of the level contained sparse and scattered platforms, the agent would likely have found it more difficult to reach any other platforms from its starting platform.

5.4 Altered Snapshots

This section examines the results of the two alterations in snapshot size (3x3 Snap and Column Snap) and examines their affect on each heuristic in turn. As mentioned previously, the goal of these configurations was to observe the behavior of WFC when it took different amounts or kinds of information out of the input image. The results were pleasing and demonstrate the amount of control that altering snapshot size has.

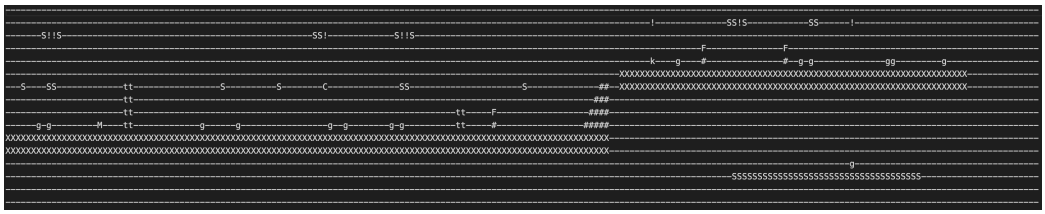


Figure 5.5: Output 183 of the Column Snapshot configuration generated from level 1.

5.4.1 Density

Because the 3x3 sized snapshot allowed WFC to capture more of the original image within each snapshot, the outputs display more recognizable occurrences of structures from the original image, like the stair-step structures from level 1 (seen in figure 5.6). However, for

level-1-produced levels, the average density of the output experiences a decrease; though the algorithm is able to generate more coherent structures, it also inserts more empty space. This is likely because a 3x3 snapshot size naturally allows for the collection of less overall snapshots from the input, and, due to the large presence of empty space, snapshots with empty space will be proportionally more common. Level 2 did not see the same decrease in density; in fact, average density increased very slightly. Level 2 contains less empty space than level 1, thus, the larger snapshot size likely did not take the same effect as in level 1. The same is true of level 3, only, in this case, the candlestick graph reveals that the maximum density value increased drastically, although the average remained largely unchanged. This is likely because, contrary to level 2, level 3 was not very dense to begin with, so the extra snapshot size did little to increase the amount of empty space in the output.

Figure 5.6: Output 7 of the 3x3 Snapshot configuration generated from level 1: an example of the more coherent stair-step '#'-tile structures seen when generated from level 1 with a 3×3 snapshot size.

Similar to the 3x3 snapshot, the column-shaped 2x16 snapshot produced varying effects on the density of each level. It did not significantly affect the average density of level 1 levels, but the candlestick graph reveals an extreme tightening in the first and third quartile, along with a heavy drop in the absolute maximum density value produced. Columns are a much more restrictive snapshot shape concerning the adjacency rules they must follow, so when

one is decided, it tends to create a straight line that more closely approximates the layout of the original level (seen in figure **??**); this keeps the average density similar, but, because it does not have as much varietal freedom as the unmodified WFC, it is unable to produce levels that are as dense at the extremes of the distribution. This phenomenon is even more clear in level-2-produced levels where the average density saw hardly any change, and the candlestick chart for standard and column snap are nearly identical. Level 3, however, saw a massive increase in density compared to standard WFC generations. Though it should be noted that, for unknown reasons, a number of columns on the right side of many of the generated levels were filled totally with solid blocks, this phenomenon (whether it be an error or a natural product of this snapshot shape on this level), was not the only reason for an increase in density. Without it, the density increase may have shrunk slightly, but the generated levels also displayed a much larger number of solid platforms than levels generated by standard WFC; moreover, many of these platforms extended much longer than they appeared in the original level. This is likely due, once again, to the restrictive nature of adjacency rules with a snapshot that spans the height of a level.

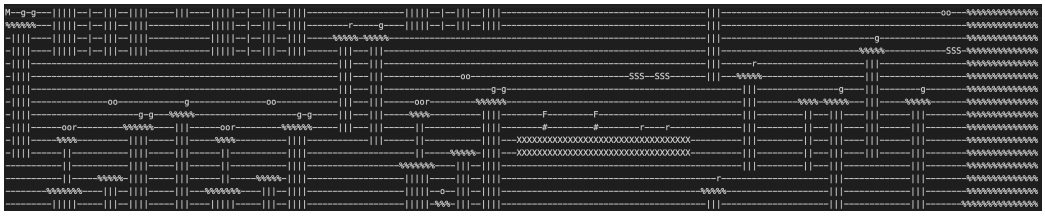


Figure 5.7: Output 174 of the Column Snapshot configuration generated from level 3: An example of a more dense level-3-produced level including longer platforms. The fill of solid bricks can also be seen on the right side.

5.4.2 Linearity

When concerning linearity, both configurations of altered snapshots resulted in about the same effect, only to differing degrees. In level 1, we see an increase in average linearity compared to the standard for 3x3 and an even greater increase for the column snapshots. The same is true of level 2, but level 3 sees a drop in its average linearity for column snapshots.

As mentioned in the discussion of density, the 3x3 snapshots resulted in a large amount of empty space for levels 1 and 3. This combined with the more frequent appearance of long linear platforms in outputs from all three levels leads to a higher average linearity as well as an increase in absolute maximum linearity for each. The same is even more true of column snapshots due to their restrictive nature that leads to a high probability of long platforms in the output. Though, concerning the drop in linearity for level 3, it is unfortunately likely that the filled-in right side of many of the level 3 outputs negatively affected the linearity calculation. That said, there are also a number of outputs that contain shorter platforms appearing at many different heights (a phenomenon also demonstrated in figure 5.7), which is a potentially more natural cause for this dip in linearity.

5.4.3 Playability

For all three levels, the effect to related playability statistics (number of successes, average completion percentage, and average completion percentage of failed attempts) varied with the use of a 3x3 snapshot size. Level 1 saw a slight decrease in most of these stats, likely due to the previous conclusions of overall empty space increasing in the output. Level 2, in contrast, saw a slight increase in most of these stats, though, of the levels the agent failed, completion percentage still decreased. Level 3 saw a more substantial decrease in these

stats; once again, it is likely that more common blank space caused the agent to fall out of the level more often.

Compared to the playability results for every other configuration, the results of the column snapshots are the most impressive. Levels 1, 2, and 3 all all saw a massive increase in the raw number of levels completed by the agent, and even the levels that weren't completed had a higher completion percentage on average than levels generated by standard WFC. The direct effects of the column snapshots have previously been discussed, but, once again, the lack of freedom concerning adjacency rules leads to long linear platforms in the level which are likely to have a goal tile near the end.

Chapter 6

Future Work

6.1 Different Heuristics

When analyzing the expressive range of WFC with different modifications, I only used two of the expressive range heuristics previously mentioned. However, expressive range is a broadening field with more available metrics, so it would be interesting to analyze the produced levels with other heuristics. In particular, leniency would be an interesting measure to use; I used an A* agent to roughly assess the playability of a level (a method which is already not perfect), but this does not produce an exact measure for the difficulty of a level. Using the leniency heuristic and comparing it with the existing playability analysis could be beneficial in learning which modifications have the most implication on difficulty.

As playability is concerned, the A* algorithm used to assess playability could do with some improvements. The algorithm implemented in the Mario framework does not attempt to find a path to the flagpole and instead concerns itself only with getting to the right hand side of the small screen that is visible to Mario. As a result, this agent is not the perfect agent for finishing levels, and when run on the the fifteen original Mario levels included in

the framework, it only manages to complete eight of them; level 3, which was used in the generation of this thesis, was one of the levels it could not complete. As such, there is a high likelihood that it failed a number of the WFC-generated levels that would otherwise be beatable by a more skilled player or agent. In the future, a more competent A* should be used to better assess level playability.

6.2 Use of Current Modifications

Though the configurations used for the two current modifications of WFC presented interesting and varied outputs, it is my belief that they have much potential for their capabilities.

I referred earlier in 3.3.1 to the forced tile placements modification as a "glorified paintbrush" and noted that I did not want to do too much of the algorithm's job when placing blocks in the level. However, it would be interesting to analyze the results of a level that had more complex structures already filled in. For instance, allowing WFC to build around a level that already has a complete floor or pre-made obstacles filled in could allow the designer to ensure playability while still receiving varied results.

Similarly, the bounded tile appearances modification could be used to restrict more than just ground blocks to the bottom areas of the level. Tiles like enemies or powerups could be confined to certain areas of the level on either axis in order to add a gradual progression of difficulty throughout a level. For instance, weaker enemies could appear further toward the left with more frequent powerups, while stronger enemies appear towards the right as powerups become more scarce.

Finally, though this was not strictly a modification of WFC, the variation of snapshot size and shape showed a significant amount of promise concerning the generation of natural-looking and playable levels. More experimentation with these parameters alone could lead

to a better understanding of how they could be used to a developer's advantage.

6.3 Further Modifications

Finally, the world of possibilities for modification of WFC is yet a broad one. This thesis by no means exhausts the potential for modification on this algorithm, and, as such, I would like to propose a few more modifications for consideration in the future.

6.3.1 Forced Tile Frequencies

This modification takes inspiration from the idea of snapshot frequencies, applies it to individual tiles, and turns it into a strong requirement as opposed to a soft one. The general idea is to consider the number of times a certain tile appears in the input image and force it to appear exactly as many times in the output. Mario levels, for instance, always contain one flagpole, so it would be useful for developers to ensure that a flagpole always appears exactly once in the generated level. For applications in other games, like *The Legend of Zelda* where a certain number of keys must appear so that a player may unlock every door, it would be helpful to feed the algorithm a level with a matching number of keys and doors and return an output with the same amount, helping to ensure the playbility of a level. Of course, this modification does not necessarily have to be limited to matching the number of a specific tile in the input; it could instead be extended to allow the designer to chose how many of a certain tile should appear in the output regardless of how many appeared in the input.

Primitive functionality of this algorithm was implemented into the generation code for this thesis, specifically concerning the Mario tile and the Flagpole tile. In order for the *Mario* framework to run a level, it required Mario and a Flagpole to be present. Therefore,

after generation, if a level lacked either of these, one was placed in (with Mario placed on the farthest left solid block and the flagpole placed on the farthest right).

6.3.2 Removal of Screen-Wrapped Adjacency Rules

The other modification is one that we believe would significantly alter the output of the algorithm. As Wave Function Collapse works currently, the input image is "screen-wrapped" on itself; snapshots at the outer edge of the input are adjacent to snapshots at the opposite edge. This is only natural, as the alternative is to generate snapshots with no adjacency rules on one or more sides. This presents a problem during the observation phase as, if one of these snapshots is chosen to appear anywhere in the middle of the image, it will immediately eliminate all snapshots in one of the adjacent stacks where it has no adjacency rules. This causes the algorithm to enter an impossible state and terminate.

Thus, screen-wrapping is difficult to avoid, and it has significant implications on the outputs it is able to generate. Consider, for example, level 1 from the canonical levels. The ground tiles only appear at the bottom of the screen, adjacent to the bottom edge. However, as far as WFC is concerned, these ground tiles are adjacent to the sky tiles at the very top of the image. As a result, when an output is generated, these tiles are allowed to appear at almost any height, resulting in a number of floating platforms.

However, with a functioning workaround to the screen-wrapping requirement, WFC would be capable of generating levels much closer to the original levels, with ground blocks only appearing near the bottom where the ground should be and only generating floating platforms when the original level contained floating platforms. The changes to output that this modification could bring are so potentially profound that this would be our first target upon further work in this topic.

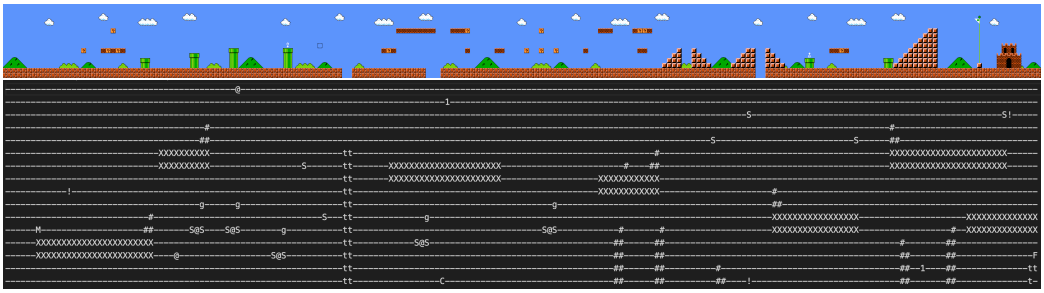


Figure 6.1: An example output (below) produced from level 1 (above) with standard WFC. Ground tiles ('X') are able to appear at any height.

Bibliography

[1] NES - Super Mario Bros. - The Spriters Resource. https://www.spriters-resource.com/nes/supermariobros/.

[2] Brian Cullen and Carol O'Sullivan. A caching approach to real-time procedural generation of cities from gis data. 2011.

[3] Steve Dahlskog and Julian Togelius. Patterns as objectives for level generation. In *Design Patterns in Games (DPG), Chania, Crete, Greece (2013)*. ACM Digital Library, 2013.

[4] Maxim Gumin. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse, April 2021.

[5] Matthew Guzdial, Duri Long, Christopher Cassion, and Abhishek Das. Visual procedural content generation with an artificial abstract artist. In *Proceedings of ICCC computational creativity and games workshop*, 2017.

[6] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. A comparative evaluation of procedural level generators in the mario ai framework. In *Foundations of Digital Games 2014, Ft. Lauderdale, Florida, USA (2014)*, pages 1–8. Society for the Advancement of the Science of Digital Games, 2014.

[7] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC Workshop on Computational Creativity and Games*, page 9, 2016.

[8] Isaac Karth and Adam M Smith. Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, pages 1–10, 2017.

[9] Hwanhee Kim, Seongtaek Lee, Hyundong Lee, Teasung Hahn, and Shinjin Kang. Automatic generation of game content using a graph-based wave function collapse algorithm. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2019.

[10] Antonios Liapis, Christoffer Holmgård, Georgios N Yannakakis, and Julian Togelius. Procedural personas as critics for dungeon generation. In *European Conference on the Applications of Evolutionary Computation*, pages 331–343. Springer, 2015.

[11] Markus Lipp, Daniel Scherzer, Peter Wonka, and Michael Wimmer. Interactive modeling of city layouts using layers of procedural content. In *Computer Graphics Forum*, volume 30, pages 345–354. Wiley Online Library, 2011.

[12] Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. Enhancing wave function collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–9, 2019.

[13] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311. IEEE, 2012.

[14] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–7, 2010.

[15] Adam Summerville, Matthew Guzdial, Michael Mateas, and Mark Riedl. Learning player tailored content from observation: Platformer level generation from video traces using lstms. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 12, 2016.

[16] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[17] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[18] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.