The Procedural Generation of City Landscapes

Connor Easterbrook
18015101

University of the West of England

March 15, 2023

Abstract

Procedural generation is an autonomous tool that is vital for the efficient production of a unique expanse, at the cost of manual design. Using the grid-based Wave Function Collapse algorithm, this project presents a procedural city generator that is fit for external use. The calculations run through a four-dimensional boolean array to check each grid slot's possibility, with an approach to allow for infinite generation. The performance and usability are evaluated through a discussion, with concluding thoughts on areas of improvement.

1 Introduction

1.1 Procedural content generation

Modelling a city can be arduous because of every consideration that needs to be considered. Every urban area contains signs it is often frequented by a large population. Garbage, road signs, public amenities, and public transportation are just four considerations of many (Parish and Müller, 2001). Kelly and McCabe, 2006 explains that, as technology improves, time spent on graphics increases, which lengthens production time. A grid-based procedural generation of a city landscape removes the effort required to place every consideration individually (Seidel, Berente, and Gibbs, 2019). It does this by using modules that contain models and places them to fill the grid slots.

1.2 Wave Function Collapse

The procedural wave function collapse algorithm is a method for generating patterns or structures that exhibit certain statistical properties, such as self-similarity or long-range correlation (Namiki and Pascazio, 1991). According to Bassi et al., 2013, it is based on the concept of wave function collapse in quantum mechanics, which describes the phenomenon of a quantum state "collapsing" into a definite state when it is measured or observed.

There is no single formula that describes the procedural wave function collapse algorithm in its entirety, as the algorithm can be implemented in a variety of ways depending on

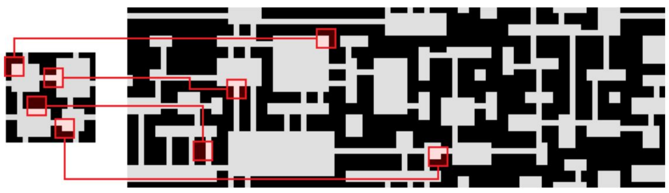


Figure 1: *Example of how the Wave Function Collapse algorithm works. Input shown on the left and output shown on the right. Similarities are highlighted. Gumin, 2016.*

the specific goals and constraints of the application. However, some common steps that are often included in the implementation of the algorithm include:

1. Initialization: Initialize the wave function to a desired starting state.
2. Iteration: Iteratively update the wave function according to a set of rules or constraints. These rules or constraints can be chosen to mimic the behavior of wave function collapse in quantum mechanics.
3. Observation: Make an observation of the system and update the wave function accordingly.
4. Repeat: Repeat the iteration and observation steps until the desired level of accuracy or convergence is reached.

In short, the procedural wave function collapse algortithm utilizes neighbour-based rules so that only the desired modules can be placed next to other modules, based on its entropy (Bassi et al., 2013). This can be seen in Figure 1. Kleineberg, 2019 details his method for this based on the procedural wave function collapse algorithm by Gumin, 2016. Concerns arose at the beginning of the project about undesirable generation stemming from missing neighbour rules which was solved through module-based constraints. This project was conducted in the Unity game engine as it provides many tools that assist in both user interactivity implementation and the creation and visualization of algorithms.

1.3 Related work

Kleineberg, 2019 produced a project featuring wave function collapse within Unity but handling it in a way that dif-

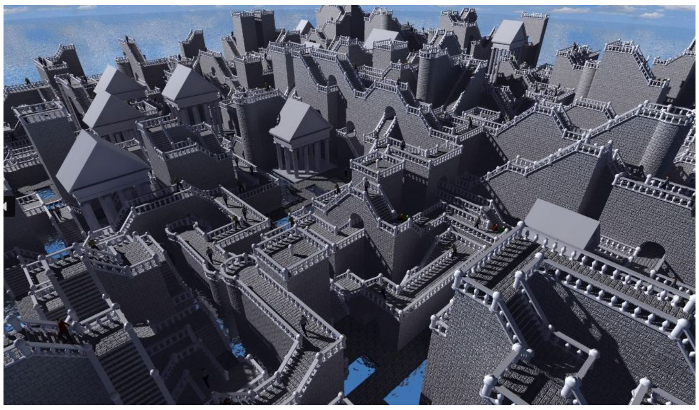


Figure 2: *Example of Paul Merrel's Model Synthesis that closely relates to Wave Function Collapse. Used as the result is very similar to Kleineberg's results but with dedicated empty areas.*

ferentiates from this project's implementation. Merrell and Manocha, 2010 published a method that is very similar to the Wave Function Collapse algorithm by Gumin, 2016 but focuses on three-dimensional objects. It can be seen in Figure 2.

1.4 Rationale

The potential applications for procedural generation of city landscapes can range from education in creating environments to entertainment and simulation (Møller and Billeskov, 2019). *Cities: Skylines* 2015, while not procedural, showcases the possible benefits from city-focused interactive software. González-Medina, Rodríguez-Ruiz, and García-Varea, 2016 use procedural city generation in robotics, while Stålberg, 2015 created a casual video game.

2 Methodology

As the Wave Function Collapse algorithm was developed intending to be used in two-dimensional space, several changes were necessary to make the algorithm architecture fit for this project, while keeping the core algorithm the same. This was done so that the algorithm remains true to nature while allowing for greater customization in procedural generation. An example of this would be the possible changing of vertical offset, simplifying the addition of decorations outside of the grid-slot module, as seen in Figure 3.

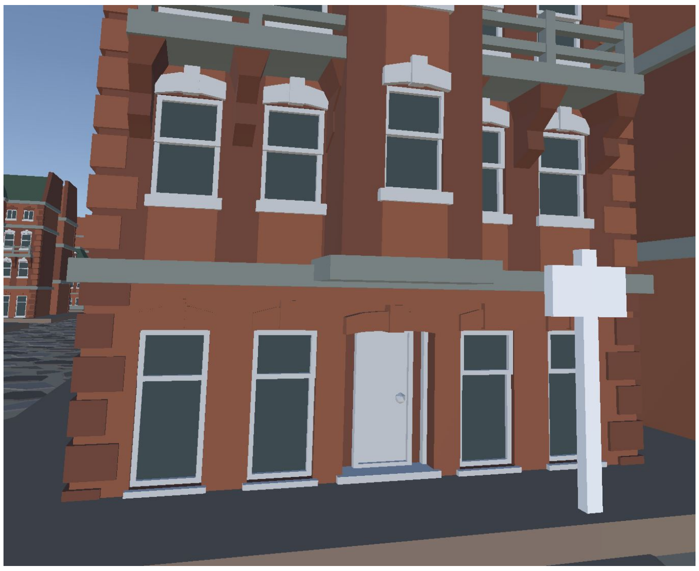


Figure 3: *Untextured sign in front of house. An Early example of using the y-Axis for implementing decorations.*

The Unity Wave Function Collapse implementation for the Unity game engine can be split into three subsections - Grid, Module, and Infinite Generation. 'Grid' specifies the creation of a map of tiles in which the modules specified in 'Module' fill in each vacant tile slot based on inputted constraints. 'Infinite Generation' details how this implementation was then expanded upon to proceed infinitely.

2.1 Grid

The grid is an essential part of the Wave Function collapse algorithm as it contains the tiles that the algorithm modules will fill following inputted constraints (Gumin, 2016). The grid must be generated first in order for the modules to be correctly placed based on the slot entropy. As a three-dimensional Wave Function Collapse algorithm was desired, the grid receives an input of width X, length Z, and height Y integers alongside a float that represents the size of each tile T to create the grid G.

$$G = (X + T) * (Y + T) * (Z + T) \tag{1}$$

A four-dimensional boolean array is then used to store the size of the grid, each grid tile face for directions, and each inputted module twice for comparison. This is done to keep all information organized and accessible.

```
private void Initialize(GameObject generator) {
    ...
    generation = new bool[gridWidth * gridLength *
    gridHeight, 6, generationModules.Count,
    generationModules.Count];
    ...
}
```

It is this generation boolean array that initializes the Wave Function Collapse algorithm and allows for all subsequent calculations to take place. Whenever grid tiles need to be specified, three integers increment through loops that represent each dimension. This is repeated until the three integers match the grid size, covering every possible tile.

2.2 Modules

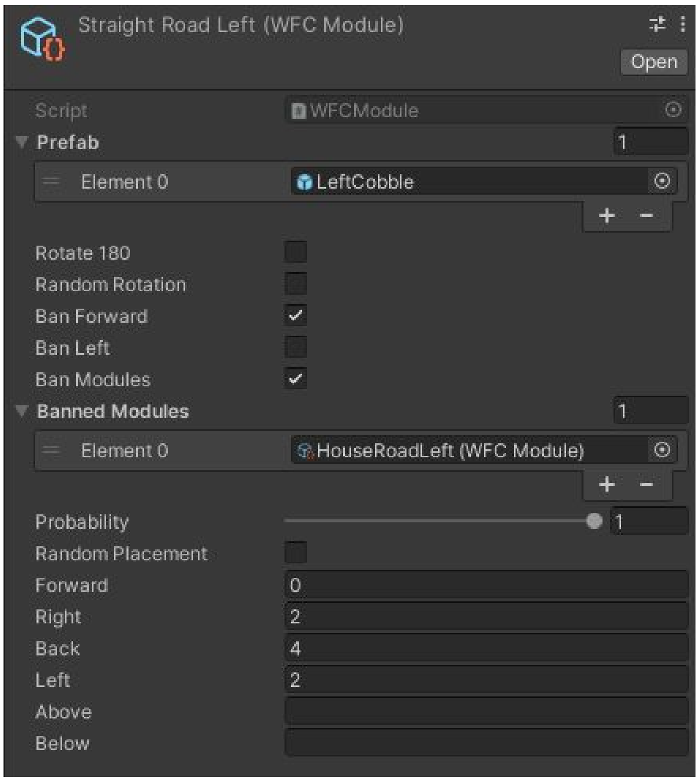
"Modules" refer to the scriptable object used within the game engine to contain the spawn data of each grid tile. Each module within the Wave Function Collapse implementation features its own rules that dictate how they are generated onto the grid (Kleineberg, 2019). The aforementioned changes made allow rules to be easily implemented and modified. These rules include, but are not limited to, banning adjacent duplicates, forcing module rotation, adding module prefab variety, and altering the module spawn probability. These rules lower the frequency of undesired generation results.

Once the grid is generated, the modules are then instantiated. This is completed by iterating through each grid tile, through each module, through each direction, and through each neighbour tile module respectively (Kelly and McCabe, 2006). Each iteration contains a boolean for whether this generation combination is possible. If it is possible then increase the entropy of the tile. This means that initially all grid tiles will have the maximum entropy by default.

To avoid a freeze from the algorithm being unsure which tile to select if there are matching lowest entropies it selects one at random. As shown in the code below, this is done by running through each tile and comparing its entropy with a randomized number below one. This solution was inspired by the implementation by Kleineberg, 2019.

```
private int NextSlot()
{
    int result = -1;
    float min = float.MaxValue;
    for (int i = 0; i < generation.GetLength(0); i++)
    {
        int entropy = this.entropy[i];
        if (entropy > 1 && entropy <= min)
        {
            float rng = Random.Range(0f, 1f);
            if (entropy + rng < min)
            {
                min = entropy + rng;
                result = i;
            }
        }
    }
    return result;
}
```

Upon selection, it is analyzed for possible modules and has one randomly selected taking probability into account. Upon module instantiation, a prefab is randomly selected from the set array of GameObjects and instantiation rules are utilized. A preview of the module inspector screen can be seen in Figure 4.


Straight Road Left (WFC Module)
Open

Script	WFCModule
Prefab	1
Element 0	LeftCobble
Rotate 180	☐
Random Rotation	☐
Ban Forward	✓
Ban Left	☐
Ban Modules	✓
Banned Modules	1
Element 0	HouseRoadLeft (WFC Module)
Probability	1
Random Placement	☐
Forward	0
Right	2
Back	4
Left	2
Above	
Below	

Figure 4: *Showcase of module rules. An early example of using a scriptable object to contain module data.*

2.3 Infinite generation

The infinite generation in this project relies on a chunk system as it was the most efficient means to generate neighbouring grid maps (Mawhorter and Mateas, 2010). Each chunk and each of the chunk's slots are given its own class to store its respective data.

The Wave Function Collapse algorithm checks each neighboring chunk and its neighbouring slots to ensure consistency in the generated world. This approach allows for an infinitely generated world with consistent and predictable features. The mathematics for this would be:

$$\forall i,j \in \mathbb{Z}, \mathrm{A}i,j = f(\mathrm{A}i-1,j^{(\mathrm{R})}, \mathrm{A}i+1,j^{(\mathrm{L})}, \mathrm{A}i,j-1^{(\mathrm{D})}, \mathrm{A}^{(\mathrm{U})}_{i,j+1}) \tag{2}$$

where $\mathrm{A}_{i,j}$ represents the generated chunk at position (i,j), f is a function that takes the existing neighboring chunks (indicated by (L), (R), (U), and (D) for left, right, up, and down, respectively) as input and returns a new generated chunk at position (i,j).

3 Evaluation

Overall this project is a standalone tool that generates an endless cityscape. Using procedural techniques and the wave function collapse algorithm is noteworthy, as it allows for the generation to be coherent and set to follow specific rules. However, the usability and performance of the project may require careful consideration and optimization, depending on the user's application. It was intended that each chunk have connected, coherent edges but unfortunately this implementation was unable to be achieved.

3.1 Usability

While the specification for this project was to create a procedural city generator, the intention was to create a tool that can be used for various types of generation. One generation usage would be being able to generate populated areas fitting a set image, an example of this would be generating a medieval village. Regarding this, there is more work to be done as two-dimensional height maps would be required in order to create more realistic population density drop-offs.

3.2 Performance

The performance of the generator is not ideal. Delaying generation bursts by one-hundred milliseconds allowed for generations to run more smoothly but instantiating game objects in Unity will always be resource intensive. The performance could be improved through threading and relying on mesh frames instead of game objects.

4 Conclusion

The system produced yields a good solution to the task specifications. A mesh is procedurally destroyed upon request and the user is able to explore this in various ways. A form of dismemberment has been implemented within the project to allow for limbs to also be procedurally destroyed, allowing for more variety in a gameplay experience. The system has been multi-threaded so that it functions without being overly resource-intensive and unobtrusive to the user's experience. The final result of this project forms a simple library that allows for easy implementation of mesh destruction in any Unity Game Engine project. Further development could be done through complicating the event portion of the mesh destruction calculations, allowing for more realistic results through a more realistic destruction path. An example of this would be utilizing a Lindenmayer system to generate unique destruction paths. In conclusion, it is believed that this project achieves its goal of being a procedural mesh destruction system in an appropriate manner, but there is room for improvement.

Bibliography

Bassi, Angelo et al. (2013). "Models of wave-function collapse, underlying theories, and experimental tests". In: *Reviews of Modern Physics* 85.2, p. 471.

Cities: Skylines (2015). Steam. Colossal Order.

González-Medina, Daniel, Luis Rodríguez-Ruiz, and Ismael García-Varea (2016). "Procedural city generation for robotic simulation". In: *Robot 2015: Second Iberian Robotics Conference*. Springer, pp. 707–719.

Gumin, Maxim (2016). *Mxgmn/Wavefunctioncollapse: Bitmap amp; tilemap generation from a single example with the help of ideas from Quantum Mechanics*. URL: `https://github.com/mxgmn/WaveFunctionCollapse`.

Kelly, George and Hugh McCabe (2006). "A Survey of Procedural Techniques for City Generation". In: *The ITB Journal* 7.2, p. 5.

Kleineberg, Marian (2019). *Infinite procedurally generated city with the wave function collapse algorithm*. URL: `https://marian42.de/article/wfc/`.

Mawhorter, Peter and Michael Mateas (2010). "Procedural level generation using occupancy-regulated extension". In: *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, pp. 351–358.

Merrell, Paul and Dinesh Manocha (2010). "Model synthesis: A general procedural modeling algorithm". In: *IEEE transactions on visualization and computer graphics* 17.6, pp. 715–728.

Møller, Tobias and Jonas Billeskov (May 2019). "Expanding Wave Function Collapse with Growing Grids for Procedural Content Generation." PhD thesis. DOI: `10.13140/RG.2.2.23494.01607`.

Namiki, Mikio and Saverio Pascazio (1991). "Wave-function collapse by measurement and its simulation". In: *Physical Review A* 44.1, p. 39.

Parish, Yoav IH and Pascal Müller (2001). "Procedural modeling of cities". In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 301–308.

Seidel, Stefan, Nicholas Berente, and John Gibbs (2019). "Designing with autonomous tools: Video games, procedural generation, and creativity". In.

Stålberg, Oskar (2015). *Cities: Skylines*. Steam.

Appendix A

Week 1 - `https://youtu.be/GASh1r6GsJc`
Week 2 - `https://youtu.be/zq4cbIbO7mg`
Week 3 - `https://youtu.be/Oxt7cvDUuXw`
Week 4 - `https://youtu.be/_hcO0cilfcs`
Week 5 - `https://youtu.be/MbXm7WIW2do`