CSC111 Project 2

Wave Function Collapse

Stefan Barna

April 1, 2024

1 Introduction

There exists an ever-growing demand for engaging and scalable content, particularly in the realm of dynamic and interactive software such as simulations and video games. The traditional means for generating this content, particularly for applications that involve randomness, is tedious and may struggle to meet complex specifications. As consequence, developers regularly use procedural content generation (PCG) techniques in producing vast and varied outputs that do not demand attention to every detail.

In general, a PCG system is one that takes in one or more design patterns and produces new ones. This often involves principles of evolutionary search, noise, and rule-based systems. Not only does PCG streamline content creation processes, but it also provides opportunities to generate dynamic content adaptable to presets and preferences.

Recent developments in PCG algorithms have enabled us to introduce artificial intelligence and machine learning to further enhance generated content. However, in the interest of both time and sanity, it is perhaps wise to avoid these advancements, and instead focus our discussion on the content generation applying a more accessible branch. One such domain is constraint-based programming. In fact, to reduce the problem further, we shall consider the processing of *images* in particular. With this in mind, we ask

> **Given an input image, how might we use procedural content generation techniques to output similar larger images?**

The approach prominent in the game development community, and the one we shall focus our efforts on, is the Wave Function Collapse (WFC) algorithm, developed by Maxim Gumin [1].

1.1 Wave Function Collapse

The WFC algorithm has been used for content generation in the games Bad North, Caves of Qud, and Townscaper, and is commonly applied for textures, images, and $n-$dimensional models among these titles and smaller ones [2]. Strictly speaking, is classified as a constraint-based greedy algorithm generating complex patterns from one or more input seeds [3]. Given the scope of image generation, the algorithm generates bitmaps locally similar to the input, meaning that

1. every N x N tile within the output image appears at least once within the input image; and
2. the distribution of N x N tile patterns in the input should match the distribution of N x N tile patterns within the output images over sufficiently many outputs.

The algorithm is inspired by the titular wave function collapse, a phenomenon in quantum mechanics induced by the observation of a wave function [4]. The wave function in the algorithm refers to the set of all possible states, or patterns, each pixel in the output image may collapse to, where the collapse itself occurs during the iterative process of selecting a particular pattern for each pixel.

2 Procedure

The Wave Function Collapse algorithm begins by identifying the $N \times N$ patterns, or tiles as we shall call them from here forth, present in the input seed. For every pixel p in the input, there is an $N \times N$ tile extracted whose top-left pixel is p, so that there are ultimately as many extracted tiles as there are pixels in the input image. This includes tiles that extend past the bottom or right border, wrapping around to the other end of the image.

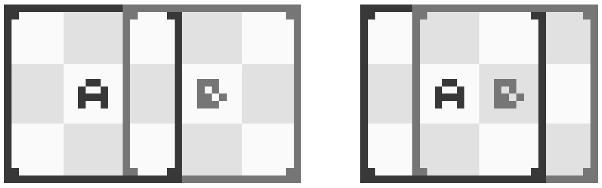

Figure 1: Overlapping tiles in the input image. Tiles A and B are not marked as adjacent in the left example, but are in the right, where A is *left*-adjacent to B and B is *right*-adjacent to A.

After extraction, every distinct tile is assigned an identifier. The number of occurrences of each tile in the input is then stored to represent the expected tile frequencies in the output, and thus adhere to the second rule of local similarity. Neighbouring tiles in the input are then identified as *adjacent* in a direction. In general, we say two $N \times N$ tiles are adjacent if they share precisely an $N \times (N-1)$ region of pixels in the input seed, as seen in Figure 1. This adjacency relationship is antisymmetric. If A is up-adjacent to B, B is down-adjacent to A. If A is left-adjacent to B, B is right-adjacent to A.

Similar to the method by which we extract tiles from the input image, the output image represents tiles by their top-left pixel. With this in mind, notice that any two adjacent tiles may be placed alongside one another in the output image while adhering to the first rule of local similarity. In particular, if B is right-adjacent to A, the top-left pixel of B will be immediately to the right of the top-left pixel of A in the input image. This compounds into the maintaining of local similarity, as we may consider then a tile C right-adjacent to B, so that *it's* top-left pixel is two pixels to the right of the top-left pixel of A in the input image and so forth. But because A and B overlap in $N-1$ pixels horizontally, we may continue this process to produce an $N-$row of pixels that appear in the input image, as in Figure 2. Of course, this works also in the vertical direction.

After all data is collected from the input seed, the output image, represented as a "wave" (grid), is initialized in an entirely unobserved state, meaning that each cell is in a superposition of all extracted tiles. That is, each cell may eventually become any of the tiles extracted. In general, a tile is included in the superposition of a cell c when it is left-adjacent to at least one tile included in the superposition of the right-neighbour to c, right-adjacent to at least one tile included in the superposition of the left-neighbour to c, and so forth. We say that a cell is collapsed when it is in a definite state, meaning that it is fixed to precisely one tile.

At this point, the algorithm iterates over the following steps until all cells are collapsed

1. Collapse the cell c with lowest nonzero Shannon entropy [5] by choosing a definite state to occupy based on the tiles this cell may collapse to and the extracted frequency data. Cells with zero entropy are already collapsed.

2. Propagate the collapse of c to its neighbours. When c collapses, the number of tiles its neighbours might collapse to may be reduced. The neighbours of these neighbours might also experience a reduction in their superposition of states. The propagation phase is complete when this chain of reductions is complete.

It is worth noting that, during the propagation phase a cell may be reduced such that it cannot occupy *any* tile. That is, there is no tile directionally adjacent to at least one tile of every neigbour. In this case, the algorithm has run into a contradiction and does not continue. It thus resets.

After the entire wave is collapsed, the generation process is complete. The program then converts cells in the wave to pixels in the output image.

Below we provide an analysis of our own WFC implementation, within which we discuss the modules and classes implemented. In particular, we consider how the implementation goes through the algorithm stages discussed, referencing concrete files, data types, and functions.

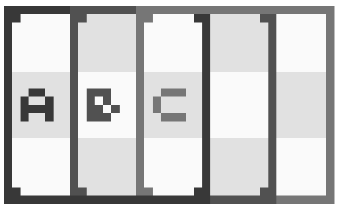

Figure 2: Producing a 3-row from the top-right pixels of tiles A, B, and C that are subsequently right-adjacent. This 3-row appears in the input image, as all three pixels appear in the tile A.

2.1 Extraction

The extraction of all information from the input seed occurs in `wfc_setup`. We call `extract()` to obtain a dictionary mapping the identifier of T to its top-left pixel for every unique tile T found in the input image. This function also produces the `tileset` 2-dimensional `numpy` ndarray of size equal to the input image, where the cell at index [i, j] stores the identifier corresponding to the tile whose top-left pixel is at index [i, j] in the input image. The remainder of the program will use tile identifiers to refer to each tile, and only in the output will these be converted to pixel data. As discussed in the Procedure above, the top-left pixel is the only important pixel in the tile when converting the output to an image.

The function process involves opening the file using the `Image` module from `PIL`, and converting it to a `numpy` ndarray. We use `numpy.pad` to wrap the input image, allowing for ease of extraction of tiles extending past the bottom or right border of the original input. From here, we simply translate $N \times N$ sections of the input ndarray into a tile identifier to be returned in the output ndarray. Computing identifiers for each tile involves hashing, where we first convert the $N \times N$ section of the input ndarray into a string, and then call Python's built-in `hash()`. This allows the same tiles to be hashed to the same identifier. The identifier is then mapped to the top-left cell of the $N \times N$ region.

After extraction, we deduce tile adjacencies and frequency data using `gen_rules()`. This function returns a directed multigraph with directional labels (2.1.1), and a dictionary mapping tile identifiers to their frequencies. The process relies on `numpy.unique()` to return all unique tiles in the `tileset` computed by `extract()`.

2.1.1 Adjacency Graph

Tile adjacency data is maintained within a directed multigraph storing directional labels on edges. Within `gen_rules()` all tiles are introduced as vertices, with an edge from A to B when A is a tile adjacent to B, meaning A is a neighbour of B in the `tileset`. This edge stores a single-character indicator 'L/R/U/D' representing the directional information of the adjacency relation. In particular, if B is up-adjacent to A, we have an edge from A to B with indicator 'U'. This is stored as a Python tuple `(vertex, indicator)`.

Because the input image may store more than one occurrence of any tile, we may find that one tile is adjacent to another in more than one direction. We therefore have the graph allow multiple edges from one vertex to another, under the condition that each edge stores a different directional indicator.

This adjacency graph is extensively used in the generation of the output wave and the propagation of a cell collapse, where the superposition of tiles for each cell is determined by the directional adjacencies to the superposition of tiles for neighbouring cells. The method `adjacent()`, accepting a tile identifier and a direction, is used precisely for this matter.

2.2 Generation

The generation of the output wave occurs in `wfc_core`, within the corresponding `Core` class. This class is initialized with an adjacency graph, frequency rules, a desired width and height for the output, and a visualizer (2.3). Upon construction, the `Core` generates a wave in the form of a 2-dimensional `numpy` ndarray. Each cell is initialized as `None`, representing that it may occupy *any* tile that has been extracted from the input image. In general, this cell is either `None` or is a Python set containing all tiles the cell may collapse to based on adjacencies with neighbouring cells. Thus, a collapsed cell is simply a singleton set, containing the single tile identifier to which it has collapsed. Note that, this wave may instead have been initialzied as a set containing all tile identifiers, but the memory cost is needlessly high.

All class methods are private beyond `generate()`, which generates and returns a collapsed wave. This method implements an iterated collapse and propagate procedure until either the wave is completely collapsed or a contradiction is reached, at which point the core resets and begins anew. At the end of each iteration, the visualizer is used to update a graphical display of the generation process. The cell chosen to collapse is that with the lowest nonzero Shannon entropy. While it is possible to iterate over all cells and compute this entropy in each iteration, this is computationally inefficient. We instead attach a `heapq` min heap attribute to the `Core`. A cell that experiences a reduction in its superposition, and thus a reduction in its entropy, during the propagation of a collapse will push its new entropy, alongside its coordinates on the wave, to this min heap. We then pop the minimum entropy from the heap when choosing a cell to collapse. It is worth noting that there is nothing stopping this popped cell from having collapsed between its push to the min heap and its pop for collapse. Hence, we continue to pop from the min heap until an uncollapsed cell is found.

2.2.1 Collapse

The collapse of a cell is handled by `__collapse()`, which accepts the coordinates to a target cell. The method collects all tiles the cell may collapse to as a list, and retrieves the weighting of each tile from the stored frequency data in the `Core`. The selection of a random tile is then left to `numpy.random.choice`.

2.2.2 Propagation

The propagation of a cell's collapse is handled by `__propagate()` which accepts the coordinates to the collapsed cell. The main process of this method is to iterate over all possibly affected cells and update their superposition of tiles. The affected cells are stored in a `deque` stack from the built-in Python `collections` library until they are appropriately updated. The method adds the immediate neighbours of the collapsed cell to the stack, and proceeds to pop and reduce elements from the stack until empty.

The reduction of a cell is handled by a helper `__reduce()`, again accepting the coordinates of a target cell. This method updates the cell's superposition of tiles based on its neighbouring cells in the wave. For every direction d, the method retrieves all tiles d-adjacent to at least one tile in the superposition of the d-neighbour. That is, one tile the d-neighbour may collapse to. This is determined using the `adjacent()` method of the adjacency graph. The method then takes the intersection of these collected tiles as the final superposition of the target cell, as a tile may only be in the superposition when it is adjacent to at least one tile of *each* neighbour.

It is possible that, during this reduction, the target cell is reduced to having no possible tiles to collapse to. In this case, the method alerts a contradiction by returning `1`. Otherwise, the method checks whether the newly computed superposition is equivalent to the currently stored superposition of the target cell in the wave. If it is, this cell was unaffected by the collapse, and thus it need not alert its neighbours of a change. If, however, the superposition was updated, we push the new entropy of the cell to the entropy min heap, and add the neighbours of the target cell to the stack so that they may also be updated.

2.3 Visualisation

The visualisation of the wave during generation occurs in the `wfc_visual` module, within the corresponding `Visual` class. This class is initialized with a dictionary mapping tile identifiers to pixels, a desired width and height for the wave to visualise, and a tile size representing the desired pixel width of each tile in the wave. The initializer admits optional flag and debug parameters, allowing for customizability of the visual representation and additional environment features useful when debugging. The flag parameter admits inputs 'off/manual/auto'. Upon construction, `Visual` generates a `Pygame` surface.

The core method of the `Visual` class is `draw()`, accepting as parameter a wave to visualise. For each cell in the wave, the method computes a representative colour by taking the average of all pixels mapped from tile identifiers in the cell's superposition. The representative pixel is then drawn on the `Pygame` surface at the corresponding location, accounting for the stored tile size. If the Visual flag is set to 'manual' the method then awaits keyboard input before terminating. If, instead, the Visual flag is set to 'off', the draw method is set to the trivial lambda in the initializer, and is thus disabled altogether.

The `wfc_visual` module contains also a `render()` function that converts a collapsed wave to an image file at a designated location. The function accepts the image file path, a mapping from tile identifiers to corresponding pixels, and a wave to render. Much like the `Visual.draw()` method, `render()` iterates over each cell of the wave. However, as the wave is collapsed, each cell contains precisely one tile identifier, which may be popped and translated to a pixel using the input mapping. This pixel is then stored in a new output `numpy` ndarray. The `Image` module in `PIL` is used to then translate the ndarray to an image at the desired location.

2.4 Modifications

Following progress on the base algorithm as discussed in the proposal submission, the difficulty in identifying bugs in the implementation was made evident. The visualisation of the wave during generation was introduced to trace the generation process and identify possible faults. Although not initially presented to be a feature in the final product, the in-progress visual component offered additional feedback to users, and allowed them to see the algorithm in progress. Consequently, the `Visual` class introduced this visualisation to the final product, and tools specific to debugging were hidden under the debug toggle in the `Visual` initializer. Beyond this adjustment, no changes to the project plan were made following the proposal.

3 Instructions

Please see the `requirements.txt` document that accompanies this submission for a collection of this project's Python library dependencies. To initiate the algorithm, simply run the `main.py` module. This will run the only method in the module, `main()`. Please ensure that the parameter `in_` is indeed a valid path to the input image. Additionally, the directory for `out` must exist, but the file does not. We encourage experimentation with the remaining parameters to yield different results.

- `n` specifies the dimensions of the $N \times N$ tiles to be extracted from the input image
- `w` and `h` specify the dimensions of the output
- `flag` specifies the form of visualisation desired during wave generation, passed to `Visual`; this flag is set to 'auto' to automatically refresh the visualisation every time the wave changes, 'manual' to await keyboard input after every wave change, and 'off' when no in-progress visualisation is desired

For more information, consult the documentation within each module. It is to be noted that the debug toggle is automatically set to `False`. If readers wish to experiment with the code and enable debug mode, they must add the additional argument `debug=True` to the `Visual` initializer.

Once the module is run, a `Pygame` window will appear. The behaviour of this window depends on the flag specified as input to `main()`. This window will not appear at all if the flag is set to 'off.' Once wave generation begins, the display will update at the end of every propagation phase, visualising the extent to which the output has been generated. If the screen resets to a monotone colour, the algorithm has reached a contradiction and reset. Once the wave is entirely collapsed, the program terminates and the `Pygame` window closes. After this, the ouput image should be found in the location specified as parameter to `main()`.

3.1 Datasets

In order to run and test any implementation of the image-based WFC algorithm, there must exist an input seed from which patterns are extracted. This project utilizes a dataset composed of images curated both by Maxim Gumin in his original development of the algorithm [1] and images of similar flair produced by us. Naturally, the algorithm functions on any image, and it is therefore encouraged for readers to run the algorithm using their own images. It is to be noted, however, that images are processed pixel-by-pixel, so that images of larger sizes will take notably longer to extract. We recommend input images of no more than 32 x 32 pixels. The algorithm works particularly effectively for images with repeated patterns.

To access the set of images used to generate the outputs in Figure 3, download and extract the `images` folder accompanying this document. When running the program, ensure that this folder is in the same directory as the accompanying Python modules submitted. The program will run as desired for any images so long as the input and output paths are valid, however we recommend any additional images fed into the algorithm also be placed in this folder.

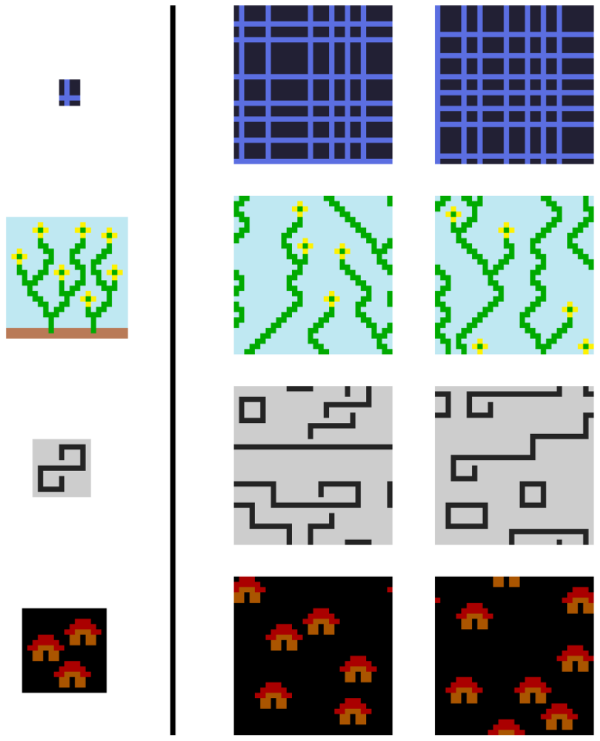

Figure 3: Example outputs and the input seeds they correspond to.

4 Discussion

Through our implementation, we successfully demonstrated a way to take an input image and produce similar outputs. Whether or not this similarity is satisfactory within a particular domain is at the discretion of developers and readers. Indeed, there exists a likeness between the inputs and outputs in Figure 3, yet one would be hard-pressed to say that the input of flowers yields an output that might pass as plants of any kind, when considering the unnatural twists in the stem and the lack of ground in the images. In contrast, the outputs corresponding to the houses input seem notably characteristic of a village, perhaps justifying the prevalence of the WFC algorithm in game development. We might conclude that our implementation is successful in creating outputs similar to the input seed, yet that similarity may be inadequate in some sense. That is, the way we define local similarity in this algorithm may not suit particular domains. Perhaps for these domains a different algorithm may be found more effective.

4.1 Limitations

Through experimentation with inputs, outputs, and tiles of various sizes, we find that WFC only works at its best when the input seed features repeated patterns and a similarity throughout itself. This way, the same tiles appear several times in the input, so that more adjacencies may be extracted amongst the same tiles. Otherwise, we arrive either at remarkably boring results or so many contradictions we might be better off saying there are no results at all. We may rationalise this: if there are notably few adjacencies between tiles, the same tiles will always appear next to one another. When they do not, we will frequently arrive at a contradiction.

A particular example of this is seen through the flowers in Figure 3. Whenever there exists a ground tile in the output, it is guaranteed that the *entire* row of pixels will be ground tiles. In fact, seeing as there are only two 'excapes' from this ground through the flower stem, it is quite feasible that no flowers sprout from the ground at all, resulting in a rather uninteresting output of ground and sky. This is all the more evident if we remove the smaller flower to the right.

A notable issue with the algorithm is simply how long it takes to generate output. The program is relatively efficient for tiles of size 2, yet even considering tiles of one size greater we note a major spike in the time taken to generate output, as well as the number of contradictions the algorithm reaches. This is all the more evident for larger output images. WFC is so slow for tiles of size greater than 3 that it is hardly worth considering these outputs at all.

We thus have a very limited domain in which WFC functions in a pleasant way. We must have inputs featuring repeated patterns, outputs of relatively small size, and tile sizes of either 2 or 3. In theory we may consider tiles of size 1, but the outputs tend to be unremarkable noise. Fortunately, some of these issues are not intrinsic to the algorithm, and may be bypassed with some optimisation.

4.2 Further Exploration

The WFC algorithm may be optimised somewhat, although this process tends to be unintuitive. Through parallel programming, we may operate on different areas of the wave simultaneously during the propagation phase, quickening the effects of a cell's collapse. In fact, there exist notable discussion on this method of optimisation, among others [6].

An immediate extension of the WFC algorithm is the introduction of rotations and reflections of tiles into the output. In particular, we may take tiles extracted from the input and transform them, introducing new tiles and adjacencies thus.

In a more exploratory direction, we may consider working with other input formats. Although we discussed WFC exclusively within the context of image processing, we might easily extend this algorithm to $n-$dimensional objects. In this case, we must change the directional indicator stored in edges of the adjacency graph, so that it may support a larger number of directions. Doing this may involve changing the method of representation to an $n-$tuple. A larger source of trouble, however, is just how we might produce adjacency data for such an object to begin with.

It may be sensible to conclude that WFC is rather versatile, with a core algorithm that simply requires adjacency information between objects and the expected frequency of each object in the result. Overlooking the time taken to produce satisfying outputs, it is no surprise, thus, that this algorithm has found itself prominent in the game development scene. Yet considering its limitations, we might wish to explore other procedural content generation algorithms that succeed where WFC fails.

References

[1] M. Gumin, “WaveFunctionCollapse,” GitHub, https://github.com/mxgmn/WaveFunctionCollapse

[2] H. Kim, S. Lee, H. Lee, T. Hahn, and S. Kang, “Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm,” *2019 IEEE Conference on Games (CoG)*, Aug. 2019. doi:10.1109/cig.2019.8848019

[3] R. Heese, “Quantum Wave Function Collapse for Procedural Content Generation,” arXiv.org, doi:https://doi.org/10.48550/arXiv.2312.13853

[4] J. J. Barton, “Wave function collapse,” Wikipedia, https://en.wikipedia.org/wiki/Wave_function_collapse

[5] “Entropy (information theory),” Wikipedia, https://en.wikipedia.org/wiki/Shannon_Entropy

[6] Y. Nie, S. Zheng, Z. Zhuang, and X. Song, “Extend Wave Function Collapse to Large-Scale Content Generation,” arXiv.org, doi:https://doi.org/10.48550/arXiv.2308.07307

[7] gridbugs, “Procedural Generation with Wave Function Collapse,” gridbugs.org, https://www.gridbugs.org/wave-function-collapse/

[8] “Pillow,” Pillow (PIL Fork), https://pillow.readthedocs.io/en/stable/

[9] “NumPy,” NumPy Documentation, https://numpy.org/doc/

[10] “Pygame,” Pygame Documentation, https://www.pygame.org/docs/