Generating Paths with WFC

Hugo Scurti

M. Sc. Computer Science

School of Computer Science

McGill University

Montréal,Québec

July 2018

ABSTRACT

Title: Generating Paths with WFC

Date: July 2018

Student: Hugo Scurti

Supervisor: Clark Verbrugge

- - -

Path planning for Non-Player Characters (NPC) is a widely discussed topic in game development, for which many solutions can be used to effectively implement desired behaviors. On another hand, using randomized path planning to generate regular or repetitive movement is not a trivial task. It often requires non-trivial programming effort or integration with pathing systems. We introduce an example-based approach to randomized path generation which requires little to no programming capabilities. The algorithm presented is a modification to the *WaveFunctionCollapse* algorithm that handles random path generation on fixed maps. We augment this modified algorithm by adding post-processing steps such as filtering small paths, path simplification and path smoothing. We show results on how the algorithm behaves when using different options and different combinations of input and output. We present a set of tools developed in the Unity engine for users, with or without programming knowledge, to utilize the algorithm.

TABLE OF CONTENTS

ABSTRACT ii

LIST OF TABLES iv

LIST OF FIGURES v

1 Introduction 1

2 Background 3

3 Method 6

3.1 Output constraints and pre-processing 6
3.2 Arbitrary colors as "Stretch space" 7
3.3 Masks 10
3.3.1 Stretch tiles around obstacles and boundaries 11
3.4 Post processing 12
3.4.1 Conversion to 3d map 12
3.4.2 Path filtering 13
3.4.3 Path Simplification 13
3.4.4 Path Smoothing 14

4 Experimentation 17

4.1 Experiment 17
4.2 Tools 18
4.3 Results 18
4.3.1 Stretch space 19
4.3.2 Frequency distribution 19
4.3.3 Uniform distribution 20
4.3.4 Different input/output combinations 21
4.4 Performance 22
4.4.1 Optimization 22
4.4.2 Result 22

5 Related work 24

5.1 WFC 24
5.2 PCG 25
5.3 Roadmap Generation 25

6 Conclusion & Future Work 27

LIST OF TABLES

Table page

4–1 Average time in seconds for varying inputs 22

4–2 Average time in seconds for varying outputs 23

LIST OF FIGURES

Figure page

2–1 Executions of the overlapping model of the original WFC algorithm. Both examples use patterns of size 3×3 and use reflection only on the y axis. Top example uses the floor option to fix the bottom line. Input images are provided by [12]. 4

3–1 Different outputs making the algorithm succeed (Output 1) and fail (Output 2) 7

3–2 Execution of the algorithm without using stretch space 8

3–3 Example of how stretch space works during the execution of the algorithm. The second selected pattern is a vertical path segment in the top figure whereas in the bottom figure, it is a stretch tile. 9

3–4 Execution of the algorithm using the notion of stretch space. To show progress, we use the same method as in the original WFC algorithm. 9

3–5 Example of overlapping patterns with masks 11

3–6 Execution of the algorithm using masks on output boundaries. Masks are represented as transparent tiles. 11

3–7 Execution of all post-processing steps in their respective order. Output map is "arena" from the *Dragon Age: Origins* benchmark set 12

3–8 Examples of our implementation of the RDP algorithm 13

3–9 Examples of our implementation of Chaikin's curve generation algorithm. 4th column is the result after 5 iterations, where the result is similar to the limit surface. 15

4–1 Workflow of high-level components of the algorithm 17

4–2 Screenshot of the Unity tools developed. The image on the left represents the input sample. The center image is the result of the main algorithm. The image on the right is the result after post-processing. 18

4–3 Comparison between output generated with and without stretch space 19

4–4 Comparison between output generated with different types of input. Rotations and reflections are disabled in order to see the differences 19

4–5 Comparison between output generated with different types of distribution. Rotations and reflections are disabled. 20

4–6 Example of execution with the Arena2 map. 21

4–7 Example of execution with the Lak519d map. 21

CHAPTER 1
Introduction

Pathfinding is a well known concept in game design and artificial intelligence. It typically involves finding the shortest path between a start and a finish point on a fixed map. However, in some cases the shortest path is not necessarily the one that we are interested in. *Non-Player Characters (NPCs)* in games tend to follow a specific linear path, such as when NPC allies follow the player's movement or when NPCs go to a certain goal. On the other hand, NPCs can also stay in a specific area and wander around, guarding something or waiting for some event to happen. In the latter case, we are interested in generating randomized meandering paths to represent random walks taken by idle NPCs.

Simple random walks, however, are unsatisfying, in tending to unrealistic, jagged, fractal-like trajectories with erratic space coverage [16]. Better results are possible by making use of higher-level navigation models, such as a waypoint graphs or flow models, but it is in general not an easy task to do for non-programmers, requiring substantial algorithmic development and significant programming effort to ensure simple, frequently desired properties, such as avoiding self-intersection, forming closed loops, and encircling obstacles.

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 [12], an example-based, constraint-based algorithm that generates textures by extracting small patterns from an input image, setting up pairs of patterns that match when overlapped alongside one another, and randomly filling the output image using the extracted patterns and their neighbouring constraints. Our goal is to use the algorithm to form a workflow that takes a representative path design and game level as inputs, and produces a usable set of paths respecting the given path properties and level constraints. Since WFC was not designed for such a task, we explain in depth how we modify and augment the algorithm to achieve this goal. Mainly, we make sure that the algorithm can be initialized with fixed size output maps instead of empty initial outputs, we use the overlapping scheme to use a specific color (*stretch color*) to fill the inside of paths in order to enforce the input path's shape to a certain extent, and we use the concept of *masks*

to loosen the output map's constraints on fixed obstacles and allow any generic input to be used with any output regardless of the shape of obstacles. We then explain how simple modifications to the input path can be done with any image editor, and can be used to control properties of the resulting paths to a certain degree. We present a set of tools developed in Unity that implement the presented model, and we describe how it works and how simple modifications to input images can control the outcome of generated output. We present various new options and demonstrate how tweaking different options on our algorithm affects the generated output in a concrete manner. We show how 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 [28].

Contributions of our work are as follows:

- We define modifications to the basic WFC algorithm to allow for small, generic images to be used as general models of path design. Basic, easily modified properties of the image then control useful properties of the path output. Our approach includes non-trivial changes necessary to accommodate the densely constrained output context of typical game levels.
- A complete realization in Unity is shown and made available [1] , including a full workflow that reads in game levels and generates abstract path data structures. This work includes a pre-processing step to read in game levels defined in ASCII form and post-processing steps to filter and smooth the output paths and convert them from a 2d image into a game-ready format.
- WFC has known scalability limitations. We incorporate recent performance improvements [9] and conduct performance experiments on actual, realistically large and complex game maps as evidence of scalability.

[1] https://github.com/hugoscurti/path-wfc

CHAPTER 2
Background

Our approach builds heavily on the *Wave Function Collapse* algorithm developed by Gumin [12], more precisely the overlapping model of the algorithm. This algorithm builds random output textures based on smaller input textures by looking at overlapping patterns in the input texture sample. Understanding this algorithm is critical to understanding our modifications, so we summarize the main steps below. Further details can be found in the code itself.

1. Extract all possible unique $n \times n$ patterns from a discretized input image. The frequency of each unique pattern from the input is stored to use later as a sampling distribution.
2. Compute all the possible overlap combinations between each pattern. An overlap is defined by 2 patterns overlapping one with another on a given offset (in both x and y directions) where each overlapping tile from one pattern have the same color as its related overlapping tile from the other pattern. This is used to filter out incompatible surrounding patterns when selecting a pattern at a fixed tile in the output image.
3. Each discretized tile in the output sample maintains a list of legal patterns that can define the pixel value in that tile. At the start of the algorithm, we set all values to true, in the sense that every pattern can be placed anywhere.
4. Choose a tile in the output sample and commit the value based on one of the legal patterns that cover the tile. This choice follows an entropy calculation that looks for the tile that has the fewest available patterns, therefore the most constrained tile. The frequency distribution of patterns computed in step 1 is then used to select a pattern on the chosen tile. This is referred to as the **observe** step of the algorithm.
5. Having fixed a specific pattern on a specific position, surrounding tiles should now have fewer possible values than before. The overlap combinations computed in step 2 are used to filter out any now illegal patterns covering adjacent positions, and this is applied recursively throughout the output image. This is referred to as the **propagate** step.

6. Repeat steps 4 and 5, observing and propagating until no further progress can be made. At that point either all tiles are fixed to one pattern (the algorithm succeeded), or at some point a tile has no more patterns available (the algorithm failed). In the latter case the algorithm is restarted with a different random seed.

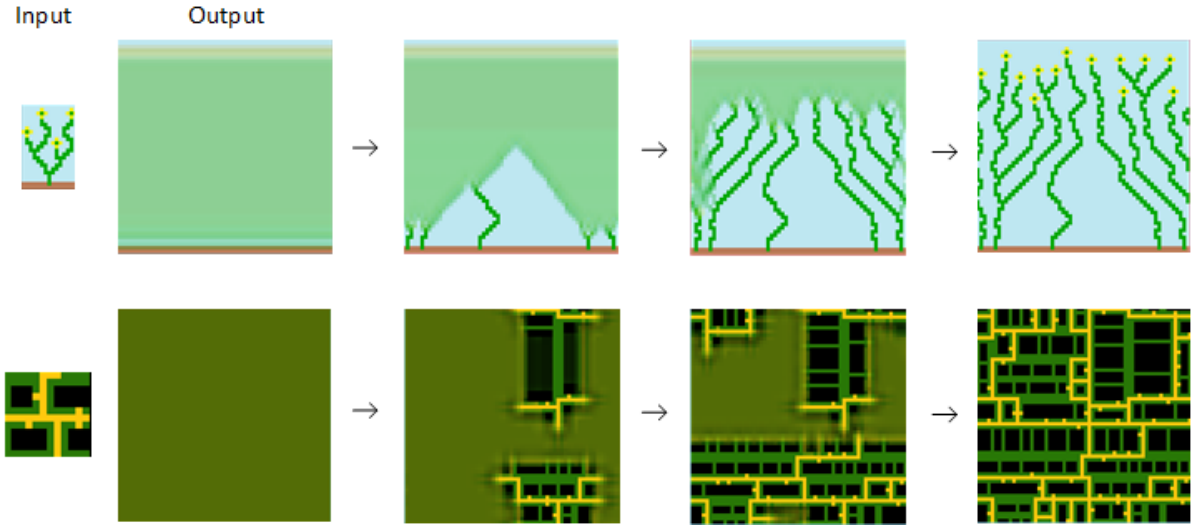

Figure 2–1: Executions of the overlapping model of the original WFC algorithm. Both examples use patterns of size 3×3 and use reflection only on the y axis. Top example uses the floor option to fix the bottom line. Input images are provided by [12].

This represents the main concept of the overlapping model of the algorithm. In most cases, patterns of size 3×3 are sufficient to capture enough overlapping behaviours between neighbour patterns without compromising running time. The original implementation adds a few options for texture generation. These options include the ability to set floor and ceiling patterns, to treat the input and output as periodic images, and to extend range of available patterns by adding rotations and reflections of patterns sampled from the input. Figure 2–1 shows the execution of the algorithm on 2 different inputs using the interactive GUI made available by Gumin[1] . During the execution of the algorithm, partial states (states that don't have a unique selected pattern yet) are shown by averaging all possible color values from surrounding patterns, for a specific position, until it is fulfilled by one unique pattern. This results in a somewhat blurry image

[1] Available here : `https://exutumno.itch.io/wavefunctioncollapse`

that is being unveiled over time while patterns are being observed and propagated. Since the algorithm starts selecting tiles where the entropy is the lowest, we generally see the algorithm being filled up from the most constrained positions. As a result, since the top example in figure 2–1 uses options the set the floor fixed, we can see that the algorithm starts filling the output from the bottom, making its way to the top. Alternatively, we cannot assess such conclusion on the bottom example since no tiles were fixed beforehand. Moreover, it is possible to infer initial constraints simply by looking at the first output image since tiles are displayed by averaging colors from legal patterns. As we can see in the top example of figure 2–1, brown floor is visible given that floor option was set. Moreover, blue ceiling is noticeable because periodic input and output was set and therefore the only extracted patterns that fit beneath the brown floor patterns were blue ceiling patterns.

The second version of the algorithm is based on tiles rather than input images. Instead of having a fixed input image from which we extract patterns, the algorithm expects a list of predefined patterns and a list of neighbours. Patterns also have a symmetry type associated with them, which helps to determine the symmetry of patterns in order to build a definite list of rotated and reflected patterns.

Finally, the algorithm is best used with the interactive GUI mentioned above. The program lets us browse through predefined samples from both versions of the algorithm and it is convenient to add additional samples to the list. Moreover, the controls are simple enough to execute many iterations fairly quickly using keys from 1 to 9 that represent the speed at which the execution goes (1 being the slowest, 9 being the fastest). Furthermore, with the overlapping model, it is possible to fix values on specific positions in a drawing stroke by holding down keys between F1 and F12. The way it works is that each extracted color is mapped to an F-key, and while the related key is pressed on a specific position the algorithm filter out patterns for which the color value on that specific position is not the color associated with the key pressed, if the state of the position is not yet fixed.

CHAPTER 3
Method

Our goal is to use the overlapping model described above in order to generate meandering paths in a fixed output map by using the input sample to describe how paths and obstacles interact with each other. However, using the WFC algorithm as it is 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's 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 paths and obstacles in inputs would either put too much or not enough constraints on the output. We deal with this by using colors on areas between obstacles and paths to represent an arbitrary distance between an obstacle and a chunk of path. Moreover, it is not possible to represent all shapes of obstacles into a single input. Therefore, we introduce the notion of *masks* in order to handle different shapes in the output map and to deal with boundaries. Finally, as the algorithm works on pixelated images, we add a few post-processing steps in order to generate actual paths on game maps and make the end result more appealing.

3.1 Output constraints and pre-processing

The original WFC algorithm generates outputs using only a fixed output dimension, but also lets users fix specific tiles in the output. However, this is done manually, and doing so for a map filled with obstacles would not be efficient. Our modified version takes maps as an output and dynamically fix all necessary tiles so that the output represents faithfully the selected game level. Maps are either represented by user-defined images or ASCII-based text files in Moving AI's map format. The output can contain 2 types of tiles : free space (represented by a white pixel) and obstacles (represented by a reddish pixel). The obstacles act as areas that cannot be altered by the algorithm, whereas free areas can be filled by the algorithm.

Since we now handle fixed output, we must let the algorithm generate output while keeping the obstacles in place. We do this by loading the output map as an array of colors and then by filtering out

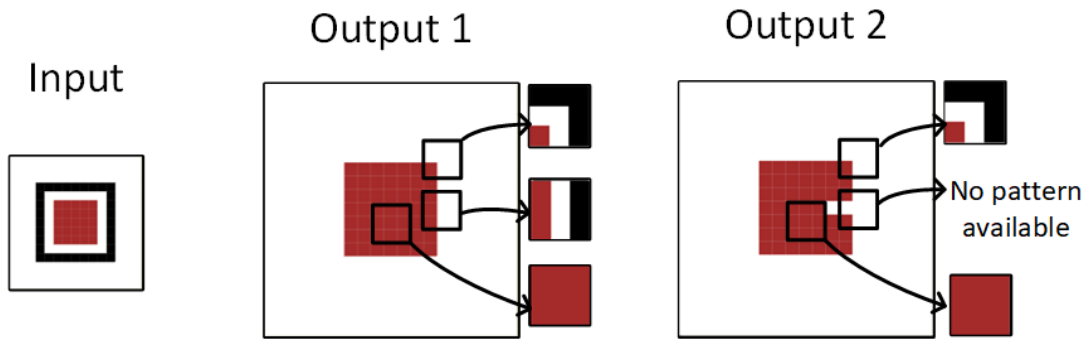

Figure 3–1: Different outputs making the algorithm succeed (Output 1) and fail (Output 2)

the patterns that don't match the output's colors on given positions. The filtering ensures that a pattern is legal on a position in the initial output only if all obstacles on the pattern match the obstacles on the output, and all non-obstacle tiles correspond to any non-obstacle tiles on the output. As shown on *Output 1* in figure 3–1, it filters down possible patterns on an obstacle's boundaries and allows only unicolored obstacle patterns in the interior of obstacles. This is analogous to using the original WFC algorithm and manually selecting the appropriate obstacle patterns for specific positions in order to simulate obstacles on a fixed map.

A drawback that arises from our implementation is that there is a restriction on the pairs of input and output that can be used together. By using only patterns found from the input, the algorithm will succeed only if all the obstacle shapes of the output can be reproduced from a local pattern found in the input. Per example, *Output 2* in Figure 3–1 shows that the notched obstacle cannot be filled by any pattern since no pattern from the input matches this specific obstacle. We will show later how to alleviate this issue by using *masks*.

3.2 Arbitrary colors as "Stretch space"

The focus of our algorithm is to generate paths that behave similarly to the ones described in the input. However, using only paths and obstacles in the input tends to generate paths that are either too strict or under-constrained. Example 1 in figure 3–2 shows an example of a under-constrained generated output where no pattern contains both obstacles and paths, therefore generated paths meander arbitrarily, more or less oblivious to the obstacle. Example 2 shows a stricter example, which will always generate the

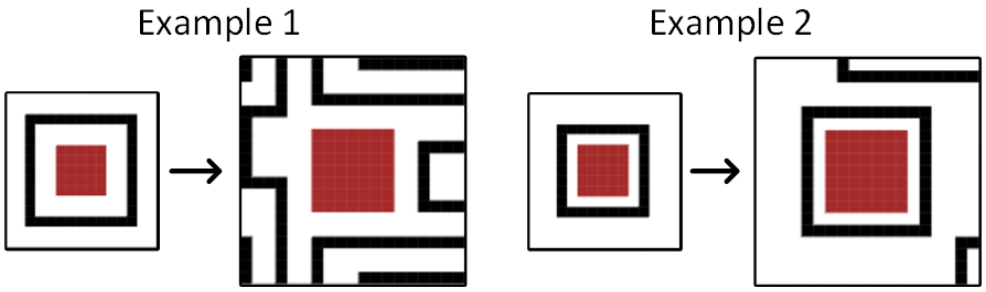

Figure 3–2: Execution of the algorithm without using stretch space

same path going around the obstacle since the only pattern that can go around an obstacle is one with a path at one unit away from the obstacle.

We introduce the notion of *stretch space* to counter this issue. We use a unique, designated color in the input image to represent the space between an obstacle and the path that goes around it. We make sure that one pattern can be filled entirely with this color so that the overlapping nature of the algorithm will make it overlap with itself, therefore making the space between obstacles and paths *stretchable*. If we would use stretch space smaller than 3, the generated output will have paths that will always be at the same distance to the obstacle as in the input, therefore having less possibilities to generate randomness in output paths.

This helps the generated paths behave more closely to their inputs because it restricts the sets of overlapping patterns for each pattern that contains a path. For example, without stretch space, a top-left corner could overlap on top of a vertical line on the right of an obstacle. However, this is not possible when using stretch space since the vertical line would have a stretch area to its left while the top left corner would have a stretch area to the right, and therefore the colors wouldn't match.

As mentioned above, white tiles in the output image can be filled by any color other than the color associated with obstacles. This means that stretch colors can be used in empty areas of the output map. Since our main focus is to generate random meandering paths around obstacles, we will use these colors to stretch the area between a path and an obstacle. As shown in figure 3–3, we use the blue color as a stretch space to let the algorithm set a random length between the obstacle and the path. This is done by either selecting another fully blue pattern to extend further the stretch area, or by selecting a pattern that closes the area with a path. As in the original algorithm, the random selection for the next pattern is done by

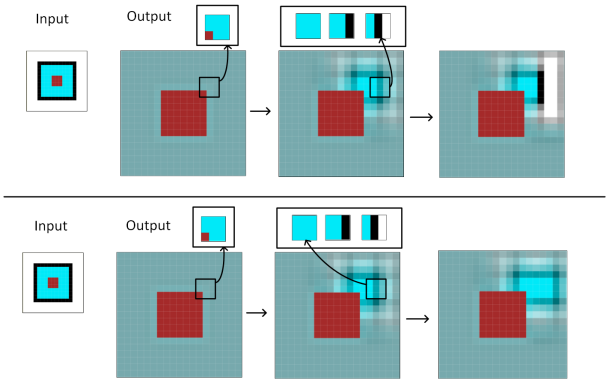

Figure 3–3: Example of how stretch space works during the execution of the algorithm. The second selected pattern is a vertical path segment in the top figure whereas in the bottom figure, it is a stretch tile.

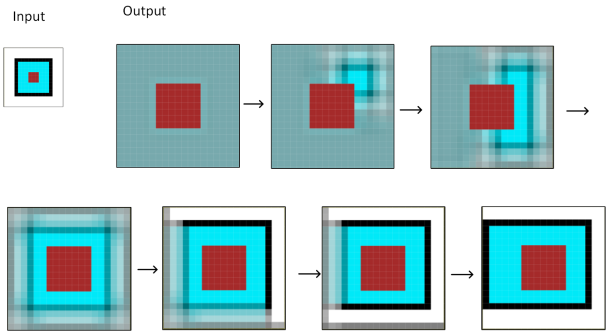

Figure 3–4: Execution of the algorithm using the notion of stretch space. To show progress, we use the same method as in the original WFC algorithm.

firstly selecting the next tile with the lowest entropy, then by choosing a random pattern from the list of possible patterns using pattern frequencies as a sampling distribution[12].

As we can see in figure 3–3, using a simple input such as a squared path around a squared obstacle and a simple output such as a bigger squared obstacle lets us generate simple paths but with considerable randomness in them. Paths generated by this pair of input/output will always be a rectangular path that will go around the obstacle. The difference in generated paths will be the space between each side of the obstacle and the corresponding path segment that goes around it. Figure 3–4 shows an example of the execution of the algorithm using the same input and output as in figure 3–3.

It's important to note that one of the drawbacks of using this notion of stretch space alongside the WFC algorithm in order to generate paths around obstacles is that given the nature of the algorithm, there can be other smaller generated paths that doesn't go around obstacles. This is due to the fact that the overlapping method works by matching patterns in terms of overlapping colors, thus making it possible for a path with stretch space to only contain stretch space inside it. This consequence can either be seen as a positive or negative effect. We will deal with this during the *post-processing steps* by enabling path filtering.

3.3 Masks

One of the main focus of this algorithm is its usage of *masks* to alleviate stricter input obstacles. Obstacles in maps have a tendency to differ from simple shapes, and input images would get very complicated if we would try to represent any type of obstacle shapes (i.e. representing any type of obstacle outline that would make patterns fit output obstacle shapes). Another option would be to tailor an input obstacle for each output map, but this would eliminate the idea of using general inputs to generate similar paths on different outputs. We introduced masks, which can represent any subset of colors. Masks can be used in 2 different ways : either it represents all colors defined in its set of colors, or it represents all colors except those defined in the set of colors. This makes comparisons between various sets of colors easier, in the sense that we can attribute a mask to a broader set of colors, or a category, and use it to verify that a particular color matches this category.

Using this idea of categories implemented with masks, we integrate it into the algorithm by instantiating masks that can be used as colors, which can replace true colors. These masked colors would then be used when evaluating all possible overlapping patterns and would filter out all patterns where a color overlapping a mask doesn't match the set of rules defined by this mask. This opens the door to more intricate combinations of patterns. We use this feature to be able to use any input for any output with obstacles of different shapes. We do this by creating patterns from the output map. We first analyze the output for all 3x3 patterns that contain obstacles. Then, for each pattern, we remove white tiles around obstacles and replace them by different masks depending on their relative position to an obstacle tile. Tiles that are adjacent (1 tile away) to an obstacle will contain a mask that allows everything but the path color. This mask ensures that no path are adjacent to obstacles. All other tiles will contain a mask that allows any color.

This feature ensures that any obstacle in the output can be dealt with and can have paths, free space and stretch space around it. Figure 3–5 shows an example of how overlapping works when using masked tiles. The tile in the top right corner containing the outline of the obstacle only has one pattern available, which is one of the masked patterns generated from the output map. If we don't use masks, then no pattern could fit at this position and the algorithm would always fail for this specific pair of samples. The upper tile shows all patterns that can be matched inside the notch. Given the way masks are generated, we can

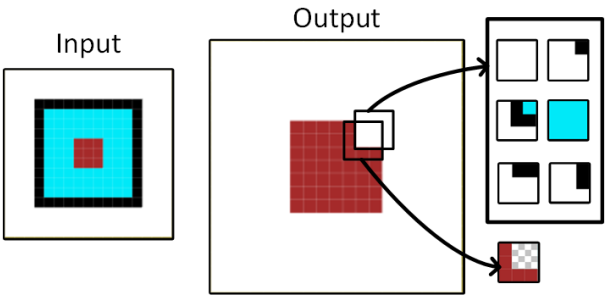

Figure 3–5: Example of overlapping patterns with masks

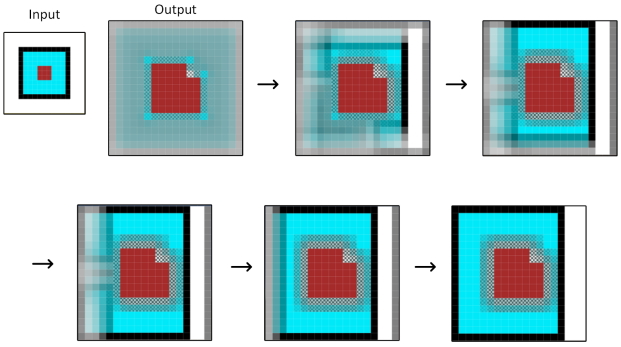

Figure 3–6: Execution of the algorithm using masks on output boundaries. Masks are represented as transparent tiles.

see that there is no pattern that contains a path on the tiles surrounding the obstacle. As a result, we can see an execution of the algorithm in Figure 3–6, which succeeds to generate a cyclic path.

3.3.1 Stretch tiles around obstacles and boundaries

The way masks are defined makes it possible to have white areas around an obstacle even if the input doesn't specify this. This is somewhat crucial when we are dealing with actual game maps, in which most of the time obstacles and boundaries are defined the same way. In maps that are enclosed in boundaries, the effect of using masks means that some stretch space can be filled on the boundaries. If this happens, the algorithm will start filling stretch tiles around boundaries, instead of starting to fill stretch space around obstacles. Depending on the ratio between stretch tiles and white tiles from the input, this can yield in cases where the whole map is filled with blue tiles and no path is generated. Moreover, since white tiles can go around obstacles, another unwanted effect can occur : some paths are generated, but no path is being generated around obstacles. To deal with these issues we've added options to force stretch tiles around obstacles and to forbid them around boundaries.

In order to achieve this, we must first detect what is considered a boundary and what is considered an obstacle. To do so, we store all boundary tiles in a queue, and then we retrieve tiles from the queue and determine if the tile is an obstacle or not. If it's an obstacle, we add its neighbours to the queue. If it's not an obstacle, then we flag this tile as a boundary. In the process, we store all obstacle tiles that we've extracted from the queue in a *visited* hash set and use this in a second pass to determine the boundaries around obstacles. If an obstacle tile has been visited, it's not considered as an obstacle, but rater as a

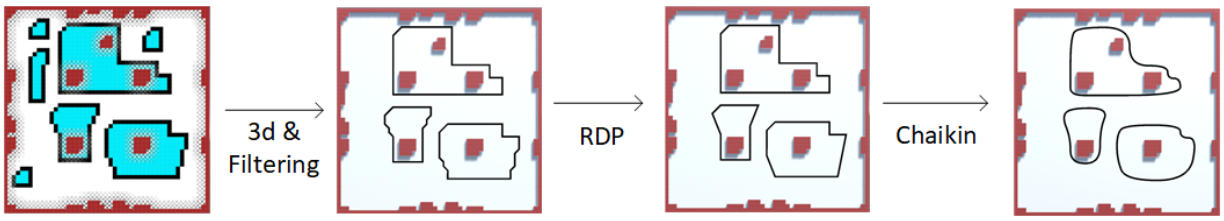

Figure 3–7: Execution of all post-processing steps in their respective order. Output map is "arena" from the *Dragon Age: Origins* benchmark set

boundary. If it has not been visited, then we apply the same process described above to find the boundaries around an obstacle. This gives us a list of both map boundaries and obstacle boundaries. We can use this in an initial pass of the algorithm to filter out patterns on boundaries with stretch space and patterns on obstacle boundaries that contain white space. We can use again the notion of masks to create appropriate masks to determine whether tiles are of a stretch space color or of a white color.

These options are allowed independently of each other because of the fact that some maps could contain obstacles that are too close to a boundaries. In this case, allowing both options would create a contradiction and make the algorithm fail all the time.

3.4 Post processing

The heuristic nature of our approach is augmented with some post-processing steps to filter out degenerate constructions, help visualize results, and generally better prepare paths for integration into a game environment. Figure 3–7 shows an execution flow for of all processing steps. With the exception of 3D conversion, all steps mentioned below are optional and can be used independently.

3.4.1 Conversion to 3d map

The first post-processing step is to convert the 2d generated image into a simple game map. We instantiate a plane to make the floor, and then we instantiate unit sized cubes for each position where there is an obstacle. Then, we must convert the black tiles that represent a path segment into actual paths. We do this by iterating over all tiles and when we encounter a black tile, we process it by adding it to a new path and by finding all neighbouring tiles that form this path, using an 8-way navigation system. Since tiles surrounding a corner tile can have more than one neighbour, it is important to start looking at non-diagonal neighbours before diagonal neighbours. We then store the path as a list of points (vertices) centered around their related position on the grid.

3.4.2 Path filtering

As a result of using overlapping patterns and masks, the generated outputs usually contain a lot of smaller paths that are not necessarily appealing. We enable the option to filter out paths that are smaller than a certain length threshold that can be selected by users.

3.4.3 Path Simplification

Depending on the input patterns, some paths can get complicated and have a significant amount of redundant turns or detours. We alleviate this issue by adding a post processing option to remove these path segments. To do so, we use the *Ramer-Douglas-Peucker algorithm* (RDP) [21, 6] to remove redundant vertices. Algorithm 1 shows an iterative implementation of the algorithm. Moreover, since we alter the generated paths, we must account for intersections between new path segments and obstacles. We do this by iterating over each obstacle cube and calculating the line-rectangle intersection with our new path segment and the obstacle (represented by function *CrossObstacle*() in algorithm 1). As shown in the algorithm, if the new segment crosses an obstacle, we disregard the new line segment as if the tolerance threshold was not attained. Therefore, as the algorithm goes, we continue to look for path simplifications on both sides of the vertex with maximum distance.

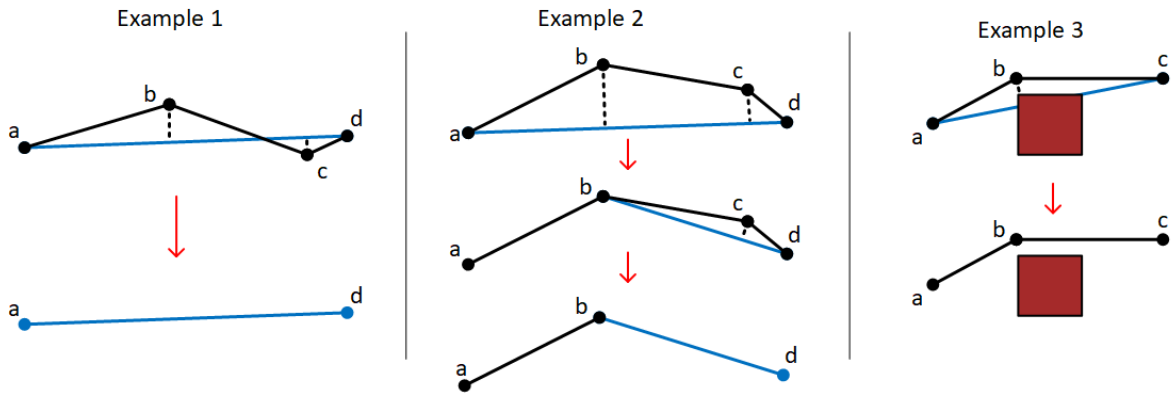

Figure 3–8: Examples of our implementation of the RDP algorithm

Figure 3–8 shows the main 3 cases that typically happen when using our modified version of RDP. In example 1, we look for the farthest point from the line between a and d, which in this case is b. Then, since the distance between b and the line is smaller than the threshold, we collapse all points between a and b. In example 2, b is bigger than the threshold, therefore we split the work in 2 and start looking

on both sides of b. c being the farthest point from the line produced by b and d, we collapse it since it's smaller than the threshold. Finally, example 3 shows that even if the point should be collapsed, if there is an obstacle that crosses the resulting line then we don't collapse the point.

Algorithm 1 Ramer-Douglas-Peucker (iterative version)

```
function RDP(List(of Point) path, float tolerance)
    queue ← (path.first(), path.last())                    ▷ queue of tuples
    while not queue.isEmpty() do
        (begin, end) ← queue.dequeue()
        maxDistance ← -1
        for each Point p between begin and end do
            distance ← LineDistance(p, (begin, end))
            if distance > maxDistance then
                maxDistance ← distance
                tempPoint ← p
        crossObst ← CrossObstacle(begin, end)
        if maxDistance ≤ tolerance and not crossObst then
            Remove points between begin and end
        else
            queue.enqueue((begin, tempPoint))
            queue.enqueue((tempPoint, end))
```

3.4.4 Path Smoothing

Paths generated with our algorithm naturally have sharp turns and corners since they are generated using 3×3 discretized patterns. We alleviate this problem by smoothing paths using Chaikin's curve generation algorithm [4]. The algorithm mainly replaces the points that form a path segment by a new set of points generated by applying masks on the previous points. For each edge going from P_i to P_{i+1}, 2 new points Q_i and R_i are generated respectively by calculating $3/4P_i + 1/4P_{i+1}$ and $1/4P_i + 3/4P_{i+1}$ [13]. We then replace the corner between R_{i-1} and Q_i by those points. Algorithm 2 shows details of our implementation. Similarly to the path simplification step, we make sure that new line segments formed by R_{i-1} and Q_i don't cross obstacles. If they intersect with an obstacle, we simply reinsert the original corner point between them in the newly computed list of points. Therefore, no smoothing is done, but R_{i-1} and

Q_i are still added to the new list of points in addition to the original point. This will let us apply a smaller smoothing between those 3 points at the next iteration.

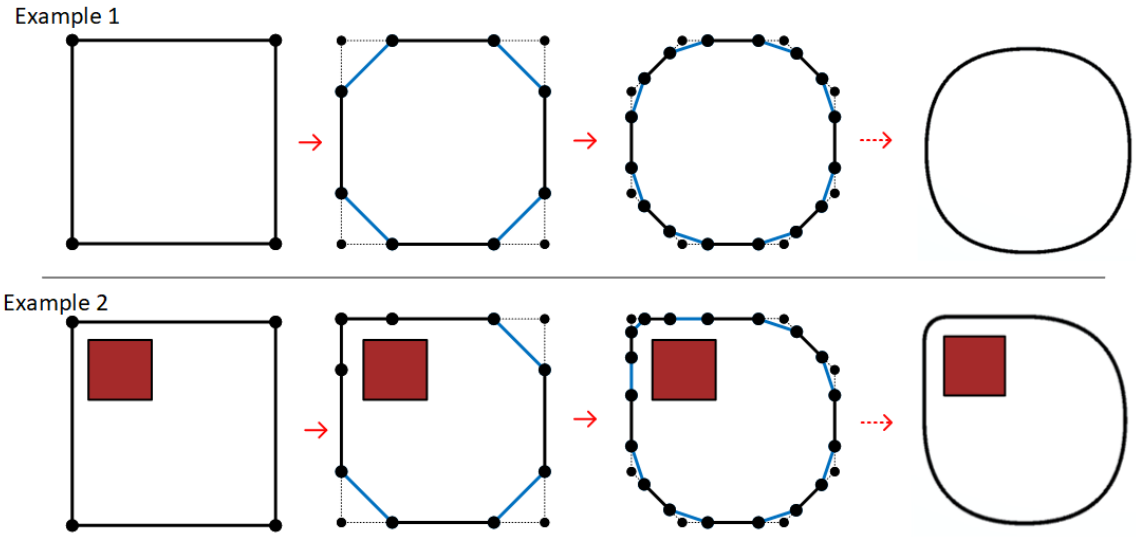

Figure 3–9: Examples of our implementation of Chaikin's curve generation algorithm. 4th column is the result after 5 iterations, where the result is similar to the limit surface.

Figure 3–9 Shows the 2 main outcomes that are generated by the algorithm. Blue lines indicate new path segment created during the current iteration, and dotted lines indicate path segments from previous iterations. In example 1, smoothing occurs normally, where the first 3 columns represent the 3 first iterations and the last column represents the final result after 5 iterations. In example 2, the top left corner is not smoothing out since the resulting path segment would cross the obstacle. In this case, we simply keep the new added points and insert back the point from the previous iteration so that smoothing occurs correctly in the next iteration.

Algorithm 2 Chaikin's curve generation algorithm

```
function CHAIKIN(List(of Point) path, int iterations)
    prev ← path
    current ← new List(of Point)
    r_prev ← NULL
    while iterations > 0 do
        current.clear()
        for i ← 0 to prev.Count − 1 do
            q_i ← 0.75 * prev[i] + 0.25 * prev[i + 1]
            r_i ← 0.25 * prev[i] + 0.75 * prev[i + 1]
            if i > 0 and CrossObstacle(r_prev, q_i) then
                current.add(prev[i])          ▷ Insert point from previous iteration
            current.add(q_i)
            current.add(r_i)
        if CrossObstacle(r_prev, current[0] then)
            current.add(prev[0])
        current.add(curr[0])                  ▷ Close the loop
        Switch references between current and prev
        iterations ← iterations − 1
```

CHAPTER 4
Experimentation

In this section, we briefly describe the experiments that we've made and the tools that we've used with them. We discuss in depth about different results of the algorithm when used with different options and setups on both inputs and outputs. We assess performance results on small to large-sized output maps, and then discuss about the limitations of the algorithm.

4.1 Experiment

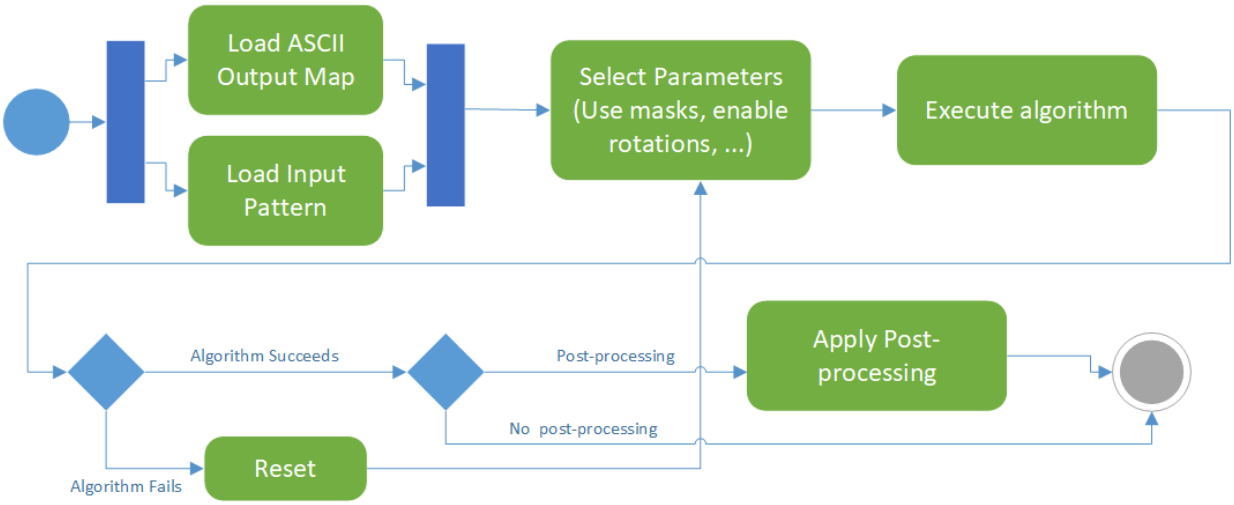

Figure 4–1: Workflow of high-level components of the algorithm

We use the same setup as in the Overlap model of the WFC algorithm. We use images to represent inputs and outputs. Each pixel corresponds to a position in the map. We chose to use patterns of size 3x3, which captures best the neighbouring relation between patterns with a reasonable execution time. Any value bigger than that would slow down the algorithm considerably.

The algorithm was implemented in the Unity Editor. We generate the input/output 2d maps using small monochromatic sprites to represent tiles and use them to 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 map, along with an agent that can walk through the generated paths. Figure 4–1 shows a high-level workflow of the algorithm, from loading the input and output images to applying post-processing.

The algorithm was tested with map bundles from the Moving AI benchmark sets [28]. Most of these benchmark sets are actual game levels from games such as *Dragon Age: Origins* and *Baldur's Gate II*. These maps give us a test bed that shows how the algorithm would interact using maps from actual games.

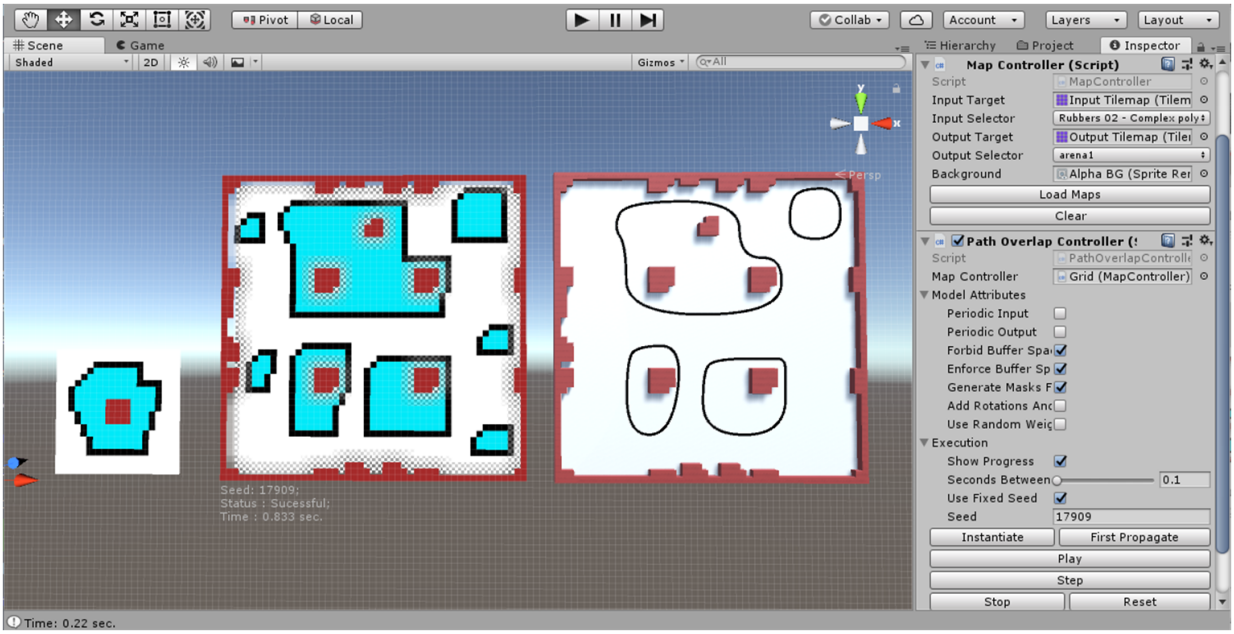

Figure 4–2: Screenshot of the Unity tools developed. The image on the left represents the input sample. The center image is the result of the main algorithm. The image on the right is the result after post-processing.

4.2 Tools

The Unity engine was used to develop the framework. We used a separate thread to execute the algorithm and used *Tilemaps* to show the image-based representation of the algorithm. We used Unity's *Job system* to implement efficient path simplification and path smoothing during post-processing. To load maps from Moving AI's benchmark sets, we implemented a simple loader that loads maps in ASCII format and convert tiles to either obstacle of free space.

4.3 Results

In this section, we discuss the results of our algorithm. We show how different options affect the outcome of the generated output, and we present results using different combinations of input and output.

4.3.1 Stretch space

We experimented with two different types of input: Inputs using only paths and obstacles, and inputs using stretch space between paths and obstacles. Both types of input generate interesting paths, however there is more randomness to paths generated without using stretch space.

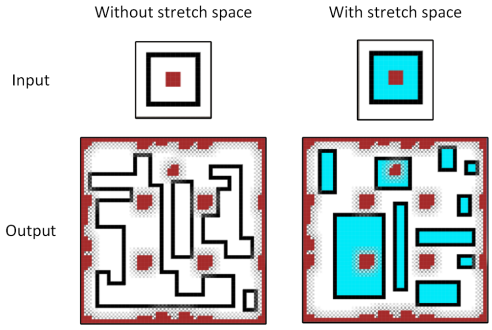

Figure 4–3: Comparison between output generated with and without stretch space

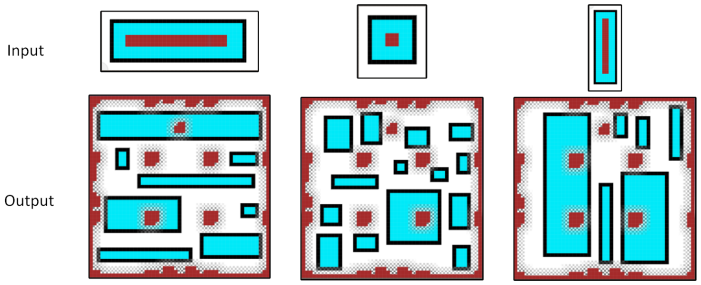

Figure 4–4: Comparison between output generated with different types of input. Rotations and reflections are disabled in order to see the differences

As shown in figure 4–3, using stretch space makes the output have only rectangular paths, whereas having a similar input without stretch space generates paths with more complex shapes. This is due to the fact that the algorithm works by letting patterns overlap if their color matches. Therefore, having different colors inside and outside the cyclic path constrain the number of overlaps between each pattern on the path's boundary. For example, the output generated without stretch space contains inner corners (corners that make the shape of the path non-convex), which are possible because of the fact that any straight line pattern can overlap with any type of corner. These types of match cannot be possible using inputs with stretch space since the colors on both sides of the line wouldn't match the ones on both sides of the corner. This then gives us two different ways of generating paths with one input, where one way is less constrained and generates a broader range of outputs, and the other way is stricter but is more similar to the shape of the path in the input.

4.3.2 Frequency distribution

As discussed above, having stretch space in input paths gives us more control over the generated output. Using weights to select patterns also helps us control the outcome of the algorithm. Our algorithm uses the same pattern selection mechanism as in the WFC algorithm : when we extract patterns from the input, we also store the frequency of each pattern in order to produce a frequency distribution. This

distribution is then used to calculate the entropy of given tiles in order to select the next tile during the **observe** step, and is used again to select a given pattern for the selected tile based on this distribution. This means that we can craft inputs in order to generate paths based on the frequency of patterns from the input. As we can see in figure 4–4, the horizontal line patterns from the input have a higher frequency since they occur more often, and this is reflected in the generated output. On the other hand, the squared path input generates an output that contains a more uniform set of horizontal and vertical rectangles, and squares.

In addition, tuning the ratio of white tiles and stretch tiles in the input sample will affect the output in terms of how many paths will there be and how big paths will get. A good ratio would be to have a similar frequency for both white tiles and stretch tiles, but it's always possible to grow or reduce one or the other by either resizing the input image to get more white tiles or by making the path bigger to get more stretch tiles. In even more complex cases, we can use obstacle tiles to cover any of the 2 types of tiles to tweak its frequency.

4.3.3 Uniform distribution

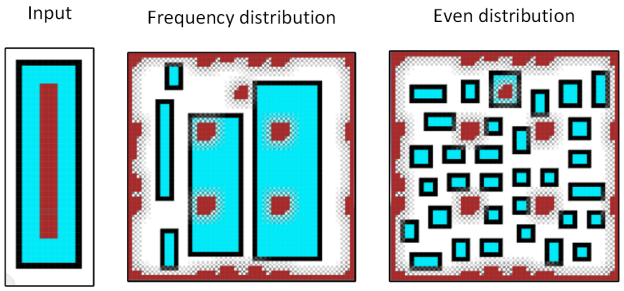

Figure 4–5: Comparison between output generated with different types of distribution. Rotations and reflections are disabled.

Alternatively, we also enable the option to select weights and entropies based on a uniform distribution (i.e. randomized weights), where the frequency of a pattern is not taken in consideration. During the process of selecting a new tile, using this even distribution implies that the calculation of entropy is analogous to selecting the tile that has the least possible patterns available. In figure 4–5, two outputs are generated from the same input using the same seed. The first output uses the frequency distribution and the second output uses an even distribution. As we can see, the first output generates vertical rectangles that look like the input, given that the frequency of vertical paths is higher than the frequency of horizontal paths.

This behaviour is not reproduced in the second output, since the distribution of pattern is the same for each pattern. This is also reflected in the fact that stretch space tiles are less frequent than in the first output. Therefore, it produces significantly smaller paths.

4.3.4 Different input/output combinations

To support the argument that the algorithm works for different combinations of input and output, we show generated 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 new input samples to show examples of different outcomes based on significantly different inputs.

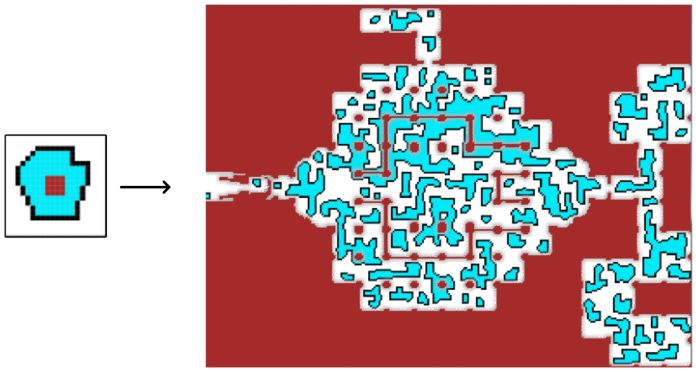

Figure 4–6: Example of execution with the Arena2 map.

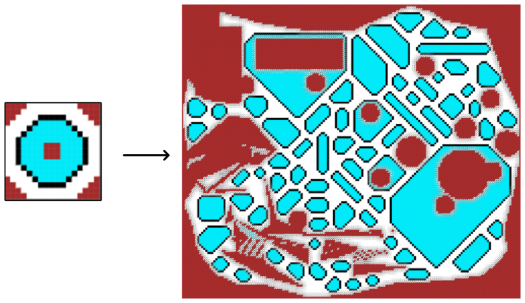

Figure 4–7: Example of execution with the Lak519d map.

Figure 4–6 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. The input tries to encapsulate every possible path movement inside a 3×3 image, and as a result we see paths with a wider range of different path segments.

Figure 4–7 shows a result on the *Lak519d* map using an octagon as the input. For this input, enabling rotations and reflections is irrelevant as they would not add additional patterns. This output has less varying path segments compared to figure 4–6 but more than with a simple square input. This gives the output a significant level of freedom. Finally, since the input is an octagon, a lot of white space can be filled in corners and this can affect the result in the sense that it would generate more blank space. We alleviated this issue by filling the corners with obstacle tiles in order to reduce the frequency of white tiles. Since this execution uses masks, adding these obstacle tiles in the input would form new patterns but they won't add anything relevant to the generated output. Moreover, since the obstacles are fixed on the

output, the fact that these obstacle patterns will have a higher frequency will not change the way the other patterns are selected when using a frequency distribution.

4.4 Performance

In this section, we present a few optimizations introduced to achieve better performances, and we show some results to compare performance based on varying factors.

4.4.1 Optimization

We integrated optimizations introduced by Fehr and Courant [9] into our algorithm to greatly improve the performance of it. Roughly, these modifications include storing the entropies of each position in an array and modifying them when filtering out patterns, and calculating overlaps only with the four non-diagonal neighbours of each pattern, instead of calculating overlaps on all possible overlaps of a $3x3$ pattern (4 overlaps instead of 8). Moreover, we experimented with Unity's *Job System* and used it to parallelize the post-processing path generation in order to improve its performance.

4.4.2 Result

Performance of the algorithm is mainly affected by 2 factors: the number of patterns extracted from the input and the size of the output. To measure performance based on these 2 factors, we run the algorithm 10 times on each test set, which involves different inputs and outputs, and compare the change in average execution time. We discarded the first execution for each test, since it could warm up the cache and give biased results.

Table 4–1: Average time in seconds for varying inputs

Input	Patterns	Average Time	Standard Deviation
Simple square	304	49.59	0.8205
Octagon	320	53.11	1.0242
Poly 1	348	58.71	0.4363
Poly 2	376	61.02	1.3880

To show the impact of the change in the number of patterns from the input, we select one output, and measure the execution on different inputs that generate different number of patterns. Table 4–1 shows executions with different inputs using the output map *Lak519d*. As shown by the average time column, the more patterns the input contains, the more time it takes to run.

Table 4–2: Average time in seconds for varying outputs

Output	Size	Average Time	Standard Deviation
arena1	49 x 49	0.89	0.0116
orz000d	79 x 137	7.61	0.1181
lak519d	168 x 145	61.02	1.3880
arena2	281 x 209	78.43	0.4308

To show the effect of changing the size of the output map, we perform tests on maps of different sizes while keeping the same input. Table 4–2 shows 4 tests on different outputs using the input image *Poly 2* (the same input used in figure 4–6). Results show that the execution time grows relative to the size of the output map. Despite this variation, execution on significantly large maps is done in a reasonable amount of time, considering that the algorithm would be best used to generate paths offline (i.e. generating paths while designing a game, not while the game is running).

CHAPTER 5
Related work

In this section, we discuss related work on the WFC algorithm. We then discuss about procedural content generation and how it is related to our work. Finally, we discuss other methods of generating random paths using generated roadmaps.

5.1 WFC

Our work relies heavily on the WFC algorithm as it is an adaptation of the WFC overlap model. The WFC algorithm has seen a few notable ports and forks. Karth and Smith released a paper on WFC in which they describe in depth the algorithm, discuss its influence since it was revealed and contextualize it in terms of constraint solving [14]. As mentioned earlier, Fehr and Courant have developed a faster version of the algorithm, written in C++. This version mainly introduces 3 modifications, including a reduction of the number of overlaps per pattern that the algorithm processes and storing entropy values in arrays instead of computing them in every iteration [9]. These optimizations were also added to Gumin's WFC repository and they are said to improve the original algorithm by an order of magnitude. Lefebvre et al. have experimented with hiding messages in the generated images from the tiled model [17], with an emphasis on the fact that embedded images are not distinguishable from normal generated images. The seed used to instantiate the model can act as a password to decode the message. Among a few others, Gumin also developed a 3d version of the tiled model to generate 3d images using tiled voxels [11].

A few ports of the algorithm were also made in Unity. Stalberg Implemented an interactive version of the simple tiled model made in Unity that runs in a web browser [27]. This interactive version shows a set of tiles on the left hand side (titled *Modules*) that are used to generate outputs on the right hand side (titled *Slots*). Increasing the solving speed enables a dynamic demonstration of how observed states are being filtered out, until one tile is left per position, which shows the final outcome. By clicking on the borders of a tile in the modules section, we can visualize all other borders that are compatible with it (defined similarly as in the xml neighbours section in the simple tiled model from the original WFC model). Moreover, we can disable any tile in order to customize the complete set of tiles used by the algorithm.

Parker Implemented a complete set of tools to use the WFC algorithm in Unity by replacing bitmaps by `gameobjects` [20]. This implementation handles both Overlap and Simple tiled models as per the original WFC, and also adds a TilePainter module which helps designing input samples in a painting-like manner. The fact that the algorithm uses `gameobjects` instead of bitmaps means we're not restricted to using 2d assets, therefore giving users the opportunity to create things such as 3d top-down isometric game maps.

5.2 PCG

Path generation itself is generically part of the field of *Procedural Content Generation (PCG)*. This field applies mostly to games and consists of algorithms that generate content such as textures, levels, terrains, quests, items, or other game-related concepts. The key idea is that the content is created in a random and algorithmic way with minimal user input, in contrast to content being manually crafted by game designers. Many generative techniques have been explored, including constraint models, based on understanding the rhythm or pacing of player experience. Smith et al.'s *Tanagra* tool for 2D platformer levels [25] explores this idea by letting users place pacing constraints on 2d platformer levels and letting the algorithm fill in the gaps by randomly generating content. This is a mixed-initiative design, where both the generative algorithm and humans collaborate together to generate playable 2d platformer levels. Moreover, machine-learning approaches can demonstrate it is possible to construct levels without domain-specific knowledge [26]. Although less common, PCG algorithms have been applied to path generation previously as well. Xu et al. [30] developed a tool to randomly place guards and cameras in stealth game levels. Their approach is concentrated on visibility and coverage properties, aiming to heuristically control level difficulty. More information on PCG can be found in Shaker et al.'s book [23].

5.3 Roadmap Generation

Higher-level navigation systems, such as roadmaps, can also be used as a basis for creating randomly wandering NPC motion, presenting seemingly goal-directed behaviour by defining random path choices at a larger scale. Paths based on optimal roadmaps, however, tend to scrape obstacles [19], which is both not human-like, and a source of error in game simulation. For games, NPC movement is better shifted away from obstacles. Convex decompositions or *navigation meshes* can be used to give more movement freedom. Triangulations are a natural choice in this respect, and many systems build roadmaps from the triangulation dual. Demyen and Buro [5], for instance, show that it is efficient to build a graph for pathfinding by using constrained Delaunay triangulations, a technique also used by Lens and Boigelot

[18] to build roadmaps with controlled distance from obstacles. Amato and Wu [1] generate roadmaps by randomly selecting candidate points on the boundaries of a shape and using the midpoint between lines connecting pairs of candidates to form a roadmap.

In defining routes that well avoid obstacles, roadmap algorithms tend to approximate the *medial axis skeleton* [2], a simplified graph that defines points equally distant from internal obstacles. This has a natural appeal for roadmaps, and thus pathing, and use of the medial axis has been experimented with by many authors [29, 10]. Singh [24] has taken this concept even further and used medial skeletons in a 3D environment (2D map with 3rd dimension as time) to generate roadmaps in a dynamic environment for stealth games.

Other pathfinding systems also lend themselves to path generation. The hierarchical pathfinding algorithm developed by Botea et al. [3] creates an abstract graph of a map by separating the map into clusters, which could be adapted to be used generatively. Search graphs of various forms can be used in a similar fashion. *Rapidly exploring random trees (RRTs)*, for example, can be used to generate random paths in fixed maps [15]. The idea of RRTs is to grow a randomized tree structure starting from a certain point in order to fill the map with many different random paths. This could be adapted to generate random meandering paths by setting a starting point and letting the RRT search grow, perhaps also including constraints on how it connects new nodes for controlling variety.

In contrast to pathfinding, dynamic systems can also be used to randomly move agents around obstacles. One can use steering behaviours [22] along with obstacle attraction [8] to make an agent move around an obstacle. In a similar fashion, using flow fields for crowd pathing [7], one can generate flow fields that gravitate around obstacles and make agents move along those flow fields.

CHAPTER 6
Conclusion & Future Work

In this paper, we have presented a modification to the Wave Function Collapse algorithm that handles random path generation using input texture samples and fixed output maps. The algorithm extends the original WFC algorithm by adding options to handle issues such as generating paths similar to the input by using stretch space, and using masks to enable new patterns based on the shape of obstacles found in the output map. We have included a few post-processing options to filter out small paths and to make remaining paths smoother. We presented a convenient tool developed in Unity that lets any user, with our without programming knowledge, experiment with the algorithm and use it for path generation purposes.

Using a texture generation algorithm to generate paths has a few inherent limitations. First of all, the algorithm doesn't have a way to specify a global constraint, in the sense that we cannot enforce a constraint on the output to satisfy the global shape of the input path. This can be seen notably in figures 4–6 (page 21) and 4–7 (page 21). The patterns are chosen based on their frequency, but this doesn't assure that the global shape of the input is respected to a certain degree. Moreover, since paths are defined by pixels, intersecting paths introduce ambiguity in converting the 2D image into waypoints for a game level. It is possible to define an input image with intersecting paths, however it is not handled in a deterministic way during post-processing. This is due to the fact that the algorithm doesn't know which pixel to choose as the next path segment when processing the area of the image where paths do cross.

Future work on the algorithm could include using heuristics on intersecting tiles to determine which neighbour is the next path segment. Further development includes the addition of different constraints to make paths meet certain evaluation criteria. Examples of criteria can include the number of turns taken, the amount of obstacles that paths go along, or even more complex constraints. We would also be interested in solving the problem of not having global constraints to enforce the global shape of the input path. One way to shift toward this idea would be to augment our algorithm with a frequency distribution on pairs of patterns. This would give the algorithm more knowledge about the selection of a neighbouring pattern based on the previously selected pattern.

Aside from WFC, one could think of using one of the roadmap generation methods presented in the previous section and augment it to be used for the same purpose of our algorithm, to generate random meandering paths around obstacles. Moreover, instead of using patterns, one could use simple obstacle avoidance and obstacle follower constraints to build a complete syntax model that would drive an algorithm to generate paths. This would still follow a similar behaviour as in our algorithm since it would be heavily constraint-based. Finally, given that those options could be developed, it would be interesting to then compare results of all these different solutions and measure which algorithm performs better, in terms of similarity to the desired initial behaviour and the resulting outcome of each algorithm. This would give us a better in-depth analysis of the general problem of path generation.

REFERENCES

[1] N. M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 1, pages 113–120 vol.1, Apr 1996. doi: 10.1109/ROBOT.1996.503582.

[2] Harry Blum. A transformation for extracting new descriptors of shape. In Weiant Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, pages 362–380. MIT Press, Cambridge, 1967.

[3] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of game development*, 1(1):7–28, 2004.

[4] George Merrill Chaikin. An algorithm for high-speed curve generation. *Computer Graphics and Image Processing*, 3(4):346 – 349, 1974. ISSN 0146-664X. doi: https://doi.org/10.1016/0146-664X(74)90028-8. URL `http://www.sciencedirect.com/science/article/pii/0146664X74900288`.

[5] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 942–947. AAAI Press, 2006. ISBN 978-1-57735-281-5. URL `http://dl.acm.org/citation.cfm?id=1597538.1597687`.

[6] David H. Douglas and THomas K. Peucker. 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, 1973. doi: https://doi.org/10.3138/FM57-6770-U75U-7727.

[7] Elijah Emerson. Crowd pathfinding and steering using flow field tiles. *Game AI Pro: Collected Wisdom of Game AI Professionals*, pages 307–316, 2013.

[8] Brett R Fajen, William H Warren, Selim Temizer, and Leslie Pack Kaelbling. A dynamical model of visually-guided steering, obstacle avoidance, and route selection. *International Journal of Computer Vision*, 54(1-3):13–34, 2003.

[9] Mathieu Fehr and Natanaël Courant. fast-wfc. https://github.com/math-fehr/fast-wfc, 2018. Github repository.

[10] L. J. Guibas, C. Holleman, and L. E. Kavraki. A probabilistic roadmap planner for flexible objects with a workspace medial-axis-based sampling approach. In *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems: Human and Environment Friendly Robots with High Intelligence and Emotional Quotients*, volume 1, pages 254–259, 1999. doi: 10.1109/IROS.1999.813013.

[11] Maxim Gumin. Basic3DWFC. https://bitbucket.org/mxgmn/basic3dwfc/src/master/, 2016. Bitbucket repository.

[12] Maxim Gumin. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse, 2016. Github repository.

[13] Kenneth I. Joy. Chaikin's algorithms for curves. http://graphics.cs.ucdavis.edu/education/CAGDNotes/Chaikins-Algorithm/Chaikins-Algorithm.html, 2000.

[14] Isaac Karth and Adam M. Smith. WaveFunctionCollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 68:1–68:10, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5319-9. doi: 10.1145/3102071.3110566. URL http://doi.acm.org/10.1145/3102071.3110566.

[15] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical Report TR 98-11, Iowa State University, 1998.

[16] Gregory F. Lawler. *Intersections of random walks*. Birkhäuser, 1991.

[17] Sylvain Lefebvre, Li-Yi Wei, and Connelly Barnes. Informational texture synthesis. https://hal.inria.fr/hal-01706539/file/infotexsyn.pdf, March 2018. URL https://hal.inria.fr/hal-01706539. working paper or preprint.

[18] Stéphane Lens and Bernard Boigelot. From constrained Delaunay triangulations to roadmap graphs with arbitrary clearance. *CoRR*, abs/1606.02055, 2016. http://arxiv.org/abs/1606.02055.

[19] Nils J. Nilsson. A mobile automaton: An application of artificial intelligence techniques. In *1st International Conference on Artificial Intelligence*, pages 509–520, 1969.

[20] Joseph Parker. unity-wave-function-collapse. https://selfsame.itch.io/unitywfc, 2016. itch.io page.

[21] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244 – 256, 1972. ISSN 0146-664X. doi: https://doi.org/10.1016/S0146-664X(72)80017-0. URL `http://www.sciencedirect.com/science/article/pii/S0146664X72800170`.

[22] Craig W Reynolds. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782, 1999.

[23] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[24] Dhirendra Singh. Using medial skeleton for path finding in dynamic stealth games. Master's thesis, McGill University, Montréal, Canada, August 2015.

[25] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216. ACM, 2010.

[26] Sam Snodgrass. *Markov Models for Procedural Content Generation*. PhD thesis, Drexel University, 2018.

[27] Oskar Stalberg. Wave - by Oskar Stalberg. `http://oskarstalberg.com/game/wave/wave.html`. Retrieved July 12, 2018.

[28] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012. URL `http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf`.

[29] S. A. Wilmarth, N. M. Amato, and P. F. Stiller. MAPRM: a probabilistic roadmap planner with sampling on the medial axis of the free space. In *Proceedings 1999 IEEE International Conference on Robotics and Automation*, volume 2, pages 1024–1031 vol.2, 1999. doi: 10.1109/ROBOT.1999.772448.

[30] Qihan Xu, Jonathan Tremblay, and Clark Verbrugge. Generative methods for guard and camera placement in stealth games. In *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2014)*, October 2014.