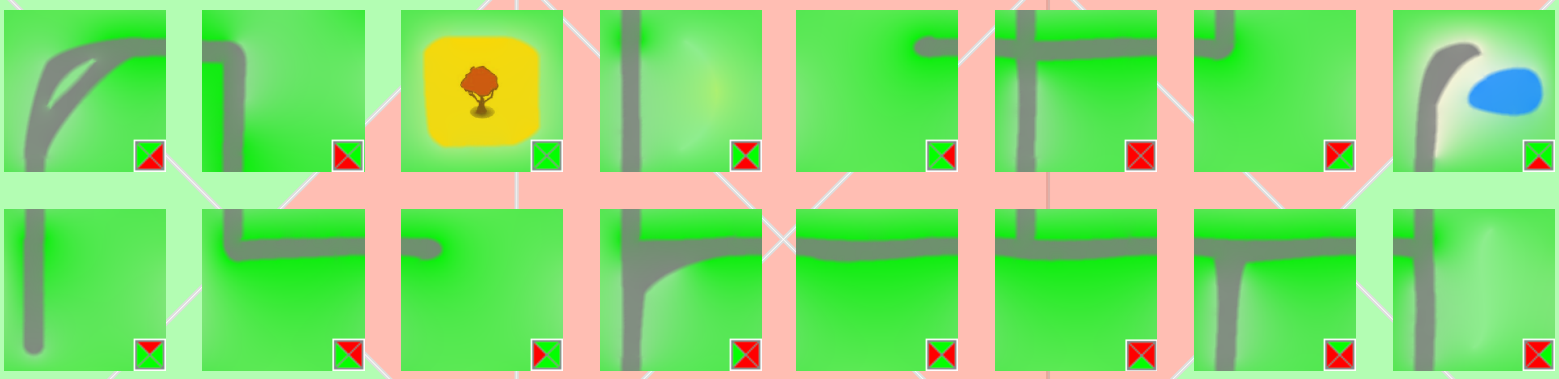
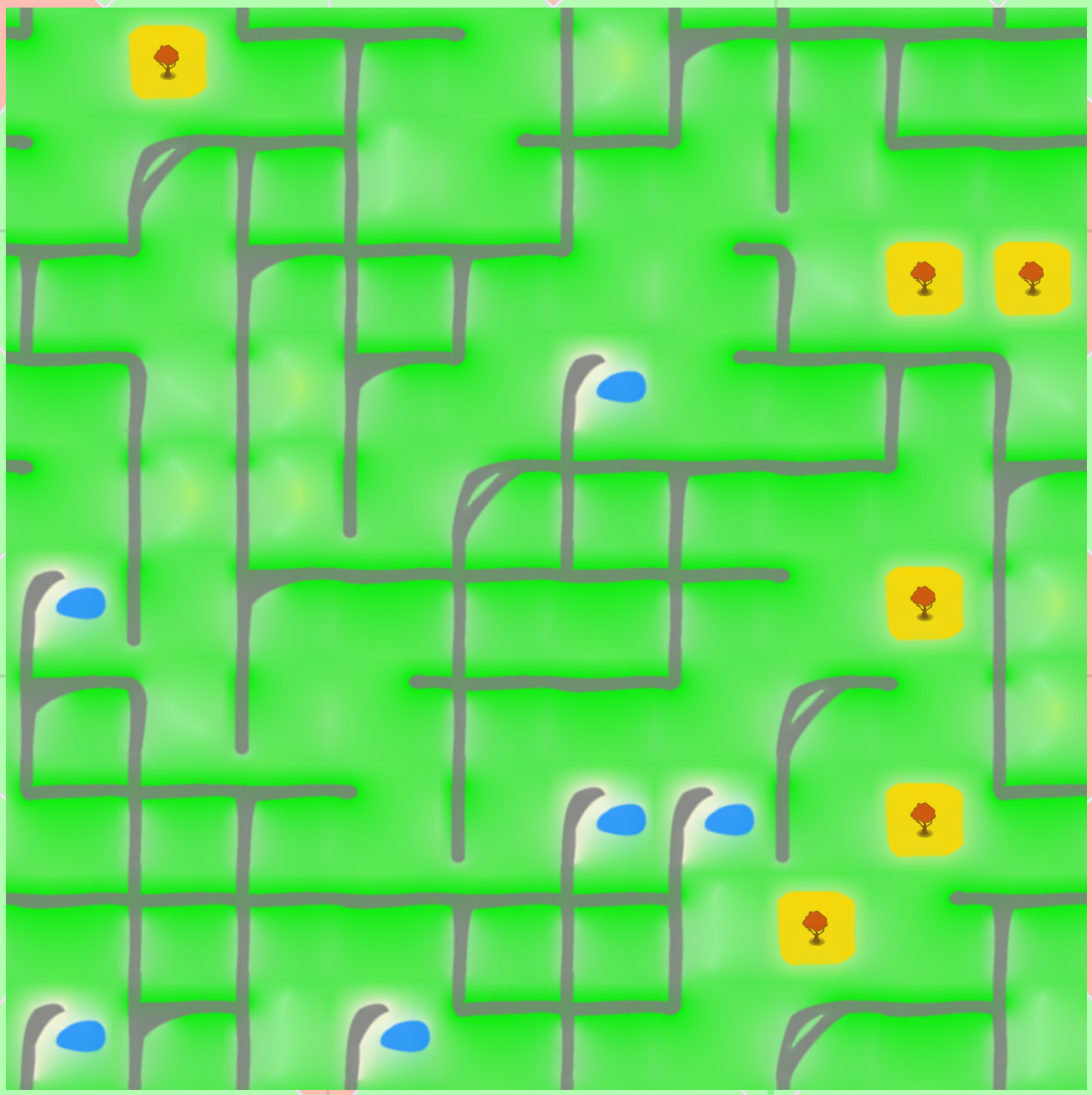


Diffusion Mosaic

Zehao Jing



Delft University of Technology

TU Delft

Diffusion Mosaic

by

Zehao Jing

to obtain the degree of Master of Science at the Delft University of Technology,
to be defended publicly on August 31, 2021 at 12:30 PM.

Student Number: 4787560

Thesis Committee:

Prof. Dr. Elmar Eisemann,	TU Delft, supervisor
Dr. ir. Rafael Bidarra ,	TU Delft
Prof. Dr. ir. Haixiang Lin,	TU Delft

TU Delft

Abstract

Large textures that can provide realistic details are widely used in modeling, gaming, art design, etc. Texture synthesis is a way to create large textures based on a small sample pattern, which can be obtained by image examples or hand-drawn work by an artist. Different methods that aim to achieve better visual effects on reducing or avoiding seams and distortions during synthesis are proposed. It has been a popular topic both in computer graphics and computer vision for many years.

However, most of the synthesis methods are automatic and take images as the input. There are few methods designed for artists who want to control and create patterns manually. The goal of our Diffusion Mosaics is to introduce a graphics tool that allows the artist to create tiles that can be seamlessly concatenated. In this way, non-periodic textures of arbitrary size can be produced at very low memory costs. We rely on a special kind of tiles called Wang tiles, which are squares with colored edges. Neighboring tiles should share the same edge color at the common border. The texture tiles are designed in such a way that if this constraint is fulfilled, the transition from one tile to the next will be seamless. It becomes possible to create a large non-periodic final texture. In our system, each tile can be filled with hand-drawn patterns using diffusion curves, which is a vector graphics primitive created by diffusing the given colors of defined Bezier curves, which has been proven to be a useful alternative to purely pixel-based design that artists typically rely on. They also match our application well, as, by adding border-color constraints to the tile, solving the diffusion process results in tiles that match the neighboring tiles naturally, hence avoiding any seams. The tool gives creative freedom to the artist that they can even draw the pattern outside the tile area, resulting in an automatic update of all other tiles when needed to ensure consistency. Finally, for more complex illustrations and backwards compatibility to software such as Photoshop, pixel-based images are also supported and can be integrated.

Acknowledgement

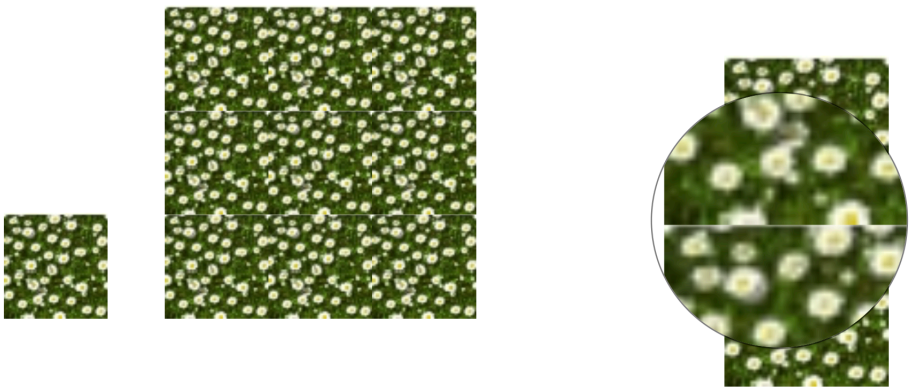
This thesis will mark the end of my special journey of my master's study. I want to thank everyone who has helped me. I want to thank Prof. Dr. Elmar Eisemann, my research supervisor, for his critical opinions and guidance, witty conversation, and academic rigor for the thesis. For me, the topic of this thesis is a new and magical field. He helped me to have a deeper understanding of computer graphics and mathematics. I am grateful to learn from him. I appreciate my friends and family's unconditional support and love during this special covid time.

Zehao Jing
Harbin, August 2021

Contents

1 Introduction

Textures are ubiquitous in the real world, and they can represent fundamental features of an object's surface. In computer graphics, texture is represented as an image that is mapped to the surface of a geometric object. Hereby, the virtual object appearance can be drastically enriched without increasing geometric complexity. How to obtain this texture information is a question worth considering. Textures can be acquired by photos, procedurally generated or drawn by artists using drawing tools. If a low-resolution (small) texture is mapped to a larger surface area, the surface texture will be distorted after the mapping. A work-around, especially for repeating textures, such as the weaving pattern of cloth, a solution could be to repeat the texture as well. Nevertheless, most textures are not perfectly uniformly repeating (e.g., grass and flowers on a meadow). A solution could be to capture a high-resolution representation but that quickly results in high memory costs that we would like to avoid. Instead, different texture patches (tiles) could be used that represent slightly different distributions and one could tile the object surface with these. Unfortunately, without a careful creation of these texture tiles, placing them side-by-side will lead to visible seams that are identified as clear artifacts by an observer as shown in Figure 1.1.



(a) (b)

Figure 1.1: (a)A large texture generated by simple repetition. (b) Visible seams and mismatch.

Some work has focused on generating larger textures or tiles from smaller exemplars automatically. However, we want to retain artistic control over the content of the texture, as it is particularly important

for content creation. Further, no method involves the artist directly in the process, instead, the provided artistic input is automatically transformed in a post process, which leaves no room to adjust the results and any later change influences the entire outcome.

In this thesis, we design an interactive large texture synthesis tool for artists, and it allows users to create patterns in the tool and generate large seamless textures at low memory cost. We want to offer an approach, in which the artist can draw the tiles directly, while the computer assists in assuring that the tiles are seamless. The patterns designed by the user can then be used to synthesize non-periodic textures of any size by a specific rule.

Our prototype implementation provides support of vector-based diffusion curves, which are a highly expressive tool and enable realistic, as well as abstract image generation. Further, we enable the use of images as input to enable the use of standard software, such as Photoshop. Users can design content for each tile, and the tile's boundary will no longer restrict the user's creation because we provide a content synchronization mechanism: when the user's content is outside of the tile, other tiles will be updated accordingly.

The large texture synthesis is a tile-based synthesis method so that as long as the basic tiles are rendered to form a tileset, large textures can be tiled through a specific tiling algorithm. The tiling algorithm can create a non-periodic seamless texture. Furthermore, this tile-based synthesis method is very efficient, and the final large textures can be stored as tile patterns and their arrangement, which requires very little memory. For example, using 8 tiles with 128x128 pixels and 8-bit RGB values can be used to generate a 16354x16354 texture at only 8*128*128*3 + (16354/128)*(16354/128) bytes = 393216+16354 bytes=409570 bytes<0.5 MB, while the full texture would require more than 260 MB - roughly a factor of 500 times more memory. Further, a small set of tile patterns is also more cache friendly, making the tiling algorithm actually quite efficient when executed at run-time.

To summarize, in this thesis, we develop an interactive texture synthesis tool, and it has the following functions:

- The user can create patterns freely by using a diffusion curve.
- The user can take images as input.
- The tool can generate large seamless and non-periodical textures.

This report has seven chapters, and the content is structured as follows: Chapter 2 will introduce some background knowledge related to this thesis, which is essential for understanding Diffusion Mosaics. Chapter 3 describes how we design the tool using a diffusion curves to generate a series of images that can be seamlessly stitched together based on the Poisson equation and create a large pattern via the tiling method. Chapter 4 will provide the details of implementing this tool and show how to use it. In Chapter 5, we will measure some performance of this interactive tool and display the visual results generated by it. Chapter 6 will discuss the shortcomings of this tool. Finally, Chapter 7 will conclude this thesis and discuss possible improvements.

2

Background

This chapter will introduce some techniques that are related to this thesis. In Section 2.1, we will discuss some popular texture synthesis methods. Section 2.2 will introduce the concept of Wang tile and its application. Then, Section 2.3 will focus on a vector-based graphics primitive called Diffusion Curve.

2.1. Texture Synthesis

Texture plays a critical role in graphics by providing various surface details for objects to make them more realistic. Those who make models need to cover the models with textures instead of constructing model and material details geometrically, which saves a lot of work and time, but is also much more efficient when displaying such a model. Currently, there are two primary sources for obtaining textures; one is hand-drawn work of the artist, the other is digitally scanned pattern images[28]. However surface texture will be distorted if the low-resolution (small) texture is mapped to a larger surface area after the mapping. One solution is to repeat the texture, but most textures are not perfectly uniformly repeating. Another solution is to capture a high-resolution texture, but this will result in high memory costs that we would like to avoid. Therefore, various texture synthesis methods that aim to generate large textures while having better visual effects like reducing or avoiding the visible seams that are identified as clear artifacts are put forward in the past several years.

Nowadays, texture synthesis has already been one of the most popular topics in computer graphics and vision. As a result, numerous researchers have proposed different methods from different perspectives. Efros et al.[8] first proposed a non-parametric sampling texture synthesis method, which takes samples from the initial image and then synthesizes a new texture pixel by pixel based on color consistency of the neighborhood. Wei and Levoy[28] improved this pixel-based synthesis method and introduced a multi-resolution synthesis model. Based on Wei and Levoy's work, Ashikhmin[1] found that when synthesizing a texture line by line, a chosen pixel from the sample image, should not only ensure that the resulting synthesized neighborhood is similar to one from the sample texture, but that also the original location of the pixel should be close to the already synthesized pixels to maintain the structure of the input image. By involving the original position and color, the algorithm leads to patches of pixels that have a consistent origin in the sample texture. The idea of using patches instead of individual pixels has sparked additional research, image quilting[7] randomly picks blocks of pixels that partially overlap in the synthesized texture and then uses dynamic programming and graph cut[18] to find the best common cut between patches to render the transition between blocks invisible. However, all these methods actually generate a large texture, which is memory and compute intensive.

While tile-based texture synthesis tries to generate large textures using different strategies, it first creates reusable tiles as a basic tileset. Once the tileset is specified, the user can create large aperiodic textures according to certain rules in real-time. The tile-based synthesis algorithms have the advantages of using less memory, short computation time, and real-time performance. Cohen et al.[4] used Wang Tiles for texture synthesis, while Dong et al.[6] used ω-tile sets as shown in Figure 2.1. They both need four sample patches and compute a cutting path to render the boundary transition natural but the patch placement is different. Wang Tiles use a diagonal cut, which can result in problems at corners. For this reason Cohen and colleagues also introduced an additional corner classification, which will

not be considered in this work. Tile-based synthesis methods have to deal with boundary conditions to avoid seams after tiling.

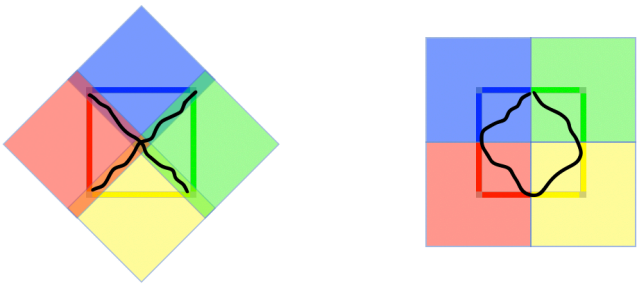


Figure 2.1: Wang tile and ω-tile. The cutting path of Wang tile will make the junction of four tiles looks prominent. ω-tile can avoid this.

In recent years, more and more texture synthesis methods[10, 19, 26] using neural networks emerged. Many of them aim to generate new textures based on the training set rather than generate large textures based on small textures, and there are also some works using deep learning approaches to create tileable textures like TileGAN[9] and Deep Tiling[25].

Basically, all methods discussed here [8, 28, 7, 4] take an image as input and generate a larger texture. Little focus lies on the artist; the creation and influencing of patterns. Ashikhmin[1]'s method is a notable exception, as it allows user interaction to control the distribution of texture patches but these interactions are still very limited and restricted to the input exemplar. To go a step further, we want to combine drawing tools with texture synthesis to achieve full flexibility and want to support a tile-based representation for lower memory consumption and efficiency at run time.

2.2. Wang Tile

Mathematician Wang Hao proposed Wang Tiles in 1961[27]. The tiles must meet the following requirements:

- Square tile
- Each edge of a tile has a color
- Only tiles with the same color on adjacent edges can be put together
- Rotation and reflection of tiles are not allowed when tiling the plane

A set of Wang tiles is called a Wang set. Since one square has four edges, if there are two colors, then there will be 2^4 combinations as shown in Figure 2.3. Wang posed a classic conjecture, which turned out to be false; Wang Tiles either generate a periodic tiling or cannot tile the plane. While it turns out that the tiling of the plane is generally undecidable, in the following decades, researchers found particular finite Wang sets that can tile the plane aperiodically[23, 5, 16]. The term 'aperiodic' of Wang tile means that there is no translation under which the pattern is invariant (i.e., there is no repetition). In 2015, Jeandel and Rao[14] presented a minimal aperiodic Wang Set that only has 11 tiles and four colors, as shown in Figure 2.2. For better visual effects, we use squares with four colored triangles to represent Wang tiles. Due to its unique properties, Wang tile is widely used in texture synthesis, map generation, and many other fields.

Figure 2.2: Aperiodic Wang Set of 11 tiles and 4 colors

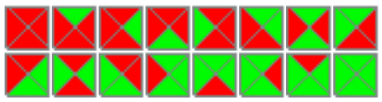


Figure 2.3: Wang Set of 16 tiles and 2 colors

2.2.1. Intuitive Stochastic Tiling

The most intuitive stochastic tiling algorithm [4] is first to randomly select a tile for top left corner. For the remaining area, select the tiles that can match its existing neighbors. Algorithm 1 shows the whole process.

Algorithm 1 Unconditional Stochastic Tiling

Require: $width,\ height,\ Tile\ Set\ T$
  for $y \leftarrow 0$ to $height$ **do**
    for $x \leftarrow 0$ to $width$ **do**
      if $x = 0,\ y = 0$ **then**
        $f(0,0) \leftarrow randomly\ choose\ a\ tile\ t\ from\ T$
      else if $x > 0,\ y = 0$ **then**
        $f(x,0) \leftarrow\ choose\ a\ tile\ t\ from\ T\ while\ f_w(x,0)\ =\ f_e(x-1,0)$
      else if $x = 0,\ y > 0$ **then**
        $f(0,y) \leftarrow\ choose\ a\ tile\ t\ from\ T\ while\ f_n(0,y)\ =\ f_s(0,y-1)$
      else if $x > 0,\ y > 0$ **then**
        $f(x,y) \leftarrow\ choose\ a\ tile\ t\ from\ T\ while\ f_w(x,y) = f_e(x-1,y)\ and\ f_n(x,y)\ =\ f_s(x,y-1)$
      end if
    end for
  end for
  return $tiling\ f$

In this algorithm, one tile can be constrained by at most two neighbors: one is its north neighbor, the other one is its west neighbor. The tile set should meet the minimal condition that more than 2 tiles

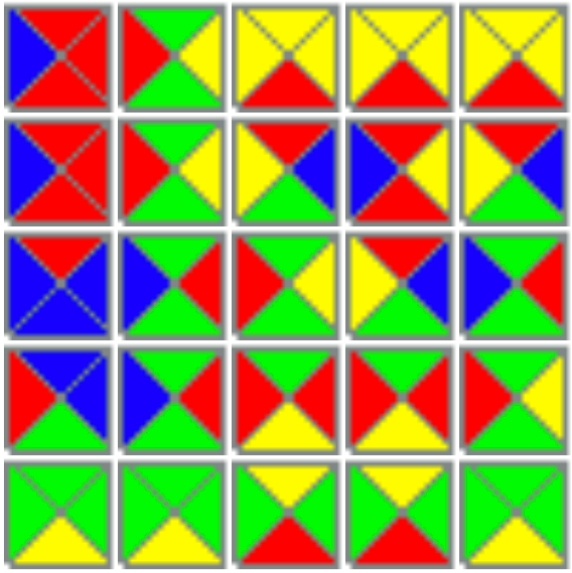


Figure 2.4: An example of a tiling by minimal aperiodic Wang Set

can be chosen when one tile's north and west neighbor are fixed. If there are $K1$ colors in the vertical direction and $K2$ colors in the horizontal direction, then there will be at most $K1^2 \times K2^2$ tiles without duplicate ones.To have a random tiling result, when a tile is constrained by its west and north neighbor, the tile set should be able to provide at least two choices, which means there should be at least two tiles for each horizontal and vertical color combination. And there are $K1 \times K2$ color combinations. Thus, the set's minimal condition for this stochastic algorithm is to have at least $2 \times (K1 \times K2)$ tiles. We will call those tile sets that have all possible color combinations as *complete Wang set*, while those sets that just meet minimum requirements as *reduced Wang set*. For example, Figure 2.5 shows a reduced Wang set, while Figure 2.3 shows a complete Wang set. When this minimal conditions cannot be met, alternative algorithms have to be taken into consideration.

Figure 2.5: A reduced Wang Set

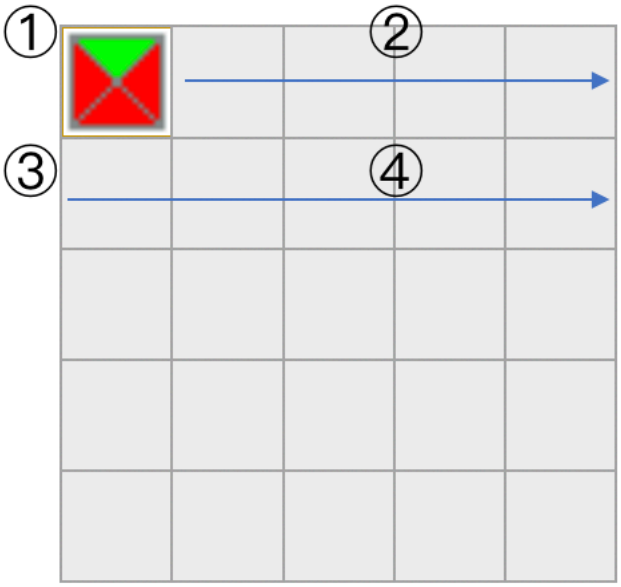


Figure 2.6: A simple stochastic tiling algorithm. Step 1: Randomly choose a tile from the tile set for the leftmost one of the first row. Step 2: Find the tile whose left edge color is the same as its left neighbor's right edge's for the remaining tiles of the first row. Step 3: Randomly choose a tile whose north edge color is the same as its northern neighbor's south edge's for the leftmost one of the next row. Step 4: Randomly choose one from those tiles whose left edge color is the same as its left neighbor's right edge's and north edge color is the same as its northern neighbor's south edge's for the remaining tiles of the row. Step 5: Repeat Steps 3 and 4 for the remaining rows.

2.3. Diffusion Curve

Diffusion curve is a vector graphics primitive proposed by Orzan et al.[21]. It uses Bézier curves as geometric primitives to represent the contour information in the image, and color and blur information is attached to the curve. These curves are generally spline curves with color and blur control points drawn by the user or extracted from the contours of existing images. The hand-drawn or extracted curves are generally the basic skeleton of the image, describing the outline of the pattern or the places, where color changes greatly.

In addition to the geometric shape, the diffusion curve should also store the colors that will be diffused in the areas on both sides of the curve. For color, the user only needs to select certain points on the diffusion curve as color control points, and assigns colors to these control points. Then the remaining colors along the curve can be obtained by linear interpolation. Because the curve has two sides, two sets are needed to store the color control points, one for the left side and the other for the right side.

With the color information, the diffusion operation can be performed. However the color transition could be very sharp at the boundary after diffusion. Therefore, a blur operation is introduced to control color transition along the curve.

The diffusion curve has the following structure:

- A geometric curve.
- Two sets of color control points.
- One set of blur control points.

After obtaining the geometric and color information of the curve, the remaining part of the image will be filled with colors through the diffusion process. The diffusion process is solved by a Poisson equation with Dirichlet boundary conditions as shown in Equation 2.1.

$$\begin{aligned} \Delta I &= \text{div } \mathbf{w} \\ I(x,y) &= C(x,y),\ if\ C(x,y)\ has\ a\ value\ (is\ next\ to\ the\ curve) \end{aligned} \tag{2.1}$$

where Δ is the Laplacian operator,I is the desired diffusion result image which is initially unknown. div is the divergence operator, **w** is the color gradient. In general, the image is smoothly diffused except on the curves, where the color changes from one side to the other, thus the gradient vector field is zero except on the curve. The Equation 2.1 can also be written in Equation 2.2 by finite differencing.

$$I_{i,j} = \begin{cases} \frac{I_{i+1,j}+I_{i-1,j}+I_{i,j+1}+I_{i,j-1}+\text{div } \mathbf{w_{i,j}}}{4} \\ \\ C_{i,j},\ if\ C_{i,j}\ has\ a\ given\ value \end{cases} \tag{2.2}$$

In order to get a smooth-shaded image, the blur value is introduced to control the blur degree of the diffusion curve. The blur value will determine whether the color transition is sharp or smooth. Since the diffusion curve only defines the blur value on the curve and does not define the blur value of other pixels of the image, the blur map for the whole image can be generated like color diffusion by diffusing the blur value on the curve, which leads to the Equation 2.3.

$$\begin{aligned} \Delta B &= 0 \\ B(x,y) &= \sigma(x,y),\ if\ (x,y)\ is\ on\ a\ curve \end{aligned} \tag{2.3}$$

where σ is the variance of the Gaussian blur specified by the user along the curve.

There are many strategies to solve Poisson equations. Usually, they can be solved by some simple iterative methods like Jacobi, Gauss-Seidel iterations, or more efficiently via conjugate gradient and multi-grid method.

Jeschke et[15] presented a GPU Laplacian solver for Equation 2.4. They also integrated seamless cloning[22] within their solver.

$$I(x,y) = B(x,y),\ if\ (x,y)\ is\ a\ boundary\ value$$
$$\nabla^2 I = 0,\ otherwise \tag{2.4}$$

Bezerra et al.[2] proposed many new methods such as color strength or diffusion barriers to control the diffusion process by reformulating the diffusion process into a constrained linear systems. With color strength a, the original color $c = (r, g, b)$ can be written as a homogeneous color[29] $c' = (ra, ga, ba, a)$. Each color channel and color strength can be diffused separately, and the final diffused color can be obtained by the projection $(r/a, g/a, b/a)$. By default, the color strength value is one.

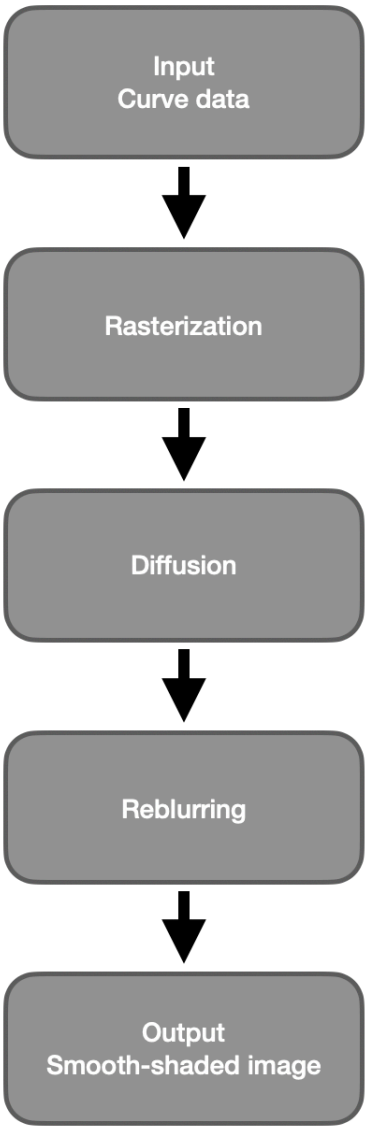


Figure 2.7: The rendering pipeline of diffusion curve

3

Our Method

This chapter will give an overview of our tool first. Then we will introduce how to design the tiles and render them. After that, tiling algorithms and graphic user interface design will be introduced.

3.1. System Overview

Although there are many existing methods for large texture synthesis, almost none of the existing synthesis techniques are specifically designed for artists. In other words, texture synthesis and pattern creation are always two completely independent processes. When the large texture is synthesized by the pattern generated by the diffusion curve, some of the features of diffusion curve can be utilized. This section will discuss how our tile-based synthesis method utilizes these features.

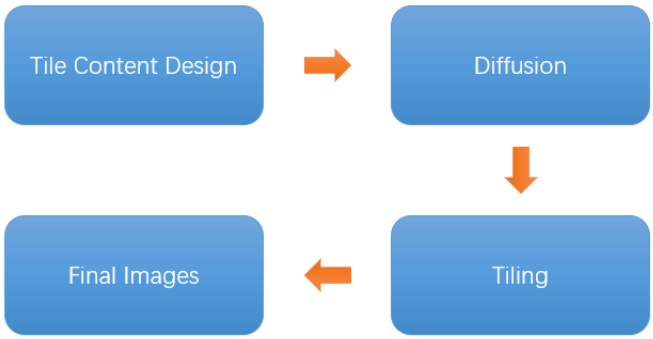


Figure 3.1: System Overview

The user designs the tile content first by using diffusion curves or images. After designing the content, the diffusion operation is executed. Having the tiles generated, the user can rely on these to tile the plane to obtain a final large image.

3.2. Tile Design

Wang Tiles provides a framework that can generate large and non-periodic textures. The next step is to fill these Wang tiles with diffusion curves and images as input. This section will show how we use the features of the diffusion curve to design these special tiles and explain why tiles generated in this way can be seamlessly stitched together.

Section 2.3 already introduces the basic knowledge of diffusion curve, and a curve should contain geometry, color, and blur information. Here, the curve is represented by the cubic Bézier spline. Figure

3.2 shows an example of a cubic Bézier curve. The curve's path is decided by these points P_0, P_1, P_2, P_3, and we will call these points the geometric control points. In other words, when these control points are fixed, the entire curve can be drawn through the De Casteljau algorithm. The Bézier curve can be defined algebraically in Bernstein form as shown in Equation 3.1.

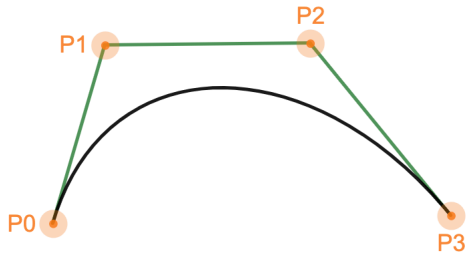


Figure 3.2: A cubic Bézier curve

$$f(t) = \sum_{i=0}^{n} \mathbf{p_i} B_i^{(n)}(t), t \in [0,1] \tag{3.1}$$

where $f(t)$ is the target Bézier curve, n is the polynomial degree, $\mathbf{p_i}$ is the control point, and $B_i^{(n)}(t)$ is the Bernstein basis function which can be expanded to Equation 3.2.

$$B_i^{(n)}(t) = \binom{n}{i} t^i (1-t)^{n-i} = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} \tag{3.2}$$

When the degree n is set to three, it is a cubic Bézier curve which is widely used in graphics applications. However, only one cubic Bézier curve may not meet the user's demand, so several curves can be joined together to form a longer piece-wise cubic Bézier spline.

In addition to geometric information, the most important element of the curve is color information. As mentioned before, there are two sets of color control points for one curve. One is the set of left color control points, and the other is right color control points. Left or right is determined according to the drawing order of geometric control points. Users can place the color control point anywhere on the curve, not necessarily at the end of each curve. After having the color control points, the colors along the entire curve are defined by linear interpolation - and the same applies to blur. When interpolating, both ends of the curve have to be considered. By default, the color is black, and the blur value is zero if no control points are placed on the curve's ends. But this will lead to unsatisfying visual results, so in this thesis the curve's ends will have the same value as its nearest color control point. Moreover, this tool introduces the color strength[2] to control the diffusion process better.

To summarize, a diffusion curve can be represented by several control points: geometric, color, and blur control points. This vector format storage is very efficient, especially in the loading and saving of patterns. In this thesis, a diffusion curve will be stored as follows:

- **Geometric control point**: $P_i(x, y)$
- **Left and right color control point**: $C_{li}(x, y, r, g, b, a)$ and $C_{ri}(x, y, r, g, b, a)$.
- **Blur control point**: $B_i(x, y, b)$

Note that a of color control point is the color strength rather than the alpha channel which would describe the degree of opacity.

The diffusion curve needs to be rendered into a pixel matrix before it can be displayed on the screen. The rendering process has three main steps: rasterization, diffusion, and reblurring[21], but

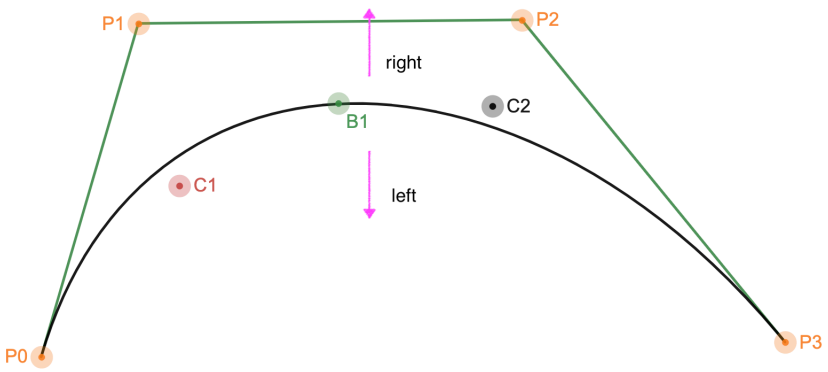


Figure 3.3: Control points of a diffusion curve. P_0, P_1, P_2, P_3 are geometric control points, C_1, C_2 are color control points, B_1 is the blur control point.

this rendering method is for diffusion curves drawn on one canvas or a single tile. Before rasterization, rendering a Bézier curve requires a discretization, which can refer to Equation 3.1, where t is linked to the step size that determines the number of samples. A sufficient number of sample points will lead to a smooth appearance after rasterization.

However, for Diffusion Mosaics, multiple tiles need to be rendered at the same time and somehow be connected. Only then, we can ensure that the resulting images can be seamlessly stitched together by a tiling method. Hence, each rendering step will be modified accordingly. In the traditional diffusion curve method, the user creation area has a limited size. As long as the user draws all curves in a tile, everything is the same as mentioned in Section 2.3. In this thesis, we will focus on creating the content for all tiles that can match each other. For example, the corresponding neighbor should update and add the curve and color source when a curve is drawn outside the tile. And this will be discussed in the following subsection. The matching of the final color, which is generated by the diffusion process guided by the Poisson Equation, will be described in detail in Section 3.3.3.

3.2.1. Content synchronization

In this thesis, the user creation will not be restricted by the tile boundaries, or more precisely, they can draw the curve outside of the tile. Figure 3.4a shows the case, where a user draws a curve crossing the north edge. Because a Wang tile is used here and each edge of a tile has a color, when the user draws outside the tile, the contents of all neighbor tiles with the same edge color have to be updated synchronously. Therefore, the system should have a content update mechanism that can determine the content of each tile. When the user draws outside the tile, this content synchronization will be triggered and the curve outside the boundary is transferred to the compatible neighbor tiles.

If the user draws the entire curve in the tile, then this case does not need special treatment. If part of the curve is drawn outside the tile, then the system needs to know which edge the curve intersects. Since the tile is square and it only has four edges, the curve must intersect with at least one edge. When a pixel on the curve appears in the north, west, east, or south area as shown in Figure 3.5, then the curve is asserted to have intersections with this edge. These intersections are stored clockwise, starting from the west edge by an array of four binary values, where 0 means no intersection, one means intersection. For example, the intersection of the curve shown in Figure 3.4b can be written as *0b0100* and *0b0000* for Figure 3.4a.

When the system detects that the curve has intersections with some edges, the content of its neighbor Wang tiles will be updated by adding an offset. In the following part, *complete Wang set* with four colors and sixteen tiles shown in Figure 2.3 will be used as an example. Figure 3.6 shows that if the

(a) (b)

Figure 3.4: a): The entire curve is inside the tile, b): Part of the curve is outside the tile.

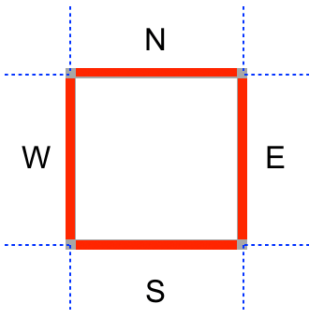


Figure 3.5: Four edges of a tile. t_w,t_n,t_e,t_s denotes all four edges(west, north, east, south). If the pixel of a curve appears in area *N* then the second value of the intersection array is 1.

user draws a curve that intersects the west edge, then all eight tiles that are possible neighbors in a tiling should be updated.

The update of the content requires the translation of the curve. For example, if a curve intersects the left side of the tile, then this curve should also appear on the right side of its western neighbor. Since a diffusion curve can be generated by several geometric and color control points, the curve in the neighbor tile can be easily updated by applying a translation vector *(offsetX, offsetY)* to all control points of the original curve. This translation process is visualized in Figure 3.7.

Figure 3.8 shows that only updating the west neighbor tiles is not enough because these tiles will affect their east neighbors in turn, and those tiles have the same color edge as the original tile also need to be updated. To distinguish between these two update methods, the update of the neighbor tiles is called *Type1 update*, and the update of the tiles with the same color edge is called *Type2 update*, and *Type2 update* does not need a translation of the curve. Note that one tile can be updated by *Type1 update* and *Type2 update* at the same time, which means when the user draws only one curve and many new curves will appear, as in Figure 3.9.

If a curve only intersects one edge, we have a simple case. However, in complicated cases, updating one curve will cause a series of tile content changes. This is because when a curve intersects more than one edge and is updated by *Type2 update*, the curve will keep the same intersection in the updated tiles, but the color of the intersecting edge may not be the same as the current tile. In other words, one curve would cause some tiles to be updated, and the updated tiles will cause more tiles to be updated, which becomes a recursive process. Note that this recursive update will cause unexpected content updates, which are not controllable for users.

A recursive process should not update a tile with the same curve twice, or it will turn into an endless loop. Therefore, the system will maintain an *identity* for each curve to distinguish them. In this system, each curve has a global id generated by a non-decreasing counter, which will increase by one when adding a new curve and will not decrease when deleting a curve. Thus, those curves derived from

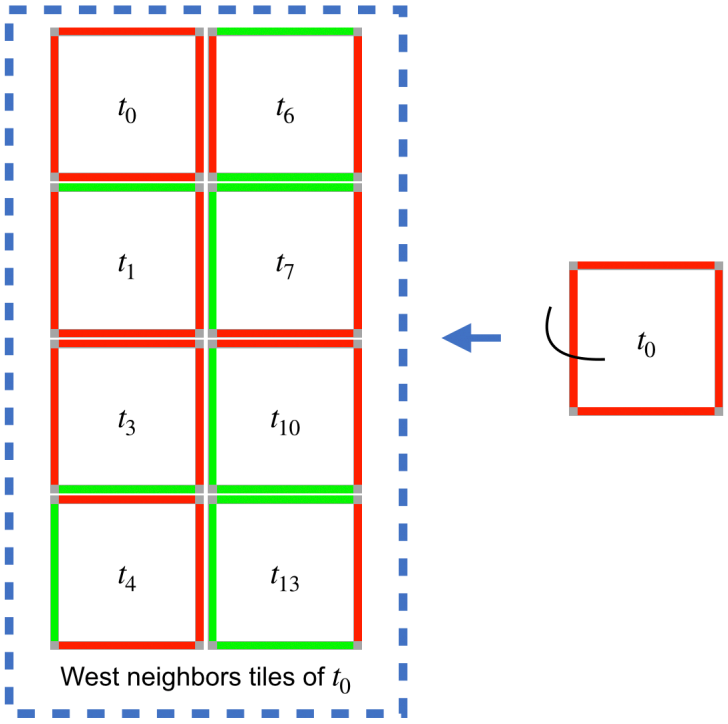


Figure 3.6: In $complete\ Wang\ set$, t_0 has eight west neighbors:$t_0, t_1, t_3, t_4, t_6, t_7, t_{10}, t_{13}$. And these neighbor tiles should be updated because their east neighbor has a curve that crosses the west border.

the same curve will share the same id, and if the user edits any curve, those with the same id will be updated correspondingly. As Figure 3.9 shows, the derived curves can appear in the same tile, so the id and the intersection binary array will be used together to identify the curve.

Through the content synchronization mechanism, it can be ensured that the geometric curve of the tile can also match correspondingly after tiling. An example of 3 tiling is shown in Figure 3.10

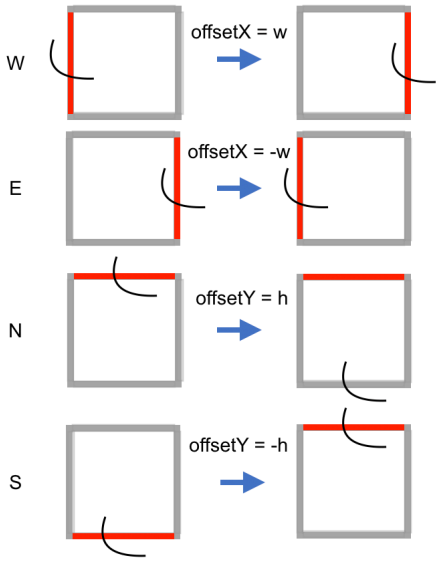


Figure 3.7: The translation process. The translation vector *(offsetX, offsetY)* depends on how curve intersects the tile.

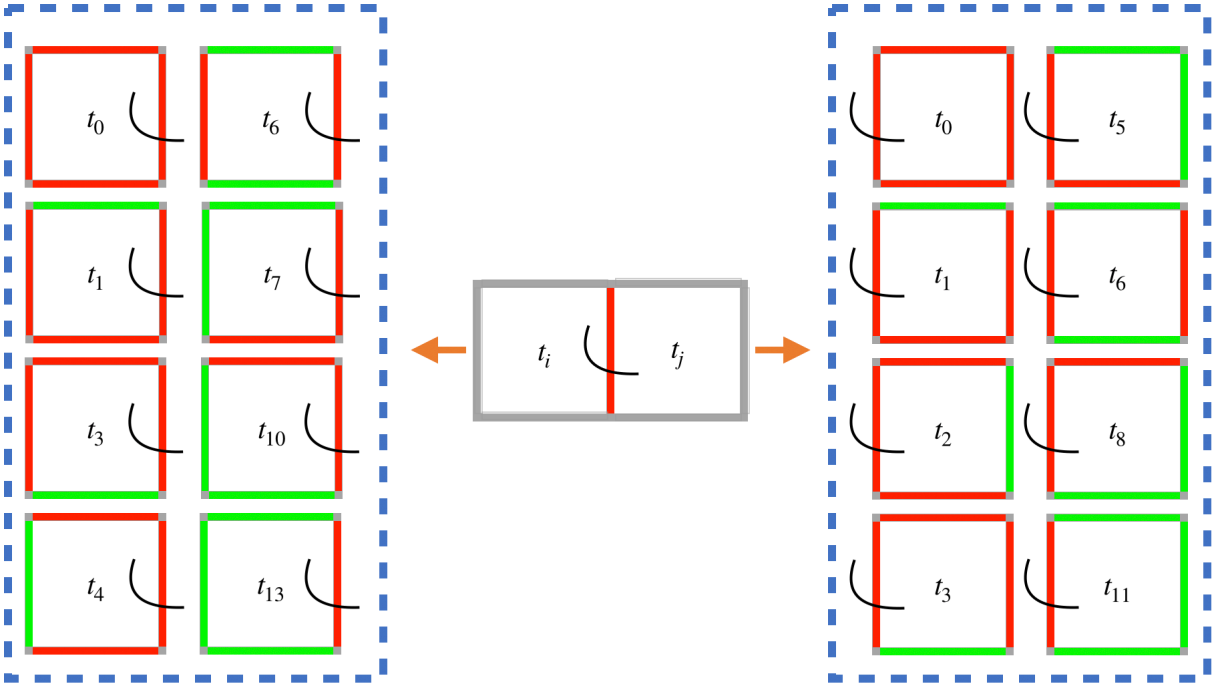


Figure 3.8: Content update. When a curve crosses a vertical edge of red color, the tiles whose east edge is red and the tiles whose west edge is red all need to be updated. When t_0 has a curve crossing the west edge, t_0, t_1, t_3, t_6 will get a translated curve by *Type1 update* and for t_1, t_2, t_3 they will copy that curve by *Type2 update*.

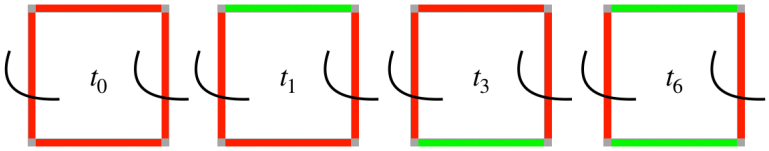


Figure 3.9: The original curve and its derived curves in tile t_0, t_1, t_3, t_6.

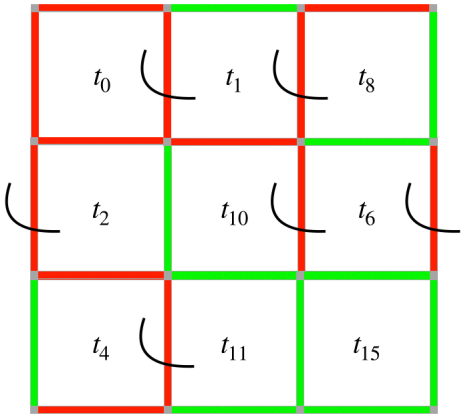


Figure 3.10: An example of 3×3 tiling with curves.

3.3. Rendering

Diffusion mosaics need to render all tiles of the selected tileset simultaneously. Furthermore, the rendering method should ensure that the final results will not have seams after generating a large tiling with these Wang Tiles by employing the algorithms in Section 3.4.

The rendering pipeline of the original diffusion curve is shown in Figure 2.7 and the rendering pipeline of Diffusion Mosaics still follows that structure. But the acquisition of color sources and the diffusion equation should be modified accordingly. In this section, the modification of color sources will be discussed first. In addition to accepting color sources of the curve as input, the system also allows users to take images as input, which will be addressed then. Last, we will introduce the new modified diffusion equation, which is the core part of the rendering in detail.

3.3.1. Color Sources

Color sources(*(x,y,r,g,b,a)*) are the result of a diffusion curve and mean the pixels that correspond to those next to the curve representing the color information on both of its sides. This is the strategy proposed by Orzan et al.[21], where each pixel is translated along the normal to avoid superposition of the color from both sides of the curve. The normal **n** of a curve is shown in Figure 3.11 and the left and right color curves are shown in Figure 3.12. The normal is perpendicular to $\overrightarrow{D2D1}$ and $\overrightarrow{D2D1}$ can be calculated according to the Equation 3.3.

$$\mathbf{d}(t) = \sum_{i=1}^{3} \mathbf{p_i} \binom{3}{i} t^i (1-t)^{3-i} - \sum_{i=0}^{2} \mathbf{p_i} \binom{2}{i} t^i (1-t)^{2-i}, t \in [0,1] \tag{3.3}$$

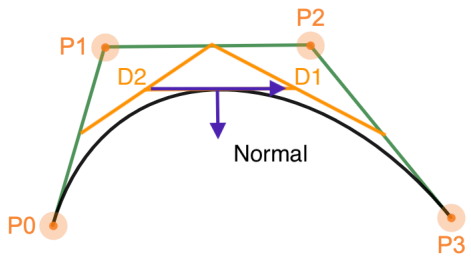
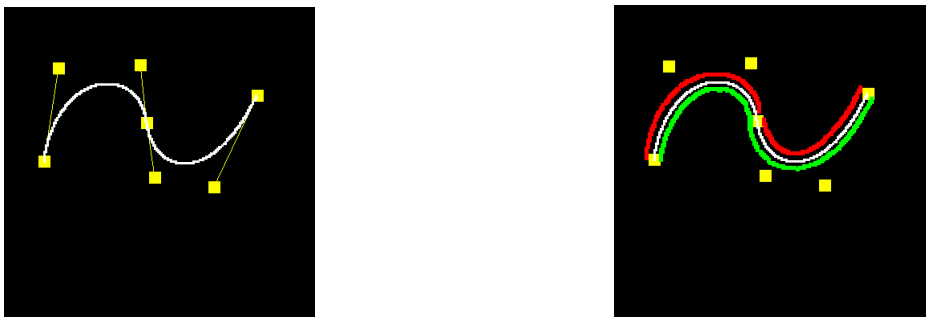


Figure 3.11: The normal of a curve.



(a) **(b)**

Figure 3.12: a): A curve without color, b): A curve with color on left and right sides.

Since the gradient is introduced, each tile stores not only the color information but also a gradient field **w** which only has a value on the curve, while it is zero elsewhere. And the gradient of those pixels on the curve can be expressed in 3.4.

$$\mathbf{w} = (cl - cr)\mathbf{n} \tag{3.4}$$

where *cl* is the left color and *cr* is the right color and **w** is the gradient that can be decomposed into w_x and w_y representing the gradient in vertical and horizontal direction.

Section 3.2 introduced a content synchronization mechanism that multiple curves will be derived from the curve drawn by the user. The colored curve still follows this mechanism. The curves with the same id will have the same colors. Furthermore, Section 3.2 has already discussed the case that some part of the curve is outside the tile. Corresponding pixels outside of the tile can simply be discarded, as they will be generated by the part of the curves inside of another tile.

3.3.2. Image Sources

Sometimes, instead of starting from scratch, an image could be used as part of the pattern. Such an image should be seamlessly integrated into the tiles as well.

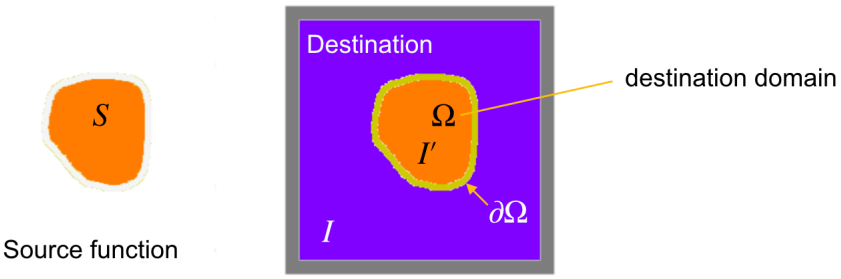


Figure 3.13: Image S is inserted to Image I. Hard constraints($I'_k = I_k$) are the colors on the border$\partial\Omega$. Soft constraints are based on a guidance vector field provided by S

Poisson image editing[22] provides a way of blending two images together seamlessly as shown in Figure 3.13. The function of the destination domain Ω is unknown and this function is denoted as I'. For each pixel p in destination image, the set of its 4-connected neighbors that are in the destination image I is denoted as N_p and the boundary of Ω is $\partial\Omega = \{p \in I \backslash \Omega : N_p \cap \Omega \neq \emptyset\}$. The source image S with *n* pixels $\{S_k | k \in 1...n\}$ is inserted into the destination domain area Ω of the destination image I, $\{I_k | k \in 1...n\}$ with the boundary $\partial\Omega$, while the remaining part of I should remain the same. Seamless image blending requires the gradient of I' to be consistent with the gradient of the source image S, and the value of I' on the border of Ω or $\partial\Omega$ is consistent with the value of I.

The resulting image I' is the solution of a Poisson equation under Dirichlet boundary conditions as shown in Equation 3.5, and an example of the two images blending is shown in Figure 3.14.

$$\Delta I' = \text{div} v \; over \Omega, with \; I_k = I'_k \; \forall k \in \partial\Omega \tag{3.5}$$

where Δ is the Laplacian operator, div is the divergence operator, v is the guidance vector field which refers to the gradient of the source function here and $v_{pq} == S_p - S_q$. The above Poisson equation can also be reformulated to a constrained minimization:

$$I' = \min_{I'|\Omega} \sum_{<p,q> \cap \Omega \neq \emptyset}^{n} (I'_p - I'q - v_{pq})^2, \; with I'_p = I_p, \forall p \in \partial\Omega \tag{3.6}$$

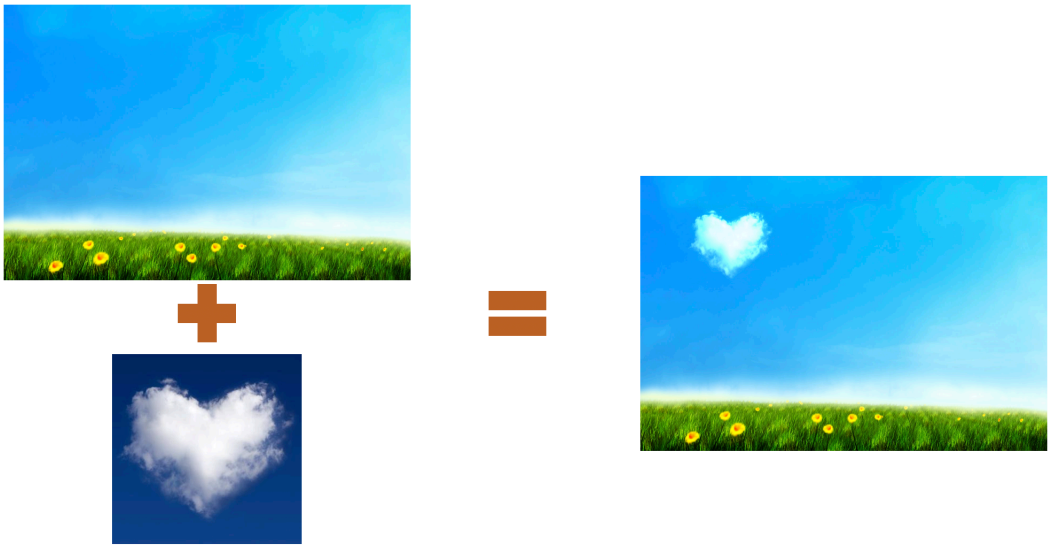


Figure 3.14: An example of image blending using Poisson image editing

In this thesis, the user can upload an image to the system directly as image sources that can be placed on the tiles.

3.3.3. Diffusion

In this section, we will build a linear equation system to solve the diffusion and image blending problem. After getting the discrete geometric coordinate and color information of the diffusion curve, the color on the curve can be diffused to other pixels of the images. This diffusion process is usually modeled by the Poisson equation framework with the Dirichlet boundary condition. The simplest case of diffusion has been introduced in Section 2.3. In this thesis, we will use the reformulated constrained linear system [2], which is described in Equation 3.15 to get the resulting tile set. However, in Diffusion Mosaics, not one tile needs to be rendered, but several tiles need to be rendered and be seamless. Hence, if the edges of a Wang tile match, the content of the tiles should also match. This means that the boundaries of tiles will not actually represent boundaries for the diffusion. The diffusion of one tile will affect the contents of its neighbor tiles, and the diffusion of its neighbor tiles will also affect its own content in turn. Our solution to get a continuous diffusion at the boundary is to add constraints that enforce that neighboring pixels of neighboring tiles should be similar. This condition is encoded via soft constraints, as illustrated in Figure 3.15, which should be applied for all west, north, east, and south edges.

Note that before the diffusion, the boundary pixels are unknown if there are no color sources or image sources. Thus, the diffusion of these tiles has to be done at the same time instead of rendering each tile one by one.

In order to satisfy these constraints, the Equation 2.1 has to be rewritten. Here, some definitions for global rendering will be given first. Let T be the tileset and the tileset T has n_t tiles. Consider each tile t_i with n pixels, $\{t_k^i | k \in 1...n, i \in 1...n_t\}$($t_k^i$ has RGB colors and color strength: (ra, ga, ba, a)). Let I be a new pixel set by putting all the pixels of all tiles together and it has a size of $(n_t \cdot n)$, $\{I_k | k \in 1...n_t n\}$. Then let $w = \{w_k | k \in 1...n_t n\}$ be the gradient set reflecting the constraints in the tile set. Most w_i are likely to be zero, as they correspond to the smooth interpolated content in the tiles. Hard constraints are pixel colors $\{C_k | k \in I_C\}$, where I_C are the colors emitted by curves.

$$I = \underset{Image\ I}{\arg\min}(\sum_{i=0}^{n_t n} |\nabla I_i - w_i|^2) \qquad , subject\ to\ I_k = C_k, \forall k \in I_C,$$
$$I_p = I_q, if\ p\ and\ q\ are\ neighbor\ pixels. \tag{3.7}$$

where ∇ is the gradient operator, w_i is the gradient of pixel i.

Remember that the RGB colors obtained after diffusion need to be divided by the color strength after diffusion.

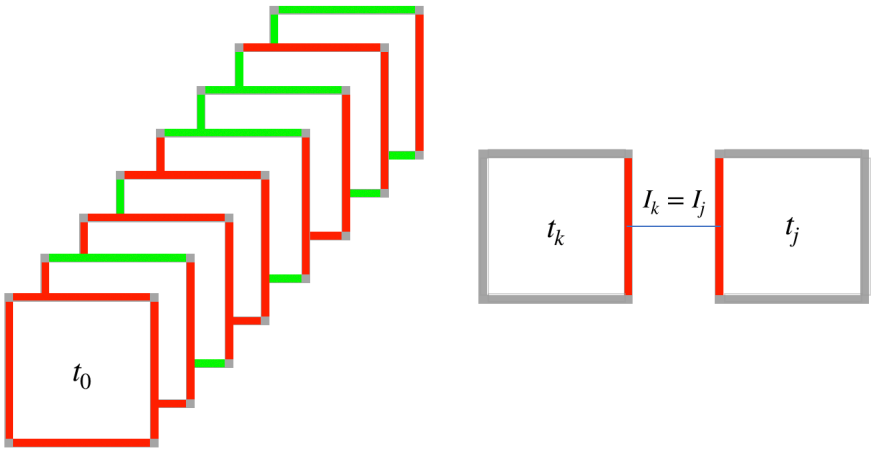


Figure 3.15: Soft constraints $I_k - I_j = 0$. The pixels I_j on the west edge of t_j should be the same as the its neighbor pixels on the east edge of t_k. Here, t_k can be $t_0, t_1, t_3, t_4, t_6, t_7, t_{10}, t_{13}$.

The least-squares solution to the linear system aims at satisfying the constraints of the color sources, border pixels, and image gradients as much as possible. Since all tiles are rendered together, the dimension of this linear system is very high, but, fortunately, the constraints matrix is very sparse, given that all constraints are local (e.g., divergence or color constraints).

3.4. Tiling Methods

Diffusion Mosaic generates large textures by small tiles, and it uses Wang Tiles as the basic tile. Due to the unique features of Wang tiles, it can create aperiodic tiling, which is very useful for reducing visual repetition. But for Wang Tiles, only tiles with the same color on adjacent edges can be put together, like in Figure 3.16, which means they cannot be tiled arbitrarily. Thus we need some tiling algorithms for the Wang tile.

(a) (b)

Figure 3.16: Wang tile match cases: a): tiles of aperiodic Wang set b): tiles of ordinary Wang set. The tile should match all its neighbor tiles

3.4.1. General Tiling Method

As mentioned before, Algorithm 1 requires tiles to meet some conditions to ensure that the tiling result is random. If not satisfied, other method needs to be taken. For example, the aperiodic Wang set as shown in Figure 2.3 has 4 colors but only has 11 tiles. Naively tiling the plane could lead to circumstances, where there are no eligible tiles remaining in the set, see Figure 3.17.

Therefore, backtracking algorithms are used to solve this problem. If there is no tile that can be chosen from the tile set for f_i, go back to previous place f_{i-1} and choose another tile. If all tiles for f_{i-1} are exhausted, then go back to f_{i-2} and repeat this process until a solution is found. In order to ensure there exists a possible tiling of the plane, the tile set should be designed carefully. The aperiodic Wang set used in this thesis, which is shown in Figure 2.2 has been proven that it can always tile the plane[14]. In this case, we can be certain that the backtracking succeeds. If the set is composed of tiles with arbitrarily colored edges, the approach might fail to tile the whole plane.

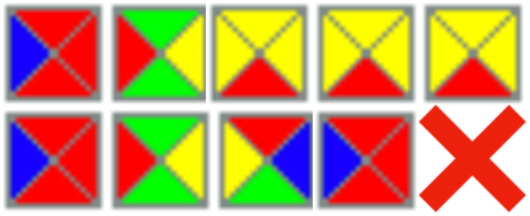


Figure 3.17: A failed tiling case: tile at $(4, 1)$ should have red colors on its north and west edge in order to match its neighbors, but there is no such tile in this set

3.4.2. Parallel Tiling Method

For a full Wang set, this tiling algorithm can be paralleled. The plane to be tiled can be divided into blocks. However, there are data dependencies when using this simple tiling algorithm because the tile

Algorithm 2 General Tiling Method

Require: $width,\ height,\ Tile\ Set\ T$
  repeat
    if $x = 0,\ y = 0$ **then**
      $T' \leftarrow\ T$
    else if $x > 0,\ y = 0$ **then**
      $T' \leftarrow tiles\ that\ satisfy\ f_w(x,0)\ =\ f_e(x-1,0)$
    else if $x = 0,\ y > 0$ **then**
      $T' \leftarrow\ tiles\ that\ satisfy\ f_n(0,y)\ =\ f_s(0,y-1)$
    else if $x > 0,\ y > 0$ **then**
      $T' \leftarrow\ tiles\ that\ satisfy\ f_w(x,y) = f_e(x-1,y)\ and\ f_n(x,y)\ =\ f_s(x,y-1)$
    end if
    if $T' = \emptyset$ **then**
      $backtracking$
    else
      $f(x,y) \leftarrow randomly\ choose\ a\ tile\ t\ from\ T'$
    end if
  until x = width, y = height
  return $tiling\ f$

should match all its north and west neighbor tiles(if they exist). With data dependencies, tiling all blocks at the same time could lead to mismatches on the boundary of the block.

Here, we divide the plane into 2 horizontal blocks. And as the boundary of two blocks, the middle row of the plane needs to be handled separately. We take a simple approach that the middle row $\frac{height}{2}$ is tiled first. Then tile the upper part of the plane and the lower part of the plane simultaneously. The tiling of the lower part is almost the same as Algorithm 1 but starts from $f(0, \frac{height}{2} + 1)$. Such blocks can be handled very similarly, only the position of the constraints changes (e.g., with only a horizontal separation into two blocks, the upper part will tile from bottom to top, as the constraints are below, while the lower part tiles from top to bottom)

In this way, mismatches on the boundary are avoided. This process is shown in Figure 3.18.

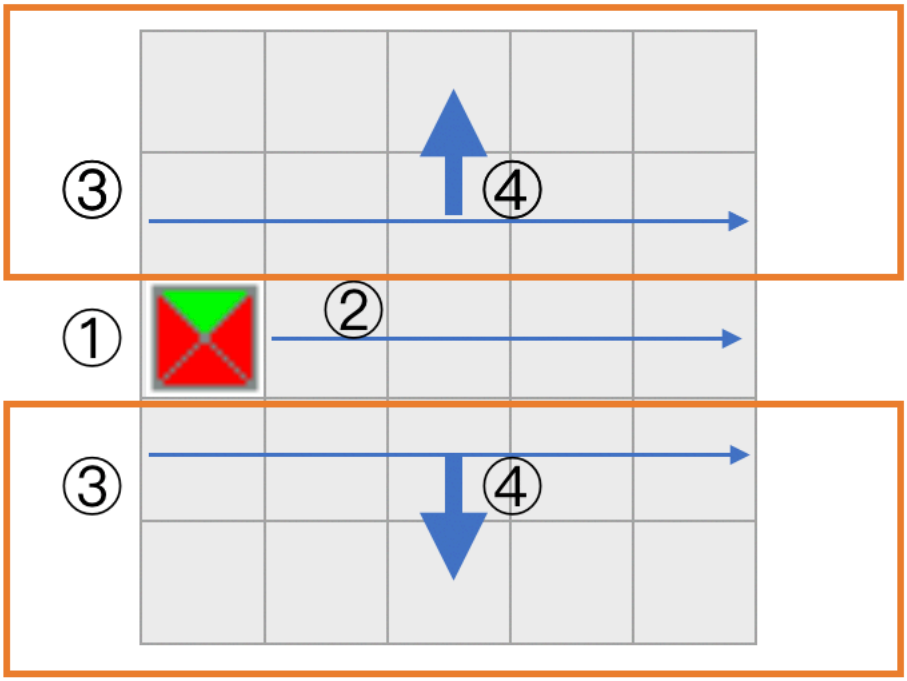


Figure 3.18: Paralleled stochastic tiling algorithm. Step 1: Randomly choose a tile from the tile set for the leftmost one of the middle row $f(0,0)$. Step 2: Find the tile whose left edge color is the same as its left neighbor's right edge's for the remaining tiles of the middle row. Step 3: Thread One chooses a tile whose south edge color is the same as its south neighbor's north edge's for the leftmost one above the middle row. Thread Two chooses a tile for the leftmost one below the middle row. Step 4: Thread One finds the tile whose left edge color is the same as its left neighbor's right edge's and south edge color is the same as its south neighbor's north edge's for the remaining tiles of the row. Thread Two finds the tile whose left edge color is the same as its left neighbor's right edge's and north edge color is the same as its northern neighbor's south edge's for the remaining tiles of the row. Step 5: Both threads repeat Steps 3 and 4.

3.5. User Interface Design

This chapter will introduce the graphical user interface design of our Diffusion Mosaic system. There are three main functions: drawing, tiling, and result-image viewing.

3.5.1. Drawing tools

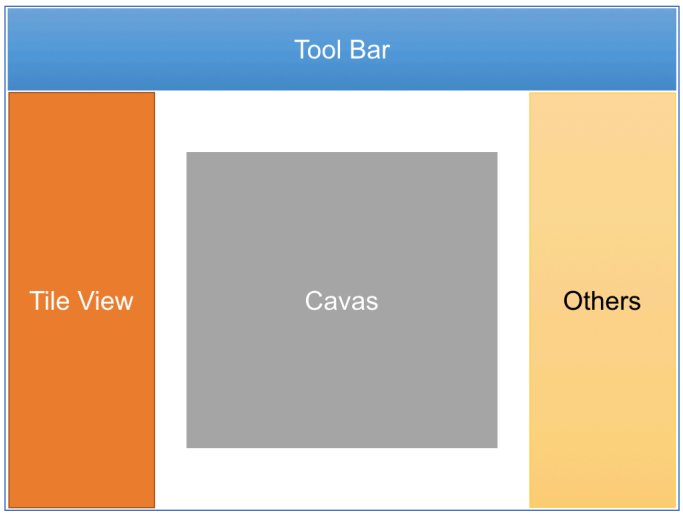


Figure 3.19: The layout of drawing tools.

The layout of the drawing tools is shown in Figure 3.19. In drawing tools, there are four main modules: *Toolbar*, *Tile View*, *Canvas* and *Others*.

For the drawing tools, a canvas is needed. Since the tool should allow the artist to define shapes outside the tile boundaries, the canvas is virtually infinite. Hence, it needs to be possible to drag the canvas and also to be able to return to the initial view focused on the current tile.

To draw different kinds of diffusion-curve control points: geometric, color and blur should be supported. This requires the program to allow for switching the control points. For color control points, RGB colors and strength can be selected in a color-picker interface. Figure 3.20 shows some of these aspects.

When creating a pattern, several curves are combined, requiring a button to start a new curve. Correspondingly, another button allows the user to delete a curve. If there are multiple curves, the user may need to select a particular curve, hence, a selection functionality is also needed. Further, two buttons are reserved to delete and edit control points. To avoid clutter, some buttons in the interface change their state according to the user action instead of statically executing one funtion. Figures 3.21 and 3.22 illustrate the interface.

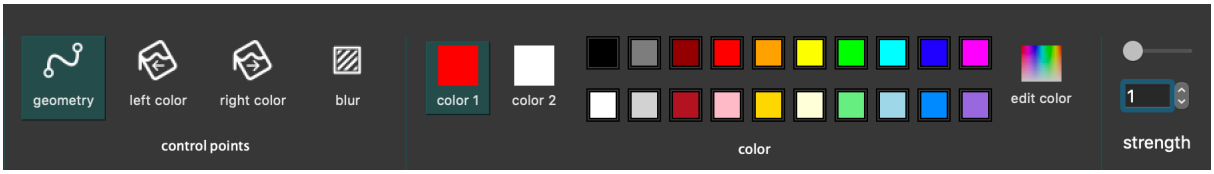


Figure 3.20: The part of the toolbar is related to the control points, color, and strength. This tool provides twenty standard colors and a button to open the palette for more colors. The strength is controlled by the spin box and slide bar.

Because the goal is to fill the Wang Tiles, the user should be able to select the tile to work on, which is conveniently presented to the user as a list view of all existing tiles. The top indicates the current tile, and the middle is a tile list with tile numbers, the corresponding Wang tile, and the tile content in form of thumbnails. The bottom is a small window with the same size as the tile that displays the content of the current tile. The tile view is shown in Figure 3.23.

In module *others* as shown in Figure 3.24, the user can choose whether to display control points, curves, color sources, and tile boundary information on the canvas. There are also image uploading,

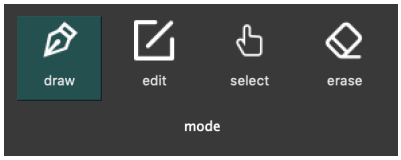
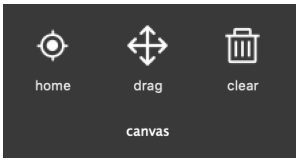
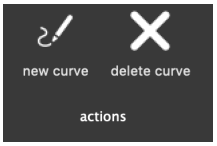


Figure 3.21: Mode switch: draw(draw control point), edit(edit control point position and color), select(select a curve), erase(erase a control point)



(a)



(b)

Figure 3.22: Buttons of a single action. a): Canvas action: home(back to the default position), drag(switch to drag mode), and clear(clear the content of canvas) b): Actions: new curve(add a new curve), delete curve(delete the current curve).

rendering, saving, and loading functions in this module.

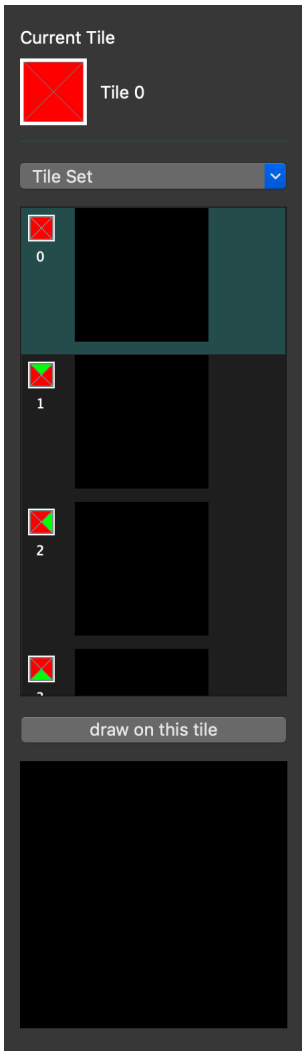


Figure 3.23: The module *Tile View* of drawing tool.

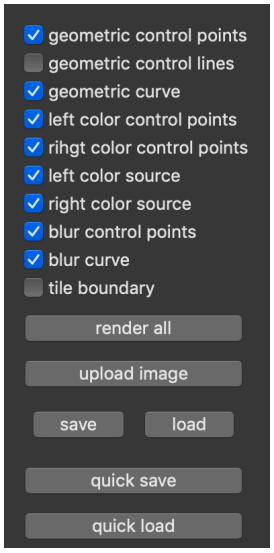


Figure 3.24: The *others* module of drawing tool.

3.5.2. Tiling tools design

The layout of the drawing tools is shown in the Figure 3.25. In tiling tools, there are also four main modules: *Tile View*,*Setting*, *Patterns* and *Wang Tiles*.

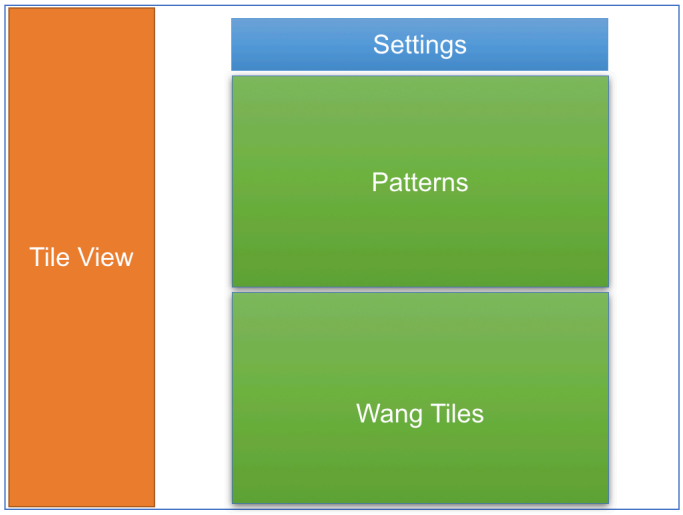


Figure 3.25: The layout of tiling tools.

3.5.3. Image Viewing design

The resulting image can already be seen in the tiling tools. Nevertheless, the pattern is superposed in form of a grid, which can be distracting. Thus, a final image viewing tool is also available to show the image without distractions. This tool also provides the option to zoom in and out, to save the image and to return to the original zoom level. The layout of the image viewing tool is shown in 3.26.

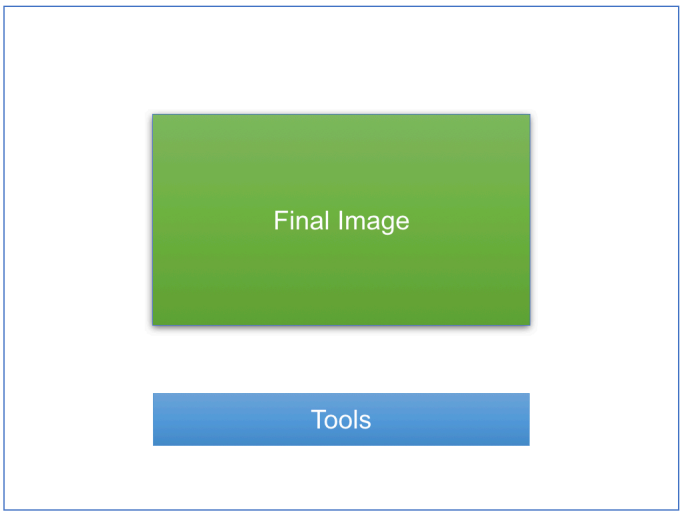


Figure 3.26: The layout of image viewing tool.

4

Implementation

This chapter will introduce how diffusion mosaic is implemented. In Section 4.1, we will introduce the programming language, library, and framework used in detail. In the Section 4.2, we will explain how the user graphical interface described in Section 3.5 is connected with specific functions.

4.1. Architecture

The *Diffusion Mosaic* is written in C++/C[12, 13]. The graphical interface is developed on the basis of QT[24] framework and OpenGL[17]. For the diffusion part, an open source program OpenNL[3] is used to solve the linear equations.

QT Framework Qt is a cross-platform application framework for desktop systems and embedded development. It includes an intuitive API, a rich class library, and integrated GUI development and internationalization tools. Applications developed by QT can be deployed on different operating systems without changing the source code and additional environment configuration. Therefore, the code of *Diffusion Mosaic* can run on the current mainstream PC operating systems like Mac OS and Windows. Signals and slots are the most core mechanism in Qt. Through the *connect()* function of the *QObject* object, communication between objects can be achieved and the UI can easily be connected to functions. Furthermore, QT provides an OpenGL module, which can be used directly without configuring the environment, and supports GLSL. Using the GUI components provided by QT, users can easily interact with the graphical interface instead of using the command line.

Multi-threading Programming Model In Diffusion Mosaic, many tasks are independent of each other, so multi-threading can be introduced to process multiple tasks at the same time. And Qt already provides a thread class *QThread*. So forked threads can handle those time-consuming operations while the main thread continues to handle GUI events. Besides, POSIX Thread (Pthread) is also used to accelerate the tiling process.

Mathematical Library Diffusion mosaic needs to solve linear equations for the diffusion process. However, writing a simple solver is relatively inefficient and cannot meet program requirements well. Moreover, since this is a really large matrix (albeit being actually sparse), simple methods will easily lead to insufficient memory. Therefore, an open-source library, Open Numerical Library(OpenNL), is used for better performance. This solver can handle sparse linear systems well and supports CUDA[20] for a suitable mapping onto graphics hardware.

4.2. Interface

As discussed in 3.5, there are three main functions: drawing, tiling, and image viewing in Diffusion Mosaic. These functions will be implemented via *tabwidget* in QT. Users can switch tabs to use different tools.

After drawing the draft of the pattern, by clicking the push button *render all*, the system will begin to compute the diffusion process, which will take some time. When diffusion is complete, the user can get their rendered image for each tile in tab *Tiling*.

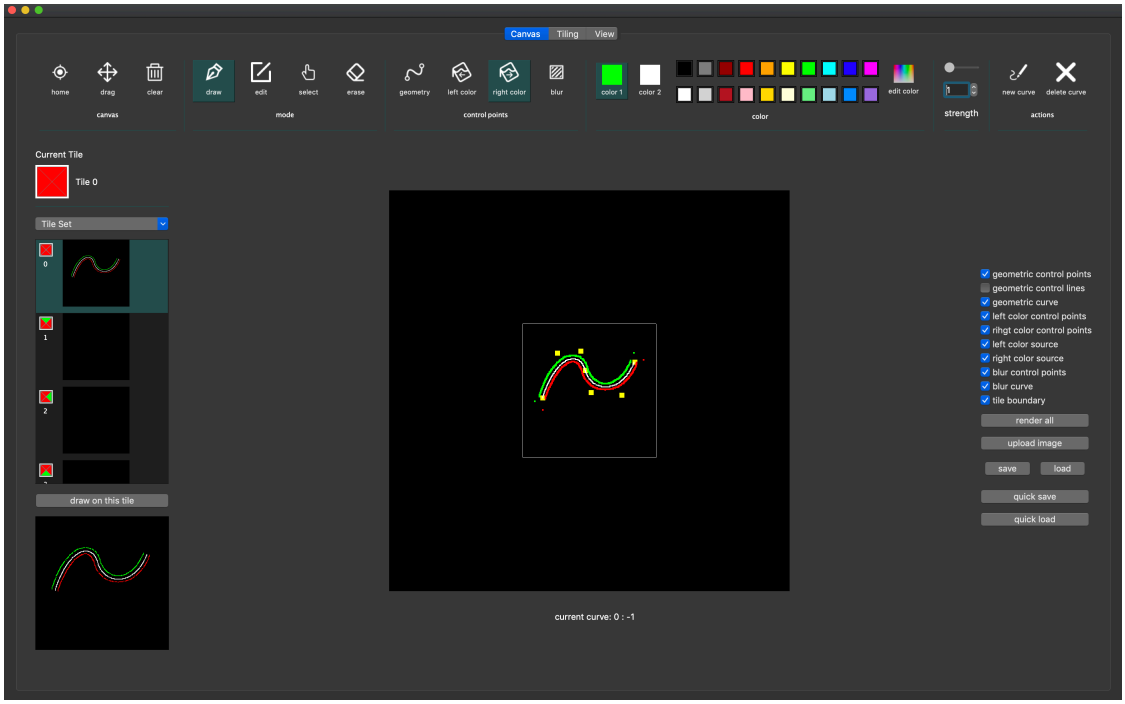


Figure 4.1: The overview of drawing tools.

By clicking the push button *generate* in *Tiling*, it will tile the plane of given width and height. When the size of the plane increases, the response time also increases. Most of the time will be spent on rendering the table widgets and computing the solution. When computing the solution in the main thread, the program would lose its responsiveness until a tiling solution is found. To improve the user experience, the computation process is handled separately by forking a new thread. Unfortunately, in the QT framework, the GUI events can only be processed by the main thread, which means the user still has to wait until the program finish rendering the table widgets. When the size becomes larger, the rendering time will become very long.

After generating a tiling, the user can go to the next tab to view the final integrated large image. The user can zoom in to view the details of the image and save it for a position specified by the user.

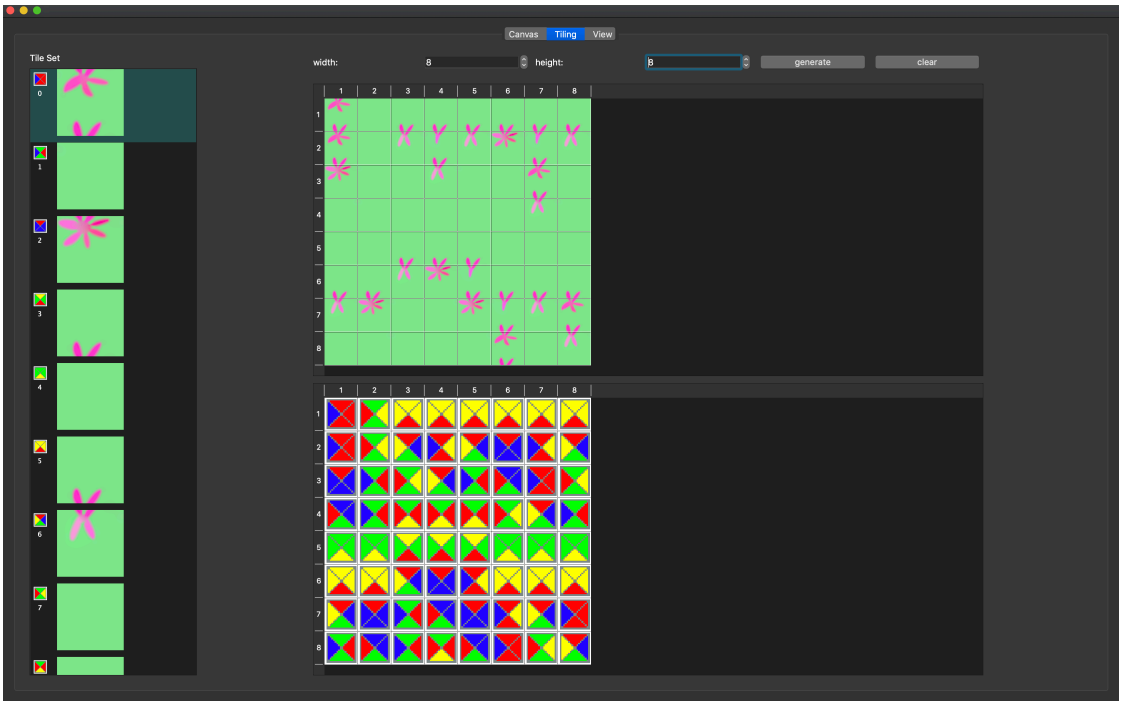


Figure 4.2: The overview of tiling tools.

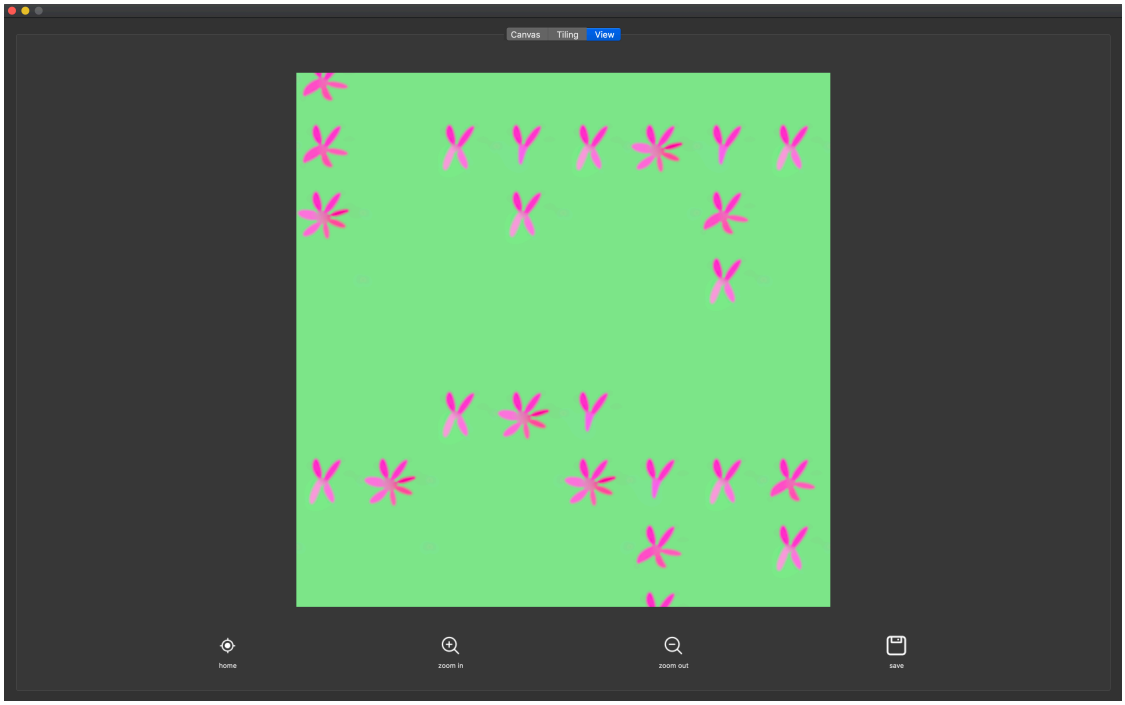


Figure 4.3: The overview of image viewing tool.

5

Results

This chapter will show visual results created by Diffusion Mosaic, including the patterns and the large synthetic textures. Also, some quantitative data for generation of these images and textures will be given. The results of this thesis are achieved in the following environment:

- CPU: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz (4 cores)
- GPU: AMD Radeon Pro 555, VRAM 2GB
- Memory: 16GB

5.1. Quantitative Results

This section presents quantitative results of the tiling algorithm. It mainly measures the cases where the target area size is 5×5, 15×15, 20×20, 25×25, 30×30, 35×35, 45×45, 50×50.

For the Wang set shown in Figure 2.3, there are two tiling algorithms available, one is sequential, and the other is parallel. Note that the parallel algorithm here is implemented in an external C file. It needs an additional file read operation to get the tile-set information and to return the tiling results. The sequential tiling algorithm is implemented in C++ with QT. The comparison of these two algorithms is shown in Figure 5.1.

The parallel stochastic algorithm has data IO operations. This part of the overhead cannot be ignored. And its computation time is shown in Figure 5.2. The data IO proportion is shown in Figure 5.3. Figure 5.4 illustrates the relationship between computation, IO time, and total time.

There is no quantitative result of the general algorithm because the general algorithm is not stable. In the best case, the tiling result can be found in one try without backtracking, and in the worst case, all possible choices have to be traversed.

After the tiling algorithm is executed, the system has to take some time to load the rendered image data and display it. The cost of this part is not small. Figure 5.5 shows the time that QT spends on setting the table widget data after getting the tiling results. The displaying process was not optimized and can take more than a second in our prototype interface.

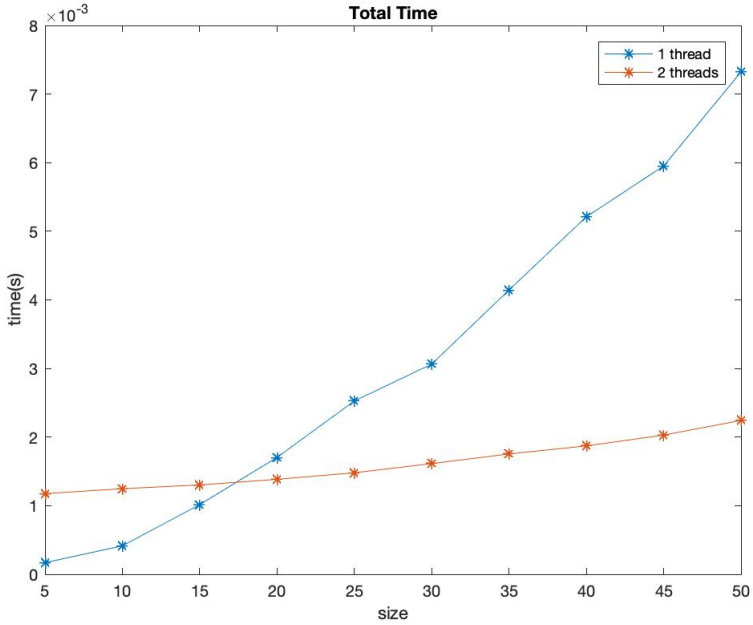


Figure 5.1: Comparison of parallel and sequential stochastic tiling algorithm: Parallel arithmetic is not fast at the beginning, but when the size is over 18, the parallel algorithm starts to be much faster than serial arithmetic

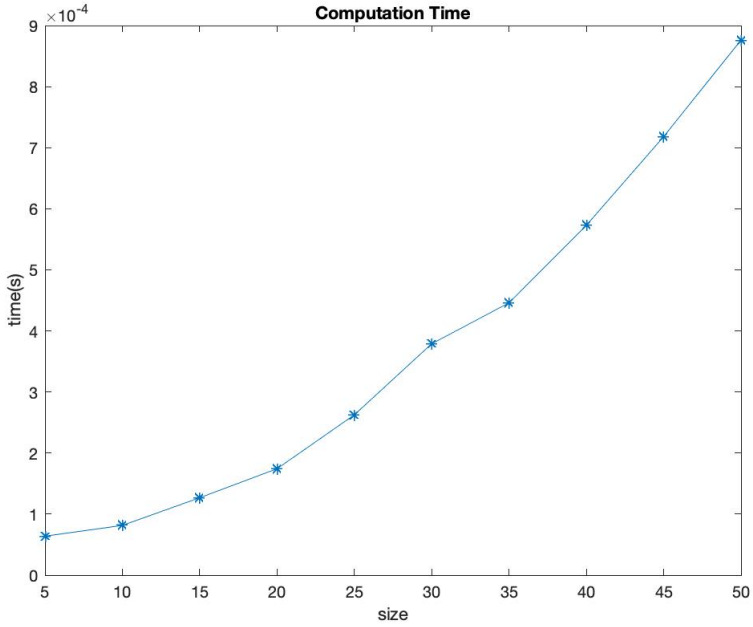


Figure 5.2: Computation time of parallel stochastic algorithm.

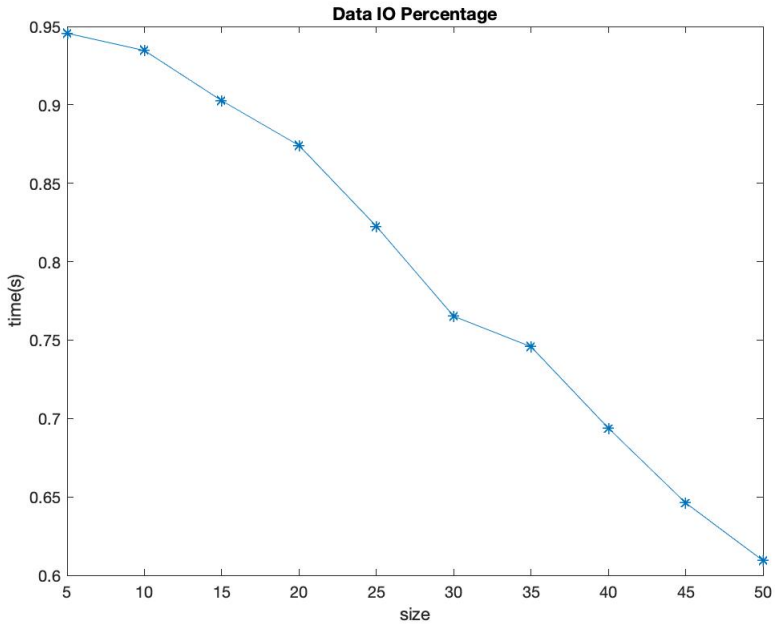


Figure 5.3: Data IO percentage of parallel stochastic algorithm.

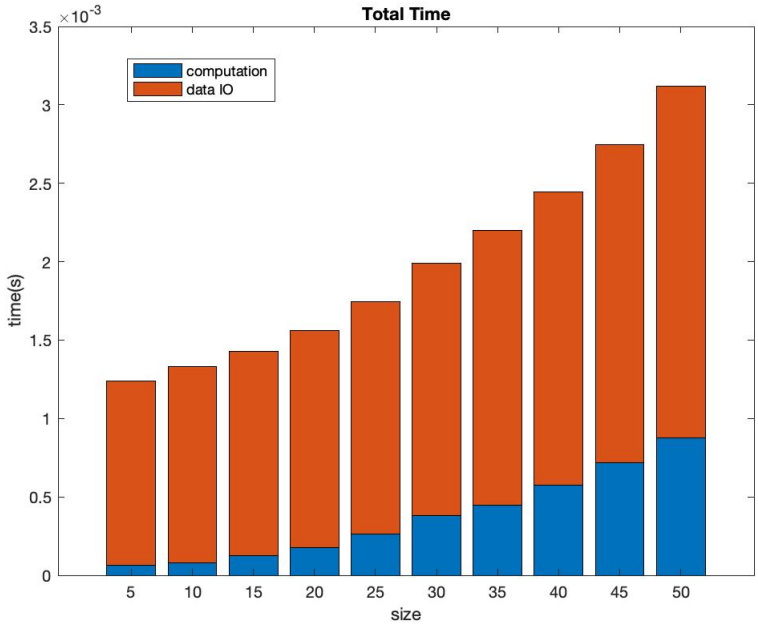


Figure 5.4: Computation time and data IO time of parallel stochastic algorithm.

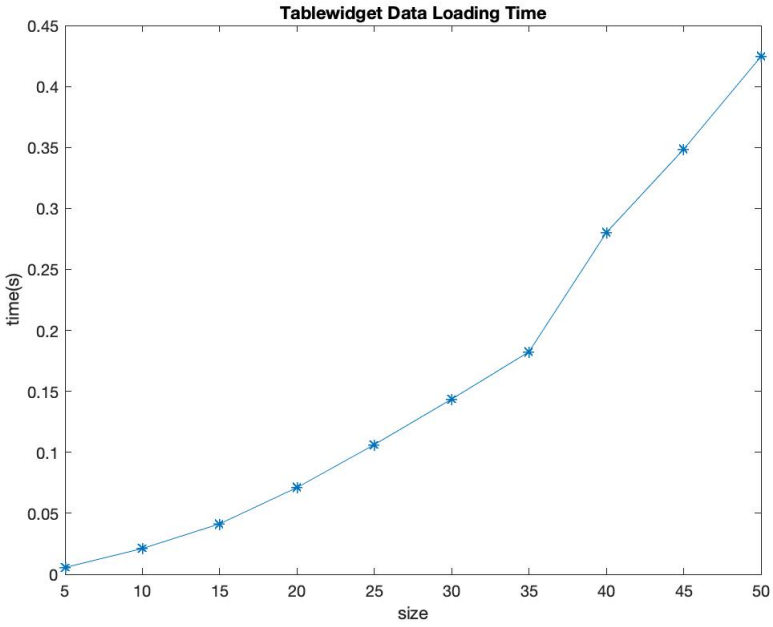


Figure 5.5: Table widget data loading time.

5.2. Visual Results

In this section, we will show the synthesis images of Diffusion Mosaic.

Case 1 Figure 5.6 shows the rendered images and corresponding Wang tile. Figure 5.7 shows the tiling and Figure 5.8 shows the final image. This case mainly shows the content synchronization update mechanism of Diffusion Mosaic, which is convenient when generating cross-tile patterns.

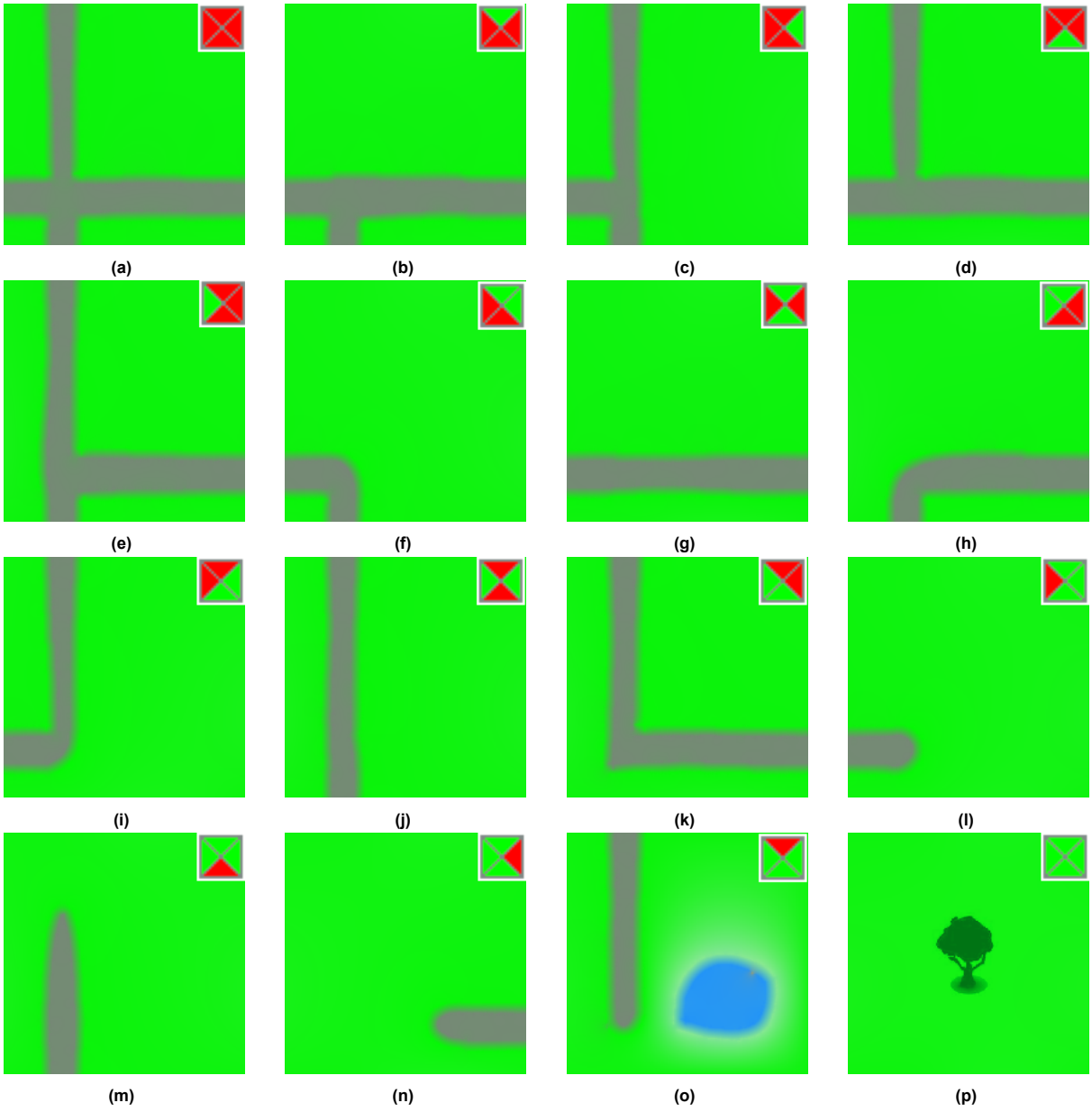


Figure 5.6: Case1: all rendered images and its corresponding Wang tile. Original tile size: 200×200

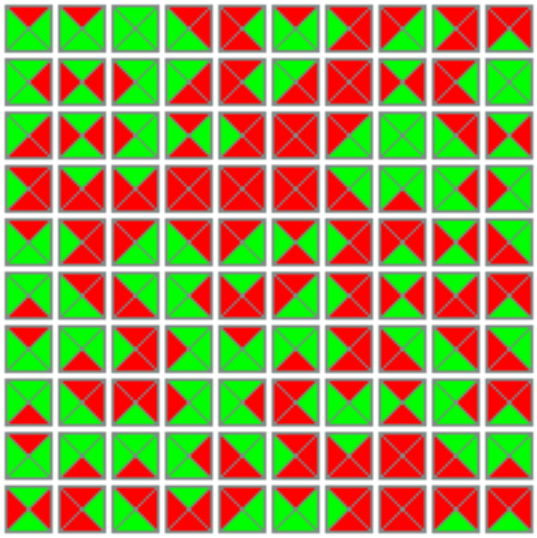


Figure 5.7: A tiling of case 1 in Wang tile form. Size: 10×10

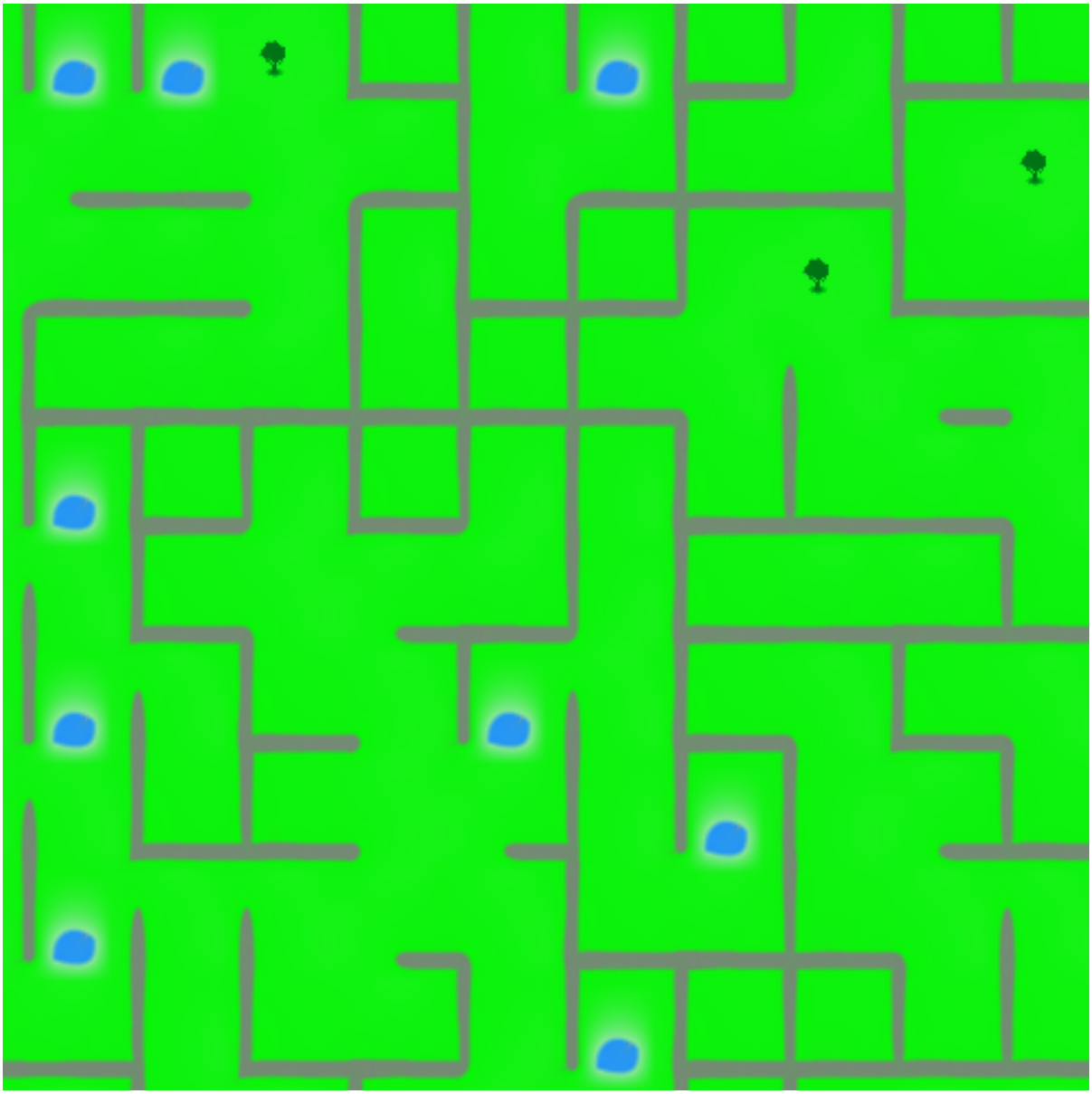


Figure 5.8: The final synthesized image of case 1. Green represents grass, gray stripes are path, and blue blocks are ponds. There are trees on the grass. Each tile is down scaled to 50×50. Size: 500×500.

Case 2 Figure 5.9 shows the rendered images and corresponding Wang tiles. Figure 5.7 shows the tiling and Figure 5.11 shows the final image. This case shows that Diffusion Mosaic can be used to create artistic patterns. The color transition between different tiles is smooth due to the cross-tile optimization.

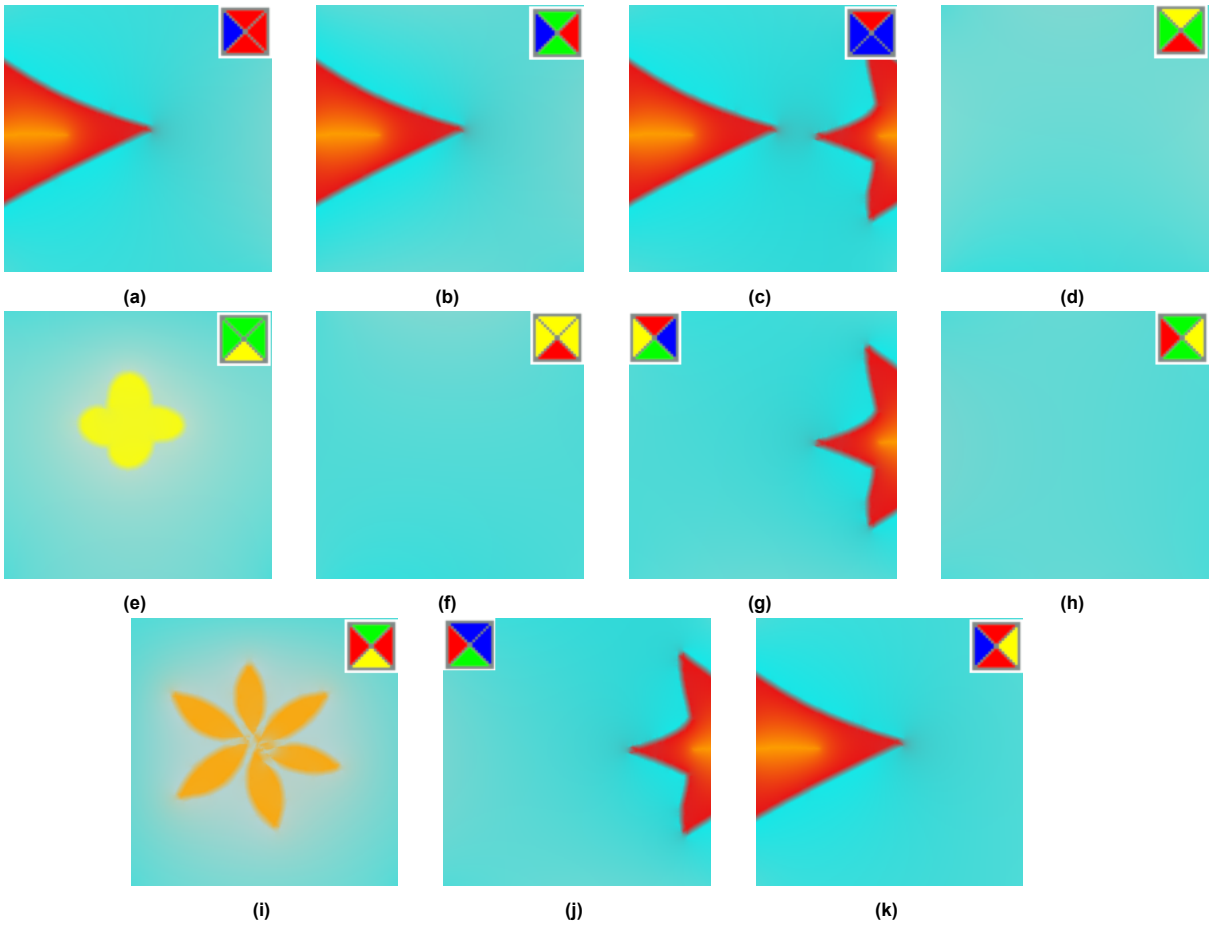


Figure 5.9: Case2: all rendered images. Original tile size: 200×200

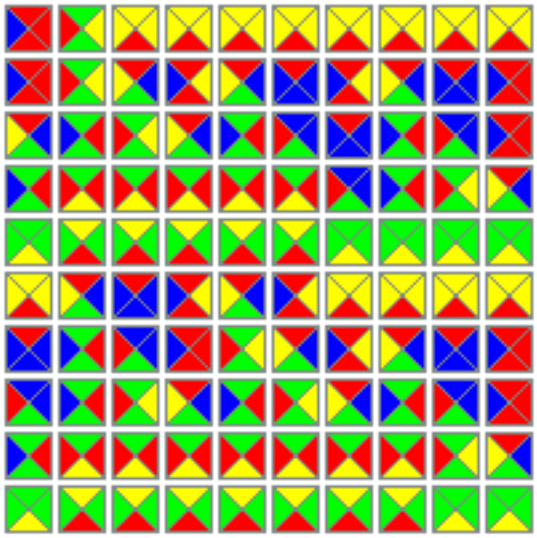


Figure 5.10: A tiling of case 2 in Wang tile form. Size: 10×10

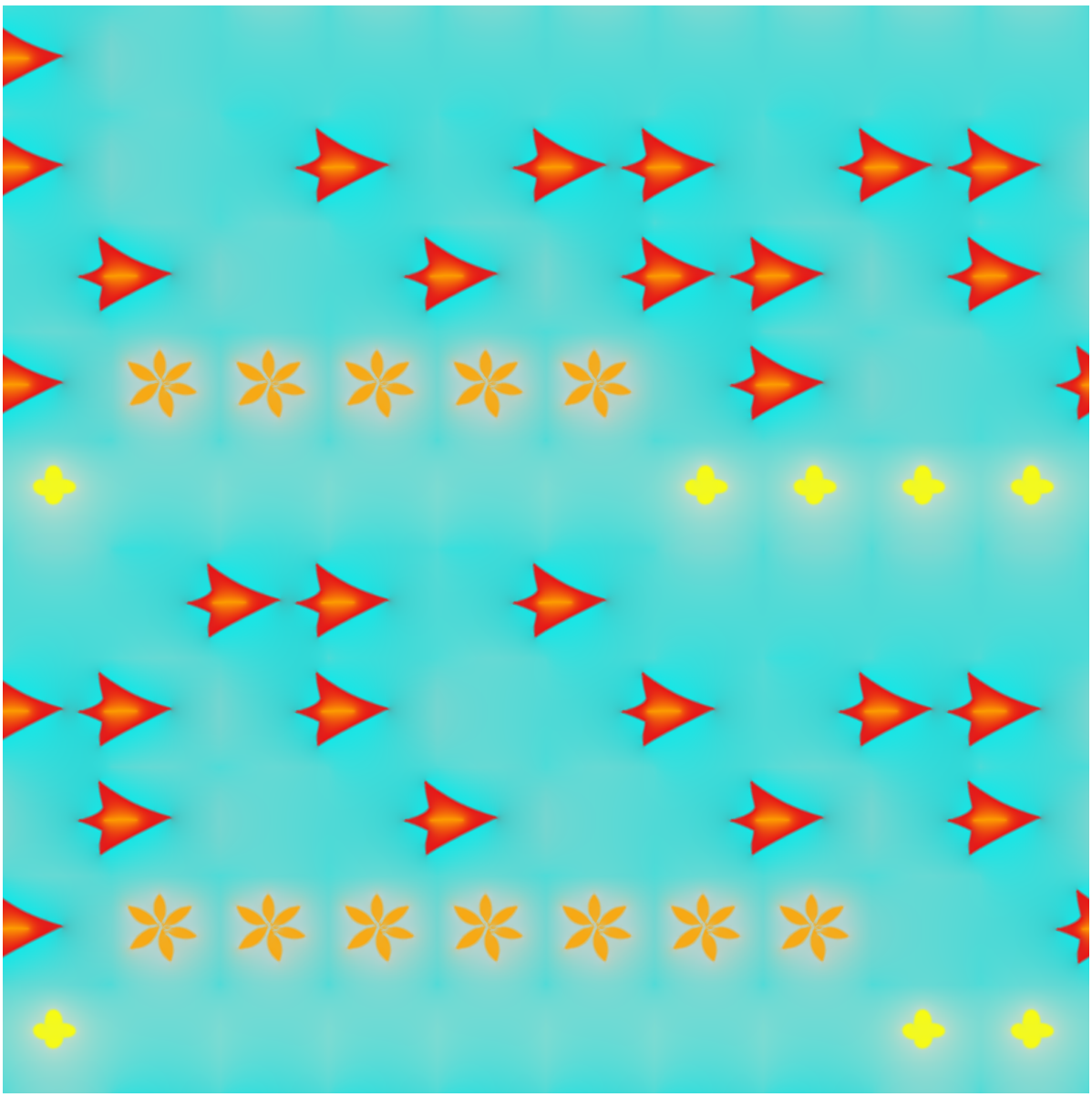


Figure 5.11: The final synthesized image of case 2. There are three different patterns in this case. And the back ground color is cyan and white. This case has a smooth color transition. Original size: 2000×2000

Case 3 This case illustrates that Diffusion Mosaic can be used to create height maps that can be converted to terrains - one of the useful additional applications of diffusion curves.

Case 3a Figure 5.12 shows the rendered images of all tiles. The tile order is the same as in Figure 5.9. Figure 5.13 shows the tiling, and Figure 5.14 shows the final image. For the height map, the image needs to be converted to a RAW format first. The terrain can then be directly generated from Figure 5.14 using Unity, resulting in the geometry illustrated in Figure 5.15. The terrain can be smoothed and textured, leading to the final result in Figure 5.16.

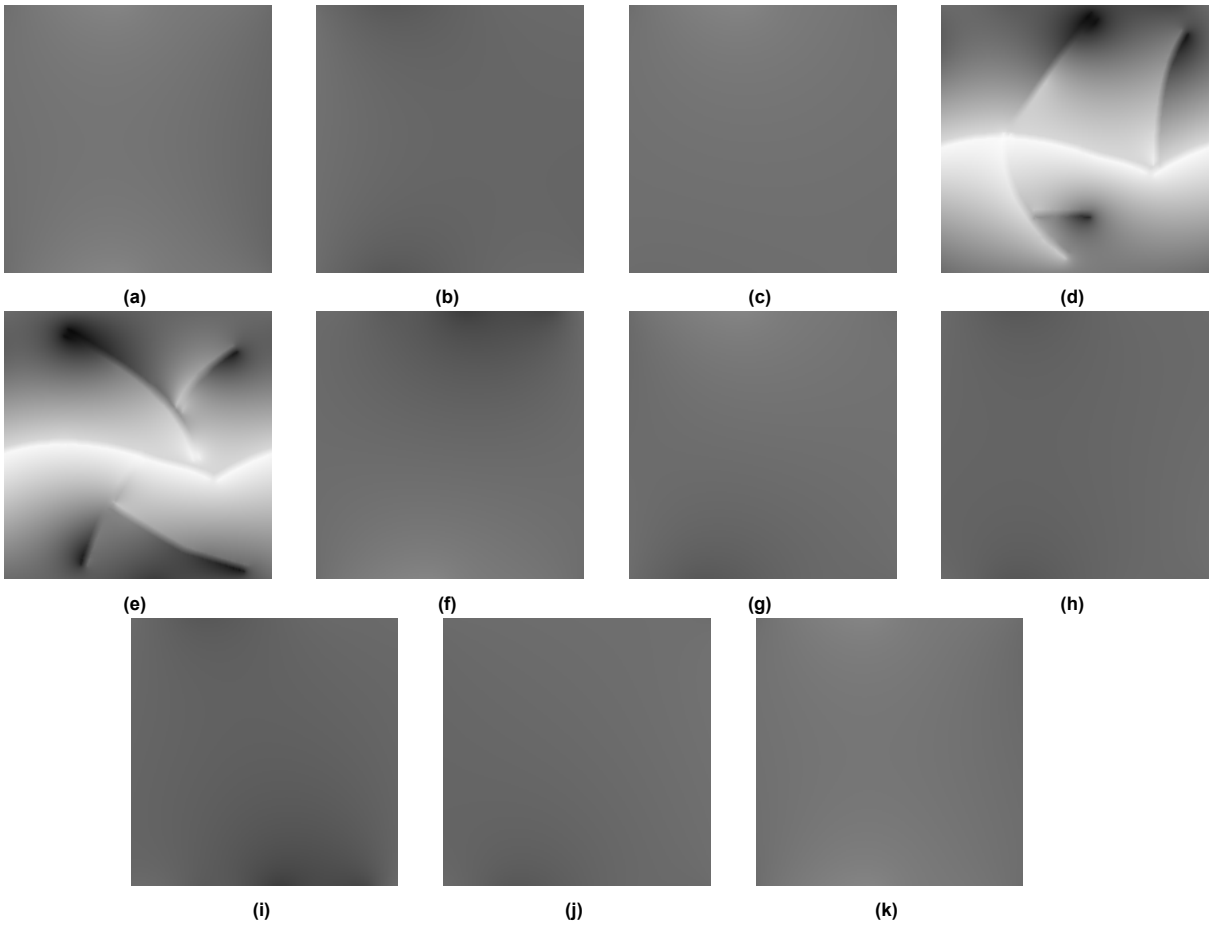


Figure 5.12: Case3a: all rendered images. Original tile size: 200×200

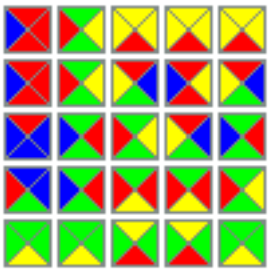


Figure 5.13: A tiling of case 3a in Wang tile form. Size: 5×5

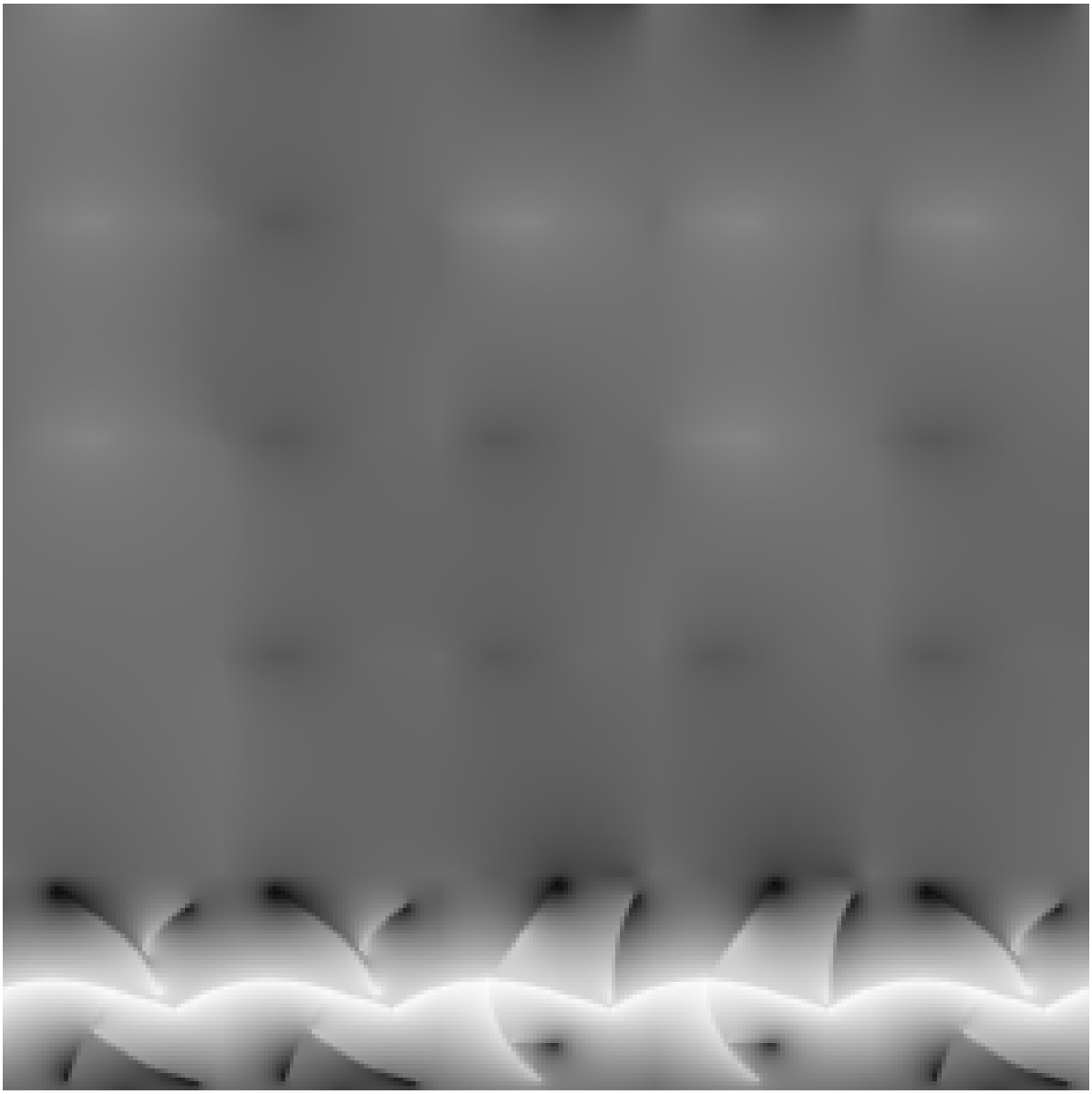


Figure 5.14: The final synthesized image of case 3a. Because the target image is a height map, the grayscale pattern is enough. Each tile is down scaled to 50×50. Size: 250×250

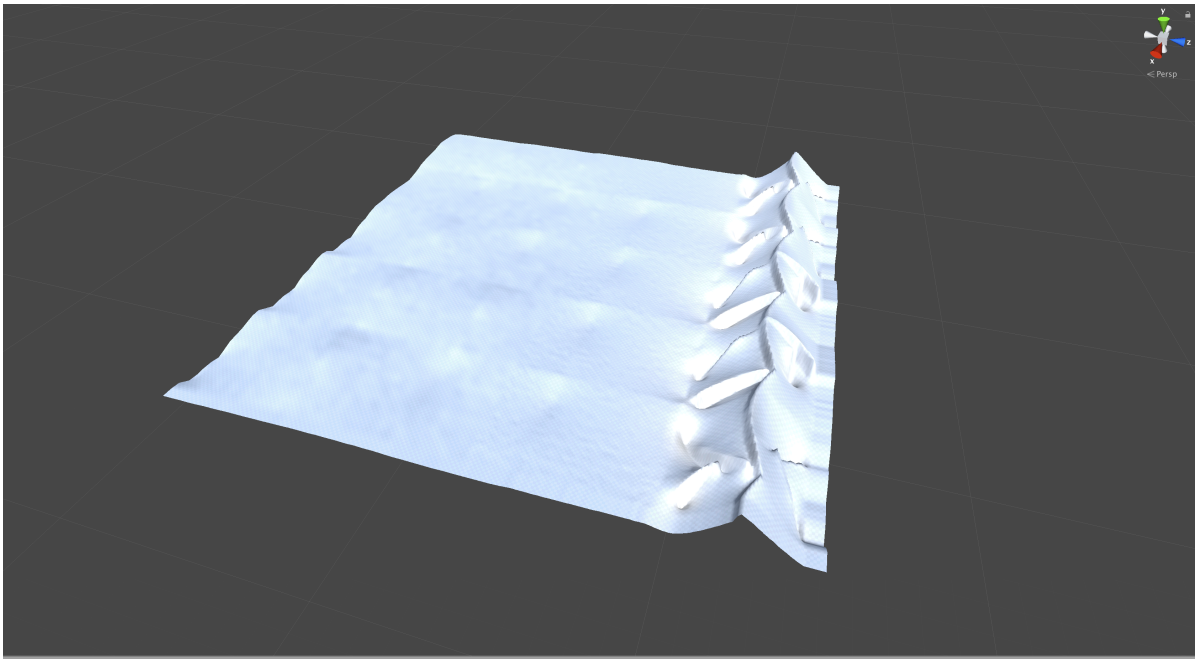


Figure 5.15: The height map generated by this pattern. Height map size: $250 \times 250 \times 30$

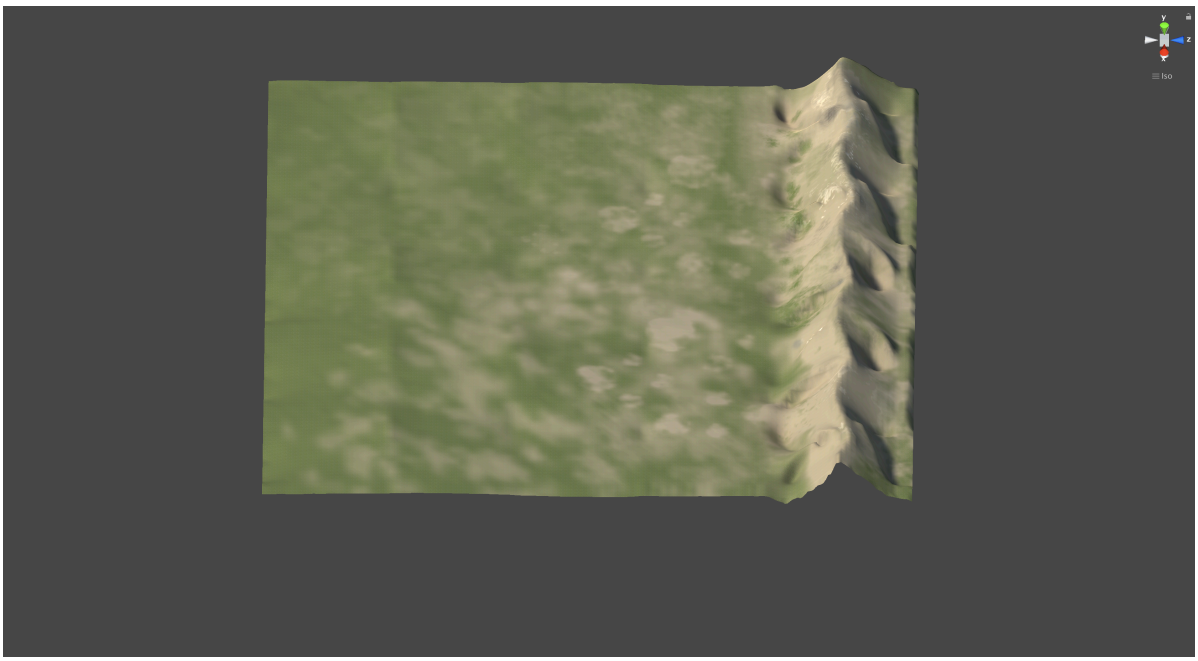


Figure 5.16: The terrain generated by this pattern with textures. 'Mountains and plains'. Terrain size: $250 \times 250 \times 30$

Case 3b Figure 5.17 shows the rendered images of all tiles. The tile order is the same as in Figure 5.6. Figure 5.18 shows the tiling, and Figure 5.19 shows the final image. The terrain(Figure 5.21) is generated from Figure 5.19 which is illustrated geometrically in Figure 5.20.

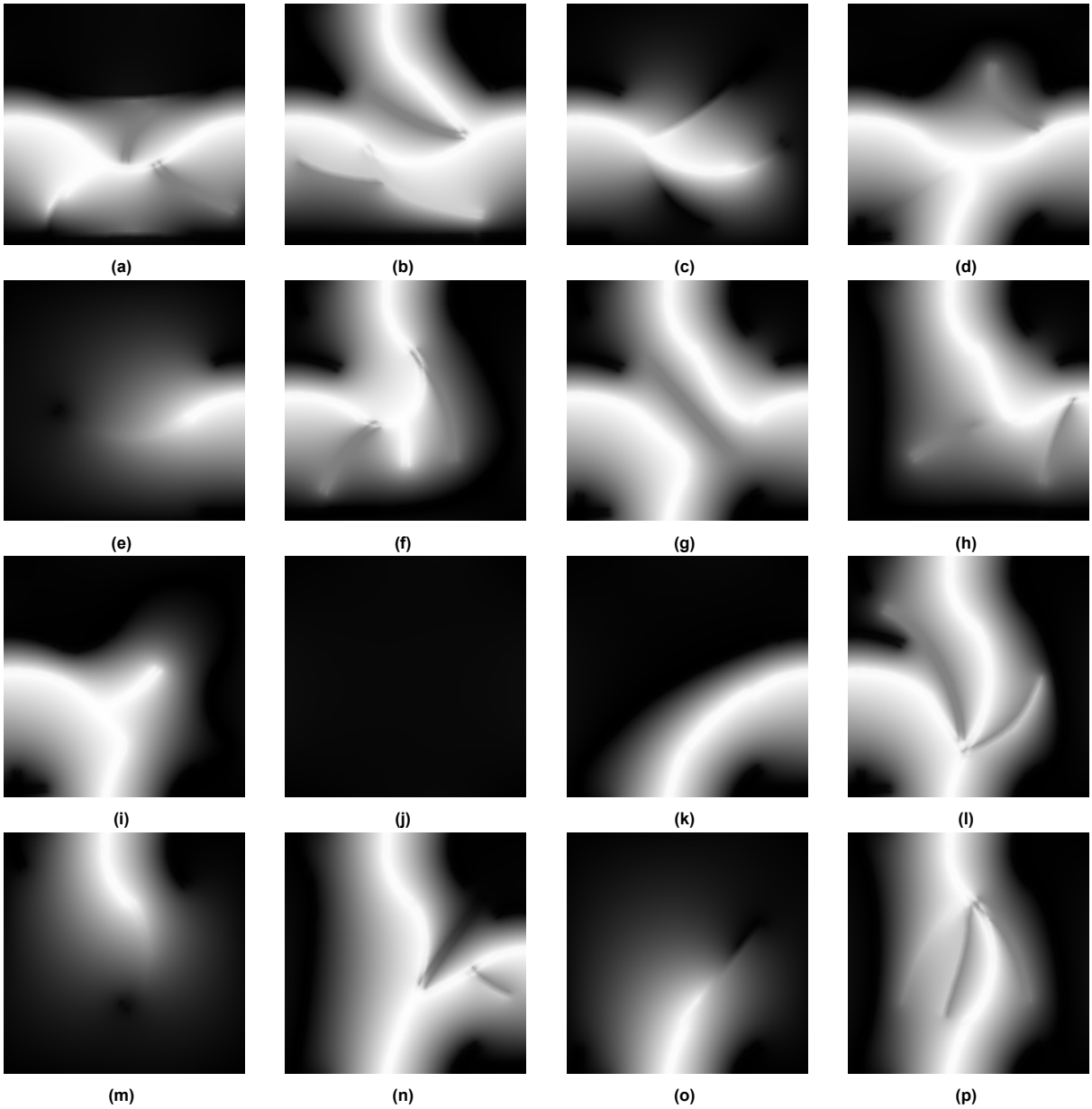


Figure 5.17: Case3b: all rendered images and its corresponding Wang tile. Original tile size: 200×200

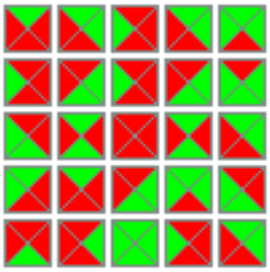


Figure 5.18: A tiling of case 3b in Wang tile form. Size: 5×5

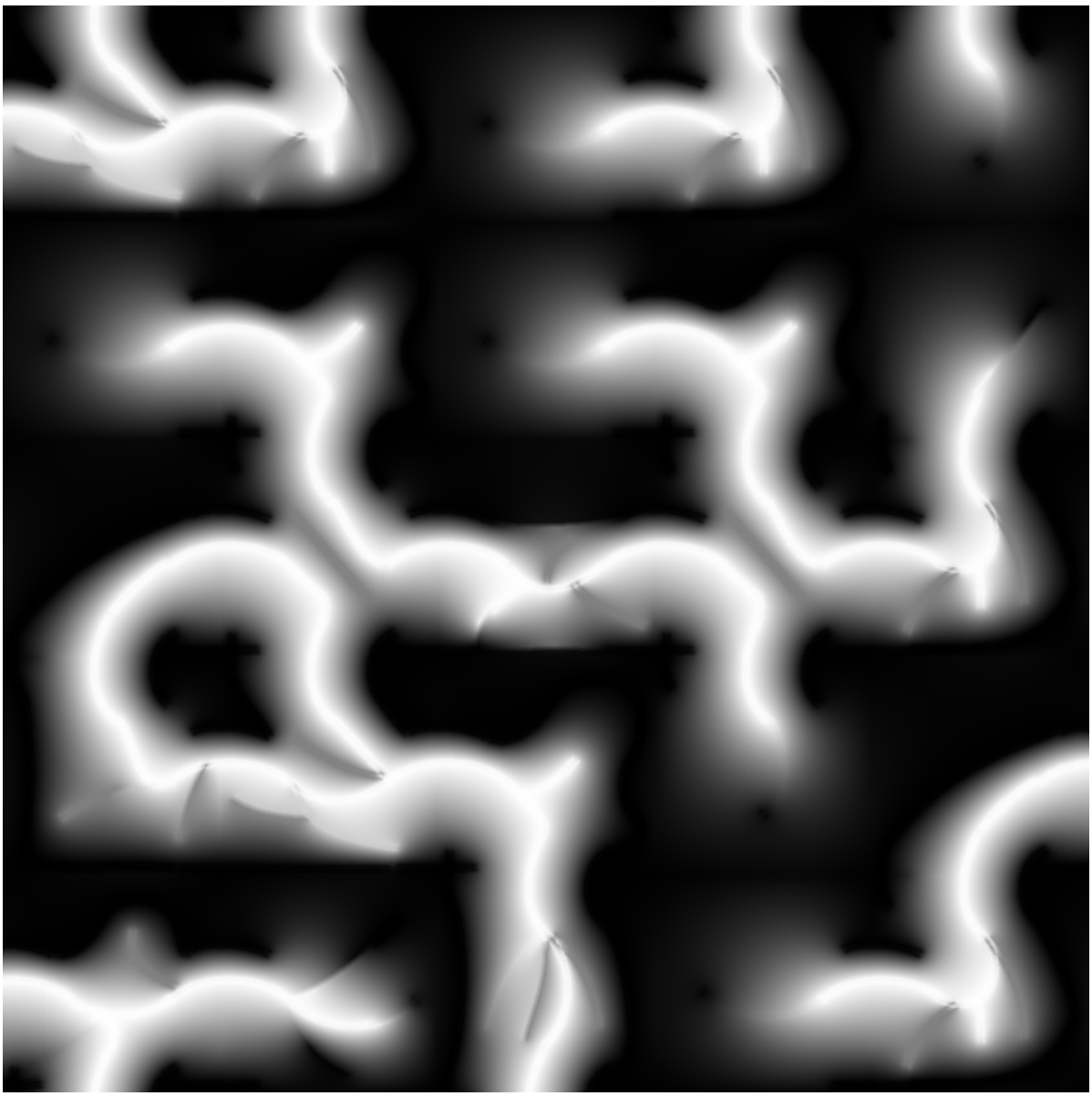


Figure 5.19: The final synthesized image of case 3b. Because the target image is a height map, the grayscale pattern is enough. Each tile is the orignial size 200×200. Size: 1000×1000

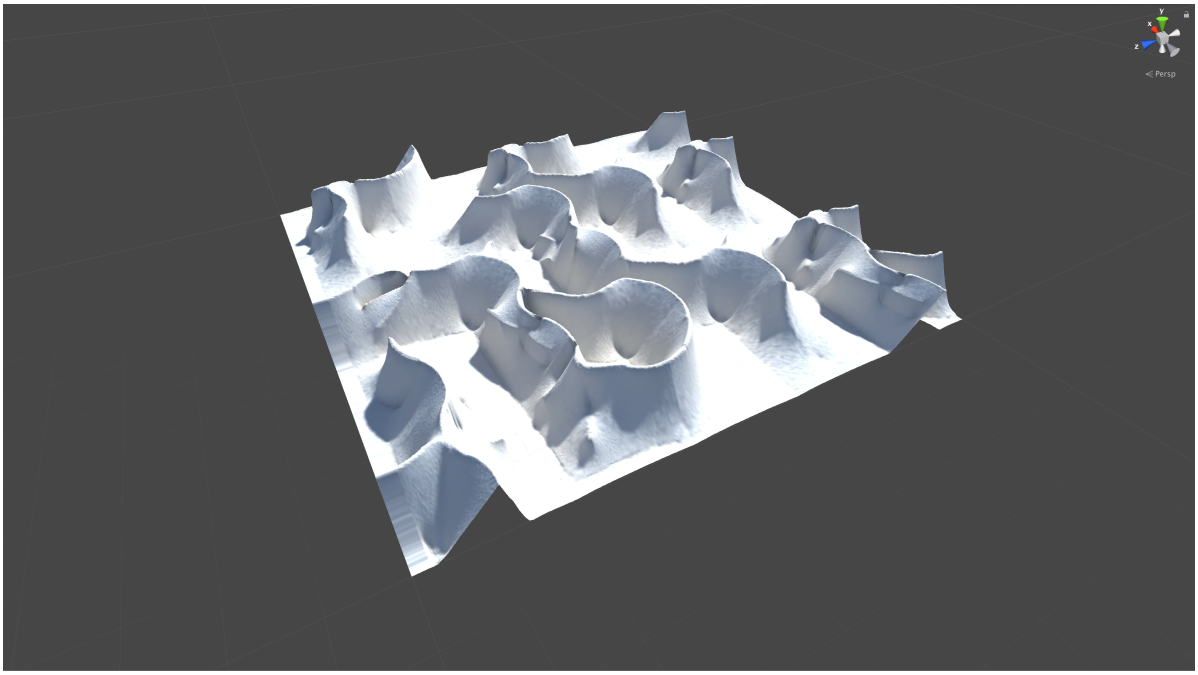


Figure 5.20: The height map generated by this pattern. Height map size: $1000 \times 1000 \times 100$

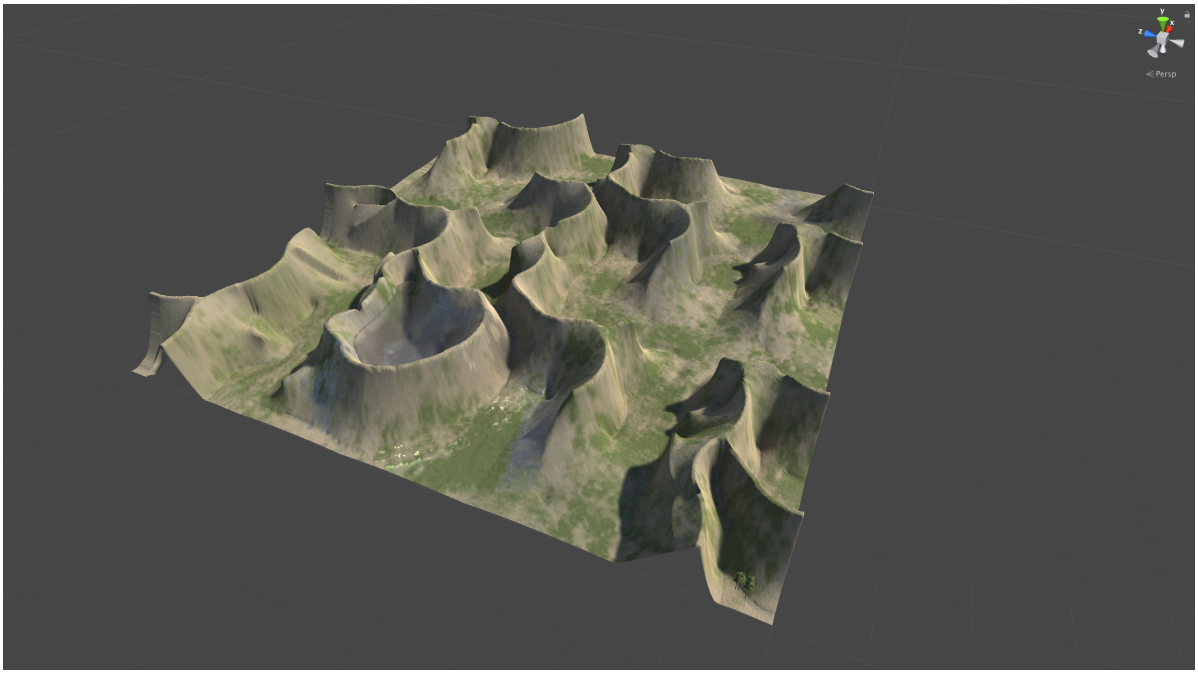


Figure 5.21: The terrain generated by this pattern with textures.'Mountains and an alpine lake'. Terrain size: $1000 \times 1000 \times 100$

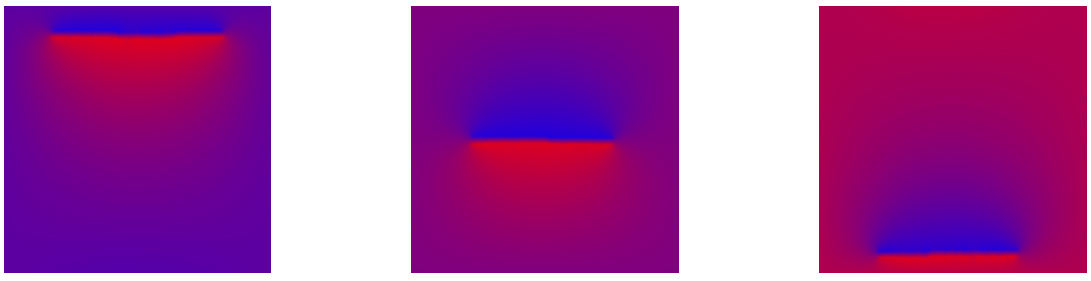
6 Discussion

6.1. Visual Results

From the results, it can be seen that if the assigned background colors are similar, the transition will be smooth after the color diffusion. But if the color difference is significant, there will be some artifacts.

Case 1 is a simple example, and it almost does not involve color blending. Case 2 has some color blending: cyan and white, but these two colors are very close. Finally, case 3 shows that the grayscale of the diffusion result can be used as a heightmap to generate a terrain. The results shows the feasibility for this application although the additional tools for the control of the landscape would be useful to add such as the format conversion tools like Photoshop, and rendering engines like Unity.

In the current diffusion system, the position of the curve in the tile will affect the final diffusion result, which can be considered a limitation. The strength of the color source, which is close to the border, becomes stronger. One possible reason is that a tile has multiple neighbors. The color sources close to the border will be the first to diffuse its color information to all its neighbor tiles. The color source of the other side needs to diffuse the entire tile first then diffuse to other tiles, which causes the color strength received by other tiles to be further away. This phenomenon only occurs when an edge has multiple neighbor tiles. If this boundary edge has only one neighbor, then this phenomenon does not exist. When there are multiple neighbor tiles, the color sources close to the boundary will be enhanced, and the more neighbor tiles, the more noticeable this phenomenon is. This phenomenon is shown in Figure 6.1. There is only one red-blue curve in one tile whose edge colors are the same.



(a) (b) (c)

Figure 6.1: The color sources near the border are enhanced.(a) The final image looks bluer. (b) The final image looks normal. (c) The final image looks more red.

There is another phenomenon which is shown in Figure 6.2. The area pointed by the orange arrow looks gray. This is because the color on the tile border is a mixture of red, blue, and green. The user can change the color or put this curve in another position to avoid this. Usually it is easy for users to predict the result of mixing two colors like blue and red, but adding other colors such as green may make the result not be easy to predict. The user should consider these in advance during the drawing process.

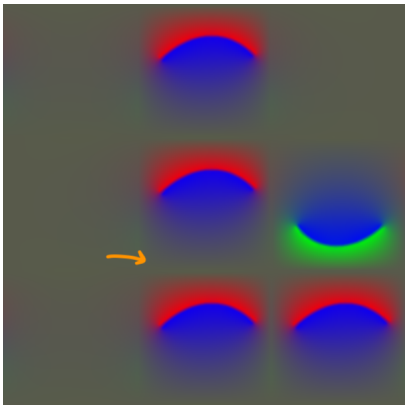


Figure 6.2: The border color looks gray instead of purple or brown.

6.2. Performance

From the results, it is obvious that the time cost of the program is mainly spent on solving the diffusion equations, and the overhead of others like rendering Bezier curves and control points is relatively small. Only when the size of the target area is large, the QT program will spend much time rendering the table widgets.

Tiling The results show that regardless of the *QThread* provided by QT or the *Pthread* of POSIX, the execution of multi-threaded and computationally intensive tasks in Qt is not necessarily faster than an external multi-threaded C program. Therefore, we wrote a multi-threaded tiling program in another file and use the C++ function *system* to execute it. We also tried to use *MPI* to parallelize the tiling process. Still, when using the function *system* to execute an external C program using *MPI*, it takes an unusually long time to initialize *MPI*.

Using an external C program to do the parallel tiling will have additional IO overhead because it cannot access the program data directly. Instead, it needs a file read operation to obtain the Wang-Tile information. For a small tile set, IO takes much more time than computation, making the parallel method slower than the non-parallel method. Through experiments, we found that when the tile set is about 18 $\times$18, the elapsed time of both ways is very close. Therefore, the tiling algorithm will be switched to the parallel method only when the plane size is larger than this size. The interesting point is that the external C program with *Pthread* can get a speedup of more than n(thread number) compared with tiling algorithms in QT. Nevertheless, a more optimal implementation would directly produce the tiling on the GPU but this is left for future work.

This thesis also proposes a general tiling algorithm, which uses a backtracking strategy, but this method may be extremely slow in some cases. In the worst case, it needs to traverse all the possible choices. Fortunately, one feature of Wang Tiles is that mathematicians claim that for certain sets, one can always tile the plane of any size, hence, the algorithm will always conclude.

Diffusion The three color RGB channels are independent of each other, but they all have a relationship with the color strength, which is only computed once. Further, if the color strength is everywhere the same, there is no need to calculate it at all. The performance of the diffusion process depends on the selected solver and whether any acceleration strategy is enabled. Our results typically take around $20 \sim 30$ seconds to compute for each color channel and color strength(on CPU).

7

Conclusion

This research led to a texture synthesis tool for artists. This synthesis method is a tile-based method, which makes full use of some characteristics of Wang Tiles. A large and non-periodic texture can be obtained. Moreover, this method makes full use of the diffusion characteristics of diffusion curves. It adds more constraints to the diffusion equation to integrate the properties of Wang Tiles, ensuring that the tile content remains seamless across compatible boundaries. Compared with the traditional synthesis method, this method utilizes an image rendering process instead of searching for patches, as done in previous work. In other words, once the content of each tile is rendered, there is no need for an additional image matching anymore, and the advantages of A tile-based synthesis method are maintained; fast and less memory.

This system integrates drawing tools and tiling tools. Users can draw patterns or add images through drawing tools without being restricted by the size of the canvas. The system also provides two sets of Wang tiles for users to choose from and offers corresponding tiling algorithms.

The most intuitive use of this system is to allow artists to create patterns. In addition, this system can also be used to create game maps and height maps which can be used to generate terrain.

Future Work While developing this tool, we noticed some problems that can be solved in the future.

The first point is usability. Because the target user group are artists, they often do not know much about computer graphics. Therefore, this tool needs to be improved to become more user-friendly. Some usability improvements are as follows:

When the user draws the curve outside the tile during the drawing process, the user knows if the curve intersects with an edge of a tile. However, if the curve intersects with more than one edge or if the curve is too long, the synchronization content update mechanism will cause many unexpected updates. Thus a more user-friendly mechanism to illustrate the possible chain of consequences of such drawing would be useful.

Users can be confused by too many control points. It is necessary to design a more convenient and intuitive element to indicate the curve's control point.

The second point is the diffusion process. The color transitions at the boundaries of tiles are smooth by our linear-system design, due to our smoothness constraints. This diffusion model is very simple and more sophisticated models could be adopted to deal with more complex situations.

Finally, the diffusion curves could be replaced with an extended definition described in Poisson vector graphics[11], to produce photo-realistic effects. Moreover, this extension enables seamless cloning and multi-layer images, which is used to solve overlaps.

References

[1] Michael Ashikhmin. “Synthesizing natural textures”. In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, pp. 217–226.

[2] Hedlena Bezerra et al. “Diffusion constraints for vector graphics”. In: *Proceedings of the 8th international symposium on non-photorealistic animation and rendering*. 2010, pp. 35–42.

[3] Luc Buatois, Guillaume Caumon, and Bruno Lévy. “Concurrent number cruncher: An efficient sparse linear solver on the GPU”. In: *International Conference on High Performance Computing and Communications*. Springer. 2007, pp. 358–371.

[4] Michael F Cohen et al. “Wang tiles for image and texture generation”. In: *ACM Transactions on Graphics (TOG)* 22.3 (2003), pp. 287–294.

[5] Karel Culik II. “An aperiodic set of 13 Wang tiles”. In: *Discrete Mathematics* 160.1-3 (1996), pp. 245–251.

[6] Weiming Dong, Ning Zhou, and Jean-Claude Paul. “Optimized tile-based texture synthesis”. In: *Proceedings of Graphics Interface 2007*. 2007, pp. 249–256.

[7] Alexei A Efros and William T Freeman. “Image quilting for texture synthesis and transfer”. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, pp. 341–346.

[8] Alexei A Efros and Thomas K Leung. “Texture synthesis by non-parametric sampling”. In: *Proceedings of the seventh IEEE international conference on computer vision*. Vol. 2. IEEE. 1999, pp. 1033–1038.

[9] Anna Frühstück, Ibraheem Alhashim, and Peter Wonka. “Tilegan: synthesis of large-scale non-homogeneous textures”. In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–11.

[10] Leon Gatys, Alexander S Ecker, and Matthias Bethge. “Texture synthesis using convolutional neural networks”. In: *Advances in neural information processing systems* 28 (2015), pp. 262–270.

[11] Fei Hou et al. “Poisson vector graphics (PVG)”. In: *IEEE transactions on visualization and computer graphics* 26.2 (2018), pp. 1361–1371.

[12] ISO. *ISO/IEC 14882:2020 Programming languages — C++*. 2020. URL: `https://www.iso.org/standard/79358.html`.

[13] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*. 2018. URL: `https://www.iso.org/standard/74528.html`.

[14] Emmanuel Jeandel and Michael Rao. “An aperiodic set of 11 Wang tiles”. In: *arXiv preprint arXiv:1506.06492* (2015).

[15] Stefan Jeschke, David Cline, and Peter Wonka. “A GPU Laplacian solver for diffusion curves and Poisson image editing”. In: *ACM SIGGRAPH Asia 2009 papers*. 2009, pp. 1–8.

[16] Jarkko Kari. “A small aperiodic set of Wang tiles”. In: *Discrete Mathematics* 160.1-3 (1996), pp. 259–264.

[17] Khronos Group. *Opengl*. 2020. URL: `https://www.khronos.org/opengl/`.

[18] Vivek Kwatra et al. “Graphcut textures: Image and video synthesis using graph cuts”. In: *Acm transactions on graphics (tog)* 22.3 (2003), pp. 277–286.

[19] Chuan Li and Michael Wand. “Precomputed real-time texture synthesis with markovian generative adversarial networks”. In: *European conference on computer vision*. Springer. 2016, pp. 702–716.

[20] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. *CUDA, release: 10.2.89*. 2020. URL: `https://developer.nvidia.com/cuda-toolkit`.

[21] Alexandrina Orzan et al. “Diffusion curves: a vector representation for smooth-shaded images”. In: *ACM Transactions on Graphics (TOG)* 27.3 (2008), pp. 1–8.

[22] Patrick Pérez, Michel Gangnet, and Andrew Blake. “Poisson image editing”. In: *ACM SIGGRAPH 2003 Papers*. 2003, pp. 313–318.

[23] Raphael M Robinson. “Undecidability and nonperiodicity for tilings of the plane”. In: *Inventiones mathematicae* 12.3 (1971), pp. 177–209.

[24] The QT Company. *QT Developer Guides*. 2019. URL: `https://wiki.qt.io/Developer_Guides`.

[25] Vasilis Toulatzis and Ioannis Fudos. “Deep Tiling: Texture Tile Synthesis Using a Deep Learning Approach”. In: *arXiv preprint arXiv:2103.07992* (2021).

[26] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. “Improved texture networks: Maximizing quality and diversity in feed-forward stylization and texture synthesis”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 6924–6932.

[27] Hao Wang. “Proving theorems by pattern recognition—II”. In: *Bell system technical journal* 40.1 (1961), pp. 1–41.

[28] Li-Yi Wei and Marc Levoy. “Fast texture synthesis using tree-structured vector quantization”. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 479–488.

[29] Philip Willis. “Projective alpha colour”. In: *Computer Graphics Forum*. Vol. 25. 3. Wiley Online Library. 2006, pp. 557–566.