Sketch 2 Tilemap: Procedurally Generated Tilemaps from a Drawing

Cole Sohn
Stanford University
csohn@stanford.edu

Abstract

Sketch-2-Tilemap generates a 3D video game level tilemap from an input sketch. This approach allows users to quickly design and prototype tiled levels by automating the manual work required to place tiles in a game engine. Sketch-2-Tilemap first generates an irregular quadrilateral grid from an input sketch where the bounds of the grid follow the contours of the sketch. Next, we implement an extended Wave Function Collapse (WFC) algorithm that works on irregular grids to choose tile placements. Finally, tiles are transformed to the appropriate grid spaces using lattice deformation to generate an output game map. Faces on an irregular grid with mismatching orientations produce problems with WFC outputs. We show this through experimentation and propose a breadth-first orientation method to optimize face alignments for more varied output tilemaps.

1. Introduction

Building large video games environments can be costly and time consuming. For many games, a large portion of this effort goes into creating art assets, which include 2D sprites or 3D models. In order to reduce costs, it is often important to create game art that is reusable and modular. Modular game assets can be fit together in new ways such that a much larger level can be synthesized from a limited number of different assets.

To maximize modularity, many games are built on a grid. Modular game assets are called tiles when they can be fit together on a grid. Grid-based games are most commonly built on a regular square grid [21], but there are many deviations from this such as triangular, hexagonal, and irregular grid-based games. [11] [4] [17]

A level designer of a grid-based game may begin designing a map with a top-down sketch of the overall level layout. Once they are satisfied by their design, they build the level using a game engine which allows for the manual placement of modular tiles. This process can be time-consuming because it requires manual tile placement by the designer. This manual effort can slow down the production of a level and inhibit creativity during the prototyping process.

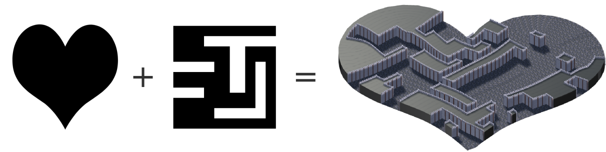

Figure 1. Sketch-2-Tilemap Input and Output

In this paper, we propose Sketch-2-Tilemap, a workflow that utilizes an extended version of the Wave Function Collapse algorithm[3] which gives the user control over generated levels. The user specifies a drawing representing the shape of the level, and a pattern which drives the structure of the level. Sketch-2-Tilemap automatically generate levels following the constraints specified in the input pattern on an irregular grid generated from the input sketch. Sketch-2-Tilemap automates the process of manual tilemap creation and allows the user to rapidly prototype and modify new level designs.

2. Related Work:

2.1. Tilemaps

Generalized tilemaps allow developers to generate unlimited unique gameplay spaces with a limited number of art assets while manually created art assets must align with specific placements and uses. Tilemaps also allow developers the freedom to swap out and make updates to tiles as needed. This can speed up the level design process by allowing designers and artists to work in tandem, and allow for large artistic changes later in development.

There are existing methods to reduce the time it takes to place tiles, but they are limited in the time they save or the ability for an artist to control the output. A common method of procedural tile selection is Adaptive Tilesets [13], which allow for automation of the process of selecting tiles, but they require unique setup for different instances and provide only a minor shortcut to the overall goal of level generation.

Tilemaps are used by many Procedural Content Genera-

tion (PCG) algorithms to automatically generate gameplay spaces through automated tile placement. Fully-procedural methods to generate tilemaps include approaches using Cellular Automate [6] and Marching Cubes [2]. These suffer from not giving the user enough control over the structure of the output.

2.2. Procedural Content Generation

Procedural Content Generation (PCG) is the automated generation of game content. PCG is a broad term that is applied to various algorithms that generate game content. These algorithms tend to fall into categories such as grammars, search, constraints, and solvers [9]. Many tile-based level generators such as Marching Cubes and WFC are constraint-based algorithms, which place tiles based on sets of rules.

Mixed-Initiative Design is a term used by Summerville et al. [18] to represent PCG algorithms that generate content with a mix of automation and user control. WFC would be considered Mixed-Initiative Design because it takes a user-generated pattern as input. Outputs generated with Mixed-Initiative Design algorithms benefit from the artistic and structural knowledge of a human designer as well as the computation power and unexpected results of a procedural algorithm.

2.3. Wave Function Collapse

Wave Function Collapse (WFC) is a constraint-based algorithm initially proposed in 2016 by Maxim Gumin [3]. Initially intended for texture synthesis, WFC has since been utilized as a PCG algorithm used to place 2D and 3D tiles to generate game levels.

WFC can syntehsize a large level from a small example pattern provided by a user. The output level matches the constraints and structure followed by the input pattern. While allowing the user to build large levels procedurally, the base implementation of the WFC algorithm proposed by Maxim Gumin [3] requires that the tiles be placed on a standard rectangular grid.

In its base form, WFC starts with an $n \times n$ input array of integers A. Each integer in the array represents the index of a color or tile used in generating the final output. Input array A represents some pattern or structure that the user would like to have control the final output. The algorithm outputs B, an $m \times m$ array which represents a texture or tilemap synthesized following the patterns and structure found in A. m can be greater than n, allowing a larger texture or tilemap to be synthesized from a small input. For examples of Gumin's initial WFC implementation, see Repository [3].

WFC first initializes B to 0. Each element of the array has an assigned array T of size equal to the number of tiles or colors available in the input. Elements of T are binary values representing if that tile can be placed in those indices of B. The algorithm iteratively selects tiles for elements of B, setting all but one elements of T_b to 0. This step is called a collapse. At each iteration, the algorithm propagates changes due to the collapse of the selected tile based on the constraints found in A. The algorithm repeats this process until all elements of B have a single 1 value in T_b, or until some T_b has all elements equal to 0, at which point the algorithm has reached a contradiction must restart.

This algorithm has been extended for various projects which incorporate three-dimensionality and various irregular grids. Marian Kleineberg used WFC extended to 3D to create an infinite city generator [7], Ryan James Smith's implemented a 3D extension of WFC in Houdini [15], and Oskar Stalberg has published many examples of WFC used to generate spaces in his games including procedurally generated islands in the game Bad North [16] and procedurally generated towns with user input in the digital toy Townscaper [17].

Tobias Nordvig Møller addressed the shortcomings of WFC in his 2019 master thesis: Expanding Wave Function Collapse with Growing Grids for Procedural Content Generation [10]. Traditionally, WFC is run on a regular grid of a rectangular shape. This implementation extends WFC by warping a simple grid to an arbitrary shape using a growing grid algorithm. This implementation deforms tiles placed by WFC using blendshapes. Møller's extension of the WFC algorithm inspired the implementation of this paper. Sketch-2-Tileset notably diverges from Møller's implementation by being generalizable to irregular grids of hand-drawn shapes. As a result, Sketch-2-Tileset follows drawn topology, matches the contours of hand-drawn shapes, and allows for holes in the input shape. A comparison of these approaches can be seen in Fig 14.

3. Methods:

Sketch-2-Tilemap converts an input sketch to a tiled level in three stages. First, an irregular quadrilateral grid is generated from an input sketch. Second, an extended wave function collapse algorithm assigns tile indices to quads based on an input constraint array. Third, square or cube tiles are deformed to match the position and shape of their assigned space on the grid. We will investigate these approaches in detail.

3.1. Image to Irregular Grid

Sketch-2-Tilemap takes as an input a binary image representing the shape of a level. First, contour detection is performed using the algorithm proposed by Suzuki et. all [19], implemented in OpenCV's `findContours()` [1] method. This returns an array of connected vertices in orientations such that shapes are to the right of the contour. Once contours are found, faces are formed with the closed contours as the edges. Contours with a counter-clockwise

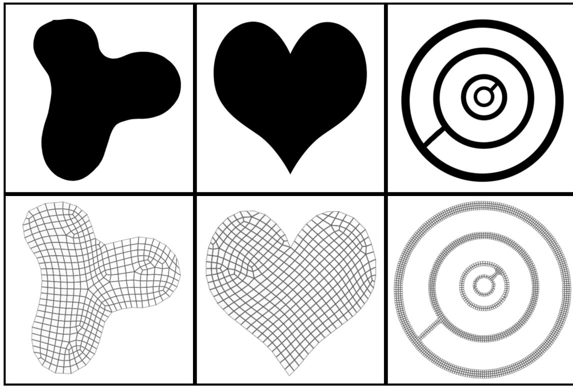

Figure 2. Input shape bitmap to 2D irregular grids

winding will result in faces with a normal vector $[0, 1, 0]$, and contours with a clockwise winding will result in faces with the normal vector $[0, -1, 0]$. A boolean union operation is applied to the faces. Due to opposite windings, this will result in a face with appropriate holes, as shown in Fig. 2.

Mesh quadrangulation is performed on the output polygon. Sketch-2-Tilemap uses the Interactive Field-Alligned Operator implementation of mesh quadrangulation proposed by Jakob et Al. [5].

This method has the benefits of topographically aligned faces near borders, and a quadrilateral only results. This approach tries to optimize vertex positions and edge orientations based on the boarder of the input polygons. This is beneficial for the resulting tilemap. Consider an input drawing with narrow shapes representing corridors, as shown in Fig. 2. It is desirable for quadrilaterals edges to follow the shape of the path so tiles can be aligned and produce traversible spaces.

This method is also optimized for runtime, which is essential for producing a realtime tilemap drawing application, and supports user sketched flowlines which can give more control to the user for the generation of the grid.

One point of difficulty for any quadrangulation algorithm is handling singularities. These are points where it is impossible to approximate a shape with a regular lattice. The Interactive Field-Alligned Operator method handles singularities by drawing T-junctions to ensure the output is comprised fully of quadrilaterals. A number of T-Junctions can be observed in Fig. 2 where the normal lattice breaks down.

Singularities pose a challenge to wave function collapse, which relies on constraints from a regular grid. One solution, used by Oskar Stalberg [17], is to utilize custom tiles and constraint patterns which provide rulesets for irregular grids. Sketch-2-Tilemap aims to make it easy for a user

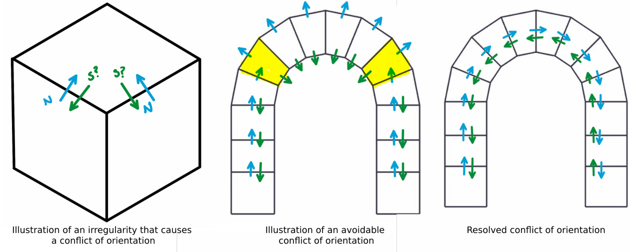

Figure 3. Examples illustrating conflicts of orientation.

to customize outputs with unique tilemaps and input pattern bitmaps. The approach used by Stalberg would handle these irregularities but make it difficult for users to specify new input patterns. Sketch-2-Tilemap implements WFC in a way that handles these irregularities and attempts to minimize orientation conflicts through custom face orientation of the quadrilateral mesh.

3.2. Orientation Conflicts

The implementation of Wave Function Collapse in Sketch-2-Tilemap relies on a set of constraints that are extracted from an input bitmap on a regular grid. As such, constraints are based on orientation. For simplicity, we assign each tile a North direction. The tiles other directions East, South, and West, are assigned clockwise from its North direction. Each tile type has a set of constraints for North, East, South, and West extracted from the input constraint pattern. Because the constraint pattern is specified on a regular grid, North is always designated as the upward direction. However, because we are performing WFC on an irregular grid, there may be conflicts of orientation. A conflict of orientation occurs when the orientation of a grid face does not match one of its neighbors. For example, a face that sees it's neighbor as the face to the north, where the neighbor does not see the original face as to its south would cause a conflict of orientation.

Example 1 of Fig. 4 shows how conflicts of orientation can be unavoidable on an irregular grid. Notice how there is no way to properly orient each face to avoid such conflicts.

However, there are instances where choosing the orientation of a face can help minimize conflicts of orientation and result in better WFC outputs. Example 2 of Fig. 4 shows an avoidable conflict of orientation caused by setting each face's North direction to the uppermost edge. Notice how the highlighted tiles have a South orientation that does not point to the face whose North orientation points to it. In this case, the conflict is avoidable, as seen in figure 3.

Search-based optimization algorithms such as Markov-Chain Monte-Carlo can be used to find an orientation of faces that minimizes conflicts of orientation. Attempted

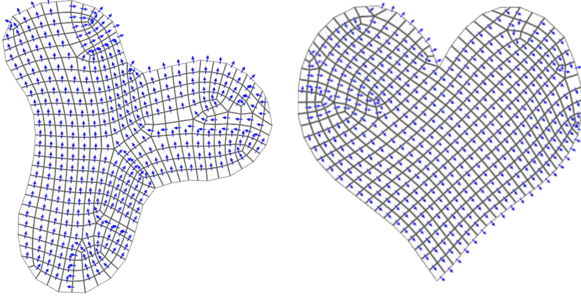

Figure 4. Breath-First orientation outputs $k = 200$.

search-based optimization methods, however, require slow runtimes and would have to be re-run on each new input sketch.

In Sketch-2-Tilemap, we perform k breadth-first orientations starting from random faces and choosing the result with the minimum number of conflicts of orientation. The result in Example 3 of Fig. 4 can be acheived with $k = 1$. For more complex meshes, the result which minimizes conflicts of orientation will be found from starting faces within the largest regular lattice-aligned patch of faces on the mesh.

3.3. Extended Wave Function Collapse

Sketch-2-Tilemap uses a custom implementation of the Wave Function Collapse algorithm proposed by Maxim Gumin [3]. This extended algorithm works for regular 2D grids, warped 2D grids (similar to the results acheived by Tobias Moller [10]), as well as irregular grids Fig. 7. The algorithm is shown in Algorithm 1. As an input, the algorithm takes G, an $n \times 4$ array where each row represents a face and each column represents the row index of the face's neighbor in a specified direction, and C, an $m \times 4m$ array where each row is a tile index and each column is a binary value representing if another tile can be adjacent to that tile in a specified direction. The algorithm outputs S, an $n \times m$ array where, if WFC is successful, each row has a single 1 value at a column whose index corresponds with a tile type.

G is generated from the irregular quad mesh built from an input sketch using the approach described in section 3.1. C is generated using the input pattern constraint texture. S is used to keep track of which tiles can be placed at which grid face, and ultimately outputs the single tile chosen by WFC.

Algorithm 1: Extended Wave Function Collapse to Generalize to Irregular Grids

```
   Input: G, C
   Output: S
 1 Def Propagate(f, S, G, C):
 2     q = [f]
 3     while q do
 4         foreach n ∈ G_{q.pop()} do
 5             if ∑ n = 1 then
 6                 continue
 7             s_new=updateState(n, G, C)
 8             if s_new ≠ S_n then
 9                 q.append(n)
10                 S_n = s_new
11     return S
12 Def Main(G, C):
13     n = |G|
14     m = |C|
15     T = n*m
16     S = 1_{n,m}
17     while T > 1 do
18         f = argmin(entropies(S))
19         S_f = collapse(S_f)
20         S = propagate(f, S, G, C)
21         T = ∏_{i=0}^{n} ∑_{j=0}^{m} S_ij
22     if T = 0 then
23         return 0
24     return S
```

When choosing a space for each iteration of the algorithm, the Shannon Entropy is calculated for each available space, and the space with the lowest Shannon Entropy is chosen using the formula in Equation 1

$$E(F) = -\sum_{t \in T_F} p(t) \log p(t) \tag{1}$$

Where $E(F)$ is the entropy of a face, t is a tile type, T_F is the set of remaining possible tiles for F, and $p(t)$ is the number of occurences of that tile type over the total number of tiles in the input constraint pattern.

Divergences from the traditional WFC algorithms include generalizable data structures as well as an `updateState()` function robust to orientation conflicts. For an irregular grid, a face's possible states depend on its neighbors constraints on it as well as its constraint on its neighbors. If a face's state does not match the constraint of its neighbors, or if none of a neighbor's possible states work with the constraints of the state, that state is eliminated from the row representing that face's possible states in the state array S.

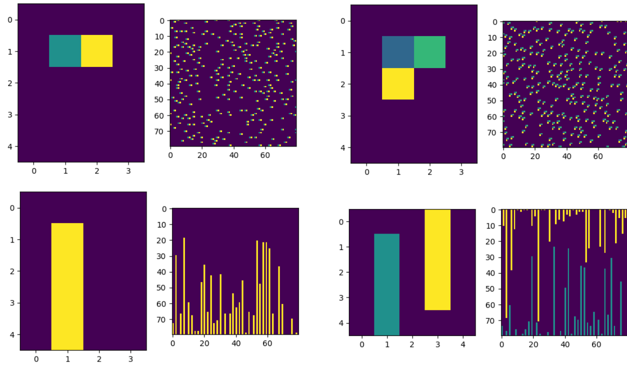

Figure 5. Output of my WFC implementation on regular 2D grids.

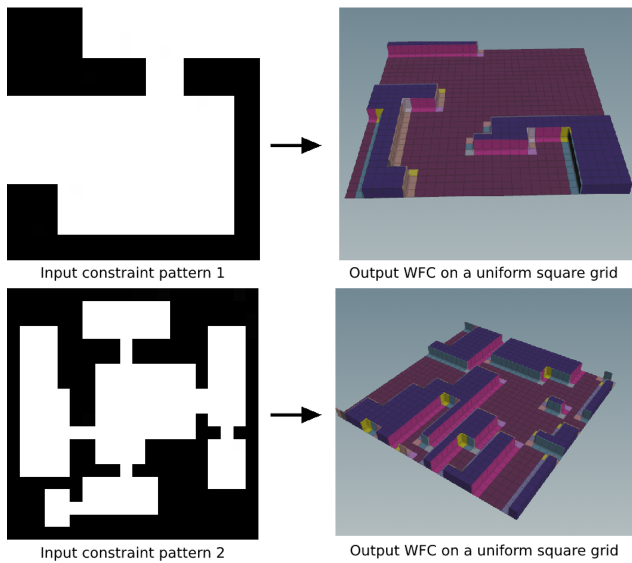

Figure 6. Outputs of WFC using Blob Tilest on regular grids

This bidirectional checking ensures that the algorithm produces legal output patterns, but limits the number of possible grid configurations. In scenarios with high ratios of face orientation conflict, it may be impossible for WFC to converge. Therefore, it is important to minimize orientation conflicts for expressive outputs using the methods discussed in Section 3.2.

This extended algorithm can produce traditional 2D WFC results Fig. 5, Fig. 6, Fig. 8 as well as results on an irregular grid Fig. 7.

3.4. Tile Deformation

A Blob tileset [21] is used with WFC to generate the results shown. This tileset includes all possible configurations of tiles containing walls and a floor such that any vertical face of the cube is a wall. In traditional WFC, tiles must be copied and translated to their correct positions. Without warping, this could be achieved through the memory-efficient approach of mesh instancing. For WFC on irregular grids, however, the tiles must be deformed to match their respective spaces. Sketch-2-Tilemap uses an 8-point lattice deformation implemented with Houidini's Lattice Geometry Node [8]. Lattices are oriented such that the North position of the face corresponds to the top edge of the input tile.

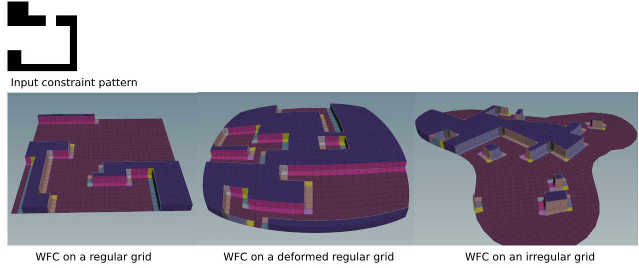

Figure 7. Output of WFC on a regular grid, a warped regular grid, and an irregular grid.

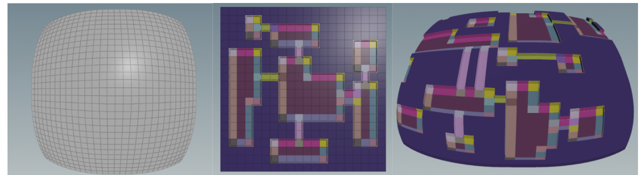

Figure 8. Another example of WFC initialized with blob tileset pattern on warped regular grid

Examples shown in this paper are created using the Wang Blob Tileset [21] included in Houdini, as well as a custom dungeon tileset built using assets from Pup-Up Productions Voxel Castle Pack [12]. Both tilesets can be seen in Fig. 11.

3.5. Digital Asset

The implementation described in this paper was built using Python and Houdini [14]. As such, it can be packaged using Houdini Digital Assets and loaded into a game engine such as Unity [20]. With this method, a user can use the tool as part of a game engine experience without knowledge of Python or Houdini. Fig. 9 shows the UI that the user will see when interacting with this tool.

Parameters exposed to the user include paths for an input sketch and constraint pattern, the number of desired faces n, the number of breadth-first orientation iterations k, and additional parameters such as a seed for the WFC algorithm.

4. Experiments:

4.1. Quantitative

We present our method for reducing conflicts of orientation in Section3.2. The hypothesis was that reducing the

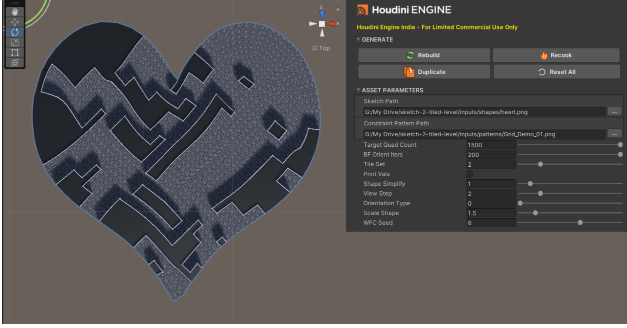

Figure 9. The Houdini Digital Asset UI for this implementation.

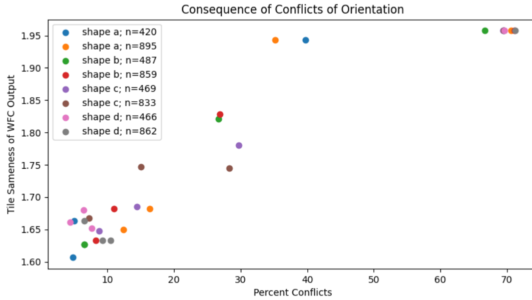

Figure 10. The impacts of conflicts of orientation on tile variation

number of conflicts of orientation would result in a better WFC output. In our case, shapes with high conflicts of orientation would often converge fully or mostly to floor or filled-in tiles, which have fewer constraints and can always be placed next to one of the same type in any direction.

These maps were sub-optimal because they did not have enough variation in tiles. To measure variation in tiles, we use average difference from the mean defined as followed

$$S = \frac{1}{n}\sum_{i=0}^{t_{len}} |t_i - \frac{1}{t}\sum_{j=0}^{t_{len}} t_j| \tag{2}$$

where S is the sameness value, n is the total number of faces in the grid, and t is a vector containing the counts of each tile type found in the WFC output.

Percentage of conflicts of orientation is found simply by

$$100 * \frac{c}{4n}$$

where c is the number of conflicts of orientation found. This is where a face's orientation does not line up with a neighbor's. c is counted a maximum of 4 times per face as each face has 4 neighbors.

We plot percentage of conflicts of orientation against our sameness values for 4 input shapes each with 2 face counts, producing varying topologies, as seen in Fig. 10. The input grids used to gather this data can be seen in Fig. 13.

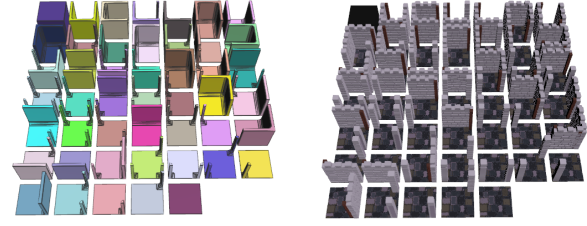

Figure 11. (Left) Wang Tile Set, (Right) Dungeon Tile Set

From Fig. 10, we see a clear correlation between conflicts of orientation and sameness of tiles output by the WFC algorithm. Therefore, in order to have more varied output, we should work to reduce conflicts of orientation. This makes intuitive sense, as conflicts of orientation effectively increase the number of constraints on the WFC output, making it difficult for WFC to converge to diverse outputs. The floor and filled in tiles are exceptions, as orientation does not matter for these tiles when places next to one another. Conflicts of orientation reduce the number of possible configurations to only tiles whose constraints are not impacted by orientation when placed next to each other.

To produce Fig. 10, we used our Breadth-First Orientation method with the k values 100 and 1, picking the orientation with the edge who had the minimum z-value as its centroid (the northmost-edge), and picking random orientations for each face. These consistently produced sameness values ranked in the order listed.

Next, we tested our implementation of Breadth-First Orientation with different k values to see the result. Fig. 12 shows the effect of varying k-values by plotting them against the resulting percent conflicts of variation averaged over 10 trials for different shapes with different n values. Interestingly, all trials plateau around $k = 50$. This likely occurs because the best alignments occur when the randomly-chosen starting face is in the largest lattice structure contained within the irregular grid. As the number of faces increase, the sizes of the lattice structures increase as well, so the probability of starting in one of these structures remains roughly the same.

4.2. Qualitative

We compare our approach to Moller et. al [10] in Fig. 14. Our approach results in an output grid shape more closely matching the input sketch. Sketch-2-Tilemap produces faces which are less deformed, requiring less deformation in the output tiles. Benefits of Moller et al.'s approach, however, include having no conflicts of orientation due to the output being a warped regular grid. In contrast, our output is on an irregular grid, so it is intractable to have no conflicts of orientation. This can result in outputs that

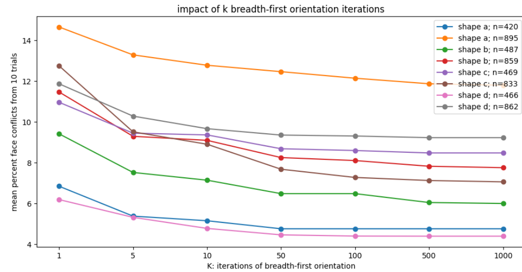

Figure 12. The impacts of increasing k on Breadth-First Orientation

potentially perform better at matching the patterns shown in the input constraint pattern.

Moller et. al's approach to WFC differs from what is shown in Fig. 14, where we run our WFC on their output grid. Their approach incorporates traversability when placing tiles on the grid. This is a topic we leave as future work.

5. Conclusion

Sketch-2-Tilemap shows the potential of running WFC on irregular input grids to generate tilemaps where artists control the overall structure by specifying an input shape and a binary constraint pattern. By reducing conflicts of orientation through breadth-first orientation, the outputs can be expressive like the outputs of normal WFC while being run on an irregular grid produced by hand-drawn inputs.
Sketch-2-Tilemap also shows how an existing procedural generation algorithm can be extended for greater user control and flexibility, and packaged in a way to make it easy for users to interact with.
There are many possible future directions to take this implementation which include synthesizing better outputs and allowing for more user control.

5.1. Future Work

Future work lies in providing the designer more intuitive control over the output. For an example of an output that is difficult for the artist to control refer to the castle input shapes of Fig. 13.

Artist control could come from allowing the user to constrain certain tiles prior to the WFC algorithm. For example, the user could select the faces along desired corridors and constrain them to wall tiles. Additionally, boundary faces can be assigned to a wall, corner, or filled tile to ensure no open edges of the map.

Additional areas of improvement for this approach include utilizing different quad meshing algorithms. The method being used [5] is prone to errors and large chunks of missing shapes. This can be noticed in multiple areas of Fig. 13, such as the tips on the tree shape C and on the bottom left circular shape on D.

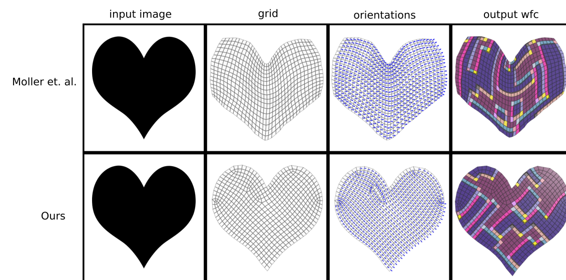

Figure 14. A comparison of our WFC on growing grids [10] and Interactive Field-Alligned Mesh grids [5]

Future work can also include a 3D WFC. The input grid could be extruded in the z-dimension to create 3-dimensional grid spaces resulting in 3D output maps with varying elevation and structures. This extension would require the consideration of a new 3D input constraint method.

References

[1] *Contour Approximation Method*. https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html.

[2] Juncheng Cui. *Procedural cave generation*. URL: https://ro.uow.edu.au/theses/3493/.

[3] Maxim Gumin. *Wave Function Collapse Algorithm*. Version 1.0. Sept. 2016. URL: https://github.com/mxgmn/WaveFunctionCollapse.

[4] *Hex Map*. https://en.wikipedia.org/wiki/Hex_map. Dec. 2021.

[5] Wenzel Jakob et al. "Instant Field-Aligned Meshes". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH ASIA)* 34.6 (Nov. 2015). DOI: 10.1145/2816795.2818078.

[6] Lawrence Johnson, Georgios Yannakakis, and Julian Togelius. "Cellular automata for real-time generation of". In: (Sept. 2010). DOI: 10.1145/1814256.1814266.

[7] Marian Kleineberg. *Infinite procedurally generated city with the wave function collapse algorithm*. Jan. 2019. URL: https://marian42.de/article/wfc/.

[8] *Lattice Geometry Node*. URL: https://www.sidefx.com/docs/houdini/nodes/sop/lattice.html.

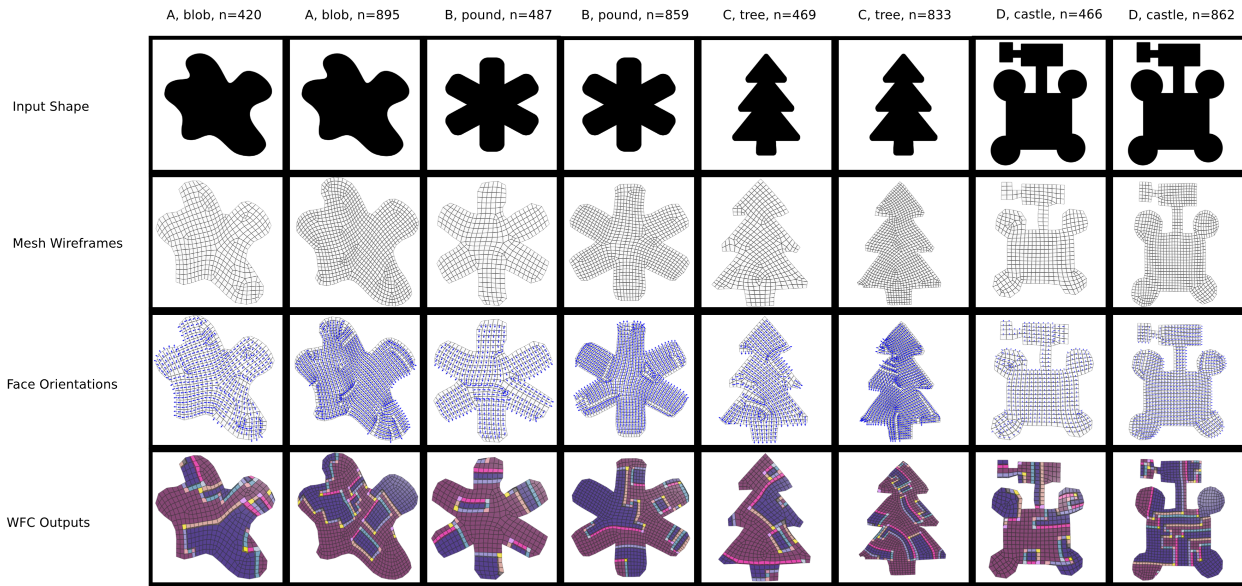

Figure 13. Inputs used in experiments shown in [Fig 10] and [Fig 12]

[9] Jialin Liu et al. "Deep learning for procedural content generation". In: *Neural Computing and Applications* 33.1 (Oct. 2020), pp. 19–37. ISSN: 1433-3058. DOI: 10.1007/s00521-020-05383-8. URL: http://dx.doi.org/10.1007/s00521-020-05383-8.

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

[11] Amit Patel. *Amit's Thoughts on Grids*. http://www-cs-students.stanford.edu/~amitp/game-programming/grids/. Jan. 2006.

[12] Pup Up Productions. *Voxel Castle Pack Lite: 3D historic*. https://assetstore.unity.com/packages/3d/environments/historic/voxel-castle-pack-lite-164189. Mar. 2020.

[13] *Rule tile: 2d tilemap extras: 1.6.0-preview.1*. URL: https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@1.6/manual/RuleTile.html.

[14] SideFX. *Houdini*. https://www.sidefx.com/.

[15] Ryan James Smith. *WFC in houdini*. Jan. 2022. URL: https://twitter.com/overdrawxyz/status/1484214381974020113.

[16] Oskar Stalberg. *Bad North*. URL: https://www.badnorth.com/.

[17] Oskar Stalberg. *Townscaper*. URL: https://oskarstalberg.com/Townscaper/.

[18] Adam Summerville et al. "Procedural Content Generation via Machine Learning (PCGML)". In: *IEEE Transactions on Games* 10.3 (2018), pp. 257–270. DOI: 10.1109/TG.2018.2846639.

[19] Satoshi Suzuki and KeiichiA be. "Topological structural analysis of digitized binary images by border following". In: *Computer Vision, Graphics, and Image Processing* 30.1 (1985), pp. 32–46. ISSN: 0734-189X. DOI: https://doi.org/10.1016/0734-189X(85)90016-7. URL: https://www.sciencedirect.com/science/article/pii/0734189X85900167.

[20] Unity Technologies. *Unity*. Version 2019.4 LTS. URL: https://unity.com/.

[21] *Wang tiles*. URL: http://www.cr31.co.uk/stagecast/wang/blob.html.