Wave Function Collapse as a Constraint Solver for Game Level Generation

CS4099 Major Software Project

Benjamin Mathias Sonnet
Supervised by David Morrison

University of
St Andrews

Computer Science
University of St Andrews
March 2024

Abstract

Wave Function Collapse (WFC) is limited by a lack of global constraints, poor performance and restriction to a finite grid. These issues are rarely addressed directly in implementations. Instead, they are worked around in an ad hoc, game-specific way that fails to exploit constraint programming techniques. Furthermore, presentation of WFC online often fails to acknowledge underlying constraint solving principles used by WFC.

This dissertation seeks to examine how these issues can be addressed and attempts to contextualise WFC within theory of constraint programming. To do this, a simple-tiled implementation of WFC using the Maintaining Arc Consistency 3 (MAC3) algorithm is presented. WFC is extended to generate infinite worlds using Infinite Modifying In Blocks (IMIB). An interface is provided for the Unity Editor that allows designers to specify their own tile set to use for generation. As example, a game themed after the popular fictional concept of *the Backrooms* is created. Weighted tile selection gives the level designer global control over the percentage of each tile in the output level. However, loading infinite worlds in real-time presents additional performance challenges and further global constraints are needed to improve control further.

Overall, this dissertation brings together WFC and constraint programming concepts with extensive examples to support understanding and further development of WFC.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 10616 words long, including project specification and plan.

Acknowledgements

I would like to thank all those who supported me throughout the time of the project, from university staff to family and friends.

Contents

Acronyms

AC3 Arc Consistency 3. vi, 30–32, 49, 50, 61

IMIB Infinite Modifying In Blocks. i, vi, 2, 21, 22, 26, 27, 35–39, 51, 54

MAC3 Maintaining Arc Consistency 3. i, vii, 2, 29, 31, 32, 48–51, 54

PCG Procedural Content Generation. v, 1, 4, 5, 7–9, 12, 13, 25, 39

RNG Random Number Generator. 48, 61

WFC Wave Function Collapse. i, v, vi, 1–4, 7, 8, 13–16, 18–27, 29, 31, 32, 43, 44, 48, 51, 52, 54, 61

List of Figures

List of Tables

Chapter 1

Introduction

Wave Function Collapse (WFC) [1] is a Procedural Content Generation (PCG) technique that has seen application across a huge number of fields, from games [2]–[4] (Figure 1.1) to poetry [5] and music [6], [7]. WFC can be described as a family of algorithms that enable generation of large game worlds from limited input through the application of constraint programming principles [8].

Figure 1.1: *Townscaper* uses WFC with player input to develop worlds [2]

WFC's application is commonly limited by its lack of global constraints, overfitting and performance. Lack of global constraints can result in no inherent overall structure to the output, making it homogeneous. Use of complex tile sets can result in overfitting, reducing output variety. Complex tile sets and large grids also result in poor performance and increased failure rate.

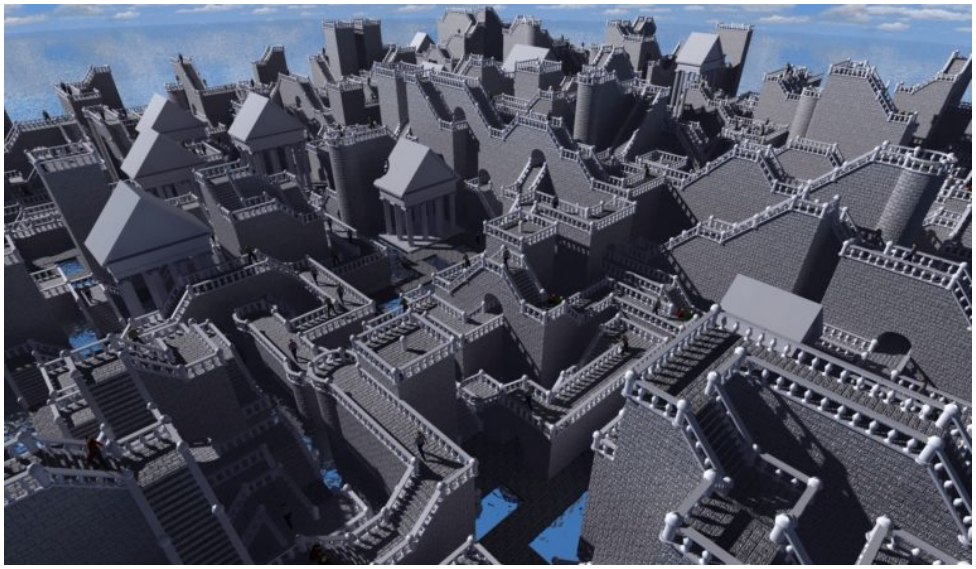


Figure 1.2: A complex Escheresque tile set that relies on modifying in blocks [9]

These issues are rarely addressed directly in implementations. Instead, they are worked around in an ad hoc, game-specific way that fails to exploit constraint programming techniques.

Presentation of WFC online often fails to acknowledge underlying constraint solving principles used by WFC. Instead, alternative wording is used, which can make forming a deep understanding of the topic challenging.

This dissertation draws an explicit connection between WFC and the Maintaining Arc Consistency 3 (MAC3) algorithm, presenting a simple tiled implementation of WFC in those terms. In simple tiled WFC, a tile set with adjacency information is used by a constraint solver. This solver attempts to fill a grid of cells with tiles, taking into account their constraints. In the rest of this dissertation, this simple tiled implementation of WFC is simply referred to as the WFC algorithm.

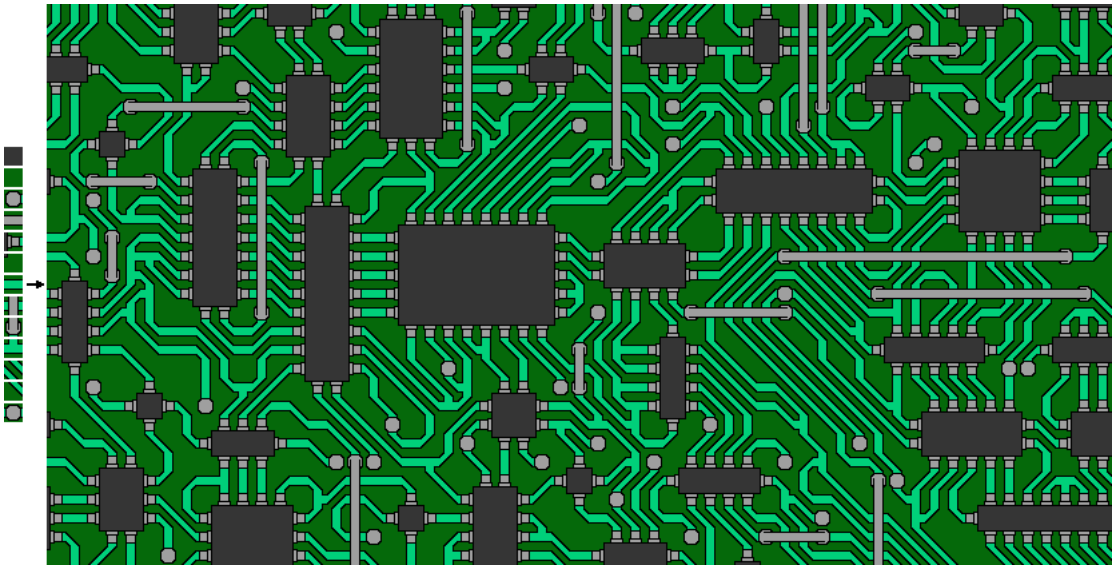


Figure 1.3: Use of simple tiled WFC to generate a circuit board graphic [1]

The simple tiled WFC implementation is extended using the Infinite Modifying

In Blocks (IMIB) algorithm, which addresses the challenge of extending WFC to an infinite space [10]. Furthermore, an interface integrated into the Unity game engine for designers to use WFC on their own tilesets is included. A simple game themed after *the Backrooms* was created in order to act as an example (Figure 1.4).

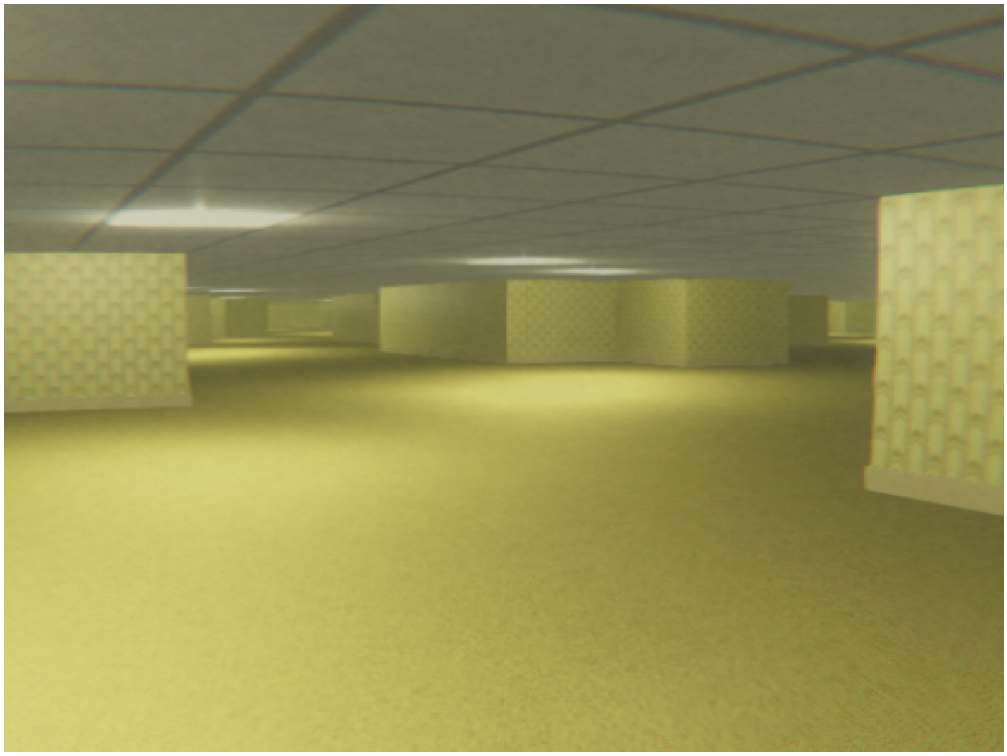


Figure 1.4: A screenshot from the game

Chapter 2

Literature Review

As part of a literature review, three key themes were identified. These include Procedural Content Generation, Constraint Programming and Wave Function Collapse. While these themes are presented in three different sections, the ideas discussed heavily overlap.

In summary, constraint programming and Procedural Content Generation have wide applications and have been a big area of recent research. For example, PCG has seen increased use of machine learning in recent years, allowing new content to be generated from large dataset models. Similarly, novel constraint programming techniques such as WFC have seen use in generating new output from very limited input data. These techniques are typically applied to video games in order to help create an ever-changing experience for the player.

2.1 Procedural Content Generation

2.1.1 Overview

PCG describes the use of algorithms to pseudo-randomly generate content. In the context of video games, this randomisation is often used to provide players with variety that can make games more enjoyable to play multiple times. Procedural Content Generation can be used in many facets of video game development. One frequent use of PCG is in randomised level generation. The term 'roguelike' is used to describe games where dungeon-style levels are procedurally generated. An example of a roguelike game is *Caves of Qud* [4]. How its developers tackled PCG issues is discussed later in Section 2.3.2. In sandbox exploration games such as *Minecraft* [11] and *No Man's Sky* [12], PCG is used to generate the player's world, offering virtually infinite locations to explore. Other uses include randomising enemy characteristics in *Shadow of Mordor* [13] and generating randomised weapons in *Borderlands* [14]. The book *Procedural Content Generation in Games* [15] is a great resource to read more about PCG.

Figure 2.1: *Minecraft* lets the player explore an infinite, procedural world [16]

2.1.2 Current Applications and Research

Many modern PCG research looks into using AI and machine learning to generate content. Some current applications of machine learning are art, music and code generation as well as chatbots [17], text-to-3D synthesis [18] and even self-driving cars [19].

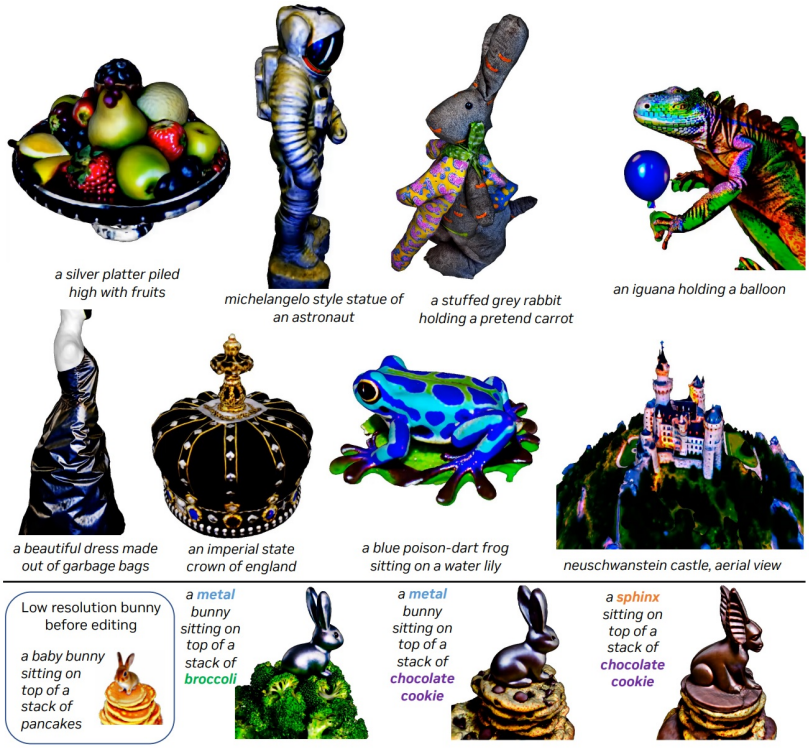


Figure 2.2: Magic3D performs text-to-3D synthesis [18]

Applications such as Stable Diffusion [20] can be used to create high quality text-to-image content (Figure 2.3). Others such as Magic3D [18] offer text-to-3D

synthesis (Figure 2.2). In terms of text-to-text, ChatGPT [21] serves as a leading language model that can be used to converse about any topic, while GitHub Copilot [22] can generate code snippets from user prompts.

Figure 2.3: Stable Diffusion offers text-to-image synthesis [20]

In the context of games, machine learning is frequently used in areas such as text, character model, texture, music and sound generation [23]. Languages such as the Video Game Description Language (VGDL) have even be used to generate entire games using AI (Figure 2.4) [24], [25]. However, by themselves, such languages have limited expressiveness, making it difficult to create interesting games [24].

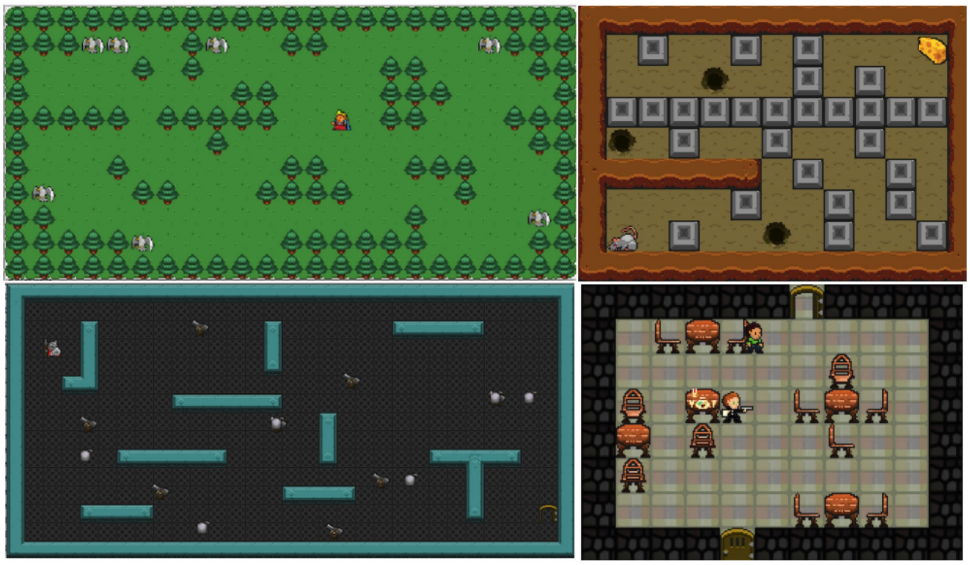


Figure 2.4: Games generated using the Video Game Description Language [24]

R. R. Torrado et al. identify that Generative Adversarial Networks (GANs) can also be used for image generation, but that it is difficult to incorporate constraints [26]. As a solution, they propose a Conditional Embedding Self-Attention Generative Adversarial Network (CESAGAN). This allows the embedding of a feature

vector to the input, enabling the network to model non-local constraints. As a result, this produces higher quality outputs with fewer duplicates as shown in Figure 2.5. In WFC, it is also difficult to enforce non-local constraints without additional modifications.

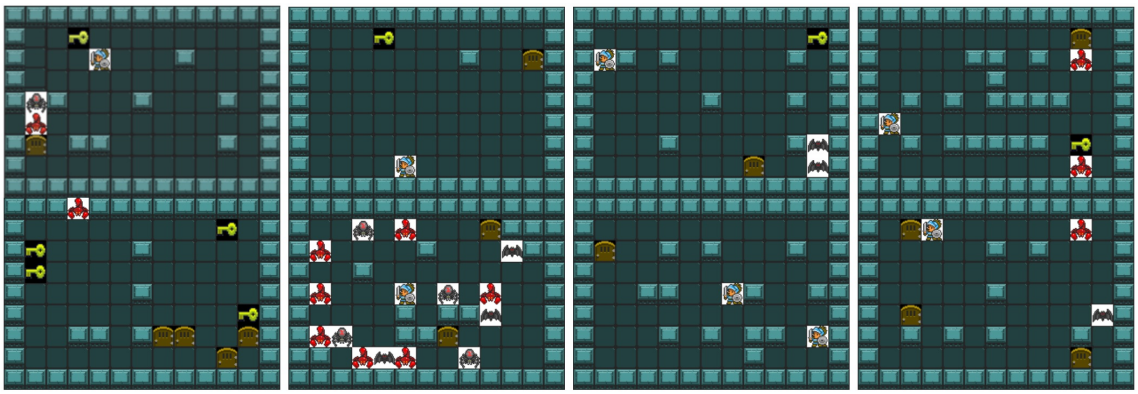


Figure 2.5: CESAGAN extends upon Generative Adversarial Networks to encourage generation of playable levels requiring additional constraints (top) instead of unplayable levels (bottom) [26]

Looking further into machine learning, A. Khalifa et al. transform 2D level design problems into Markov decision processes [27]. This approach aids reinforced learning to produce high quality output levels (Figure 2.6). The authors suggest that this reinforced learning could be applied to self-play agents to improve the content generated through simulated playtesting. Three other papers similarly suggest that machine learning is useful for evaluating content through methods such as simulated playtesting [23], [25], [28]. In the context of WFC, it might be useful to apply machine learning content evaluation and adjust the output to increase its quality.

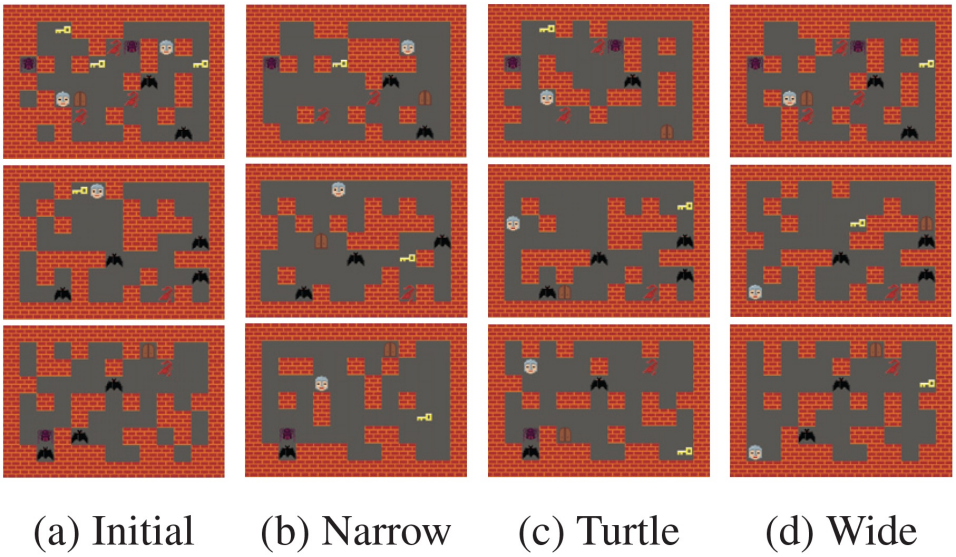


Figure 2.6: Procedural Content Generation via Reinforcement Learning used to create *Zelda* style levels. An initial random layout (a) is used with three different Markov Decision Process Representations (b), (c) and (d) to create playable levels. [27]

A. Summerville et al. comment specifically on two limitations of PCG via machine learning [28]. They state that the playability of output produced through machine learning is not guaranteed to be playable, but rather biassed towards generating playable content through the input. Similarly, whether WFC output is playable or not is not always guaranteed but heavily depends on how the input is defined. As a result, both machine learning PCG and WFC require carefully tweaked input data and an output evaluator when applied to level generation. The second limitation is that most machine learning has been applied to 2D content. Once again, the core WFC implementation and many of its offshoots exhibit the same limitation of only supporting 2D content generation [1]. Investigating further, A. Summerville et al. present a taxonomisation of Procedural Content Generation Machine Learning techniques, shown in Figure 2.7.

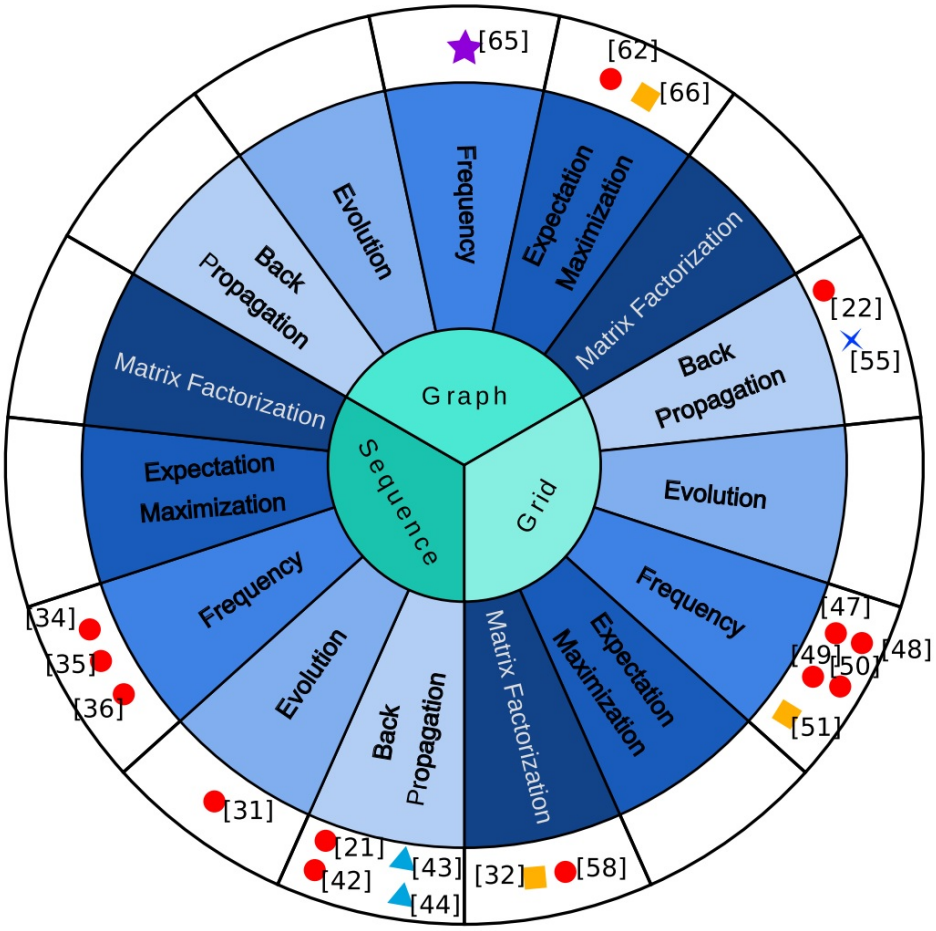


Figure 2.7: A taxonomisation of Procedural Content Generation Machine Learning techniques. There are two categorisations: the underlying data structure (graph, grid, or sequence) and the training method (matrix factorisation, EM, frequency counting, evolution, and backpropagation). Marks are colored for the specific type of content that was generated: red circles are platformer levels, orange squares are "dungeons," the dark blue x is real time strategy levels, light blue triangles are collectible game cards, and the purple star is interactive fiction. Citations for each are listed. Citation numbers correspond to those in the cited paper. [28]

2.2 Constraint Programming

2.2.1 Overview

Constraint programming deals with modelling problems through constraints and then running a solver to find solutions. In the context of PCG and video games, the use of constraints can be useful to tailor the output of PCG as desired and generate new content based on old content. Constraint programming has also been used to solve game tasks. The use of constraints allows for well-defined outputs to be created, but pay for this with more unpredictable generation times when compared to other methods [29]. Furthermore, constraint programming has been applied to a large variety of problem categories, ranging from combinatorial mathematics to logistics and scheduling [30].

2.2.2 Combinatorial Problems

Q. Cappart et al. take a deeper look into combinatorial problems. These problems deal with finding an optimal solution among a finite set of possibilities. The authors acknowledge that deep reinforcement learning has been used to tackle these problems, but that it only provides approximate solutions [31]. To find optimal solutions, they combine deep RL with constraint programming, detailing its use for problems such as the travelling salesman problem with time windows and the 0-1 knapsack problem.

P. Spracklen et al. comment that large scale combinatorial problems may have huge search spaces, resulting in low solver performance [32]. To address this, they introduce an automated process to add streamliner constraints, which focus effort on searching promising parts of the search space to improve performance.

Furthermore, a survey of combinatorial problems and attempts at modelling and solving them effectively identifies, for example, that algorithm selection techniques can achieve significant performance improvements for combinatorial search [33]. The survey presents a model for the algorithm selection problem, shown in Figure 2.8.

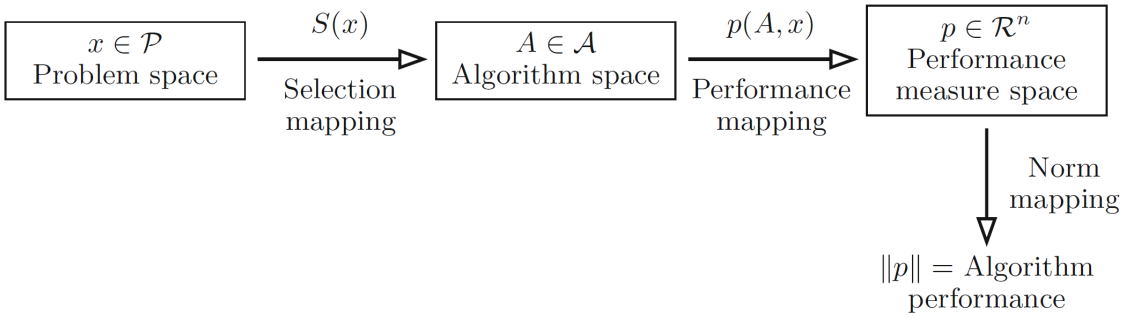


Figure 2.8: Basic model for the Algorithm Selection Problem [33]

Ö. Akgün et al. recognise the difficulty in testing constraint solvers due to the vastness of searches they may perform [34]. As a solution, they use metamorphic testing, which generates new test cases from existing ones. However, they also express the limitation that this should be used with other forms of testing. The

authors state that this is because metamorphic testing does not recognise when a solver falsely identifies a problem as unsolvable.

2.2.3 Logic Programming Languages

Another interesting application of constraint programming is to AI deep reinforced learning. G. De Gasperi et al. explore the use of the Prolog logic programming language to generate data sets to aid this learning, finding positive results in trained AI agent performance [35]. As part of this process, they convert user-specified constraints into Prolog queries through a Python program that generates JSON house plans (Figure 2.9). The pipeline for this is shown in Figure 2.10.

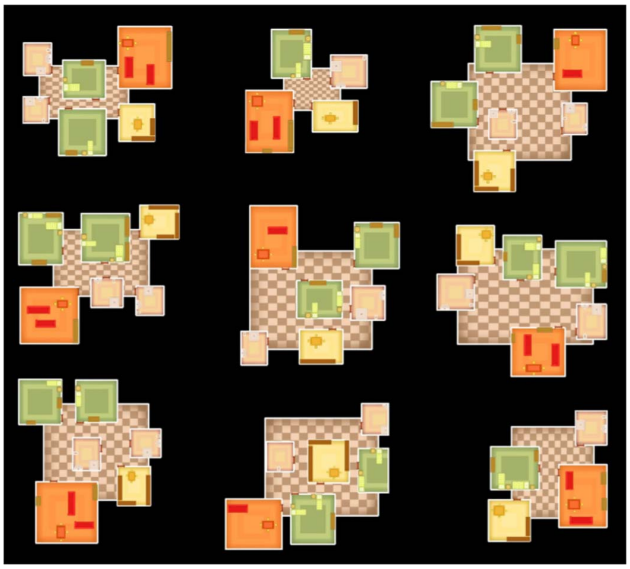


Figure 2.9: Examples of generated house plans [35]

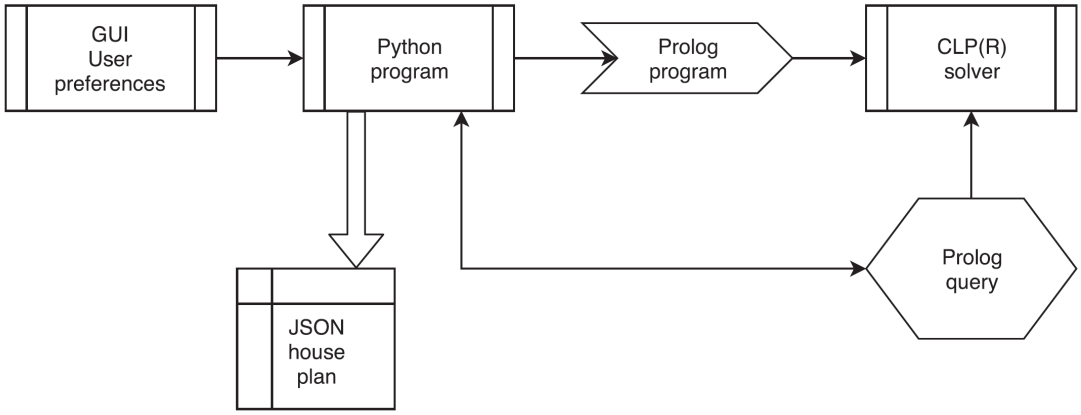


Figure 2.10: Python converting user-specified constraints into Prolog queries [35]

Other languages similar to Prolog, such as Answer Set Programming (ASP) and the Video Game Description Language (VGDL), extend Prolog's concepts with application to video games. For example, ASP can be used to generate constrained dungeons (Figure 2.11) [15]. A range of constraints are encoded. Altars (the golden

A tiles) are constrained to have four empty tiles around them. Wall tiles must have at least two neighbouring walls, encouraging the formation of larger wall segments. Gems (the green G tiles) must have three adjacent walls, making them stuck in wall segments. Finally, there must be a path with a minimum length between altars and gems, as well as a path diagonally across the level through this. All of these constraints work together to generate interesting, playable levels.

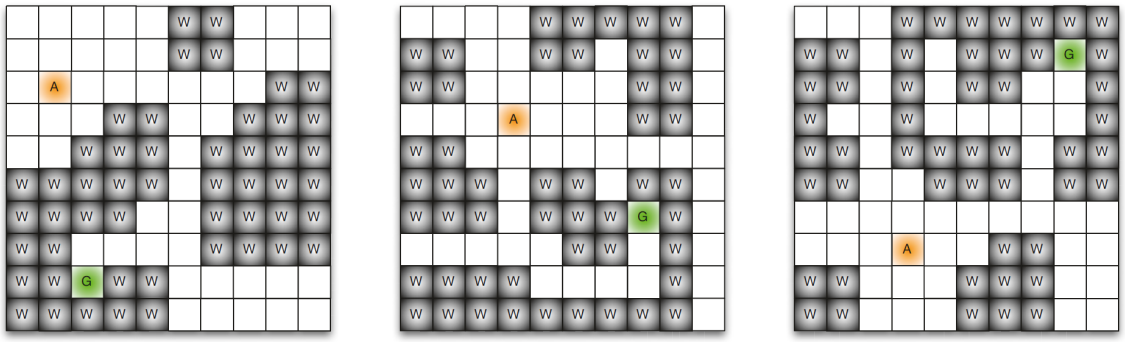
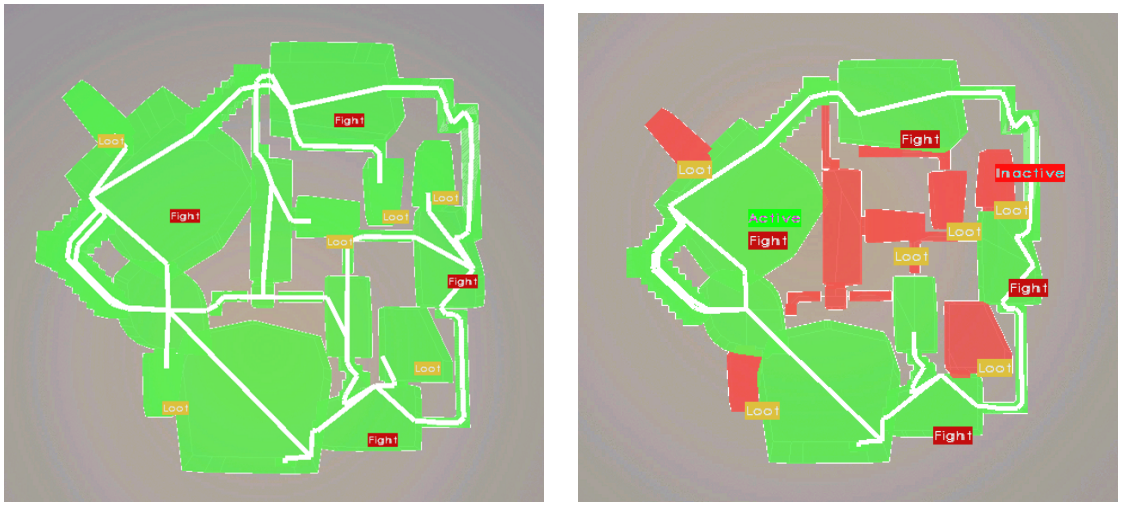


Figure 2.11: Examples of dungeons generated using ASP [15]

G. Glorian et al. also apply constraint programming to dungeon generation, but instead use a graph constraint to generate high quality dungeons with distinct areas [36]. Hundreds of variations can be generated from one dungeon with labelled rooms. The dungeon is represented as a graph, which lets areas be selectively turned off and rearranged while meeting design constraints. The concept is shown in Figure 2.12.



(a) The full dungeon (b) One variation

Figure 2.12: Generating a variation from a larger dungeon [36]

A. Khalifa et al. build on the General Video Game AI framework (GVG-AI) and the Video Game Description Language (VGDL) to create general video game generators [37]. These generators aim to save time by eliminating the need to custom-build a generator for each game. Instead, a game description is given as input and used to generate a level for the game. The results of three generators for

a *Zelda*-style game are shown in Figure 2.13. Here, the aim is to collect the key and get to the exit while avoiding attacks by monsters. The player has a sword that can be used to attack monsters and gain additional points.

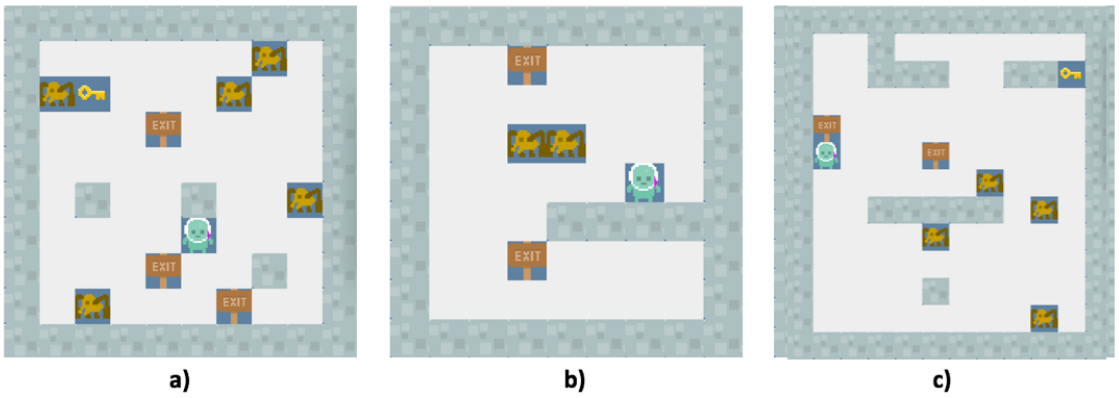


Figure 2.13: *Zelda*-style levels being generated by a Random Level Generator (a), Constructive Level Generator (b) and Search-Based Level Generator (c) [37]

2.2.4 Further Application to Video Games

In the context of video games, constraint programming research has looked into applying constraints to PCG and solving game tasks. J. Espasa et al. aimed to solve a planning problem presented in the game *Plotting* [38]. In *Plotting*, the player must plan a sequence of actions to clear blocks of varying types from a grid. The authors model the problem in two modelling language, namely the widely used Planning Domain Definition Language (PDDL) and the novel Essence Prime. Their effectiveness is then compared using several solvers, finding a SAT solver to be most effective for the problem as shown in Figure 2.14.

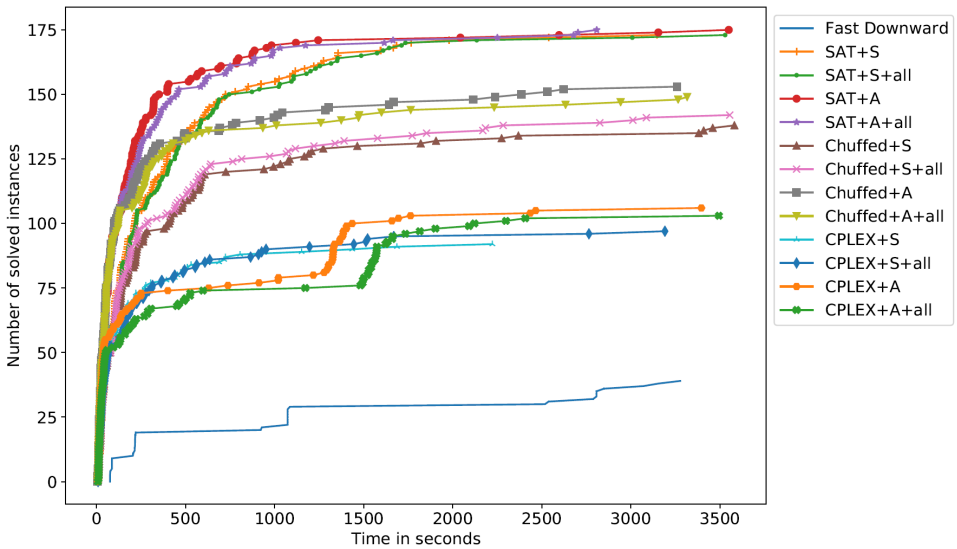


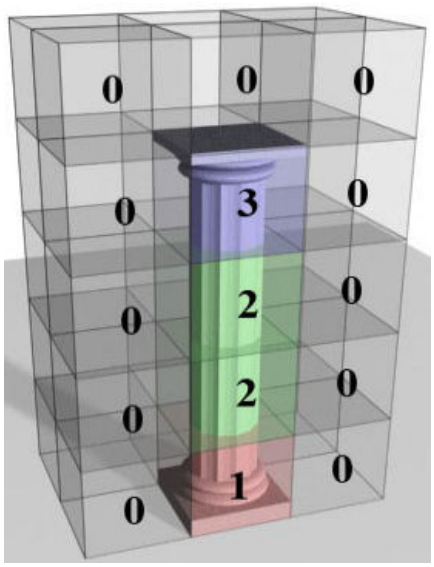
Figure 2.14: Comparing performance of *Plotting* models and solvers [38]

2.3 Wave Function Collapse

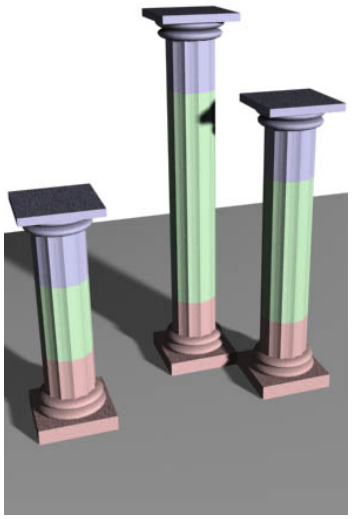
WFC [1] is a modern PCG method that has found use in games such as *Townscaper* [2] and *Bad North* [3]. It can be described as a family of algorithms, rather than one specific algorithm [8]. As such, a large variety of implementations are available online, each with their own specialisations to solve the problem for which they were designed. I. Karth and A. M. Smith note that WFC is often used as a black box, being incorporated into a workflow without being altered [29].

Figure 2.15: *Bad North* uses WFC to generate islands traversable by AI [3]

WFC builds off of the concepts of Model Synthesis, which is a method for procedurally modelling complex 3D shapes [9], [39]. In Model Synthesis, the user defines an input model detailing various dimensional, geometric and algebraic constraints. This is then used to create output satisfying the modelled constraints. A simple example using a pillar model is shown in Figure 2.16.



(a) An input model with model pieces arranged and labelled 0 to 3



(b) Pillar variations using the input model pieces

Figure 2.16: Generating pillars of different lengths from input model pieces [9]

2.3.1 Implementation Variations

Data Input

The data input stage is likely the WFC stage with the most variation across implementations. The original implementation [1] supports both an overlapping and simple tiled model. The overlapping version takes a sample image and defines overlapping patterns of $N \times N$ tiles, where the output is constructed of a random arrangement of these patterns. The overlapping WFC pipeline is shown in Figure 2.17. The simple tiled model instead defines single tile patterns, where each tile has its own defined set of possible neighbours. An example of simple tiled generation was shown in Figure 1.3.

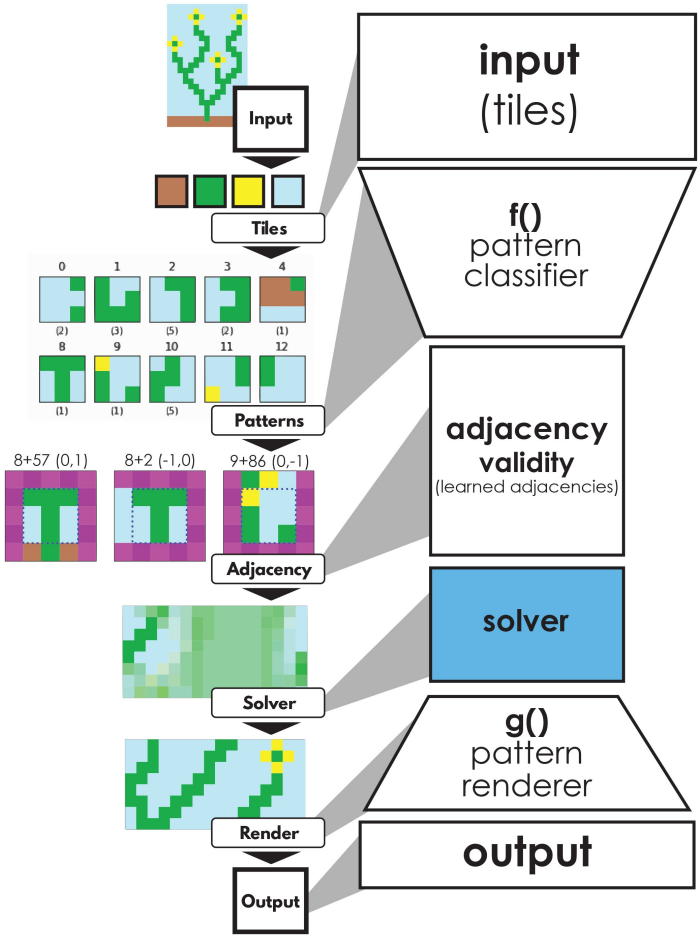


Figure 2.17: The overlapping WFC pipeline with 3×3 overlap [8]

2D vs 3D

While a lot of implementations focus solely on 2D input tiles and grids, fewer implementations support 3D input or output. This makes it a challenge to apply WFC to 3D environments. Furthermore, the increased complexity from a 3D environment can lead to an increased failure rate, which requires techniques such as backtracking or modifying in blocks to counteract. This was highlighted in Figure 1.2.

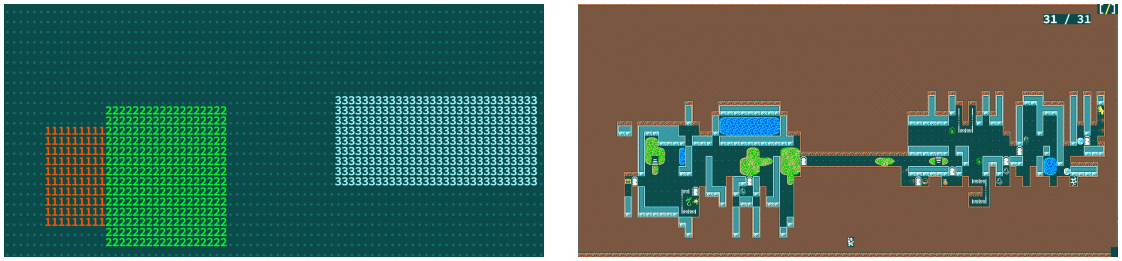
2.3.2 Limitations of WFC

Research papers on WFC frequently attempt to identify and find solutions for problems that implementations of the algorithm commonly face. Some of the most common problems are a lack of global constraints, overfitting and performance. These problems, as well as some other challenges, are discussed below.

Lack of Global Constraints

One problem with WFC is that, while output can be tailored to satisfy local constraints, global constraints are not inherently supported. This results in there being no inherent overall structure to the output. In other words, the output can be homogeneous.

In some applications, such as the game *Caves of Qud* [4], WFC is used only after other algorithms have defined distinct regions of the map (Figure 2.18) [8], [40], [41]. This multi-pass approach allows WFC to be used to its strengths.

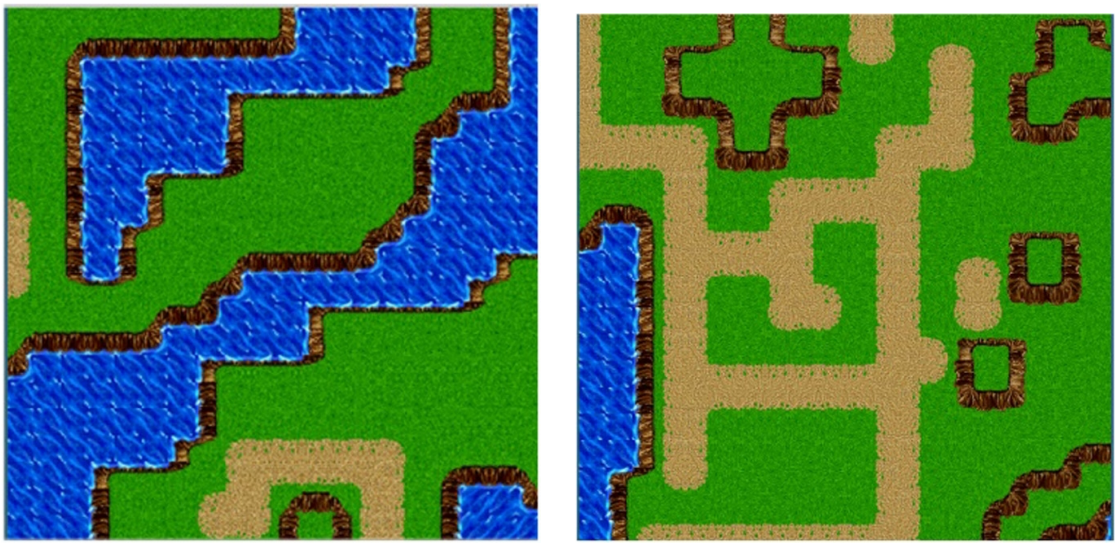


(a) Areas of the level set before using WFC (b) Interiors generated using WFC

Figure 2.18: *Caves of Qud*'s multi-pass approach to avoid homogeneity [41]

To tackle the lack of global constraints, D. Cheng et al. add constraints on minimum tile count, maximum tile count and object distance [42]. These are checked after each observation step.

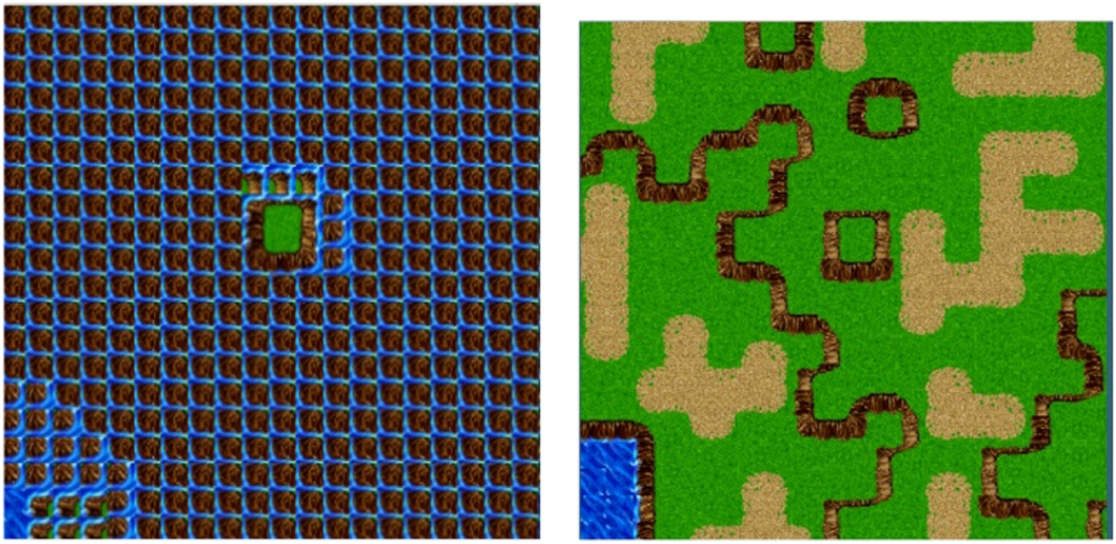
The global maximum tile count constraint allows balancing of tile counts. This constraint is analogous to the use of weighted tile selection. Figure 2.19 shows the global maximum tile count constraint being used to reduce the amount of water in the level to match the desired design.



(a) Generation without the maximum constraint (b) Generation with the maximum constraint

Figure 2.19: Use of the global maximum constraint to limit water tiles [42]

The global minimum tile count constraint gives more control on how tiles should be placed in the level. The way it is presented in the paper shows it being used to preset tiles before generation. This enables the definition of key areas that should be present in the output. Figure 2.20 shows the global minimum tile count constraint being used to place water in the bottom left and land in the middle of the level.



(a) Pre-setting water and land tiles (b) Generation with the minimum constraint

Figure 2.20: Use of the global minimum constraint to pre-place tiles [42]

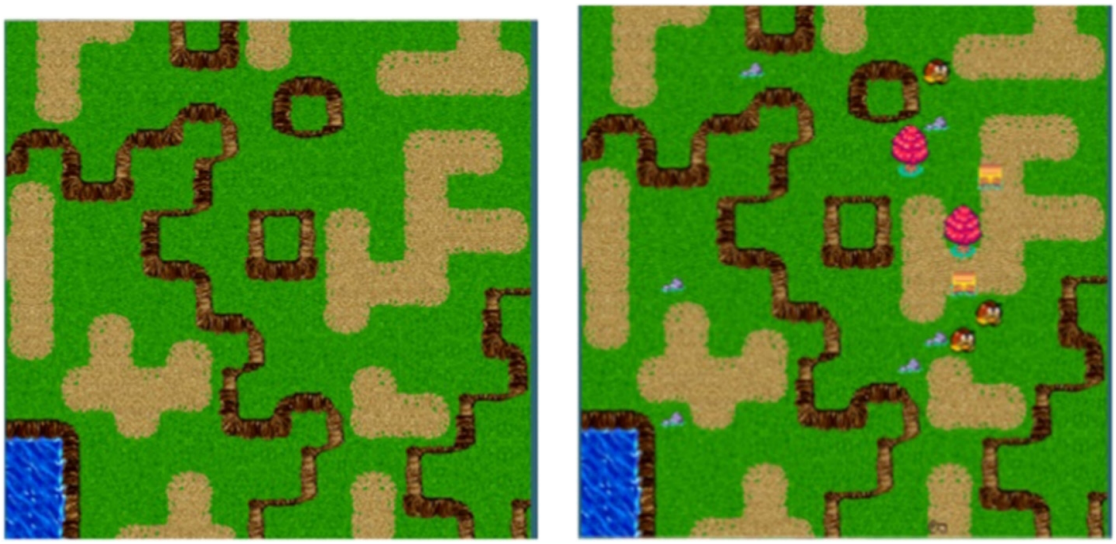
The object distance constraint is used to create areas of interest, rather than scattering items equally throughout the level. Figure 2.21(a) shows how, without an object distance constraint, enemies and items are scattered at random throughout the level. In contrast, Figure 2.21(b) shows how enemies are clustered around a treasure chest. To achieve this, enemies are constrained to be fewer than 10 units away from chests. Furthermore, keys are constrained to be more than 10 units away from chests. This means that keys will not simply spawn next to chests, making finding them more interesting as the player must search the map.

(a) Generation without the distance constraint (b) Generation with the distance constraint

Figure 2.21: Use of the object distance constraint to improve object spawns [42]

One additional option included is to carry out generation in two passes. This helps create levels of certain styles and to reduce conflicts arising from trying to generate everything in one pass, such as placing an object on an unsuitable tile. In

Figure 2.22, double-layer generation is used to spawn enemies on grass and chests on dirt roads. Furthermore, rock and dirt decor is placed on suitable tiles.



(a) Single-layer generation (b) Double-layer generation

Figure 2.22: Use of the double-layer generation to ease object spawning [42]

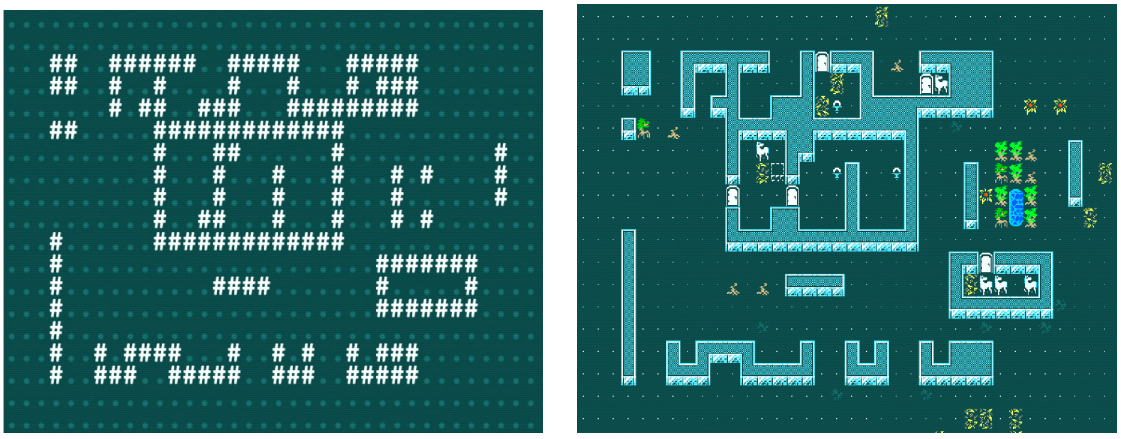
A. Sandhu et al. achieve the same goal as minimum and maximum tile count constraints through the use of a weighted choice [40]. In addition to using entropy to choose the most constrained tile after each propagation step, assigning a weight to each choice can encourage the algorithm to choose a different balance of tiles. Furthermore, the authors introduce a second observation step. This performs a second, smaller scale WFC algorithm, which can be used to refine item placement and create subregions within a map.

Solutions altering WFC directly to support global constraints are faced with the issue that the additional constraints can have a negative impact on performance. However, by combining such solutions with those discussed in the Performance Subsection (2.3.2), the impact can be reduced [8].

Overfitting

When adding a lot of detail to the input, such as through using complex tile sets, the output may become too constrained. This can result in overfitting and an increased failure rate.

One solution is to use a multi-pass approach. This not only reduces the risk of overfitting, but also helps to globally constrain the output. This is done by reducing the detail of the input and instead adding additional details to the output in a second pass once WFC has run. *Caves of Qud* applies this approach by generating architecture using WFC and then generating details using additional passes (Figure 2.23) [41].



(a) Architecture generated using WFC (b) Details filled using additional passes

Figure 2.23: *Caves of Qud*'s multi-pass approach to avoid overfitting [41]

Performance

Another issue identified with WFC is performance. While the chance of success is high with small inputs, larger inputs are much likelier to fail, especially with more complicated tile sets [8]. This can result in a significant generation time for large output grids as generation must be restarted several times. Several solutions to WFC's scalability problem have been proposed.

One of the most common solutions is to include some form of backtracking, which allows further searching of the search space upon a contradiction instead of having to restart. However, with complex tile sets, care must be taken to reduce the chance of backtracking exploring an unpromising search space for an extended time as in the 3D Escheresque example shown previously in Figure 1.2. Using a search heuristic could help with this. I. Karth and A. M. Smith compared the performance of WFC with and without backtracking and global constraints, finding that backtracking is critical to improving performance when using global constraints (Figure 2.24) [8].

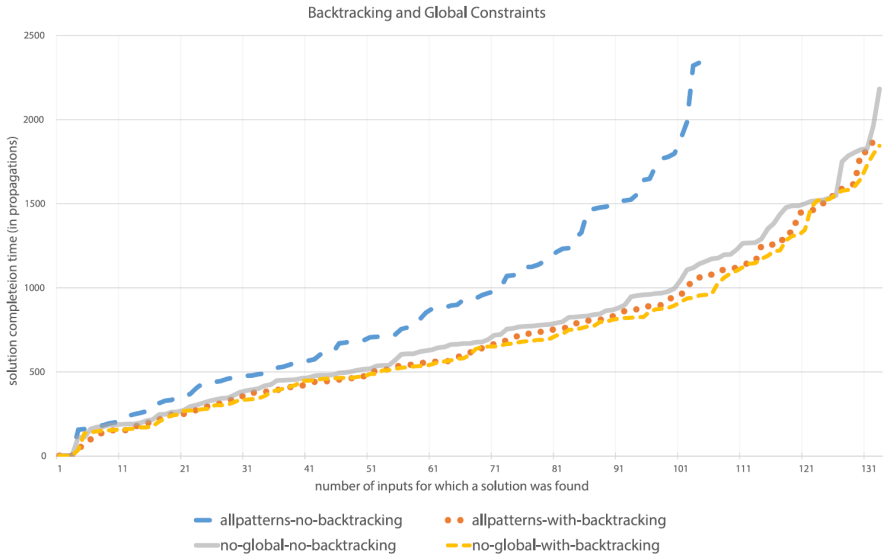


Figure 2.24: Comparing performance of WFC with and without backtracking and global constraints. When using global constraints, backtracking significantly improves performance. [8]

Nested WFC (N-WFC) [43] is one technique that aims to improve scalability of WFC. It splits a larger grid into smaller grids, evaluating sub-grids diagonally from the top left (Figure 2.25). Each sub-grid overlaps constraints from its left and upper neighbour to satisfy constraints between adjacent sub-grids. This can be extended to an infinite space by overlapping new areas with old areas (Figure 2.26). While this technique does improve the performance of WFC, a large number of conflicts from edge data still occur.

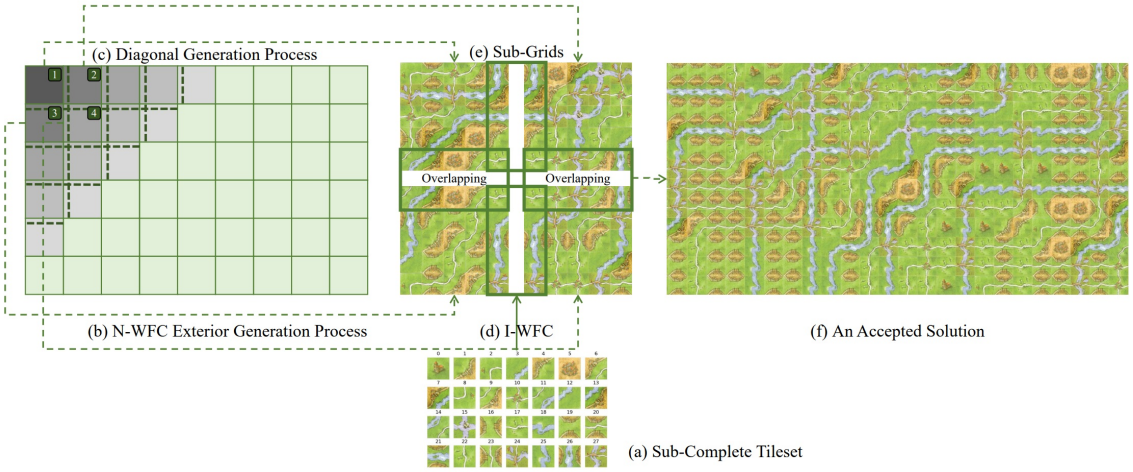


Figure 2.25: Large-scale game implementation with N-WFC and sub-complete tile set. First, it requires (a) one sub-complete tile set. Then the (b) Exterior Generation Process uses (c) Diagonal Generation Process to start generating. Each (d) sub-grid uses (e) I-WFC to find an accepted solution and overlap its edge with the adjacent sub-grids, forming a final solution. [43]

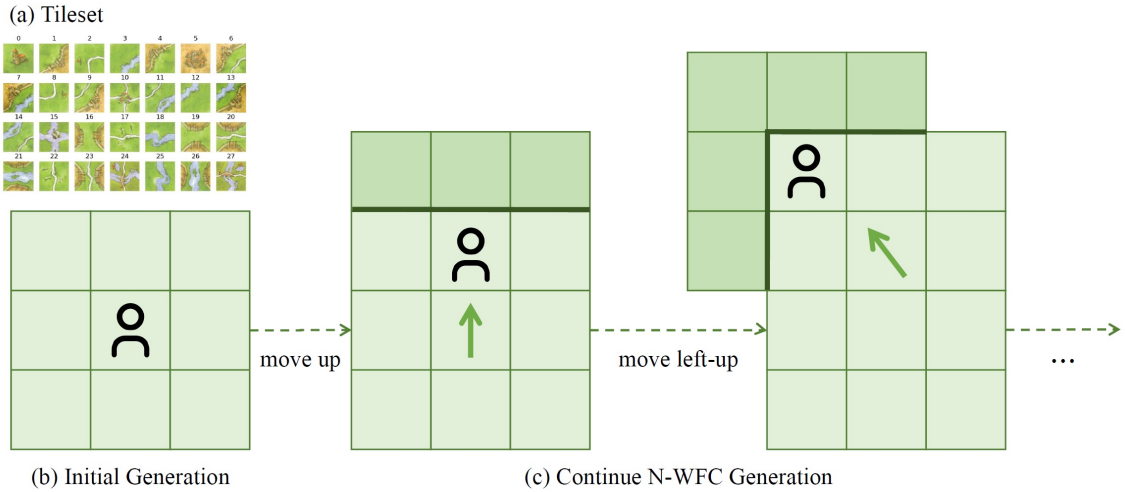
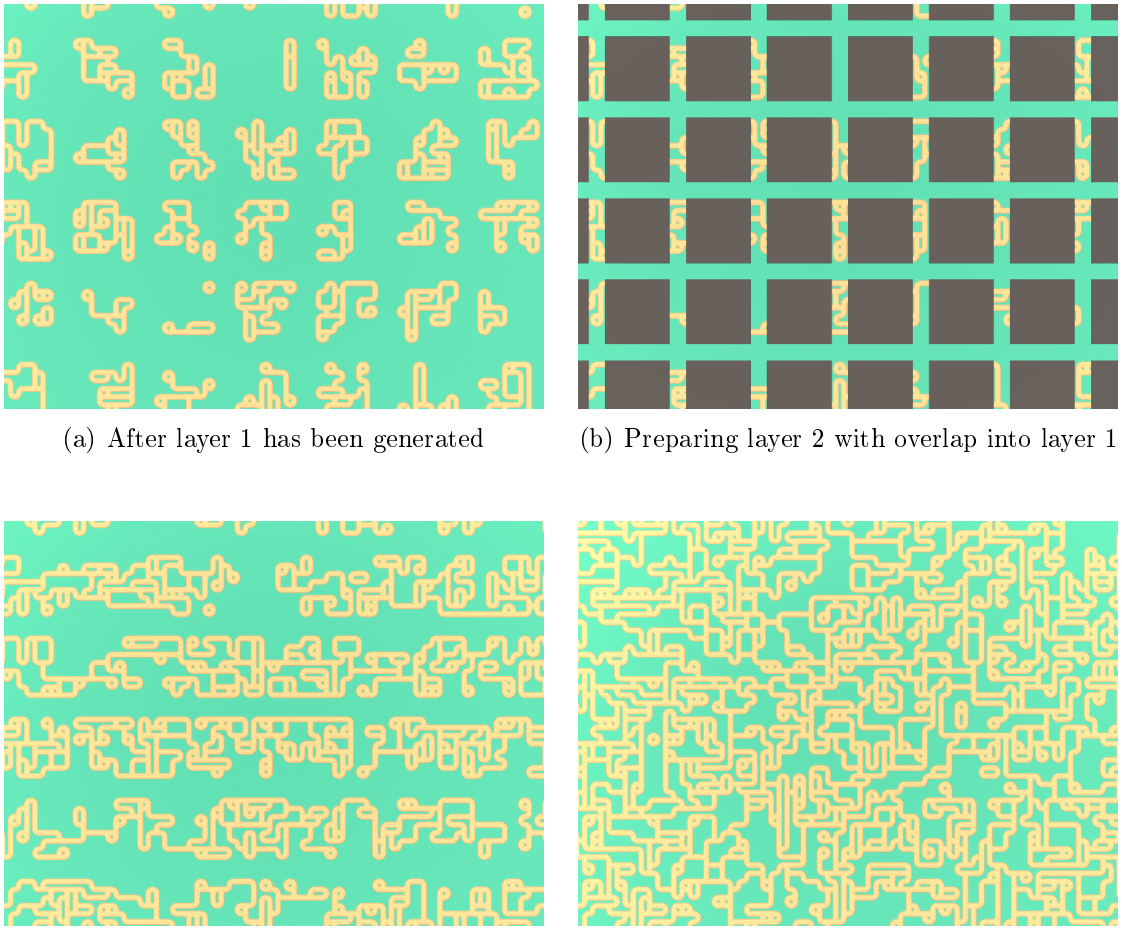


Figure 2.26: Infinite game implementation with N-WFC and sub-complete tile set [43]

Another technique, Infinite Modifying In Blocks [10], applies WFC multiple times in small chunks. Keeping the chunks small keeps the performance of generation high, while running WFC in four layers per chunk ensures that constraints are satisfied between adjacent chunks. The layering also addresses the limitation of conflicts that Nested WFC struggles with. As each chunk is made up of four WFC layers, failed layers can usually be ignored rather than having to be regenerated. However, this comes with computational overhead from running WFC four times per chunk. An overview of the method is shown in Figure 2.27, with detailed discussion in Section 6.1.7.



(a) After layer 1 has been generated

(b) Preparing layer 2 with overlap into layer 1

(c) After layers 1 and 2 have been generated

(d) After all layers have been generated

Figure 2.27: A glimpse into the IMIB pipeline. Each layer defines a small part of each chunk to run WFC in. By clearing and running four overlapping layers, a full grid is generated. [10]

Other Challenges

Environments of a certain style, especially those trying to create a realistic feel, may struggle from WFC's use of a grid structure for its output. However, if this regular grid is transformed into an irregular quadrilateral grid, more complex shapes can be used. H. Kim et al. achieve this by using a graph-based data structure, which can be integrated with a navigation mesh in 3D as shown in Figure 2.28 [44]. However, this solution is limited due to a lack of control over solution order.

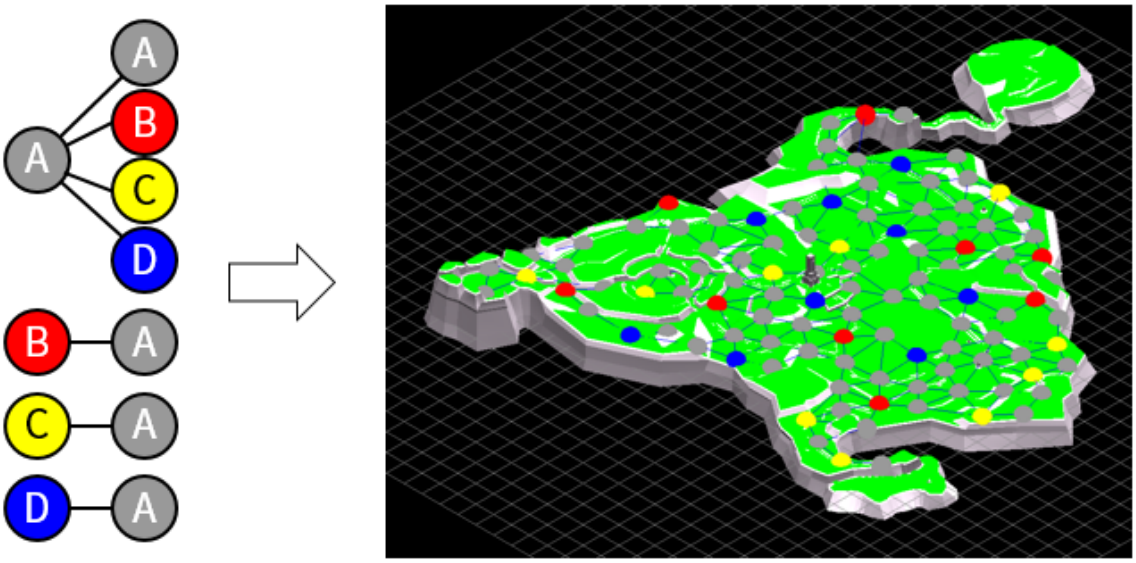
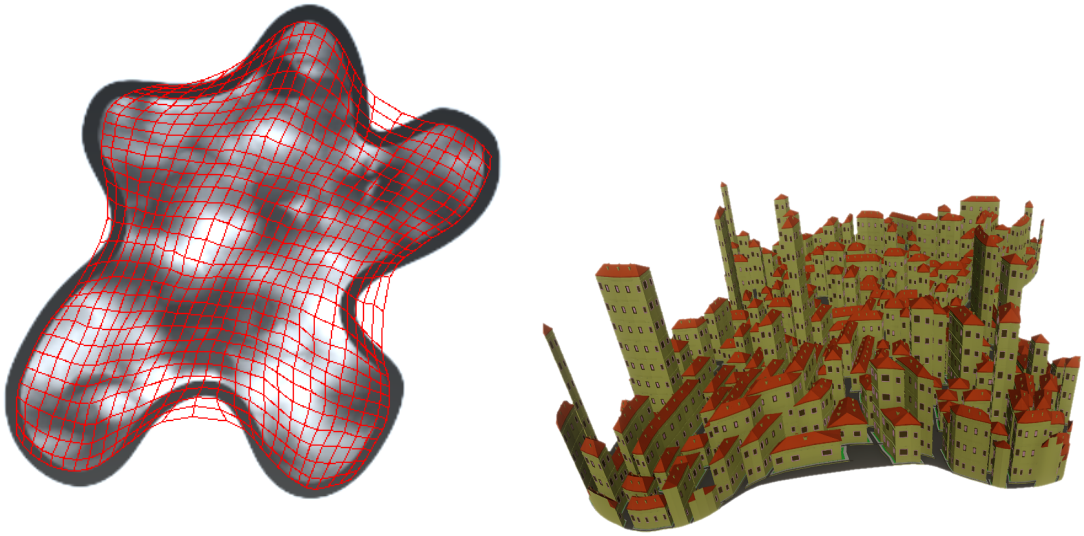


Figure 2.28: Placing nodes on a navigation mesh using graph-based WFC [44]

Taking the idea of irregular quadrilateral grids further, T. Møller and J. Billeskov explore the use of a growing grid neural network to augment WFC [45]. Growing grid can be used to create irregular quadrilateral grids that fit a given input shape (Figure 2.29(a)). Here, gradients can be used to control grid density. WFC output can then be fit onto an irregular quadrilateral grid to create more interesting worlds (Figure 2.29(b)). The authors also found that players have higher self-confidence in navigating irregularly-shaped maps and an increased ability to form mental maps of their environment.



(a) Growing grid creating an irregular quadrilateral grid fitting an input shape

(b) Fitting WFC onto an irregular quadrilateral grid to create more complex worlds

Figure 2.29: Using growing grid and WFC to generate more complex worlds [45]

Very simple implementations may ignore symmetry when defining tile neighbours in the simple tiled method [46]. Instead, every neighbour for each direction of each tile is listed explicitly. While this keeps the code simple, it means that a huge

amount of work is required when defining neighbours for complex tile sets, with high chance of human error. The original WFC implementation and several others define a symmetry type for each tile to tackle enumeration of large tile sets. This means that much less data about the input has to be provided, lessening the work required and chance of human error. The original WFC implementation defines five symmetry types, which it applies to a variety of 2D images. D. Cheng et al. instead define nine symmetry types as in Table 2.1, which supports a greater variety of tile sets [42].

Name	Symmetry type	Initial tile's equivalent transformation number	New symmetry type after transformation
F	No symmetry	0	F/F/F/F/F/F/F/F
S	Centrosymmetric	0/2	S/S/S/S/S/S/S/S
T	Vertical axis symmetry	0/4	T/B/T/B/T/B/T/B
L	Counter-diagonal axis symmetry	0/5	L/Q/L/Q/Q/L/Q/L
B	Horizontal axis symmetry	0/6	B/T/B/T/B/T/B/T
Q	Main diagonal axis symmetry	0/7	Q/L/Q/L/L/Q/L/Q
I	Horizontal and vertical axis symmetry	0/2/4/6	I/I/I/I/I/I/I/I
/	Double diagonal axis symmetry	0/2/5/7	'/'/'/'/'/'/'/'/'/'/'/'/'/'/'/'/'
X	All 8 transforms are identical	0/1/2/3/4/5/6/7	X/X/X/X/X/X/X/X

Table 2.1: A symmetry dictionary, proposed by [42]

Chapter 3

Requirements Specification

The following objectives were outlined in the Description, Objectives, Ethics and Resources (DOER) document at the start of the project. Primary objectives were chosen as core requirements of the project, with secondary objectives serving as additional goals if time allowed.

Primary Objectives

- Create a game that uses Procedural Content Generation.
- Use novel PCG methods such as Wave Function Collapse.
- Extend on at least one PCG method.
- Use assets to give the game a full set of graphics and audio.

Secondary Objectives

- Make levels navigable by AI opponents.
- Allow customisation of level generation and other gameplay elements via an in-game menu.

Chapter 4

Software Engineering Process

4.1 Methodology

4.1.1 General Overview

The project was carried out with use of Agile methodologies. Weekly supervisor meetings were held, in which the past week's work would be evaluated. This was then contextualised within the overall time frame of the project. This critical analysis helped to identify and set goals for the next week and beyond. Agile development suited the nature of the project as the full progression of the project was not clear from the start. For example, initially it was planned to apply a Wave Function Collapse implementation directly and focus more on extending it to aid game design. However, while there were many implementations of WFC available online, many of them had poor documentation and did not work out of the box due to missing assets and errors, while others could not be applied to a 3D tile set. The official Wave Function Collapse GitHub contains links to other WFC implementations [1]. Four implementations investigated were the original implementation, two forks by Joseph Parker [47] and Maksim Priakhin [48], as well as a simplified implementation by Garnet Kane [46].

4.1.2 Semester One

The key areas of focus in the first semester were carrying out a literature review, setting objectives, reviewing ethics, designing the game and implementing the WFC algorithm. As described, the goal of the project was initially to extend upon an existing implementation of WFC. As this was unsuccessful, the focus changed to implementing an algorithm from scratch. After the core of the constraint solver was finished, WFC's lowest entropy cell selection and random weighted tile selection were added. The meeting notes for semester one are available in Appendix Section C.1.

4.1.3 Semester Two

The key areas of focus in the second semester were finishing the game and documenting the project in this report. The core generation had already been fully implemented, but assets still had to be created and put into the generator. Furthermore, extensions on the constraint solver, such as Infinite Modifying In Blocks

and dynamic chunk loading, had to be added to make levels infinite and playable. Additional work during the holidays involved studying Infinite Modifying In Blocks, adding graphics filters and starting block modelling. A list of tasks was created during this time and extended during semester two. This list and the meeting notes for semester two are available in Appendix Sections E and C.2 respectively.

4.2 Tools and Technologies

4.2.1 Unity (C#)

Unity was used as the game engine for development. This was chosen as Unity and its scripting API language C# have commonly been used for implementations of Wave Function Collapse, including the original WFC repository by Maxim Gumin [1]. WFC has also been adapted to other engines such as Unreal Engine [49]. Two issues facing later development using Unity were its poor support for multithreading and importing `.fbx` models from Blender.

4.2.2 Blender

Blender was used to create tile models as it is a free but powerful modelling software. Each model could be created in Blender and exported as an `.fbx` file. These files were imported into Unity and unpacked. Each model could then be used as a GameObject and have any additional assets such as lights and audio sources attached. Due to Unity's poor support for loading files from Blender, materials had to be reassigned in Unity.

4.2.3 GitHub

GitHub was used for version control. This was useful for comparing new and old code and tracking progress over time.

4.2.4 Document Management

Google Drive and Google Docs, both part of Google Workspace, were used to hold most documents relating to the project. This included a tasks document (Appendix Chapter E), game design document (Appendix Chapter D), weekly meeting notes (Appendix Chapter C), literature review research document and credits document for any external resources used. Furthermore, Overleaf was used to write up the majority of this dissertation. These cloud platforms enabled work from multiple locations and more effective evaluation with the project supervisor as the latest versions of documents were always available. One issue with Overleaf is that it has a compile time limit. The full version of the disseration would time out on Overleaf, so instead had to be edited and compiled locally. GitHub was again used to provide version control.

Chapter 5

Ethics

There are no ethical considerations. All questions on the self-assessment form could be answered with "No". This included the following declarations:

- The project did not use any secondary datasets.
- The project did not involve research with human subjects.
- No potential physical or psychological harm, discomfort or stress to researchers or participants was foreseeable.
- No conflicts of interest arose in the project.
- The project was not externally funded.
- The project did not involve the use of living animals.

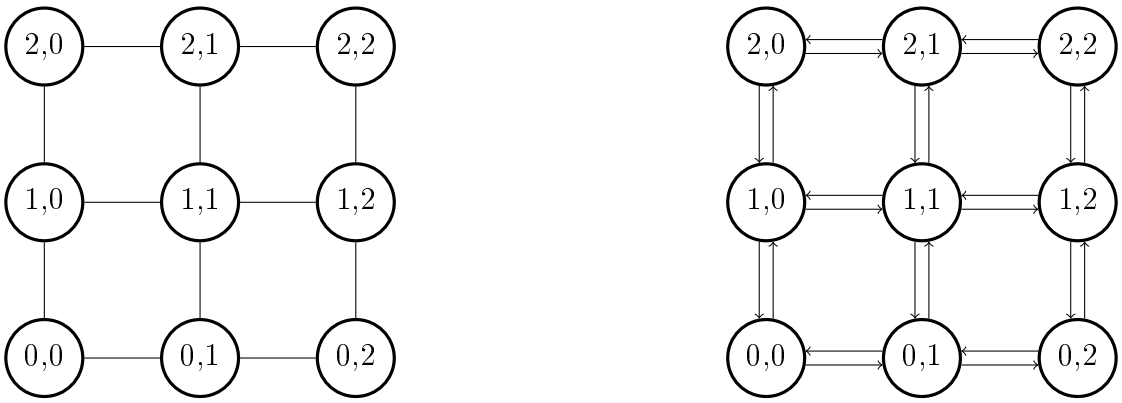
Chapter 6

Design

6.1 Level Generation

6.1.1 Contextualising the Wave Function Collapse Algorithm

The Wave Function Collapse (WFC) algorithm can be viewed as an extension on the ideas presented by the Maintaining Arc Consistency 3 (MAC3) algorithm. What is referred to as a grid of cells with tile choices in Wave Function Collapse is analogous to a graph of variables / nodes with a domain of possible values. Constraints between variables can then be viewed as edges in an undirected graph (Figure 6.1(a)).



(a) Undirected graph

(b) Directed graph

Figure 6.1: A 3×3 cell grid as an undirected and directed graph

6.1.2 Arcs

An edge in an undirected graph can alternatively be viewed as two opposite edges in a directed graph (Figure 6.1(b)). Arcs apply this concept to constraints, with each arc representing one of the two edges making up a constraint in the directed graph. For an arc to be locally arc consistent, all the values in the first variable's domain must be supported by at least one value in the second variable's domain.

For example, suppose the use of an extremely simple tile set with only cubes and empty tiles, where any tile choice is possible at the start. Furthermore, say that cubes can only have other cubes as neighbours and that empty tiles only have empty tiles as neighbours. This corresponds to Figure 6.2.

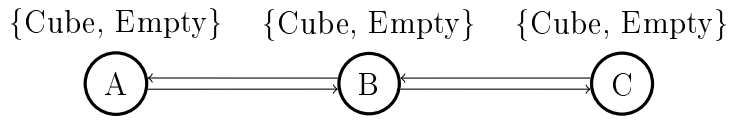


Figure 6.2: Example starting state, in which all arcs are consistent

Then, assume that node A is set to be a cube, removing the empty tile from its domain. Now, the arc <A,B> is consistent as B still has a cube as a support value. However, the arc <B,A> is no longer consistent since the empty tile does not have any support in A's domain, which only has the cube in it. This corresponds to Figure 6.3.

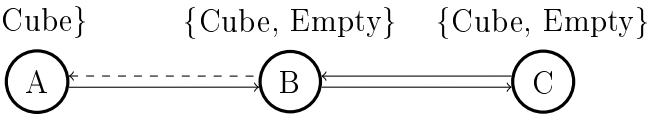


Figure 6.3: After assigning Cube to A, <B,A> is no longer consistent

6.1.3 AC3

The Arc Consistency 3 (AC3) algorithm enforces arc consistency in the grid of cells. The algorithm is given a queue of arcs to enforce arc consistency across. Each arc is checked for support and unsupported domain values pruned. If any domain changes occur, then all arcs targeting the primary variable of the current arc are re-added to the queue. This is required as the domain change may have resulted in support for arcs to this variable being lost.

Back to the example, assigning Cube to A will trigger arc revision with all the arcs incident on A. In this case, this initialises the queue with arc <B,A>. Checking the arc shows that B's Empty value is no longer supported by A (Figure 6.4).

1. Check arc <B,A>:
   (a) B's value Cube has support in A through its value Cube.
   (b) B's value Empty does NOT have support in A! Remove it from the domain.

Figure 6.4: Revising arc <B,A>

Performing this single revision leaves the graph in the state as shown in Figure 6.5. Revising <B,A> results in <C,B> becoming inconsistent, highlighting the importance of adding targeted arcs to the queue after a domain change. In this case, <C,B> must be added to the queue. <A,B> does not have to be added since the domain change happened while revising <B,A>. This exploits the fact that arc support is bi-directional, meaning that if a value is supported on <B,A>, then it must be supporting some value on arc <A,B>.

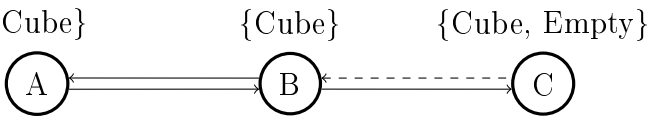


Figure 6.5: Revising <B,A> results in <C,B> becoming inconsistent

Revision of <C,B> proceeds similarly to revision of <B,A> (Figure 6.4). Similarly, C's domain change from revision of <C,B> does not require re-checking of arc <B,C>. After this, the queue of arcs to revise is empty and as such the consistency before assignment has been maintained. The state after revision is shown in Figure 6.6.

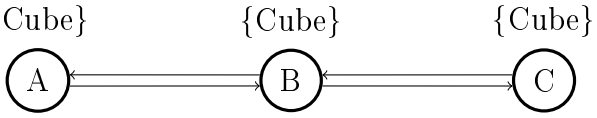


Figure 6.6: The graph after AC3 has been carried out

An additional property to note is that each node now only has one value in its domain. Given that each arc is consistent, this means that this is a valid solution to the original problem where each node could either be a cube or empty tile. In the context of WFC, this means that all cells have been collapsed with a valid tile choice and thus that a valid map has been generated.

6.1.4 MAC3

Maintaining Arc Consistency 3 (MAC3) uses Arc Consistency 3 (AC3) to create a full constraint solving algorithm. To begin, AC3 is run with all arcs to ensure global arc consistency at the start. Then, a variable and value are chosen in an attempt to find a solution. This choice is like that done in the example shown previously in Figure 6.3. Arc consistency is then maintained by running AC3 with the all the arcs targeting the variable that had a value assigned (Figure 6.4). This ensures that these arcs maintain local arc consistency. As global arc consistency was enforced at the start, maintaining local arc consistency after each assignment is enough to also maintain global arc consistency. If any variable assignments lead to a domain wipeout (an empty domain) when enforcing arc consistency, then backtracking can be performed and another value chosen. Once each variable only has one value left and all arcs are consistent, a solution has been found. If not stopped by a timeout, MAC3 will continue running until a solution has been found or all choices have been attempted, meaning that there is no solution.

6.1.5 Playing Sudoku as an Analogue to Constraint Solving

Sudoku can serve as an example to make the constraint solving process clearer. Someone solving Sudoku might pencil in the possible number for each cell to help make guesses and fill the grid. This is analogous to giving each cell a set of possible values.

At the start, given numbers help the player pencil each remaining cell. Like this, the possible numbers of each cell are reduced. If a cell only has one possible

number remaining, then the player knows that the cell must hold that value. This is analogous to running AC3 at the start to ensure each cell has a consistent set of possible values.

Eventually, the player is likely to face the situation where no more numbers can be inferred from the given clues. Just like the MAC3 solver, the player must make a guess as to what the correct number might be. After making a guess, the implications of it can be carried forward. This is analogous to running AC3 after making a variable assignment.

If this guess was incorrect and leads to a cell with no choices remaining, the player must undo the guess. This is analogous to having a domain wipeout and backtracking. After backtracking, the player knows that the choice they made was incorrect, so can pencil out the number they tried. From this, they may gain more clues about other cells in the grid. This is analogous to unassigning a value and running AC3 again.

If the player keeps making guesses and inferring additional information from them, they will eventually solve the Sudoku. If the player were to ignore the implications of a guess, then their solution may not be valid. This highlights the need to maintain consistency between each pair of constrained cells in the grid.

6.1.6 Variable and Value Choice

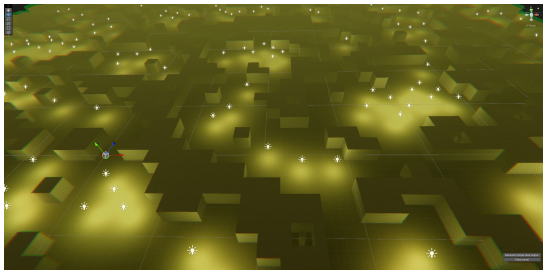
The manner in which a variable and value are chosen can be implemented as desired. The most effective method often depends on the specific problem. A common method used in Wave Function Collapse is lowest entropy cell selection together with weighted random tile selection.

Lowest Entropy Cell Selection

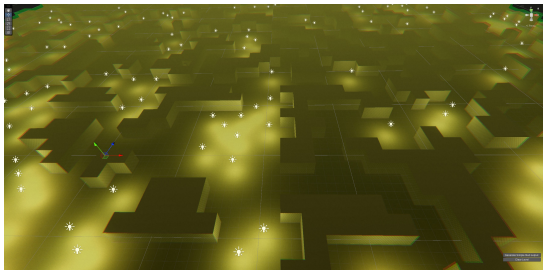
The lowest entropy method can be viewed as choosing the most constrained cell at each step as it takes into account the weight of each cell. This increases the chance to find a solution quickly as any contradictory assignments are reached faster and backtracked from with fewer steps. The weight of each tile is defined in the level editor and represents the target rate of occurrence of a tile in the generated level. Equation 6.1 shows the Shannon Entropy of a cell, which is lowest for cells with a small number of remaining tile choices of imbalanced weighting.

$$\text{Shannon Entropy} = S = \log(\sum \text{weight}) - \frac{\sum(\text{weight} \times \log(\text{weight}))}{\sum \text{weight}} \tag{6.1}$$

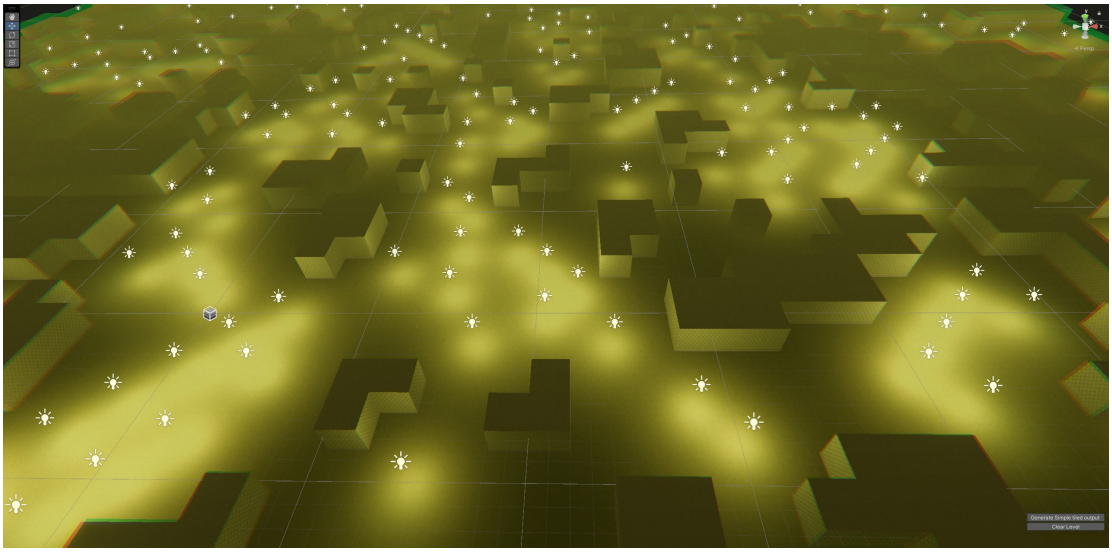
Figure 6.7 shows outputs of three different cell selection techniques. The lowest entropy cell selection generates a distinctly different level to the other two techniques. The two empty tile types have relative high weights. Furthermore, empty light tiles may only have empty tiles as neighbours. This means tiles near empty tiles will have a relatively low entropy, making them chosen before other tiles. In effect, this creates bigger clusters of empty space compared to the other selection techniques.



(a) Ascending Cell Selection



(b) Smallest Domain Cell Selection



(c) Lowest Entropy Cell Selection

Figure 6.7: Examples of levels with three different cell selection techniques

Weighted Random Tile Selection

Assigning each tile a weight gives the level designer global control over the percentage of each tile in the output level. Once a cell has been chosen, weighted tile choice is carried out by summing all tile weights and then choosing a random number within that range. This gives higher weight tiles a higher chance to be selected. If the tile choice did not include any randomness, then unplayable levels such as a completely empty level could be generated (Figure 6.8). In this case, the first tile in the tile set is the empty tile, generating an empty level.

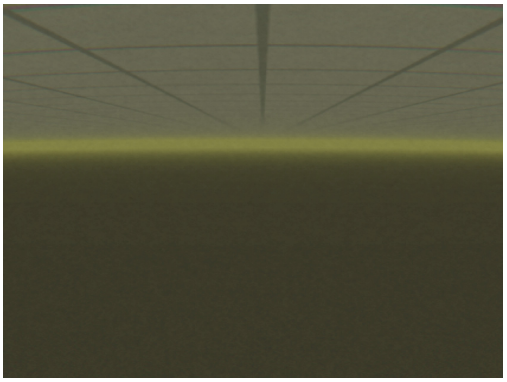
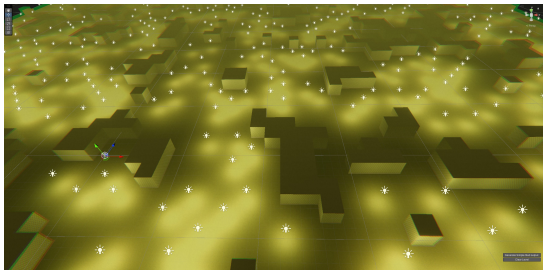
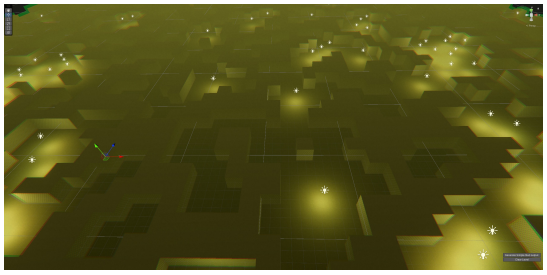


Figure 6.8: Ascending tile selection has a high chance to generate unplayable levels such as this empty level

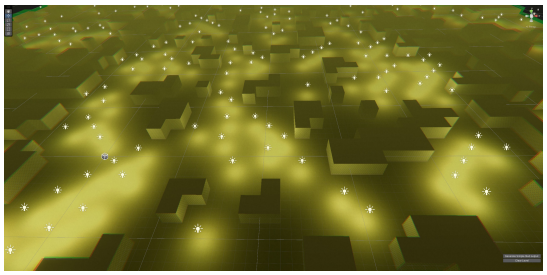
The chance to generate unplayable levels highlights the challenge in applying a simple constraint solver to level generation in a game. A level that satisfies all constraints for the solver does not necessarily translate to a good level. Some additional constraints could be added in, such as requiring a specific block in generated chunks. However, other constraints such as requiring a certain percentage of a specific block type in the entire level are harder to enforce effectively and efficiently. Such constraints can be encourage implicitly in generation, through features such as the weighted random tile selection. With this, output is not guaranteed to satisfy block percentages, but instead strongly biased towards creating outputs that roughly match these constraints. Figure 6.9 shows how different tile weight configurations can create vastly different levels. Careful tweaking of weights is required to create an appealing level. Having too dense of a level can create inaccessible areas as in Figure 6.9(b). An alternative tile selection technique is to simply pick tiles at random (Figure 6.9(d)). This is analogous to giving tiles equal weights. In this case, adjacencies contribute much more to how commonly certain tiles appear.



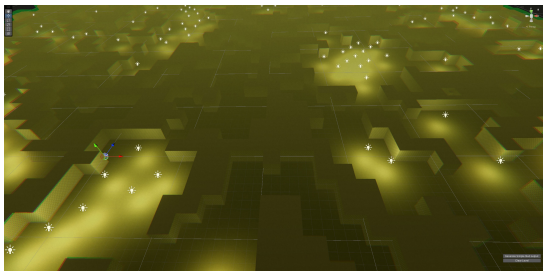
(a) Increased weights for empty tiles creates a sparser level



(b) Increasing weights for solid tiles creates a denser level, with increasing chance of unreachable areas



(c) A balance of weights for all tiles creates a level with some sparse and some dense areas

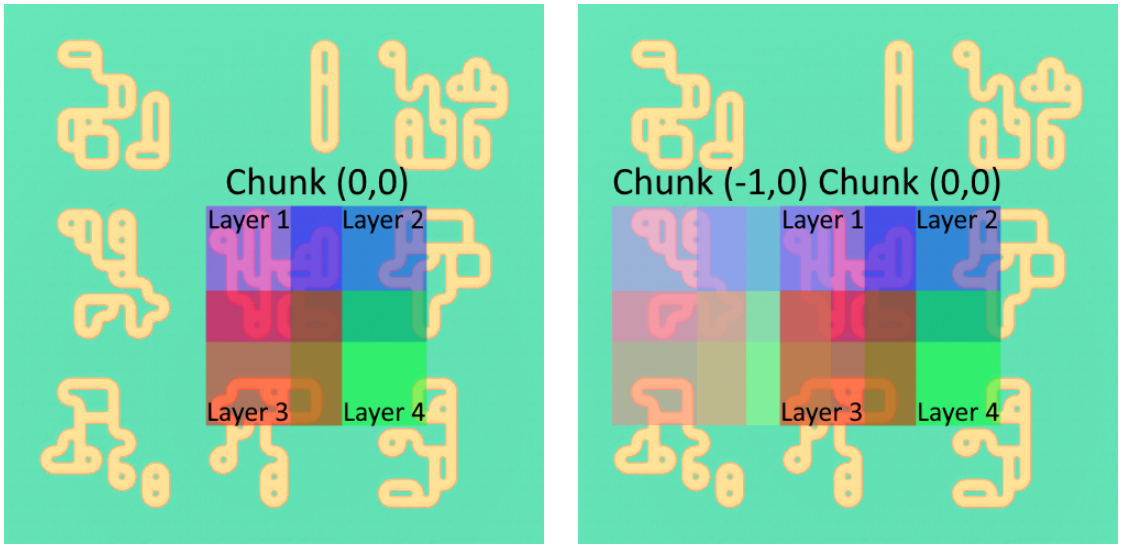


(d) Having equal weights (randomly selecting tiles) gives less control over tile percentages

Figure 6.9: Examples of levels with four different tile weight configurations

6.1.7 Infinite Modifying in Blocks

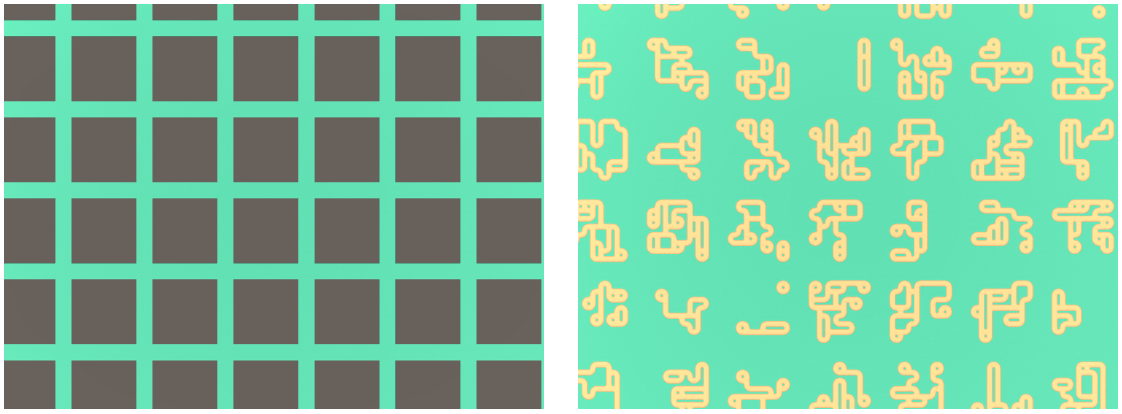
Infinite Modifying In Blocks works by splitting up generation into overlapping chunks, each of which are split into four layers. These layers are offset such that they do not interfere with each other. This can be used to ensure that generation is deterministic and optionally process each layer in parallel. As example, Figure 6.10(a) shows how a single chunk is formed of four overlapping layers. Extending on this, Figure 6.10(b) shows how chunks overlap with each other, highlighting the layers of another chunk in lighter colours. The images from the original source were edited to show more clearly the concepts of chunks and layers as implemented.



(a) The four layers making up a single chunk (b) Overlap between adjacent chunks

Figure 6.10: IMIB uses a grid of overlapping chunks consisting of overlapping layers

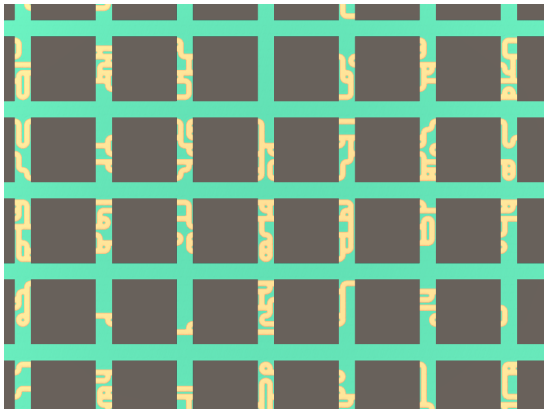
The algorithm is run layer by layer, finishing the layers of all active chunks before moving to the next layer. First, a layer is prepared. This involves calculating the area inside of the chunk that forms the layer and clearing all cells inside of it (Figure 6.11(a)). Second, the solver tries to find a solution for the layer (Figure 6.11(b)). This takes into account adjacency information from the cells next to the layer. If a solution is not found, the layer is left as it was before. This is not an issue for most tile sets as single layer failures are hidden by other layers successfully generating.



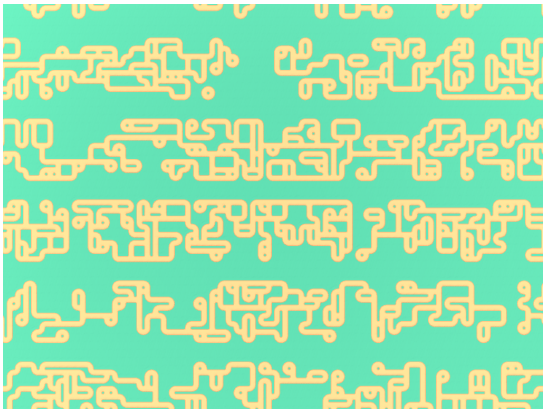
(a) The first layer is defined and cleared (b) The first layer is generated

Figure 6.11: Clearing and generating layer 1 as part of IMIB

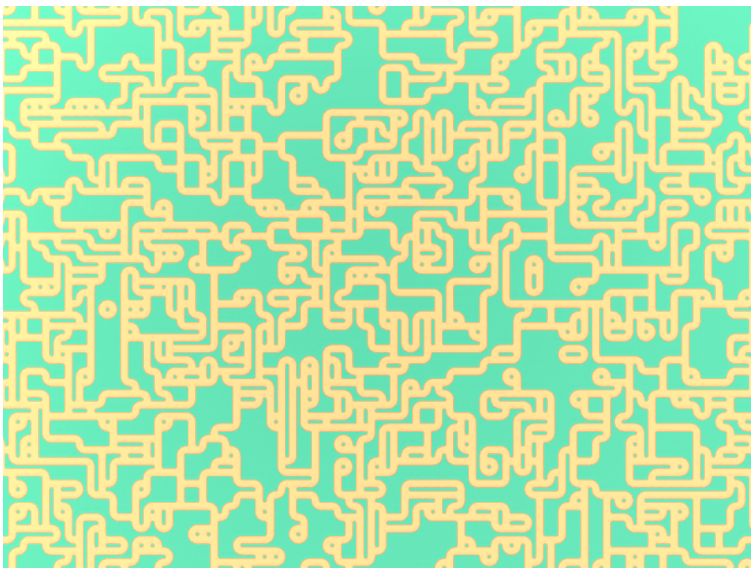
Once each chunk has generated its first layer, the second layer is cleared (Figure 6.12(a)) and generated (Figure 6.12(b)). This process is repeated until all layers have been generated (Figure 6.12(c)).



(a) The second layer is defined and cleared



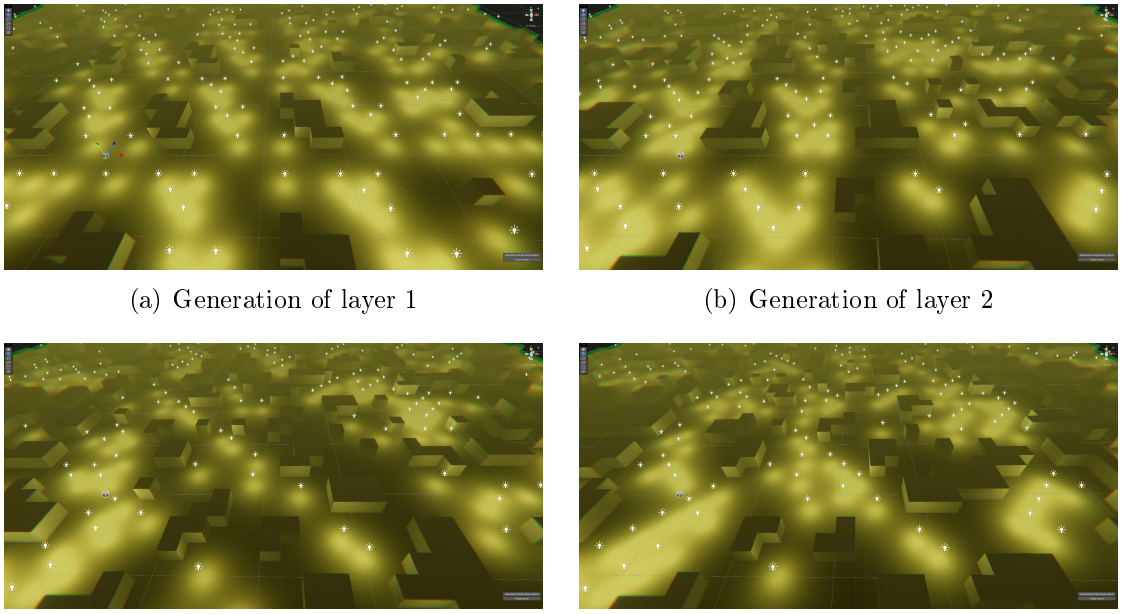
(b) The second layer is generated



(c) The result after all layers have been generated

Figure 6.12: Each IMIB layer is generated in turn to compose a full result

As further example, Figure 6.13 shows IMIB applies with *the Backrooms* tile set.



(a) Generation of layer 1

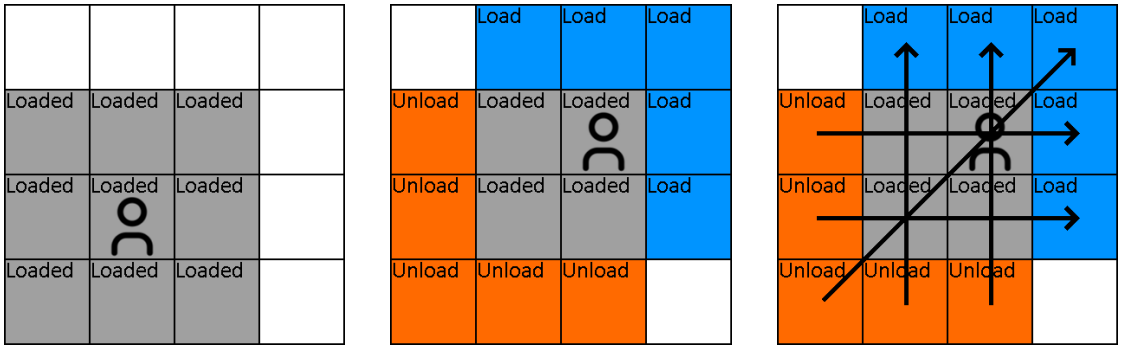
(b) Generation of layer 2

(c) Generation of layer 3

(d) Generation of layer 4

Figure 6.13: Running IMIB with *the Backrooms* tile set

Another component to consider is how to load new chunks and unload existing chunks when the player moves about. To do this efficiently, direct calculation of new chunks to load relative to old chunks is performed. As example, Figure 6.14(a) shows chunks loaded around the player. When the player moves across to another chunk, new chunks must be loaded (blue) and some of the existing chunks unloaded (orange) (Figure 6.14(b)). By calculating the horizontal and vertical difference in chunk coordinates between the player and the old chunks, the coordinates of the new chunks can be mapped from the old chunks (Figure 6.14(c)).



(a) Chunks loaded before the player moves

(b) Chunks after the player moves. Existing chunks are shown in grey, chunks to unload in orange and chunks to load in blue.

(c) Arrows showing how chunks are mapped to each other

Figure 6.14: The chunks to unload and load in an infinite world can be mapped directly from their relative position to the player

6.2 Unity Editor and Tile Representation

The Unity Editor was used to provide a platform with which level designer could specify parameters for level generation. First, the level designer should attach the Level Generation Manager component to a Unity game object. The designer may specify the size of chunks use in Infinite Modifying In Blocks. Tiles are split into three different components across the entire specification to generation pipeline, with the tile set for generation specified in steps 1 and 2.

1. The level designer must first specify tiles by creating Unity game objects and then attaching a subclass of the Tile component to it. The Cube subclass exists for fully-symmetric tiles and the NonSymmetric subclass exists for all other tiles. The designer can then specify tile adjacencies by dragging other tile game objects into the neighbours arrays and specifying the desired rotation of the neighbouring tile. Base rotations of tiles should have their solid face at the back of the tile. The base rotation for corner tiles has the back of the corner on the left and back of the tile. Each game object should then be dragged into the tile set array of the level generator. Additionally, an empty tile must be specified directly.

2. After specifying tile adjacencies, the level designer should import FBX models to form a second set of game objects. These should be given a collider to avoid the player falling through the level. The designer can also add any additional components such as lights or audio sources. These model game objects can then be referenced in the matching tile script component. This tells the level generator to use the desired model for a given tile. The tile size specified in the level generator must match the size of the tile models in order for the generator to put them together properly.

3. Finally, the tile set that the level designer specified is converted from the designer-specified semi-explicit tile set to a fully-explicit tile set. The level generator takes each specified tile and rotates it in steps of 90 degrees, adjusting adjacency data to suit. This gives each possible rotation of a tile its own tile, hence the term fully-explicit. In effect, this enables the generation of maps without having to perform intermediate calculations on tile rotation, simplifying the algorithm at the cost of a larger tile set.

6.3 Game Design

The premise of the game was inspired by the fictional concept of *the Backrooms*. These encompass an endless collection of 'levels', each a potentially infinite space with a unique theme centred around invoking a feeling of liminality. This was deemed a suitable theme to use for a game exploring novel Procedural Content Generation techniques. The first level of *the Backrooms*, 'Level Zero', resembles an empty office-like space. Yellow wallpaper, light brown carpet and bright fluorescent lights combine to give it an unsettling monotone appearance and soundscape. The player controller was taken from an online tutorial [50], [51]. This allows the player to move around the level.

6.4 Graphics

6.4.1 Models and Textures

The models with texturing for each tile were made in the software Blender using a YouTube video as a guide [52]. The tiles have distinct shapes, but share common wallpaper [53], ceiling tile [54] and carpet textures [55]. These textures are mapped so that they appear continuous when tiles are placed next to each other. The carpet texture does not connect seamlessy when rotated, so a workaround is used. This spawns the floor separately for each tile, ensuring each carpet piece has the same orientation. Each of the models used is shown in Figure 6.15. From left to right, the models include a cube, corner, corner pillar, wall, centre pillar and empty tile.

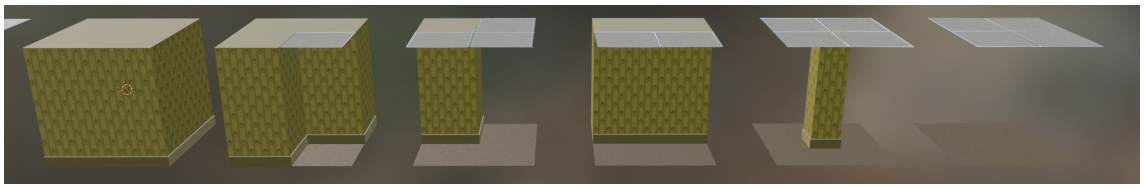


Figure 6.15: The model tile set in Blender

6.4.2 Visual Effects

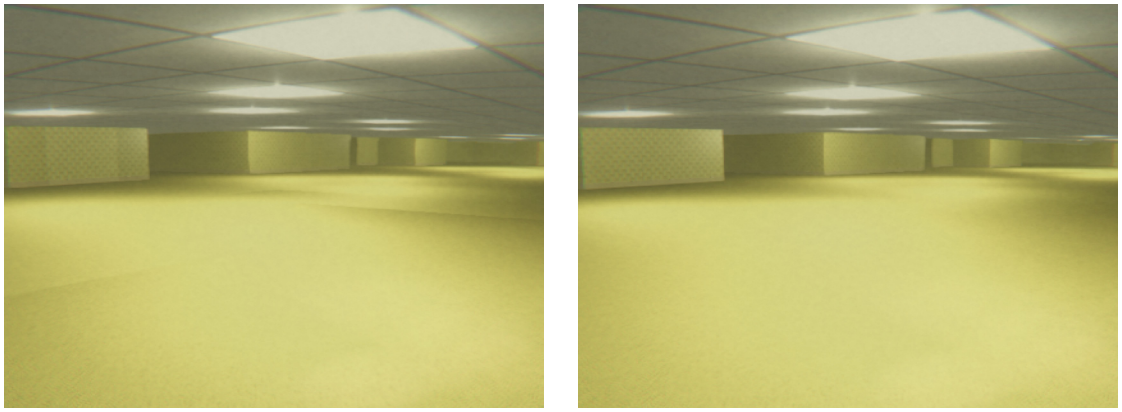
Unity's Universal Render Pipeline was used to give additional control over rendering and visual effects. To simulate the appearance of a video recorded onto a VHS tape, a range of camera effects were used. These are listed in Figure 6.16. Furthermore, the player's camera is rendered to a render texture with a resolution of 640 by 480 [56] and a letterbox used [57], further enhancing the image.

- Depth of Field: Blurs objects that are very close to the camera, simulating depth.
- Bloom: Increases the effect of light on the scene, emulating high sensitivity. Also includes a lens dirt texture [58].
- Motion Blur: Simulates unclear image capture from moving the camera.
- Tonemapping (ACES): Maps colour tones differently to give a filmic effect, making the image appear more realistic.
- Lens Distortion: Shifts projection around the centre of the image to simulate capture through a lens.
- Film Grain: Adds noise to the image to simulate similar noise real cameras can capture.
- Chromatic Aberration: Disperses red, green and blue colours at the edge of the screen to simulate the same effect lenses can exhibit in real life.
- Panini Projection: Shifts projection to better render perspective views in wide angle scenes. Acts as additional distortion to make the image appear more organic.
- Color Adjustments: Reduces exposure to make tones more neutral. Also adds a colour filter to shift colours slightly more towards monochrome yellow to fit the level theme.
- Fog: Obscures distant objects to hide the border of currently generated chunks.
- VHS Overlay Videos: Adds additional noise seen when playing VHS tapes [56], [59], [60].

Figure 6.16: Camera Effects

6.4.3 Lighting Model

A deferred lighting model was used to light the level [61]. Unity's default lighting relies heavily on baked lighting to provide for high quality lighting. Baking lighting describes pre-calculating lighting before runtime. Textures can then be illuminated cheaply at runtime using this baking data to help give levels a realistic look. As the project uses procedurally generated levels, it must heavily rely on realtime lighting instead. This describes lighting that is calculated at runtime as opposed to baked lighting. However, forward rendering (Unity's default rendering method) only allows for a limited number of realtime light sources to be rendered at once. Deferred lighting instead supports many realtime lights at the cost of less accurate lighting. Figure 6.17 compares forward and deferred rendering in a generated level, showing that deferred is more suited to rendering many light sources.



(a) Forward rendering fails to compute lighting accurately with many light sources

(b) Deferred lighting is more suited to many light sources

Figure 6.17: A comparison of forward and deferred lighting. Forward lighting fails to light the scene properly, showing distinct lines on tiles.

6.5 Audio

To give the game area an immersive soundscape, a collection of ambient effects and sound effects were used. Ambient effects are those sounds that are played equally at all times, while sound effects are those sounds that are placed inside of the environment. The most distinct sound effect of Level Zero of the Backrooms is the buzzing of the fluorescent lights. Each empty tile with a light is given an audio source that plays such a sound [62]. Each light is given its own random offset so that the audio does not poorly overlap and cause constructive interference. For ambient sounds, a VHS sound is included [63]. All audio assets were taken from `freesound.org` as this site offers sounds free for use in games.

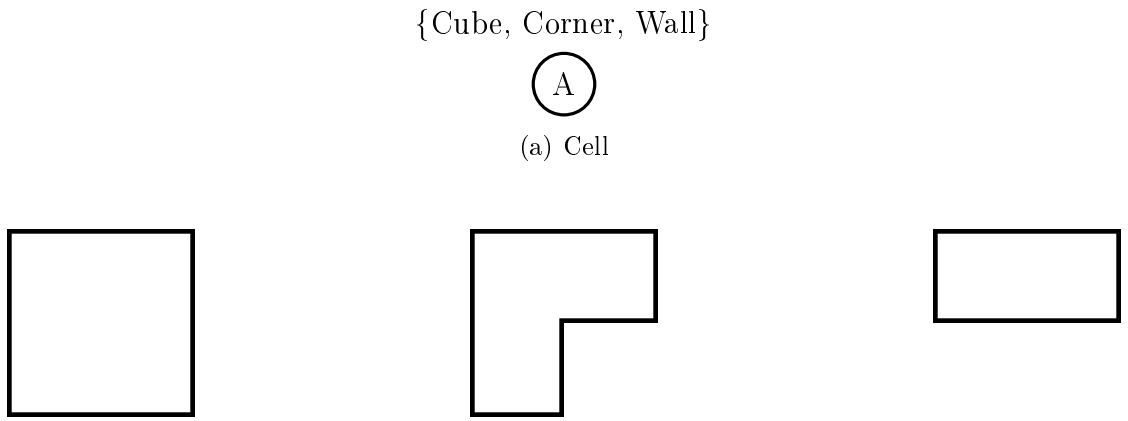
Chapter 7

Implementation

7.1 Cells

7.1.1 Cell Class

The cell class contains integer coordinates, a set of possible tile options and a tile prefab game object. These variables allow the cell to be used in a grid for level generation. The set of tile options is initialised with the entire fully-explicit tile set. As WFC is run on a grid including the cell, tiles that become invalid options are removed. Once the cell has been collapsed, the assigned tile is instantiated. Additionally a reference to the instantiated tile is preserved in the tile prefab field.



(a) Cell

(b) Cube Tile

(c) Corner Tile

(d) Wall Tile

Figure 7.1: A cell with cube, corner and wall tiles as options

One additional property to note is that the cell class does not inherit from Unity's MonoBehaviour class. This means that it can use constructors that pass basic information needed to initialise each cell. Initialisation data includes the coordinates of the cell within the current layer, the cell's global grid coordinates and the current chunk using the cell. This data is needed for constraint calculations while solving and is updated each time a new layer is spawned.

Objects that do inherit from MonoBehaviour cannot use constructors safely and must instead use Unity's Start or Awake functions. However, the Start and Awake functions cannot easily be passed initialisation data. The cost of not inheriting

from MonoBehaviour is that any such instances cannot be attached to any objects spawned in the world.

7.1.2 CellArc Class

The cell arc class is used to represent arcs in the grid. Each arc contains references to two cells. The constructor is used to ensure that the arc is valid. This is done by comparing the x and y coordinates of the cells when creating the arc. An exception is thrown when the cells are not adjacent. Cells being adjacent is specific to the context of WFC. In Sudoku, for example, a pair of constrained cells need not be directly adjacent.

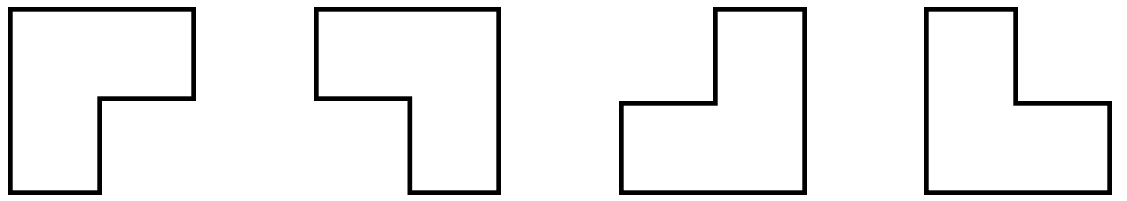
7.1.3 CellReference Class

The cell reference class contains a reference to a cell. As mentioned in Section 7.1.1, the cell class does not inherit from MonoBehaviour and as such cannot be attached to any objects in the world. By instead inheriting from MonoBehaviour, a cell reference instance can be attached to a tile. Having this cell reference attached to a tile allows the cell's data to be used in generation of later layers.

7.2 Tiles

7.2.1 Rotation / Cardinality

Each spawned tile has a cardinality of either 0, 1, 2 or 3. These cardinalities correspond to rotations of 0, 90, 180 and 270 degrees respectively. As example, all cardinalities for the corner tile are shown in Figure 7.2.



(a) Cardinality 0 (0 Degrees)

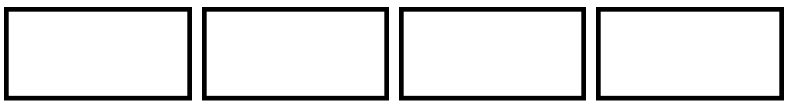
(b) Cardinality 1 (90 Degrees)

(c) Cardinality 2 (180 Degrees)

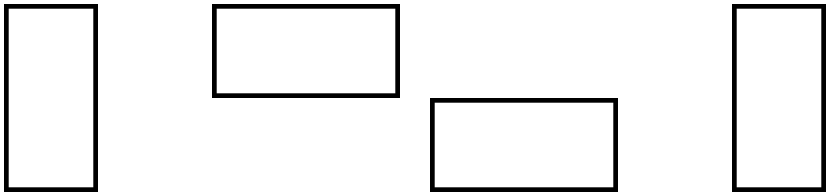
(d) Cardinality 3 (270 Degrees)

Figure 7.2: The corner tile with all possible rotations (counted clockwise)

Cardinality information is used to determine which combination of two tiles and their rotations fit together. For example, a single wall tile constituting part of a longer wall will require any adjacent wall tiles to be of the same cardinality (Figure 7.3(a)). When specifying the neighbours of a tile, the level designer must also set the cardinality of each neighbour.



(a) Valid wall placement (Cardinalities 0,0,0,0)



(b) Invalid wall placement (Cardinalities 3,0,2,1)

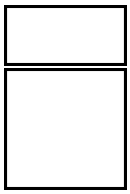
Figure 7.3: Comparison of valid wall placement to invalid wall placement if adjacent walls must have matching cardinality

7.2.2 Tile Symmetry

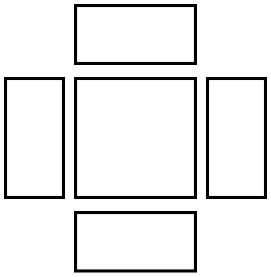
To reduce the amount of adjacency data to be specified by the level designer, as well as the chance for human error, the level generator includes the functionality to convert a semi-explicit tile set into a fully-explicit tile set. This fully-explicit tile set is created at the start of generation in several steps.

First, it is checked whether the empty tile was specified and is included in the semi-explicit tile set. After confirming this, the main conversion stage begins.

Second, a non-symmetric copy is created for each tile. Non-symmetric tiles may have different neighbours on each side. Any symmetric tile can be represented as a non-symmetric tile. For example, a tile with equal symmetry on all sides can be converted by copying the single list of neighbours onto all sides and incrementing the cardinality per side as required. Specifically, cardinalities on side 1 are incremented by 1, those on side 2 by 2 and those on side 3 by 3. This fully symmetric tile is called a cube tile in the code.



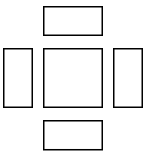
(a) A symmetric cube tile with only a wall neighbour of cardinality 2 on the back side specified



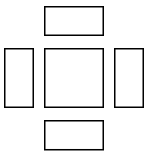
(b) A non-symmetric cube tile variant with a wall neighbour specified on each side. Clockwise from the back, cardinalities of the walls are 2, 3, 0 and 1.

Figure 7.4: A symmetric cube tile converted to a non-symmetric tile. For simplicity, the cube is only shown with a wall as its possible neighbour.

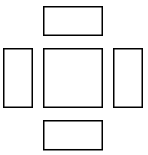
Third, an array for each possible tile rotation is created (cardinalities 0, 1, 2 and 3). This array is attached to the original tile to set explicit neighbours later. The instantiated tile is used as the base rotation (cardinality 0) and referenced in the array. For cube tiles, the base rotation tile can be referenced for each rotation instead of having to create additional copies. The filled in array for cube tiles is shown in Figure 7.5.



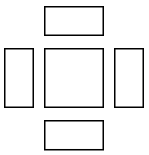
(a) Cube Tile Explicit Variant 0



(b) Cube Tile Explicit Variant 1



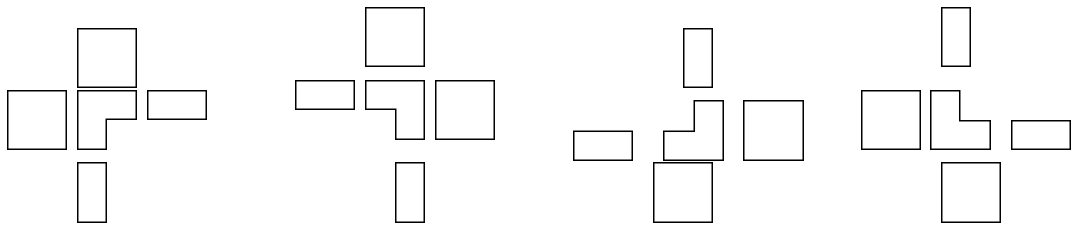
(c) Cube Tile Explicit Variant 2



(d) Cube Tile Explicit Variant 3

Figure 7.5: The array of explicit variants for the cube tile with neighbours shown. Due to the tile's symmetry, no extra work is needed to calculate variants for cardinalities 1, 2 and 3.

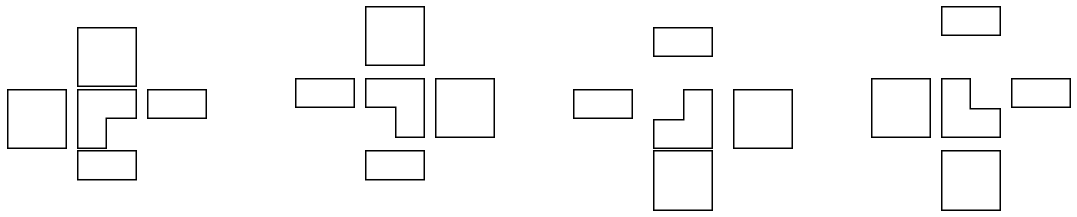
For non-cube tiles, variants for the remaining cardinalities (1, 2 and 3) must be created. Each new variant copies the previous one. This allows a loop to be used that always performs 90 degree steps. First, the tile is rotated by 90 degrees. Then, neighbour data and rotation arrays are swapped. This sets back neighbours as left neighours, left neighbours as front neighbours and so on. Finally, the cardinality values of all neighbours are incremented by 1 to mark the 90 degree rotation. The filled in array for corner tiles is shown in Figure 7.6.



(a) Corner Tile Explicit Variant 0 (b) Corner Tile Explicit Variant 1 (c) Corner Tile Explicit Variant 2 (d) Corner Tile Explicit Variant 3

Figure 7.6: The array of explicit variants for the corner tile with neighbours shown.

These explicit variants do not yet have tile rotation implicitly encoded in their neighbour data. The original tile neighbours in the tile array must be replaced with their explicit variants matching the rotations in the rotations array. To do this, the array assigned to each tile containing all the explicit variants is used once all explicit variants have been created. For each neighbour of the explicit tile, the reference to the non-explicit neighbour is replaced by the explicit neighbour with the correct orientation. As example, the explicit corner tile variants are shown with the original cube and wall neighbours in Figure 7.7. The references are changed to explicit cube and wall neighbours to match Figure 7.6.



(a) Corner Tile Explicit Variant 0 Before Neighbour Update (b) Corner Tile Explicit Variant 1 Before Neighbour Update (c) Corner Tile Explicit Variant 2 Before Neighbour Update (d) Corner Tile Explicit Variant 3 Before Neighbour Update

Figure 7.7: The explicit corner tile variants before updating neighbours to their explicit forms

7.2.3 Collapsing Cells

To collapse a tile into a cell, the tile's model is instantiated. Then, the model is set to follow the level generation manager's transform (position, rotation and scale) and has its local position reset to the cell's world coordinates. Doing the placement in this way ensures that the model lines up with the world grid while maintaining the correct scale and rotation. Finally, a Unity box collider is added to the model, a second reference to the tile added to the cell and the model activated to render it in the world. The second reference ensures that the spawned tile can be recovered if the generation of a layer fails. Unity's 'Physics.OverlapBox' function can be used to detect the box collider given the world coordinates of the cell. This box overlap can be used to obtain the cell reference during the generation of later layers.

7.3 Chunks

Chunks are defined by their own chunk coordinates, which are converted from world coordinates using the chunk size as a divisor. From this, each of the chunk's four layers can be defined. Layer one is aligned with the chunk's world coordinates, while layer two is offset by half a chunk in the x direction. Layer three is similarly offset by half a chunk in the y direction, while layer four is offset both in the x and y directions. Each layer is given its own layer spawner instance, which runs WFC on the portion of the chunk defined by the layer.

Each chunk has its own Random Number Generator (RNG), which is used by the layer spawners when making tile choices. The seed for the RNG is defined deterministically through the chunk's coordinates. The global level generation seed is added to allow generation of multiple levels while keeping determinism. This deterministic use of RNGs when spawning chunks is what allows the level generation manager to spawn chunks identically, even when unloading and loading a chunk again.

7.4 Level Generation Manager

7.4.1 Prerequisites

To start generation, the level designer must have defined a tile set for the generator to use. This must contain an empty tile and be convertible into an explicit tile set as detailed in Section 7.2.2.

7.4.2 Solver (Layer Spawners)

The MAC3 solver code is held in the layer spawner class. This matches the idea of layers and chunks modifying the world in an infinite set of blocks. To allow accurate placement of tiles in the world, a starting cell coordinate must be passed when initialising the layer spawner. Furthermore, a reference to the layer's parent chunk is kept in order to utilise its RNG.

7.4.3 Grid Initialisation

Before starting generation of a layer, the grid of cells composing it must be initialised. Additionally, a list of cells left to assign is initialised to simplify cell choices when solving. Padding on the layer size is used to include any previously collapsed cells. Added cells that have not been collapsed are treated as empty tiles. This padding means that non-padded cells take into account full adjacency information. If the layer were not padded, then border cells would not have arc consistency with cells outside of the layer as these arcs would not be checked by the solver. The non-padding cells in the centre of the grid are refreshed by restoring each cell's tile options set to include all possible tiles. This allows each each layer to re-generate its cells. In effect, this connects singular, overlapping layers into an infinite, continuous grid.

7.4.4 Dealing with an Infinite Grid

Were every cell stored in a global grid, then it would be trivially easy to get previously collapsed cells. This is unfeasible due to the requirement of letting the player navigate an infinite world. As such, once cells are collapsed, the grid used by the solver is discarded. Instead, the cell reference class described in Section 7.1.3 is used to preserve a reference to the cell. Each tile has its own collider that can be used to obtain the cell reference as described in Section 7.2.3.

7.4.5 Setting up MAC3

With the layer's grid fully initialised, the main stage of the solver begins. First, a stack of state changes is initialised. This tracks changes to cells after each assignment by pushing a new state change to the stack. Second, global arc consistency must be enforced. To do this, AC3 is run with the starting queue consisting of all arcs in the grid.

7.4.6 MAC3 Recursion

Once global arc consistency has been enforced, the first iteration of the recursive MAC3 algorithm is called. Pseudocode of this is given below. How each step works and its purpose is explained in more detail in subsections following the pseudocode.

1. If all cells have been assigned, return.
2. Choose a cell and tile to assign as in Section 6.1.6.
3. Enter a new state and assign the tile to the cell, pruning other tiles from its tile options set.
4. If the cell's tile options set changed, run AC3.
5. Recurse if AC3 was not run or there was no domain wipeout during AC3.
6. If all cells have been assigned, return.
7. We are now in the case that a domain wipeout must have occurred. Revert the state.
8. If the cell we tried to assign still has other options left, remove / 'unassign' the tile we tried from the options, run AC3 and recurse again.
9. If all cells have been assigned, return.
10. It is now the case that all possible tile values for a cell have led to a domain wipeout. This means that an earlier choice must be to blame. Restore the tile we tried both to assign and unassign and return.

Managing State Changes

To track state changes, each layer spawner holds a stack of state changes. After each time a new cell is chosen to be assigned, a new state is entered. A state change class is used that holds the cell being assigned and any domain changes occurring across the grid. Domain changes can happen to any cell and can consist of multiple tile changes. Hence, a dictionary that maps each cell to a hash set of removed tiles is used. To track a new state change, a new key and hash set are added to the dictionary if required. After this, the hash set can be obtained from the dictionary through use of the cell and any tile changes added.

To revert a state, the domain changes dictionary is iterated through, with each cell having any removed tiles restored to its domain. Furthermore, the cell that was assigned in the state change is re-added to the list of cells left to assign.

Assigning and Unassigning a Tile to a Cell

Assigning a tile to a cell involves pruning all tiles except the one being assigned. These tiles are removed from the tile options set. Furthermore, each domain change is recorded in the current state changes. Finally, the assigned cell is removed from the list of cells left to assign. After this, AC3 can be run to maintain arc consistency.

Unassigning a tile to a cell first requires the state to be reverted. After this, the opposite choice is taken. This is done by pruning only the tile choice that was previously assigned. This demonstrates the binary branching nature of the implementation. Rather than exploring all possible tile values in one level of search, each tile value is either assigned or unassigned.

Visualising Recursion

The recursive MAC3 method can be visualised as consisting of three parts as shown in Figure 7.8.

1. Making a new assignment.
2. Making the opposite assignment (“unassignment”) after the first one failed.
3. Both assignments having failed.

Figure 7.8: The three parts of the recursive MAC3 method

The finishing state of the grid is checked before making the new assignment and after each of the assignments. This ensures that the recursion is finished immediately after consistency has been enforced from an assignment and each cell only has one value left.

Chapter 8

Evaluation

The goal of the project was initially to extend upon an existing implementation of WFC. As this was unsuccessful, the algorithm had to be implemented from scratch. Implementing Wave Function Collapse and extending it with Infinite Modifying In Blocks presented a huge technical challenge. Ultimately, this detracted from work on the game itself as the focus of the dissertation changed. This was reflected in a change of dissertation title from 'Procedural Content Generation in Video Games' to 'Wave Function Collapse as a Constraint Solver for Game Level Generation'. The objectives set at the start of the project also suffered from this.

8.1 Against Requirements Specification

8.1.1 Primary Objectives

Create a game that uses procedural level generation

The final project includes a very basic game in which the player can explore an infinite world generated through procedural level generation. The game is themed after *the Backrooms*, a fictional concept detailing an infinite expanse of 'levels'. The first of these levels, 'Level Zero', is included in the game.

Use novel PCG methods such as Wave Function Collapse

Wave Function Collapse was studied thoroughly throughout the project. WFC was contextualised within theory of constraint programming and a simple-tiled implementation created using the MAC3 algorithm with lowest entropy cell and weighted random tile selection. Alternative selection strategies are included in the code.

Extend on at least one PCG method

Standard Wave Function Collapse is finite in grid size and has increasing complexity with larger grid sizes. WFC was extended upon through the addition of the Infinite Modifying In Blocks algorithm. This helped generate output consistently, where standard WFC might require extensive backtracking in certain cases. Furthermore, it allowed WFC to be applied to generate an infinite world by generating many small, finite chunks as the player moves around in the world.

Use assets to give the game a full set of graphics and audio

A small set of models were created to serve as tiles for WFC. These match the design of *the Backrooms* through the use of online assets. Furthermore, a basic soundscape is created with some tiles given fluorescent lamp sound effects. Ambience sound played consistently throughout the game also add to this.

8.1.2 Secondary Objectives

Make levels navigable by AI opponents

No attempts were made to introduce AI opponents to the game.

Allow customisation of level generation and other gameplay elements via an in-game menu

The final game does not include any in-game menu, but the generation interface allows the level designer to set the seed of the level generator, adjust weights, tiles and chunk properties.

8.2 Designer-facing Components

8.2.1 Editor Interface and Code Quality

The final project includes an interface for level designers to run the WFC implementation with their own tile set. Efforts were made to comment code clearly, which is something other WFC implementations miss. This is also an issue facing the original WFC implementation, which does not include any comments in its code [1].

8.2.2 Ease of Use

The level generation code includes exceptions that can tell the level designer what is incorrect with their setup. However, despite this additional guidance, the level generation manager interface can be confusing. Especially concepts such as cells vs tiles, tiles vs models and tile cardinality are likely to confuse users of the project's code. This problem is likely worsened since WFC is often treated as a black box by people from a range of backgrounds [29]. These users may not be experts in reading code and simply want to set up the generator with their own tile set. It is possible that these users are the likeliest to struggle with correctly setting up the level generation manager.

8.3 Player-facing Components

8.3.1 Gameplay

Players are able to explore an infinite map. While this may be interesting for a while, the experience has a significant issue with homogeneity. The low variety of models used means that the geometry of the level does not form many interesting structures. The lack of multiple levels also limits the amount of enjoyment players

may get out of the game. One idea during development was to add puzzles and give each level its own sub-areas with different tile designs.

8.3.2 Performance

While the procedural level generation is functional, it suffers from a lack of optimisations that would make it more suitable to application for a video game. For example, chunk loading could be multi-threaded. However, implementation of this is difficult due to poor support by Unity. Furthermore, the use of Unity coroutines could be explored to let chunk loading take place across multiple frames. This would avoid lag spikes, which can be very off-putting to players. Another potential optimisation might be alternative chunk management. Carrying data from existing chunks to new chunks to be loaded directly could significantly improve performance over the current indirect method. Having a data structure that efficiently manages this would be key to making such an optimisation work.

Chapter 9

Conclusion

Wave Function Collapse (WFC) is limited by a lack of global constraints, poor performance and restriction to a finite grid. Furthermore, presentation of WFC online often fails to acknowledge underlying constraint solving principles used by WFC. This dissertation examined how these issues can be addressed and contextualised WFC within theory of constraint programming.

A simple-tiled implementation of WFC using the Maintaining Arc Consistency 3 (MAC3) algorithm was presented and extended to generate infinite worlds using Infinite Modifying In Blocks (IMIB). Weighted tile selection gives the level designer global control over the percentage of each tile in the output level. These concepts are demonstrated in a game themed after the popular fictional concept of *the Backrooms*. These additions to WFC show that the lack of global constraints and restriction to a finite grid can be addressed. The included game and generation interface illustrate the potential application of WFC with these additions to video games.

However, loading infinite worlds in real-time presents additional performance challenges and further global constraints are needed to improve control further. The solver could be optimised through the addition of multi-threading and lazy loading across frames. Furthermore, code could be reworked to more efficiently carry generation data across layers. The ease of use of the generation interface in the Unity Editor could be improved by making it simpler to specify a tile set for generation in the Unity Editor. The game could be further developed through the addition of more models and levels.

In summary, this dissertation contextualised WFC within theory of constraint programming and addressed WFC's limitations through IMIB and weighted tile selection, extending generation to an infinite space and applying this in a game themed after *the Backrooms*.

Bibliography

[1] M. Gumin, *Wave Function Collapse Algorithm*, version 1.0, Sep. 2016. [Online]. Available: https://github.com/mxgmn/WaveFunctionCollapse.

[2] O. Stålberg, *Townscaper*, 2020. [Online]. Available: https://www.townscapergame.com/ (visited on 13/09/2023).

[3] O. Stålberg, *Bad north*, 2018. [Online]. Available: https://www.badnorth.com/ (visited on 13/09/2023).

[4] Freehold Games, *Caves of qud*, 2023. [Online]. Available: https://www.cavesofqud.com/ (visited on 30/10/2023).

[5] M. O'Leary, *Oisín: Wave function collapse for poetry*. [Online]. Available: https://github.com/mewo2/oisin (visited on 13/03/2024).

[6] R. B. Pál Patrik Varga, *Procedural mixed-initiative music composition with hierarchical wave function collapse*. [Online]. Available: https://graphics.tudelft.nl/Publications-new/2023/VB23/WFC_audio_PCG_WS_demo.final.pdf (visited on 13/03/2024).

[7] B. Baltaxe-Admony, *Wfc piano roll*. [Online]. Available: https://github.com/bbaltaxe/wfc-piano-roll (visited on 13/03/2024).

[8] I. Karth and A. M. Smith, 'Wavefunctioncollapse: Content generation via constraint solving and machine learning,' *IEEE Transactions on Games*, vol. 14, no. 3, pp. 364–376, 2022. DOI: 10.1109/TG.2021.3076368.

[9] P. C. Merrell, *Model synthesis*. [Online]. Available: https://gamma-web.iacs.umd.edu/papers/documents/dissertations/merrell09.pdf (visited on 05/03/2024).

[10] Boris, *Infinite modifying in blocks*, Jun. 2022. [Online]. Available: https://www.boristhebrave.com/2021/11/08/infinite-modifying-in-blocks/.

[11] J. Fingas, *Here's how "minecraft" creates its gigantic worlds*, Jul. 2019. [Online]. Available: https://www.engadget.com/2015-03-04-how-minecraft-worlds-are-made.html (visited on 12/09/2023).

[12] H. Alexandra, *A look at how no man's sky's procedural generation works*, Oct. 2016. [Online]. Available: https://kotaku.com/a-look-at-how-no-mans-skys-procedural-generation-works-1787928446 (visited on 12/09/2023).

[13] 2. Game DeveloperStaffJanuary 01, *7 uses of procedural generation that all developers should study*, Jan. 2016. [Online]. Available: https://www.gamedeveloper.com/design/7-uses-of-procedural-generation-that-all-developers-should-study (visited on 12/09/2023).

[14] W. Yin-Poole, *How many weapons are in borderlands 2?* Jul. 2012. [Online]. Available: https://www.eurogamer.net/how-many-weapons-are-in-borderlands-2 (visited on 12/09/2023).

[15] N. Shaker, J. Togelius and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016. DOI: 10.1007/978-3-319-42716-4. [Online]. Available: https://www.pcgbook.com/ (visited on 12/09/2023).

[16] Minecraft, *Minecraft caves and cliffs snapshot twitter post*. [Online]. Available: https://twitter.com/Minecraft/status/1414978446229393420 (visited on 17/03/2024).

[17] Y. Cao *et al.*, *A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt*, 2023. arXiv: 2303.04226 [cs.AI].

[18] C.-H. Lin *et al.*, *Magic3d: High-resolution text-to-3d content creation*, 2023. arXiv: 2211.10440 [cs.CV].

[19] A. Gambi, M. Mueller and G. Fraser, 'Automatically testing self-driving cars with search-based procedural content generation,' in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 318–328. DOI: 10.1145/3293882.3330566. [Online]. Available: https://doi.org/10.1145/3293882.3330566.

[20] Stability AI, *Stable diffusion generative models*. [Online]. Available: https://github.com/Stability-AI/generative-models (visited on 26/10/2023).

[21] OpenAI, *Chatgpt*. [Online]. Available: https://chat.openai.com/ (visited on 26/10/2023).

[22] GitHub, *Github copilot*. [Online]. Available: https://github.com/features/copilot (visited on 26/10/2023).

[23] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis and J. Togelius, 'Deep learning for procedural content generation,' *Neural Computing and Applications*, vol. 33, no. 1, pp. 19–37, Jan. 2021. DOI: 10.1007/s00521-020-05383-8. [Online]. Available: https://doi.org/10.1007/s00521-020-05383-8.

[24] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius and S. M. Lucas, 'General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms,' *IEEE Transactions on Games*, vol. 11, no. 3, pp. 195–214, 2019. DOI: 10.1109/TG.2019.2901021.

[25] X. Neufeld, S. Mostaghim and D. Perez-Liebana, 'Procedural level generation with answer set programming for general video game playing,' in *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, 2015, pp. 207–212. DOI: 10.1109/CEEC.2015.7332726.

[26] R. Rodriguez Torrado, A. Khalifa, M. Cerny Green, N. Justesen, S. Risi and J. Togelius, 'Bootstrapping conditional gans for video game level generation,' in *2020 IEEE Conference on Games (CoG)*, 2020, pp. 41–48. DOI: 10.1109/CoG47356.2020.9231576.

[27] A. Khalifa, P. Bontrager, S. Earle and J. Togelius, 'Pcgrl: Procedural content generation via reinforcement learning,' *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 16, no. 1, pp. 95–101, Oct. 2020. DOI: 10.1609/aiide.v16i1.7416. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/7416.

[28] A. Summerville *et al.*, 'Procedural content generation via machine learning (pcgml),' *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, 2018. DOI: 10.1109/TG.2018.2846639.

[29] I. Karth and A. M. Smith, 'Wavefunctioncollapse is constraint solving in the wild,' in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, ser. FDG '17, Hyannis, Massachusetts: Association for Computing Machinery, 2017. DOI: 10.1145/3102071.3110566. [Online]. Available: https://doi.org/10.1145/3102071.3110566.

[30] C. Jefferson *et al.*, *Csplib problems library*. [Online]. Available: https://www.csplib.org/Problems/categories.html (visited on 31/10/2023).

[31] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz and A. A. Cire, 'Combining reinforcement learning and constraint programming for combinatorial optimization,' *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 5, pp. 3677–3687, May 2021. DOI: 10.1609/aaai.v35i5.16484. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/16484.

[32] P. Spracklen, N. Dang, Ö. Akgün and I. Miguel, 'Automated streamliner portfolios for constraint satisfaction problems,' *Artificial Intelligence*, vol. 319, p. 103 915, 2023. DOI: https://doi.org/10.1016/j.artint.2023.103915. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370223000619.

[33] *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach* (Lecture Notes in Computer Science), en. Cham: Springer International Publishing, 2016, vol. 10101. DOI: 10.1007/978-3-319-50137-6. [Online]. Available: http://link.springer.com/10.1007/978-3-319-50137-6.

[34] Ö. Akgün, I. P. Gent, C. Jefferson, I. Miguel and P. Nightingale, 'Metamorphic testing of constraint solvers,' in *Principles and Practice of Constraint Programming*, J. Hooker, Ed., Cham: Springer International Publishing, 2018, pp. 727–736.

[35] G. De Gasperis, S. Costantini, A. Rafanelli, P. Migliarini, I. Letteri and A. Dyoub, 'Extension of constraint-procedural logic-generated environments for deep Q-learning agent training and benchmarking,' *Journal of Logic and Computation*, exad032, Jun. 2023. DOI: 10.1093/logcom/exad032. eprint: https://academic.oup.com/logcom/advance-article-pdf/doi/10.1093/logcom/exad032/50530031/exad032.pdf. [Online]. Available: https://doi.org/10.1093/logcom/exad032.

[36] G. Glorian, A. Debesson, S. Yvon-Paliot and L. Simon, 'The Dungeon Variations Problem Using Constraint Programming,' in *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, L. D. Michel, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 210, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 27:1–27:16. DOI: 10.4230/LIPIcs.CP.2021.27. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/15318.

[37] A. Khalifa, D. Perez-Liebana, S. M. Lucas and J. Togelius, 'General video game level generation,' in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16, Denver, Colorado, USA: Association for Computing Machinery, 2016, pp. 253–259. DOI: 10.1145/2908812.2908920. [Online]. Available: https://doi.org/10.1145/2908812.2908920.

[38] J. Espasa, I. Miguel and M. Villaret, 'Plotting: A Planning Problem with Complex Transitions,' in *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, C. Solnon, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 22:1–22:17. DOI: 10.4230/LIPIcs.CP.2022.22. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16651.

[39] P. Merrell and D. Manocha, 'Model synthesis: A general procedural modeling algorithm,' *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 6, pp. 715–728, 2011. DOI: 10.1109/TVCG.2010.112.

[40] A. Sandhu, Z. Chen and J. McCoy, 'Enhancing wave function collapse with design-level constraints,' in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, ser. FDG '19, San Luis Obispo, California, USA: Association for Computing Machinery, 2019. DOI: 10.1145/3337722.3337752. [Online]. Available: https://doi.org/10.1145/3337722.3337752.

[41] B. Bucklew, *Math for game developers: Tile-based map generation using wavefunctioncollapse in caves of qud*. [Online]. Available: https://www.gdcvault.com/play/1026263/Math-for-Game-Developers-Tile (visited on 10/03/2024).

[42] D. Cheng, H. Han and G. Fei, 'Automatic generation of game levels based on controllable wave function collapse algorithm,' in Jan. 2020, pp. 37–50. DOI: 10.1007/978-3-030-65736-9_3.

[43] Y. Nie, S. Zheng, Z. Zhuang and X. Song, *Extend wave function collapse to large-scale content generation*, 2023. arXiv: 2308.07307 [cs.AI].

[44] H. Kim, S. Lee, H. Lee, T. Hahn and S. Kang, 'Automatic generation of game content using a graph-based wave function collapse algorithm,' in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–4. DOI: 10.1109/CIG.2019.8848019.

[45] T. Møller and J. Billeskov, 'Expanding wave function collapse with growing grids for procedural content generation.,' Ph.D. dissertation, May 2019. DOI: 10.13140/RG.2.2.23494.01607.

[46] G. Kane, *Easy wave function collapse*. [Online]. Available: https://github.com/GarnetKane99/WaveFunctionCollapse (visited on 30/10/2023).

[47] J. A. Parker, *Unity-wave-function-collapse*. [Online]. Available: https://selfsame.itch.io/unitywfc (visited on 25/09/2023).

[48] S. K. Maksim Priakhin, *Unity-wave-function-collapse-3d*. [Online]. Available: https://github.com/oddmax/unity-wave-function-collapse-3d (visited on 20/09/2023).

[49] Epic Games, *Wave function collapse*. [Online]. Available: https://docs.unrealengine.com/5.0/en-US/BlueprintAPI/WaveFunctionCollapse/ (visited on 13/03/2024).

[50] D. Morman, *How to make an fps player in under a minute - unity tutorial*. [Online]. Available: https://www.youtube.com/watch?v=qQLvcS9FxnY (visited on 05/10/2023).

[51] D. Morman, *Fpscontroller*. [Online]. Available: https://github.com/dustinmorman/FPSControllerTutorial/blob/main/Assets/Scripts/FPSController.cs (visited on 10/11/2023).

[52] Default Cube, *Youtube video*. [Online]. Available: https://youtu.be/_u0dy3TMZ_s (visited on 30/12/2023).

[53] Scans and textures, *The backrooms wallpaper texture*. [Online]. Available: https://sketchfab.com/3d-models/the-backrooms-wallpaper-texture-f7d2757432f94a2bb366c9a41503647b (visited on 31/12/2023).

[54] TextureCan, *Office ceiling tiles texture*. [Online]. Available: https://www.texturecan.com/details/131/ (visited on 19/01/2024).

[55] TextureCan, *Wool carpet texture*. [Online]. Available: https://www.texturecan.com/details/148/ (visited on 18/01/2024).

[56] Carrots In The Sky, *How to make realistic vhs style graphics really fast for your game*. [Online]. Available: https://youtu.be/BK_NnKr4r04 (visited on 20/01/2024).

[57] rabidgremlin, *LetterBoxer repository*. [Online]. Available: https://github.com/rabidgremlin/LetterBoxer/ (visited on 15/01/2024).

[58] ActionVFX, *Lens dirt overlays*. [Online]. Available: https://www.actionvfx.com/collections/free-lens-dirt-overlays (visited on 18/01/2024).

[59] Enchanted Media, *Vhs effect overlay video*. [Online]. Available: https://www.enchanted.media/downloads/free-vhs-effect-overlay-video/ (visited on 20/01/2024).

[60] Haro Games, *Vhs retro effect - unity tutorial*. [Online]. Available: https://youtu.be/CHUS0udL9Nk (visited on 20/01/2024).

[61] Unity, *Forward and deferred rendering*. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@17.0/manual/Forward-And-Deferred-Rendering.html (visited on 20/02/2024).

[62] J. Bezouska, *Factory fluorescent light buzz*. [Online]. Available: https://freesound.org/people/janbezouska/sounds/537769/ (visited on 20/01/2024).

[63] flacfile, *Vhs hum sound*. [Online]. Available: https://freesound.org/people/flacfile/sounds/596531/ (visited on 20/01/2024).

Appendix A

Testing Summary

A debug mode flag is included with the level generation manager. This prints information on the solver during initialisation and runtime. Debug mode was heavily used during development to ensure each part of the solver works correctly. Debug mode is suited for testing extremely small levels. For bigger levels, the debug statements printed to console result in an extremely significant increase to generation time.

While unit testing would potentially be useful for small parts of the solver, such as individual arc revisions and runs of AC3, testing the system as a whole is more challenging. The same issue of ensuring output is playable faces any sort of unit test of WFC. In the context of video games, the qualitative elements of what makes an output level enjoyable are similarly difficult to test. Furthermore, the cell and tile selection strategies used in the implementation use RNGs to form output, complicating testing further.

Appendix B

Player Guide

Premise

Embark on an infinite journey through the enigmatic depths of *the Backrooms*. Take in the eerie solitude in this procedurally generated liminal space, where the boundaries between reality and the unknown blur into obscurity. Traverse endless rooms in search of escape at the elusive escape points scattered throughout. Will you manage to find an exit, or will you become lost forever in this maze of yellow hues and buzzing lights?

Controls

Mouse:	Look
Left Shift:	Run
WASD Keys:	Move

Appendix C

Meeting Notes

C.1 Semester One Meeting Notes

Weeks 0 to 3

- Research and narrowing down project aims.
- DOER and ethics submissions.

Week 4 - 03.10.2023

- Create a design outline document.
- Try to adapt one of the WFC implementations to 3D.
  - https://github.com/GarnetKane99/WaveFunctionCollapse
  - Up, down, right, left, front, back neighbours.
  - Each prefab in the neighbours array should have a weighting too. Use an entropy calculation.
- Get a simple, one-room playable demo.

Week 5 - 10.10.2023

- No meeting ILW.
- Adapt research into a draft context / literature review for the final report (see email for lit rev guide).
- Get WFC working with weights by the next meeting.

Week 7 - 24.10.2023

- Try to finish literature and make an introduction to WFC in the report.
- Implement your own WFC.

Week 8 - 31.10.2023

- Continue working on WFC algorithm.

Week 9 - 10.11.2023

- Fix last bugs with MAC.
- Either get in symmetries or give each neighbour data explicitly for now to get generation working properly.
- Lowest entropy.

Week 10 - 17.11.2023

- Implement weighting into the algorithm.
- Have another look at what is meant by entropy.
- Make notes on the directional artefacts (probably a result from overfitting / tileset neighbour bias).

Week 11 - 24.11.2023

- Visual debugger to help with artefacts (diagonal, edge only shapes...)
- Refocus a bit on the game element (texturing, filter, ...)
- Extra: Infinite modifying in blocks, sub-wfc.

C.2 Semester Two Meeting Notes

Week 1 - 17.01.2024

- Goal block
- Have player start in middle of grid with empty blocks around at start of generation.
- Look more into the lighting. Replace placeholder carpet and roof. Check performance of tiling roof.

Week 2 - 24.01.2024

- Main Goals:
  - Continue working towards chunk generation.
    - Let chunks that are separately generated overlap information.
      - Chunks have trouble generating. Maybe the overlap method is not fully correct. Think about overlap size and simply getting information vs delete some part.
      - Infinite modifying in blocks with good chunk sizes might help without backtracking.
    - Allow backtracking for chunks.
  - Script goal to let player win.
    - Detect when the player touches the goal.
    - Show a message and go back to the menu / teleport the player somewhere else.

- Extra:
  - Add in a mini sub-region to test (i.e freezer).
  - Think about making a variant for empty vs empty light.
  - Graphics improvements:
    - Fog / render distance if needed later.
      - Adjusted fog colour.
      - Should anything else be adjusted? For now, no render distance cust. needed.
    - Make vhs effect screen space rather than plane.
      - Seems to be difficult to overlay onto the render texture without more cameras.
    - Skybox lighting for ambient light.
    - Balance light settings and bloom a bit more.
  - Audio improvements:
    - Script the lights to have random offset.
      - Should probably also replace the buzz with something that doesn't tick.
    - Add VHS SFX to the player.

Week 3 - 31.01.2024

- Only discussed progress per email.
- Implement infinite modifying in blocks for next week.

Week 4 - 07.02.2024

- Do report writing.
- Polish game and finish any 90% implemented features.
- ((Test performance of 2D grid vs linear grid. Might be changed automatically by C#. Probably done automatically))

Week 5 - 15.02.2024

- Focussing more on other projects at the moment.
- Tried to do parallelisation, but it wasn't simple. Focus on other things first.
  - Generate the level as the player moves about.
- Continue report writing when able.

Week 6 - 22.02.2024

- Very busy with other courseworks.
- Continue the report in chunks over the break.

Appendix D

Design Document

Outline

- Backrooms-style game using Wave Function Collapse (WFC) to infinitely generate rooms.
- Different tilesets can be used to generate different levels.

Generator

- First generate a fixed size level of a few tiles. Later, generate in chunks (modifying in infinite blocks) to support infinite generation as well as with more tiles.
- Generate geometry for each level. Give "doors" a weighting to go into another sub-level.
- Is it worth integrating some more traditional methods? Traditional generators may work better for rooms. Something more random like WFC might work better for Backrooms.

Modules / Blocks

- Each a 1x1 tile, Empyrion style block with adjacency constraints and weighting (how?) for each.
  - Full
    - Floor (full with carpet on top)
    - Ceiling (full with ceiling on bottom)
    - Could do all three in one block type? Or maybe use texture variations somehow.
  - Wall (half block)
    - Crouch hole (rotated wall)
  - Corner (half)
  - Corner (quarter)

  - Thin Wall (quarter)
  - Corner pillar (half)
  - Corner pillar (quarter)
  - Ceiling tiles
    * A single tile consisting of multiple tiles (2x2?). Above empty, give each sub-tile a chance of of being emissive.
  - Empty
  - Centre pillar
  - Miscellaneous half block variants that fit other normal blocks for variety.

Regions / Structure

- Make a "door" tile that is a floor tile with adjacencies that connect several sublevels.
- Is there a better way of doing this that maintains modularity?
- How might room structures be made while maintaining modularity?
  - Not needed for some areas of the backrooms (i.e. starting area) as they are not well structured.
- Use a density noise map that defines how empty or full an area of the map is. Helps give natural variety.
  - Can also apply noise to lights and grunge for textures.

Goal

- Collect key items across sublevels (and levels?) to unlock new areas / objectives (Monstrum escape route style).

Appendix E

Holiday and Semester Two Task List

- Algorithm
  - Make the algorithm visual (intermediate steps)
  - Give control of generation to the player via keyboard when debugging.
  - Infinite modifying in blocks
    - Think about how this will affect modularity. Will implementing this mean changes to the base algorithm will be harder to make? Or will this just always work on top? It will probably work separately.
  - Sub-WFC
  - Generate block variants for symmetries from the initial block.
- Audio
  - Get ambience sounds
  - Footstep sounds
  - Audio filter
- Game
  - Objectives
    - Spawn items for objectives
    - Spawn key locations among random generation
  - Set up player in Unity to start in the middle of the maze
  - Make a menu screen to start the game.
- Graphics
  - Textures for each block
    - Got a low res wallpaper texture and simple colour for rims. Should help performance.
    - In the future, use AI art for some textures? (Just a possibility)
  - Filters for the camera
    - First round of basic effects. Could probably be enhanced in quality later.

  - Graphics for floor and lights.
    * Have a flat texture for the whole map? Or break it into block chunks too? Performance?
    * Get proper lighting in that is also performant.

- Report
  - Break into sections.
    * Game design, algorithms, discuss different options and things tried, audio and graphics...