Proceedings of the Fourteenth Artificial Intelligence and
Interactive Digital Entertainment Conference (AIIDE 2018)

Generating Paths with WFC

Hugo Scurti, Clark Verbrugge
School of Computer Science
McGill University
Montréal, Québec, Canada
hugo.scurti@mail.mcgill.ca, clump@cs.mcgill.ca

Abstract

We describe a tool based on the *Wave Function Collapse* algorithm that performs example-based path generation on fixed maps. Our design aims at a practical system usable by non-programmers, and includes both easy input control and multiple post-processing steps. The design is implemented in Unity and enables users to easily visualize the results of experimenting with different path descriptions and game levels.

Introduction

In this work we describe an example-based approach to path generation with a simple, flexible interface. Our design is based on the *Wave Function Collapse (WFC)* algorithm (Gumin 2016), an example-based approach to texture generation. We developed a complete realization in Unity, with available source code (Scurti 2018). We modify and augment WFC to form a workflow that takes a representative path design and game level as input, and produces a usable set of paths respecting the given path properties and level constraints. Simple modifications to the input path design can be done with any image editor, and can then be used to control properties of the resulting paths. The algorithm scales to significant, practical level sizes. For easy use and experimentation, the tool is able to read in actual game levels expressed in ASCII form, as from the *Moving AI* benchmark suite (Sturtevant 2012). For more information, a detailed analysis is available at https://arxiv.org/abs/1808.04317

WFC Background

Our approach builds heavily on the *Wave Function Collapse* algorithm developed by Gumin (2016), more precisely the overlapping version of the algorithm. This algorithm builds random output textures based on smaller input textures by looking at overlapping patterns in the input texture sample. The original implementation adds a few helpful options for texture generation, such as the ability to set boundary patterns, wrap periodically, and extend the range of available patterns by adding rotations and reflections of patterns sampled from the input. Further detail can be found in the code itself.

The WFC algorithm has seen a few notable ports and forks. Our design is influenced by Karth and Smith's paper on WFC in which they describe in depth the algorithm and relate it to constraint solving (2017). We also made use of Fehr and Courant's modifications to improve performance (2018).

WFC for Path Generation

The intent of our approach is to allow a user to supply a single path sketch showing an example path, and to then apply it to an existing output level, filling the level with paths that have characteristics similar to the original example. However, using the WFC algorithm to achieve this would introduce a few issues. First, we need to be able to fix tiles in the output to represent the output map on which the algorithm is executed. While it is possible to fix tiles in WFC, doing so manually for each obstacle in the output would be tedious. We solve this by automatically generating static outputs that represent the output map on which the algorithm is executed. Furthermore, using only the precise paths and obstacles in inputs would either put too much or not enough constraints on the output. We deal with this by using *stretch space*, which is a defined color between obstacles and paths to represent an arbitrary distance between them. It is also not possible to represent all shapes of obstacles into a single input. We therefore introduce the notion of *masks* in order to handle different shapes in the output map and to deal with boundaries.

Since the algorithm works on pixelated images, we add post-processing steps in order to generate actual paths on game maps and make the end result more appealing through simplification and smoothing (Ramer 1972; Douglas and Peucker 1973; Chaikin 1974).

Implementation

Our tool is based on a Unity implementation of our modified WFC algorithm. We use custom inputs, extracting patterns of size 3×3 as sufficient to represent useful path properties, without introducing major scaling issues. Input and output 2D maps use small monochromatic sprites to represent tiles and interactively show the execution of the algorithm. When the execution succeeds, post processing steps can be applied to generate paths in a 3D representation of the output

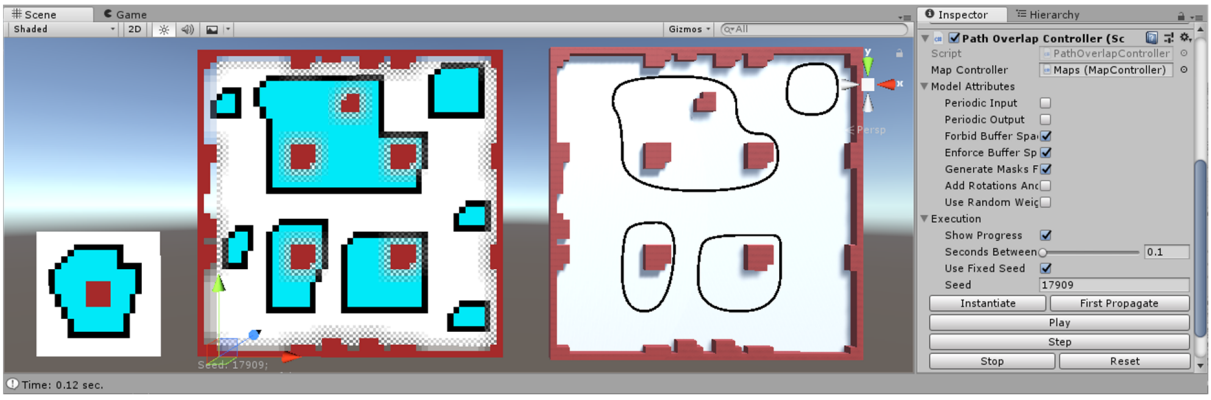

Figure 1: Screenshot of the Unity tool developed. First image on the left is the input. The image in the middle is the result of the main algorithm. The image on the right is the final result after post-processing.

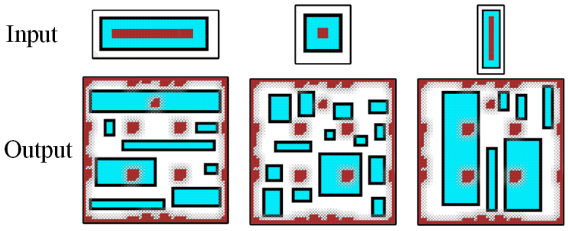

Figure 2: Comparison between output generated with different types of input. Rotations and reflections are disabled

map, along with an agent that can walk through generated paths. Figure 1 shows the interface. The algorithm was tested with maps representing game levels from *Dragon Age: Origins* and *Baldur's Gate II*, as extracted from the Moving AI benchmark sets (Sturtevant 2012).

Results

Output of the algorithm depends on characteristics of the input and can be controlled by different user options. Below we show the effects of some major choices.

Frequency distribution

Using weights to select patterns gives us some control over the outcome of the algorithm. Our algorithm uses the same pattern selection mechanism as in the WFC algorithm: the frequency distribution of patterns found in the input is used to bias the choice of patterns applied to the output. As we can see in figure 2, the difference in number of horizontal or vertical components of the input patterns is thus reflected in the generated output.

Different input/output combinations

To support the argument that the algorithm works for different combinations of input and output, we show examples on 2 larger inputs, namely *Arena2* and *Lak519d* from the *Dragon Age: Origins* benchmark set, which are respectively of size 281×209 and 168×145 pixels. We also introduce 2

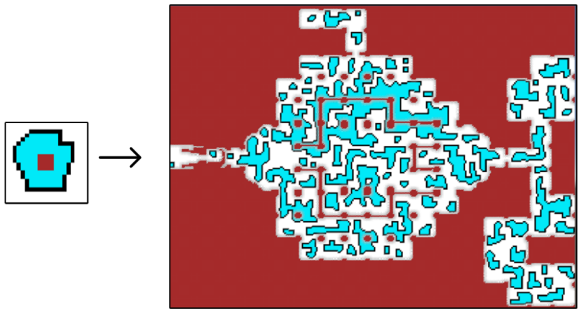

Figure 3: Example of execution with the Arena2 map.

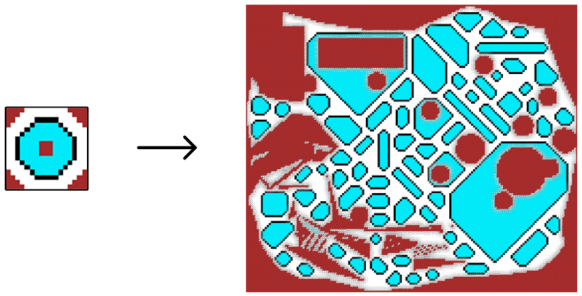

Figure 4: Example of execution with the Lak519d map.

new input textures to show examples of different outcomes based on significantly different inputs.

Figure 3 shows a result of the algorithm on the *Arena2* map using a complex polygon as the input, and with rotations and reflections enabled. As a result, generated paths have more complex turns and routes.

Figure 4 shows a result on the *Lak519d* map using an octagon as the input. For this input, enabling rotations and reflections is irrelevant as it would not add additional patterns. Compared to figure 3, this output has less varying path segments. However, in terms of path shapes, paths in this output are more similar to its corresponding input.

References

Chaikin, G. M. 1974. An algorithm for high-speed curve generation. *Computer Graphics and Image Processing* 3(4):346 – 349.

Douglas, D. H., and Peucker, T. K. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization* 10(2):112 – 122.

Fehr, M., and Courant, N. 2018. fast-wfc. https://github.com/math-fehr/fast-wfc. Github repository.

Gumin, M. 2016. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse. Github repository.

Karth, I., and Smith, A. M. 2017. WaveFunctionCollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, 68:1–68:10. New York, NY, USA: ACM.

Ramer, U. 1972. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing* 1(3):244 – 256.

Scurti, H. 2018. Path-wfc. https://github.com/hugoscurti/path-wfc. Github repository.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144 – 148.