**AN INVESTIGATION INTO WAVE FUNCTION COLLAPSE**

CT6007 Individual Research Project

By: Anton Ziegler

S1907905

**Table of Contents**

Introduction ……………………………………………………………………………………...4

Aim and Objectives ...…………………………………………………………………..4

Literature Review ………………………………………………………………………..……..5

What is Wave Function Collapse (WFC)? ............................................................5

How does Wave Function Collapse work? ...........................................................6

Drawbacks of Wave Function Collapse ...……………………………………………7

Where is Wave Function Collapse Now? ..............................................................9

How Does Wave Function Collapse compare to other algorithms? ....................12

Conclusion .....…………………………………………………………………………………14

Output ..………………………………………………………………………………………...14

Glossary ………………………………………………………………………………………..16

References ………………………………………………………………………………….…17

**Table of Figures**

Figure 1: Example of Wave Function Collapse (Gumin, 2019) ...………………….5

Figure 2: A slide from the Game Developers Conference, explaining the method which WFC works (Bucklew, 2019) ...………………………………………………...7

Figure 3: A slide form the Game Developers Conference, showing the issues with tight constraints (Bucklew, 2019) …..…………………………………………...8

Figure 4: A Slide form Game Developers Conference showing the issue of Homogeny (Bucklew, 2019) ...…………………………………………………………8

Figure 5: An in-game image of Bad North (Frushtick, 2018) ...………...………...10

Figure 6: Martin O’Leary’s 2019 implementation of Wave Function Collapse using sentences ..…………………………………………………………………………….11

Figure 7: A Koch curve generated by an L-system (Shaker, Togelius and Nelson, 2016) ..………………………………………………………………………………….12

**Introduction**

In the formative years of game creation, Procedural Content Generation (PCG), which is the automatic creation of content such as levels, was increasingly used with the intention of saving disk space (Scholz, 2019). As compacity increased, the need for PCG started to decrease. Developers also found it gave them a lack of control and preferred not to relinquish themselves to its unpredictable nature (van der Linden, Lopes and Bidarra, 2014). Despite this, PCG has continued to be developed as it still has uses like game replay ability, time or cost savings, or player adaption (van der Linden, *et al.* ,2014). With this continued motivation, a new method called Wave Function Collapse (WFC) was formed by Maxime Gumin and started receiving much attention in 2016 and 2017 (Karth and Smith, 2017). With this method, a small input sample could be used to create large maps all rather quickly and effortlessly (Bucklew, 2019). Combining the small input with a small algorithm allowed for major advantages, quick prototyping and a vast array of diverse content, along with development time and money saved (van der Linden, *et al.* ,2014). The algorithm also allowed for a modicum of control for the developer, meaning they also in turn would be satisfied with its implementation (Kim, H., Lee, S., Lee, H., Hahn, T. and Kang, S).

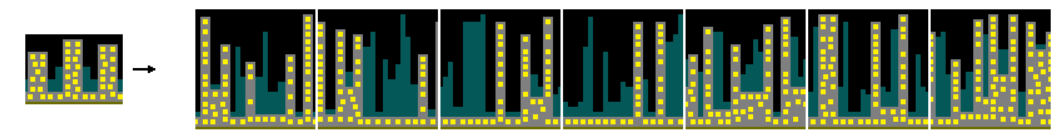
**Aim and Objectives**  
 The aim of this paper is to gain a better understanding of WFC. The paper will explain how the algorithm works, explaining what its inputs are, how its outputs are created. This will also examine the constraint-based nature of an implementation. This paper examines each step that that is used when the program runs and explains some of the various drawbacks that can be faced when using WFC. Along with the drawbacks, it will also examine some ways that the drawbacks can be mitigated when implementing the formula. Some of the implementations of the formula that have been shown with some success will be examined as well. This includes Oscar Stolberg’s use of the formula in 2018’s Bad North, Martin O’Leary’s implementation using the formula to generate poetry from inputs, Breanne Baltaxe-Admony’s version that plays music based on an output created from the algorithm, and Sylvain Lefebvre, Li-Yi Wei and Connelly Barnes version that attempts to hide information in the images. It is also going to exam two other Procedural Content Generation algorithms: Perlin Noise, and L-systems. This will look at the strengths and weaknesses of them, when compared to WFC.

**Literature Review**

**What is Wave Function Collapse (WFC)?**

Wave Function Collapse is a constraint driven algorithm. It takes in a small sample input, generally in 2-D or 3-D and, using a list of constraints, outputs a much larger version of the input (Karth and Smith, 2017 ; Scholz, 2019). The algorithm was created by Maxim Gumin and gained notoriety in 2016 from some hobby and indie game developers, the goal being to create an image where none of the constraints are violated when the image is complete (Karth and Smith, 2017). This works by using adjacency constraints (Scholz, 2019). By taking an image, and creating smaller snippets of the larger piece, you can create an image that matches, by specifying how the snippets all fit together, as long as all edges can in some way connect with each other.

In figure 1, we can see how inputting a small version of a building, and allowing everything that connects to populate out, we can end up creating a sort of skyline effect. The effect takes in small image, and can be used to create many, larger scale versions of the sample image. This still matches the constraints set out by the developer when creating both the sample input and an expected output. This does not allow for any blending, which also maintains the original image (Karth and Smith, 2017). Without the blending the output resembles the original input however it appears to have grown organically.



**Figure 1:** Example of Wave Function Collapse (Gumin, 2019).

**How does Wave Function Collapse work?**

The original implementation that Maxime Gumin created for Wave Function Collapse was written in C# and totalled less then 1,000 lines of code (Gumin, 2019). This is rather simple to get up and implement in a project, and is easily understood, what each step in the program is completing. There are generally 5 steps that the algorithm goes through to complete.

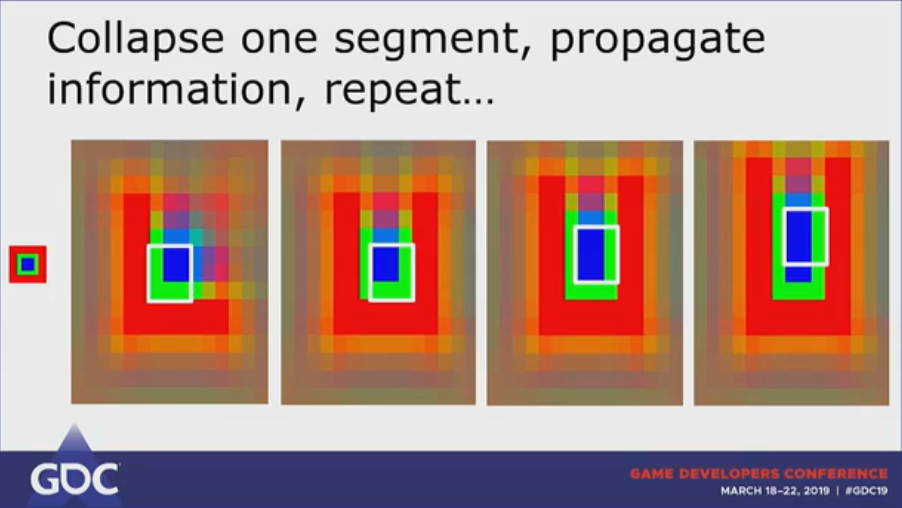
The algorithm begins by taking the input and splitting it up into smaller chunks of a determined size N (Bucklew, 2019). These chunks sized NxN are what helps calculate the constraints of the project, as it allows it to figure out which chunks can fit together. Each chunk is given a constraint for each side. Stalberg showed, in his 2018 talk an implementation that uses colour, determined at run time, to match each side. (Stalberg, 2018) This is one implementation, constraints can de given prior to running, or can be left to the program to decide.

The next step is to fill in the whole map with every possible chunk (Bucklew, 2019). This is simply allowing all things to be unconstrained. Prior to putting in any constraints into any part of the map all possible permutations are available, before adding any further constraints. It gives an overview of how any and all possible version will look, before beginning the process of filling in each spot.

It then begins by selecting the smallest N sized area of the map, at random from the output and is filled in (Bucklew, 2019). The program selects an area, and then puts a small chunk in, anywhere it wants. At this point the program has no constraints of any kind, and as such, it is free to do whatever it wants with the first piece it places, before putting down any constraints. This first piece is selected randomly, as there is no difference between any of the chunks, meaning any selection will do.

The program then propagates to the surrounding area and removes possibilities that don’t fit the constraints (Bucklew, 2019). This is where the constraints now come in. Once a chunk has been placed in, the program can now tell each surrounding chunk what options are still available to it, and those now know what options can be placed in. There constraints affect each surrounding piece, letting them know what has been placed down before it.

The program then begins anew. It skips the division of chunks, selecting a new place to put a chunk, however now there are some constraints in place, it only places chunks that would fit the available constraints. In figure 2 the input image is shown on the left. The program has begun to fill in some of the chunks of the output, which are denoted by a single solid colour, as they have now been constraint. The white outline is the area that the algorithm has decided to fill in. Around the white area, where the colour is not solid, it shows all the possibilities that can still be placed, based on the current constraints.



**Figure 2:** A slide from the Game Developers Conference, explaining the method which WFC works (Bucklew, 2019).

**Drawbacks of Wave Function Collapse.**

WFC isn’t without a couple drawbacks. Notably, if the input image is too complex, it tends towards completing the same image repeatedly. In figure 3 we can see a star input and the output images, which is simply the star repeated. This is due to the constraints being too tight, only one possibility exists at any point while it fills in, making it propagate the same, with little difference. This is an issue of simply having too complex an input without enough options for it to make different formulations. However, it can be fixed, by either making the chunks larger, or by making the input smaller.

A screenshot of a cell phone

Description automatically generated

**Figure 3:** A slide form the Game Developers Conference, showing the issues with tight constraints (Bucklew, 2019).

WFC also has a habit of being homogeneous. In figure 4 we can see a simple input, with a large output area, this output has a side effect of densely packed small structures, and no larger ones, to break up the similarity, causing it to be busy, or noisy. The program doesn’t create new structures, only expanding upon the ones there. This arises when the output area is too large, and not enough input has been given. This can be mitigated by breaking up the area with different fills from WFC. Split the output into different parts and give the program different inputs for each individual area.

A close up of a sign

Description automatically generated

**Figure 4:** A Slide form Game Developers Conference showing the issue of Homogeny (Bucklew, 2019).

The final issue that the formula encounters is that there is a possibility that, at any point, a chunk of the program may end up with constraints where no single possible option can conceivably fill it (Stalberg, 2018). In this case, it is very difficult for the formula to do much, other than restart as it is not a recursive formula, hence it has no idea what steps to undo that caused the failure in the constraints. This can be mitigated by creating a search function (Kim *et al.*) If a chunk would be added that would cause a failure state, so that the image was no longer completable, it tries to place a different chunk that fits the constraints, which can stop the failure from occurring. Adding a function like this can be time consuming, as it does need to check that at least 1 piece can go everywhere and this does not completely rectify the issue; however, it does aid in mitigating a failure state, or at least creates a scenario where such a state is difficult to reach, by attempting to pre-empt it, which can be worth the extra expenditure of resources.

**Where is Wave Function Collapse Now?**

Wave Function Collapse has many spin offs and forks from the original project (Gumin, 2019). In total There are at least 50 known implementations that various people have used for different things, which includes uses in Indie Games, AAA games, music production, poetry, and others. (Gumin, 2019). It has been used in at least one indie developed game, Bad North by Oscar Stalberg, released in 2018. In Bad North, Stalberg runs the algorithm at game run time to generate each level uniquely as the player is joining in to play (Stalberg, 2018). In Figure 5 we can see a map that was created in game, using a 3-D implementation of the formula. It creates elevations and is even used to create things like the vegetation growing around the in-game area (Stalberg, 2018). This demonstrates how multiple uses of the formula, can lead to a more natural effect, such as things like vegetation growth. This implementation also includes heuristics that allow the map to be navigable (Gumin, 2019). This is a form of constraint that has been added in, causing certain areas to be required for the player to be traversable, or there would be no way for the game to be multi-tiered. It would have simply created a flat playing area, and no elevations, as such some form of tiering had to be added, and by adding in a simple constraint requiring sometimes the tiers be traversable adds a whole other dimension to the game. This allows for greater level diversity, as now levels can be more then one tier, and also adds in game dimensions, with players being forced to make choices based on the heights.

**Figure 5**: An in-game image of Bad North (Frushtick, 2018).

Other implementations of this algorithm can be seen in Martin O’Leary’s version which takes sentences and creates poems from it (Gumin, 2019). It creates this using sentence fragments and mashing them together to produce poems with fixed metric forms (O’Leary, 2019). The implementation represents a taking a tile sheet, that is single dimensional, as opposed to the 2-D NxN implementation previously described in this report. Your input is the text of a sentence, or words from a book. It is then treated as any other implementation, and the output is based on syllables of the words or can also be the number of words per line. In Figure 6, we can see an incomplete verse, with how many words will be placed in each spot, and how it is filling in each sentence. The start of the first line, and the second half of the fourth last line are from the same point, and show the same thing, a natural repetition. These repetitions are things people enjoy seeing, which makes some of the visual of this happening so appealing (Karth and Smith, 2017). This will fill out to a complete text and is poetry. This can also be used to create limericks, a version of poetry based upon syllables. The input would be similar, however it would need to include the number of syllables of each word, to facilitate the output being coherent.

A close up of text on a white background

Description automatically generated

**Figure 6:** Martin O’Leary’s 2019 implementation of Wave Function Collapse using sentences.

WFC has also been applied to music generation (Gumin, 2019). In this example, WFC is used to produce a set of sample images, which are then played as music (Baltaxe-Admony, 2019). This is derived from the colours and patterns that are found in the output image based on the other colours, interposed form the background. This method also allows the user to change the output method, based on scale, harmonic or major (Baltaxe-Admony, 2019). This allows more control over the output and gives some variation from even the same input image, allowing it to be used multiple times. In this instance adding in a music player to synthesize music from a tiled image shows some of the breadth of possibilities from the algorithm, and its possible other uses.

Sylvain Lefebvre, Li-Yi Wei and Connelly Barnes have also examined adding coded messages to tile sets, specifically adding them to WFC (Lefebvre *et al.*, 2018). These messages are encoded by created by constraints added to the colours of a pattern and fill in the surrounding area (Lefebvre *et al.*, 2018). The choice of each constraint is determined by a secret key (Lefebvre *et al.*, 2018). With the key and input it is possible to determine how each constraint is made, determining information, such as true or false. This can be used to send messages or other information that can be placed in the tiles as they are created. These messages could be decoded or used for various things, to pass information from system to system. One possible thing is to create Quick Response (QR) codes, in the tile set (Gumin, 2019).

**How Does Wave Function Collapse compare to other algorithms?**

WFC is not the only algorithm that can be used to create Procedurally Generated Content. Some of the other algorithms, including but not limited to, are L-systems (Shaker, Togelius and Nelson, 2016), and Perlin Noise (Rose and Bakaoukas,2016). Both these algorithms have their distinct strengths and weaknesses, in comparison to WFC, and can be examined on what they can be used for. This section aims to have brief overviews of what each algorithm is good for, and how that changes compared to WFC.

A screenshot of a cell phone

Description automatically generated

**Figure 7**: A Koch curve generated by an L-system (Shaker, Togelius and Nelson, 2016).

L-systems are a grammar-based algorithm, meaning that by input a string which takes a left and right-side value, by changing the left value by the right value every time it comes across it, it creates a new form (Shaker *et al*., 2016). What this means is, if you have a value of A on the left, and AB on the right, every time you run it, it will provide AB, which then leads to ABB, and on. In Figure 7 we can see an implementation of the algorithm to create a spiralling effect. The major issues this algorithm faces are that it follows a drawable line (Shaker *et al.*, 2016). The output that is created must be done in one continuous piece, and while it can be stepped back upon with instructions which can be used to create blank spaces, it is still going to be some form of line, as even a blank space follows a consistent structure when traversing. This means it is good for creating things like vegetation or missions (Shaker *et al.*, 2016). Vegetation simply needs to appear to be slightly different from each other while still creating a lot of them, to fill in the area. Like WFC, this algorithm gives some control back to the developers (van der Linden, *et al.*, 2014). A change in the variables will be reflected in the output, and it isn’t all up the algorithm with how the output will look. L-systems, as they are a grammar system, rely heavily on hard coding (van der Linden, *et al.*, 2014). In this instance it can be very difficult to change rules or parameters that could impact the results, especially as the system becomes more complex. This is the major difference from WFC in that, once WFC is set up, it is much easier to change the output each time.

Noise-based Procedural content was created to produce terrain content, which can be difficult to do, as terrain is not a natural Euclidian shape, they are curved, and not straight lines (Rose and Bakaoukas, 2016). As terrain is not usually things like triangles and squares, it can be harder for a computer to generate because these shapes do not lend themselves to natural looking areas, which are of a more random nature. Terrain is in fact fractal in nature (Rose and Bakaoukas, 2016). This means that the forms are similar in type, yet different in size and forms. Perlin Noise was created to solve this issue. It creates a gradient value and applies it to a map at random to create slopes, which appear to be natural (Shaker *et al.*, 2016). This method does not pick points on a map, to go up or down, and then fill in information between. It instead does the opposite, taking the slopes in-between and applying heights or dips to it from there. The issue with this, however, is that the map can become distorted (Kass and Pesare, 2011). As you move around the map to create content, areas can be passed over a couple of times, creating very noisy areas. These areas do not process well, and generally end up with jagged distortions. This is one of the key issues that Perlin has, it is a sort of blending between areas, and can result in maps that don’t feel natural, but are much more mechanical in nature. This is not an issue WFC faces. Blending is not a possibility in content generated from WFC. The trade-off is that memory requirements for Perlin Noise are low (Rose and Bakaoukas, 2016). No matter the size of the content being generated from the program, Perlin Noise will always be extremely lightweight to run, a feature that WFC does not have, as the larger the generated area is, the more information from all the constraints it must keep track of when it is producing content.

**Conclusion**

To conclude the research, the evidence gives a better understanding of the algorithm and various parts of it. There is a dive into how a traditional implementation of the algorithm going through each of the different steps that it takes to run through one complete pass through, and create an output that is similar in nature, yet different in design from the input. It also shows some of the various issues that can be faced when implementing WFC, from complexity to homogeny and the possibility that it will fail, while providing possible fixes for each of the issues that can crop up. Along with this was how some developers have chosen to implement WFC, from its use to hiding information in pictures by knowing all the key information, by playing music from the output created from the formula, poetry that can be created from an input, and a game being created and released using it. Finally, we looked at two other procedural content generation techniques, in Perlin Noise, and L-systems, and how they compare to WFC. Each of these algorithms have their advantages, and their disadvantages and looking at them gives us a better view of WFC, where it fits in, and what it is good for.

**Output**

For demonstrating the way that Wave Function Collapse is implemented, and to test the viability of the how to solve the failure rate potential issue. The algorithm will be implemented using the Unity Engine, version 2019.2.10f1 (Unity, 2019). This implementation will be able to take sample input images, and create expanded upon output images, that can be used for 2D level creation. Unity has been chosen as it is readily available, and its prototyping capabilities. It also includes some key elements to facilitate the building of the output, the ease of importing and exporting images, so that they can be accessed more readily, and speed of testing, where it is possible to view changes rapidly when testing. This culminates in an easy to use and explain platform to facilitate the algorithm’s implementation.

When implementing the algorithm there are several issues that have been noted, and this implementation will attempt to address an issue that has been shown, by implementing the given fix that has been proposed, to asses it’s viability. The goal is to test and see the search functionality that was proposed by Kim et al, as a fix to the possibly of a failure state (Kim *et al.*). This implementation is to gauge the viability, whether the trade off increasing the taxation on the system, is worth the decrease in failure rate, and being forced to restart anew. Determining this may require implementing the algorithm twice, one without the functionality, to view the differences between both versions.

For testing the viability of this implementation, this will be tested on a Lenovo Legion y520 notebook, with an Intel Core i5-7300HQ CPU, 16 GB RAM and a dedicated NVIDIA GeForce GTX 1050ti GPU, running Windows 10 (64-bit). The output expects to have some decrease in performance over Scholz (2019) implementation, as the Search function will be taxing on the system, however it remains to be seen whether the drop is great enough to not warrant the implementation.

**Glossary**

Euclidian – a shape that is not mechanical looking, generally has curves.

Fractal – A repeating chaotic nature or pattern, not mechanical looking.

Fork – Where someone creates a version of a program that deviates from the original.

Hard coding – coding that puts information directly in, rather then using a variable, to allow simple changes.

L-system – An algorithm that follows grammar rules, such as A -> B results in every pass turning A into B.

Perlin Noise – A noise-based algorithm created by Ken Perlin.

Procedural Content Generation – Algorithms to create content by creating code to generate the content, instead of designing it directly.

QR code – A Quick Response code that can be read by a device, similarly to a bar code.

Wave Function Collapse – A constraint-based content generation method, devised by Maxime Gumin.

**References**

Baltaxe-Admony, B. (2019). *bbaltaxe/wfc-piano-roll*. [online] GitHub. Available at: https://github.com/bbaltaxe/wfc-piano-roll [Accessed 5 Nov. 2019].

Bucklew, B. (2019). Math for Game Developers: Tile-Based Map Generation using Wave Function Collapse in 'Caves of Qud. In: *GDC*. [online] Available at: https://www.youtube.com/watch?v=t1O0\_yHe-6Y [Accessed 11 Oct. 2019].

Cook, M., Colton, S., Gow, J. and Smith, G., General Analytical Techniques For Parameter-Based Procedural Content Generators.

Frushtick, R. (2018). *Bad North is the chillest RTS I’ve ever played*. [online] Polygon. Available at: https://www.polygon.com/2018/8/21/17764170/bad-north-impressions-nintendo-switch-rts-strategy [Accessed 11 Nov. 2019].

Gumin, M. (2019). *mxgmn/WaveFunctionCollapse*. [online] GitHub. Available at: https://github.com/mxgmn/WaveFunctionCollapse [Accessed 12 Oct. 2019].

Karth, I. and Smith, A. (2017). WaveFunctionCollapse is constraint solving in the wild. *Proceedings of the International Conference on the Foundations of Digital Games - FDG '17*.

Kass, M. and Pesare, D. (2011). Coherent noise for non-photorealistic rendering. *ACM Transactions on Graphics*, 30(4), p.1.

Kim, H., Lee, S., Lee, H., Hahn, T. and Kang, S., Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm.

Lefebvre, S., Wei, L. and Barnes, C. (2019). *Informational Texture Synthesis*. [online] Hal.archives-ouvertes.fr. Available at: https://hal.archives-ouvertes.fr/hal-01706539/ [Accessed 5 Nov. 2019].

Lorensen, W. and Cline, H. (1987). Marching cubes: A high resolution 3D surface construction algorithm. *Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH '87*.

Lucas, S. and Volz, V. (2019). Tile pattern KL-divergence for analysing and evolving game levels. *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '19*.

O'Leary, M. (2019). *mewo2/oisin*. [online] GitHub. Available at: https://github.com/mewo2/oisin [Accessed 5 Nov. 2019].

Rose, T. and Bakaoukas, A. (2016). Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques. *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*.

Scholz, D., 2019. *Tile-Based Procedural Terrain Generation* (Doctoral dissertation, Technische Universität Wien).

Shaker, N., Togelius, J. and Nelson, M. (2016). *Procedural Content Generation in Games*. Cham: Springer International Publishing.

Stalberg, O. (2018). Wave Function Collapse in Bad North. In: *Everything Procedural Conference*. [online] Available at: https://www.youtube.com/watch?v=0bcZb-SsnrA [Accessed 11 Oct. 2019].

Unity Technologies, 2019. *Unity*, Available at: https://hadoop.apache.org.

van der Linden, R., Lopes, R. and Bidarra, R. (2014). Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, [online] 6(1), pp.78-89. Available at: http://ieeexplore.ieee.org.glos.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=6661386&isnumber=6766678 [Accessed 11 Oct. 2019].