ORIENTATION GUIDED TEXTURE SYNTHESIS USING PATCHMATCH

By

Rosemary L Dutka

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science — Master of Science

2013

ABSTRACT

ORIENTATION GUIDED TEXTURE SYNTHESIS USING PATCHMATCH

By

Rosemary L Dutka

Texture describes the unique structural patterns that we perceive in the world. Various surface geometric details such as animal fur, plant leaves, and carpets can be thought of as texture. In computer graphics, textures stored as images are ubiquitously used to decorate boundary surfaces of objects. There are multiple approaches to acquire realistic and aesthetically pleasing textures. One of the most popular methods is a process known as texture synthesis, in which we produce seamless nonrepetitive textures from a small patch of texture sample.

In this thesis, we present an orientation guided fast texture synthesis based on an image editing tool, PatchMatch, which is included in PhotoShop. Given an example image, our model adopts a hierarchical process to improve retention of structural texture features at multiple scales. We generalize PatchMatch by using orientation to guide the alignment of texture features, indicated by a planar direction field, in the creation of large texture patches. To demonstrate the effectiveness of our approach, we first apply our algorithm in designing new textures with two and four-way symmetry which can be extended to n-way symmetry, and then in enhancing latent fingerprints. Furthermore, our results show empirically that orientation guided PatchMatch has the advantages of providing control over the density of singularities without knowing the exact locations and reducing spurious singularities.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Yiying Tong for his guidance and assistance throughout my work on my master's degree. I appreciate the opportunity to absorb some of his knowledge, skills, and neverending quest for quality.

Thanks to Dr. Xiaoming Liu, Dr. Anil Jain, and Dr. Yiying Tong for teaching some of my favorite and fascinating courses during my studies at Michigan State University.

I would like to specially thank the Computer Science and Engineering 101 course instructors, Kevin Ohl and Monika Sobolewska, for their advice and support. Thank you, Monika, for encouraging me to keep going when I would rather have given up.

Couri Vandam, Lacey Best-Rowden, and Beibei Liu, thank you for your guidance, support, advice, and friendship during my time at Michigan State University and beyond. I would also like to thank my Alma College co-worker, Kimberly Lauffer, for her guidance, support, and friendship.

Thank you Aaron Warsaw for your support, advice, many hours of proof-reading, and listening to my endless dramatic rants.

Finally, I am very grateful to my loving and supporting parents Charlene and Robert for teaching me the importance of knowledge and wisdom.

Thank you,

Rosemary

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

N-RoSy N-Way Rotational Symmetry

DEC Discrete Exterior Calculus

EM Expectation-Maximization

GPU Graphics Processing Unit

IBFV Image Based Flow Visualization

LIC Line Integral Convolution

MRF Markov Random Field

NNF Nearest-Neighbor Field

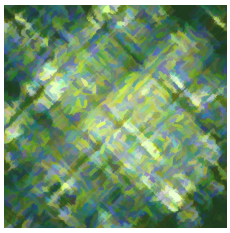
RANSAC Random Sample Consensus

SSD Sum-Of-Square Difference

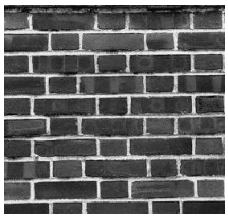
Chapter 1

INTRODUCTION

Texture defines the inherent geometric and photometric properties of objects. Examples are tiger stripes, carpet grooves, leopard spots, and scenic ridges and valleys. A long history exists in the classical scope of art which employs the use of textures to draw attention to and explain the subject matter of the artwork. More recently, this artistic approach has been adapted to create computer rendered images.



(a)



(b)

Figure 1.1: For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this thesis. Images (a) and (b) are examples of texture. (a) is an artist drawing with a green color palette. (b) is a digital photograph of bricks.

Texture (see Figure 1.1) can be generated manually through photo scanning, hand-drawn images, or by an artist using a graphics tool, e.g. Adobe Illustrator. Unfortunately, manually designing natural textures can be time consuming and difficult; therefore, an automated approach which produces a large texture from a small texture patch is desirable. This process is known as example-based texture synthesis, demonstrated in Figure 1.2. In practice, user-specified parameters are used to control various surface properties, e.g. regularity, of the

synthesized output texture.

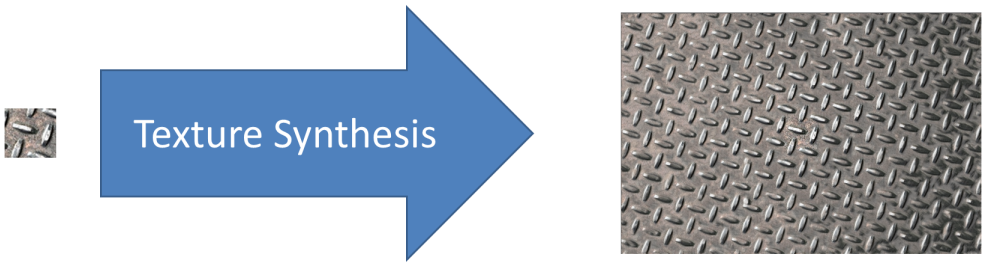


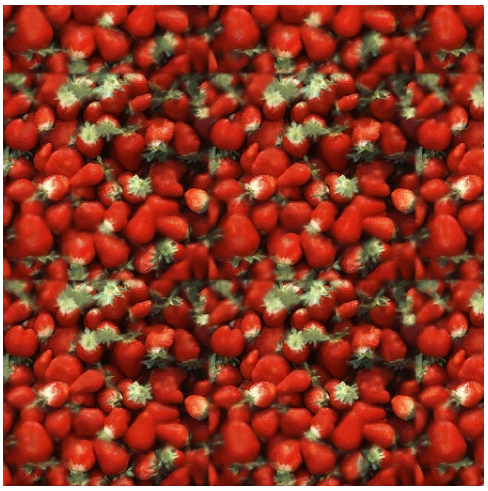
Figure 1.2: An example texture is used as input into a texture synthesis algorithm, which produces a new synthesized texture.

A multitude of applications can benefit from texture synthesis. Such examples are rendering, image and video editing, segmentation, and classification. Hence, many different approaches have been proposed in texture synthesis to meet the various requirements [7, 11, 12, 22, 24, 26, 29, 45]. Yet, all approaches have in common a practical methodology (see Figure 1.2); that is, the synthesis process starts from an example texture and combines key structural elements of the example to produce a new large texture. Every texture synthesis algorithm shares three common goals:

- The structure of a new texture must contain patterns that are similar to the example as shown in Figure 1.2.
- No visible artifacts are present in the new texture. Typical artifacts include incorrect boundaries, seams, and blocks as demonstrated in Figure 1.3b.
- The new texture must not be a periodic tiling of the example texture.

In addition, many texture synthesis algorithms fail to meet key requirements for interactive use. Example approaches, [4, 11, 40, 46], are lacking the efficiency to provide realtime

(a) Example Texture



(b) Bad Synthesized Texture

Figure 1.3: An example of a bad synthesized texture (b). Seams are visible around the boundaries of tiles of example image (a).

feedback. Others are not flexible enough to be extended to a wide variety of image editing protocols, such as hole-filling and image "reshuffling". Image "reshuffling" allows the user to select a portion of the image and move it to a new location within the same image. However, Barnes et al. provide an algorithm, PatchMatch, that is fast and flexible for interactive image editing [2]. In fact, it is known as Content Aware Fill in Adobe Photoshop. Later in this thesis we will elaborate on the texture synthesis algorithm, PatchMatch.

Striving to meet the goals of texture synthesis and interactive editing with the additional flexibility of feature alignment, we will describe a new framework, Orientation Guided PatchMatch for producing 2D textures. We employ a hierarchical method as suggested in [2] to improve the preservation of texture structures at different scales. We depart from the original method by introducing control over directions of the patterns in the texture. Orientation vectors in the x, y-plane guide texture placement within the synthesized texture. As the orientation varies with the location in the output texture, we describe it by a vector field. A vector field can be discretized to a 2D array of arrows in x, y plane (see Figure 1.4).

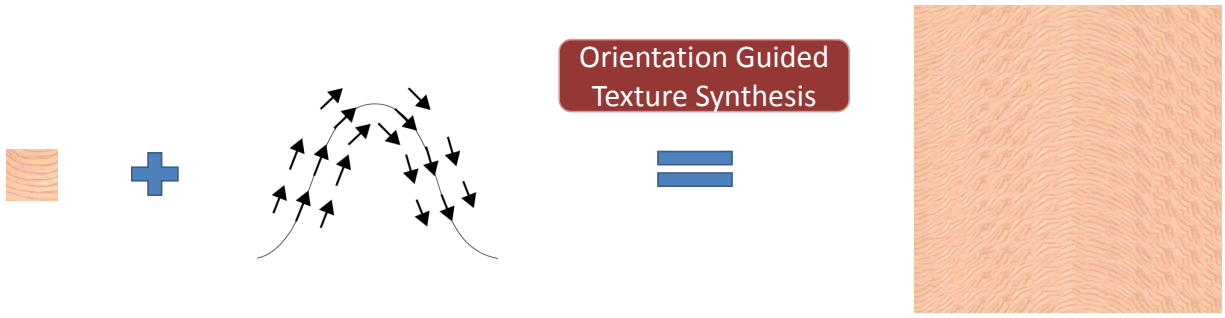


Figure 1.4: An example texture and a vector field together produced a new orientation guided texture through our orientation guided texture synthesis algorithm.

Vector field topology is mostly concerned with the layout of singularities and their relations, and is of ultimate importance in various areas of science and mathematics. Visualizing the topology of vector fields help scientists perceive intuitively the global behaviors and salient features present within a field. One application of Orientation Guided PatchMatch is that it could be adapted to vector fields on curved surfaces. Figure 1.5 demonstrates a texture synthesized on a bunny mesh. Flow texture synthesis, or time varying vector field

(a) (b) (c)

Figure 1.5: Two images of a bunny mesh and an image of a texture atlas. (a) bunny without texture and (b) bunny rendered with the texture. (c) the texture atlas containing charts of textures for patches on the bunny.

synthesis, can be applied to view and reproduce storm cloud formations. One important example is the movement of wind within a tornado. By understanding the properties of

the vector field within a thunderstorm, we could improve forecast predictions and storm tracking, allowing more time for individuals in the path of the storm to seek shelter. Further innovations, such as improving structural building components, can be developed from studying wind vector fields.

Another potential application of texture synthesis is in enhancing latent fingerprints, which are of importance to crime investigators. Latent fingerprints are partial fingerprints found at crime scenes and often contain noisy features within it, while rolled fingerprints are impressions caused by rolling the finger from side to side, or nail to nail. Figure 1.6 shows a latent fingerprint (a) next to its rolled counterpart (b). Minutiae, defined as ridge endings

(a) (b)

Figure 1.6: Two types of impressions of the same finger. (a) Latent fingerprint and (b) a rolled fingerprint.

and bifurcations, are the most important features of fingerprints, commonly used for matching a latent fingerprint with a rolled fingerprint. They can be automatically computed on rolled fingerprints in databases, but often have to be manually found by examiners on latent fingerprints. Orientation fields have proven to be significant in enhancing and reconstructing latent fingerprints for use in identifying fingerprints [14, 15, 19, 48]. By synthesizing the latent fingerprint along the orientation fields, our results may potentially help examiners locate

minutiae automatically in fingerprint matching. Thereby it can lead to a reduction in time and amount of errors. In Chap. 2, we discuss further orientation fields for fingerprinting.

The rest of this thesis is organized as follows: Chap. 2 discusses various texture synthesis and vector field design methods in more detail. Chap. 3 outlines the algorithm PatchMatch and our modifications necessary for orientation-guided synthesis. Chap. 4 details our results from altering PatchMatch to synthesize along vector fields or their generalizations for features with rotational symmetries. Finally, Chap. 5 serves as a conclusion and describes our future work.

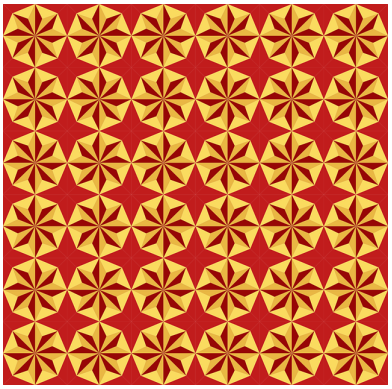
Chapter 2

BACKGROUND

2.1 Texture Synthesis

Texture incorporates a wide range of photographic characteristics creating a unique appearance on the underlying surface. These features can be distinguished in the texture as landscapes, fauna, animal furs, and even paper fibers. A spectrum of textures is defined by Lin et al [27], ranging from stochastic to regular textures. Stochastic textures appear as random noise across the entire image. Such an example is beach sand in Figure 2.1a. Regular

(a)



(b)

Figure 2.1: Two types of textures. (a) Stochastic texture, sand, and (b) regular texture, design parquet lakeside inlay floor designs.

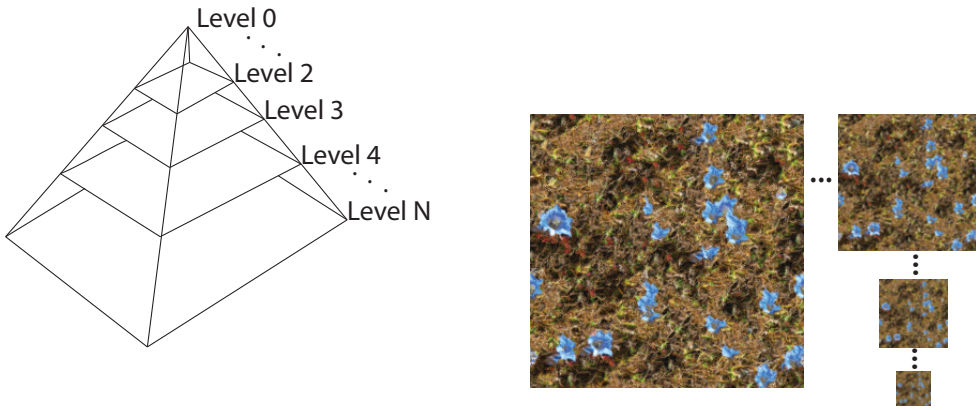
textures have structured patterns. An example of structured texture can be described as floor tiling in a house (see Figure 2.1b). Texture is utilized by multiple disciplines within computer science. Pattern recognition uses texture as features to classify an image [10].

Similarly, within computer vision, texture can assist in understanding the image and locating various objects in the image [36]. In computer graphics, texture mapping provides rich visual details of 3D surfaces at minimal cost, when producing realistic 3D images accelerated by graphics hardware [13, 23, 31, 40, 39, 44].

Many approaches are used in practice to acquire new textures. One of the most common methods is example-based texture synthesis. This process designs larger textural images from the features found in smaller example images [1, 11, 12, 21, 22, 43]. Example images can also effectively be tiled in the output image and seamlessly stitched together to render an image similar to the example-based image [12].

Multiresolution, or hierarchical, methods in texture synthesis often start by constructing texture pyramids, i.e. stacks of images at different resolutions, for analysis of the exemplar texture features. Pyramids are typically constructed by downsampling, filtering or resizing the image in half, until the last image contains one pixel. Figure 2.2 shows the base image and successively smaller sub-images. The top of the pyramid is the coarsest level and the bottom is the finest resolution level. Heeger and Bergen [17] presented one of the earliest work in texture synthesis using hierarchical methods. They use a combination of Laplacian pyramid for edge detection and steerable (oriented transform) pyramid with histograms from each pyramid level to synthesize a similar texture pyramid. Unfortunately, their technique was limited to synthesising stochastic textures with little structure. De Bonet's method [8] overcomes this limitations by restricting the flow of texture features in a top down fashion, similar to Paget and Longstaff [30]. That is, the sampling of texture features is conditioned on the coarser levels in the pyramid. Although their results can synthesize a wider range of textures, tuning the parameters is not as straightforward as Paget and Longstaff's method. Together their pioneering work led to the popularization of multiresolution in texture syn-

thesis [1, 2, 11, 25, 24, 28, 43].



(a) (b)

Figure 2.2: Image pyramid (a) and (b) with base image and a series of successively smaller sub-images, each being half the resolution of the image at the level below.

An effective approach for accelerating texture synthesis is through leveraging image coherence. Ashikhmin [1] is one of the first papers to propose the use of image coherence. The idea is straightforward. When texture features are copied from the example texture to the new texture, the probability that they will be assigned random locations is small. Instead, the features should be grouped together in the new texture. While Ashikhmin's method produces decent results and gains in speed, it is only applicable to natural patterns. Tong et al. [37] improved the original idea by extending it to k-coherence. That is, a k-nearest-neighbor lookup table of the example texture features is used to facilitate the search for a feature with the minimal Euclidean distance from a given feature. K-coherence produces quality images with an improvement in synthesis speed only when k is small (for textures with high frequency features). Barnes et al. [2] inspired by k-coherence, use a similar idea in their propagation search step of PatchMatch, which we discuss in detail later in this thesis.

Constrained texture synthesis has been employed by image editing tools [11, 26, 43]. In general, photographs contain regions with flaws. A flaw can be noise on a photograph,

blurred objects, or an unwanted object in the image. Hole filling is the most common type of constrained texture synthesis [11, 26, 43]. A hole, or region, is missing from a texture and needs to be filled in using example textures from the surrounding areas (see Figure 2.3). Wei and Levoy [43] state that this method must meet two requirements: the new region must appear similar to the surrounding area and no visual boundaries should occur between the old and new regions. More recently, Barnes et al. [2] discuss additional constraints, such as image retargeting and reshuffling. Image retargeting, also known as seam carving, allows user defined seams both vertically and horizontally to be preserved when reducing image size. Image reshuffling moves a user-specified region from one location to another location in the image.

(a) (b)

Figure 2.3: Constrained texture synthesis using PatchMatch. Image (a) shows the object, a tower, that we want to remove from the image. Image (b) shows the result of deleting the image and filling in the background.

Image editing tools have been developed that allow for convenience of end-users [2, 28]. The primary focus of research in editing tools has been on three objectives; user friendliness, efficiency, and quality of the final synthesized image [2, 28]. These texture synthesis

algorithms have been developed to complete the described editing process in less than half a second [2]. From the end-users' perspective, the tool's runtime performance is fast enough to be interactive. The quality of the synthesis image must be acceptable to the user. It is required that these images be nonrepetitive, aesthetically pleasing, and remain close to photo-realistic.

Various approaches have been proposed in the last two decades for texture synthesis. A short summary of the mainstream works is provided in the following paragraphs.

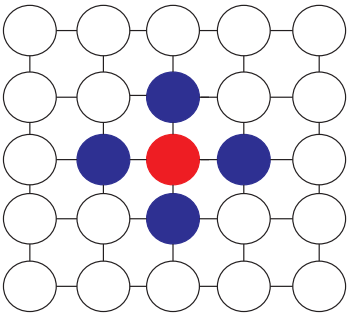


Figure 2.4: A simple example of Markov random fields. The color of each pixel is considered a random variable conditionally dependent on its neighbors, and the combined probability satisfies a certain Markov property. For instance, with local Markov property, the colors of the neighboring pixels (blue) determines the probability distribution of the color of the center pixel (red).

Markov Random Fields Textures are modeled using the Markov Random Fields (MRF) theory (Figure 2.4) and synthesis is done through computing higher order statistics of the example-based texture [7, 30]. The goal, based on the realization of a local and stationary random process, is to produce (per pixel) a spatial neighborhood similar to at least one neighborhood in the given example. As a result of the implementation using Markov Random Fields, a near-neighbor dependence is achieved and therefore it produces high-quality results given the approximation for various textures [7, 30]. A drawback of this approach is

that synthesis is computationally costly and thus these algorithms are not appropriate for interactive image editing tools. Many of the following approaches have their roots in MRFs, such as Texture Synthesis By Non-parametric Sampling [11], Graphcut Textures [22], Parallel Controllable Texture Synthesis [24], as well as many others [26, 37, 40, 44], but use various tools to improve efficiency while maintaining the quality of the result.

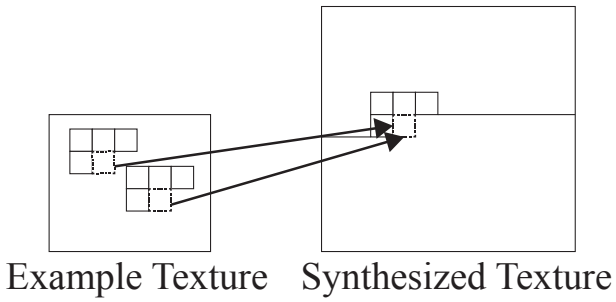


Figure 2.5: Demonstrating pixel-based synthesis process in [43]. The value of a pixel p is determined by comparing its neighbors $N(p)$ in the L-shaped region with all neighborhoods of the same type in the example texture.

Pixel-based Synthesis Pixel-based texture synthesis algorithms sample pixels based on a similarity metric for neighborhoods, typically sum-of-square difference (SSD). Instead of using a Markov Random Field built from the exemplar texture explicitly, this method refines the synthesized texture by picking the color of the center of the neighborhood in the exemplar most similar to the neighborhood of a pixel being processed. This approach has been shown to be simple to implement and produce decent results [11, 24, 43], but, lack in efficiency. Generally, these algorithms synthesize a texture in scan-line order by sequentially processing each pixel of the output texture based on its current neighborhood, via the implicit Markov Random Field (see Figure 2.5). Thus, after a few iterations, neighborhoods contain similar structure as the example image [11, 24, 43]. Some methods have employed a multi-resolution scheme, i.e. building a texture pyramid to keep structures at all scales

consistent between the exemplar and the output [8, 17, 24, 43]. Wei and Levoy [43] proposed accelerating this method via tree-structured vector quantization for quick nearest neighbor searching. Turk [40] demonstrates how pixel-based synthesis can be extended to curved surfaces. Additionally, he supplements the process with a directional field design tool on the mesh to help guide the surface texture feature directions. Until Lefebvre and Hoppe proposed their parallel approach [24], these algorithms had been insufficient for iterative image editing tools. Lefebvre and Hoppe's method generates high quality results by using multiscale Gaussian image stacks and parallel texture synthesis running on Graphics Processing Unit (GPU). They further enhance their results by transforming the exemplar texture into a novel appearance space, to include nonlocal information, and perform dimensionality reduction to provide efficient neighborhood comparisons [25]. However, their method cannot be straightforwardly modified for constrained texture synthesis.

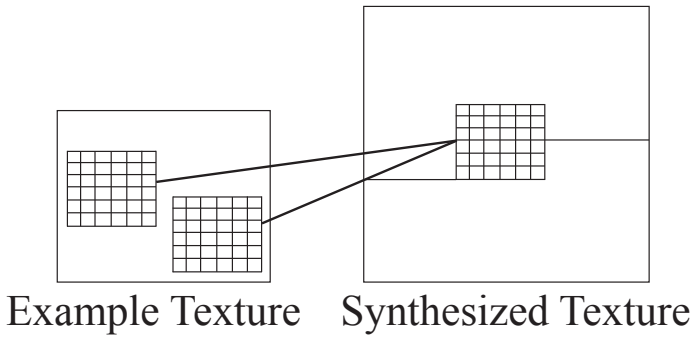


Figure 2.6: Demonstrating patch-based synthesis process. Candidate patches in the example texture are compared against the patch in the new texture.

Patch-based Synthesis A variant to pixel-based synthesis is the patch-based synthesis, which processes the output by blocks, or patches, instead of the pixel-by-pixel procedure (see Figure 2.6) [2, 3, 12, 21, 22, 26]. New textures are designed through compiling various sub-patches from the exemplar at differing displacements, or offsets, that are optimized with

a similarity metric (typically, SSD) [2, 3, 26]. Kwatra et al. [21] determine a unit of pixel values by optimizing a quadratic energy function in an Expectation-Maximization (EM) approach. EM is an iterative method for finding the maximum likelihood estimate. Boundary conditions are handled in a number of ways: determining a minimum cut between two neighboring sides in the patch place [22], cutting and joining patch edges along a boundary where the difference in pixel values is minimal [12], blending overlapping patches [26], or by a patch-voting EM-like algorithm which averages the neighbor information thereby smoothing the boundary [2]. These algorithms have shown to produce quality images and synthesize textures much faster than the pixel-based methods [2, 26]. The patch-based methods have also been implemented with multi-resolution approaches to improve feature preservation in the final synthesized image [2]. A recent approach proposed by Kim et al [20] is symmetry guided texture synthesis, in which approximate symmetries are obtained from an additional exemplar texture and are then transferred to a new texture, producing results that combine the texture pattern of one exemplar and the symmetry of another. One limitation of this approach is that the texture perturbation model (patch-based texture synthesis and as-rigid-as-possible deformation) may not be efficient enough to synthesize a large texture interactively.

Reaction-diffusion Systems Reaction-diffusion equations are a set of partial differential equations describing the process in which chemical agents diffuse and react with each other. It has been suggested to use these equations to describe how patterns form on animal skin or fur by Alan Turing [38]. The "chemical agents" are called morphogens [39, 45]. A simple outline of this system demonstrates how two different morphogens expand at different rates and their reactions with each other that leads to the breakdown or building of the

morphogens [39, 45]. The system attempts to achieve a state of equilibrium between the two morphogens. These models have been shown to create new textures that are structurally similar to mammal fur and fish skin [39, 45]. This approach creates very high quality textures with natural features but can be inefficient and cannot produce results with artificial texture (see Figure 2.7). This technique has been recently used to enhance fingerprints [14].

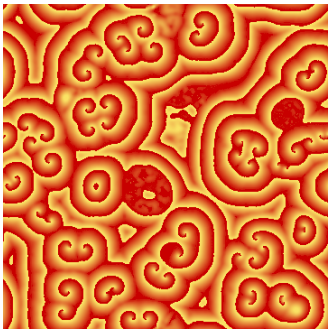


Figure 2.7: Synthesized texture generated from an interactive reaction diffusion application by Weckesser [42]. The image shows the morphogens as they diffuse and react with one another.

The following table presents a comparison of various texture synthesis algorithms.

Table 2.1: Comparison of Texture Synthesis Algorithms

Authors	**Processing Time (seconds)**	**Parameters**	**Input Size (pixels)**	**Output Size (pixels)**
Ashikhmin [1]	1.3 for basic; 20 to 30 for user control	5x5 neighborhood	200x200	500x500
Barnes et al. [2]	0.68	5 iterations, 7x7 patch size	0.1 megapixels	0.1 megapixels

Table 2.1 (cont'd)

Cabral and Leedom [4]	3,000 cells processed per second	$L = 10$ pixels (length of local streamline)	Not reported	256x256
De Bonet [8]	Not reported	Thresholds are of the form: $T_i^j = \alpha/i^\beta$, where $\alpha \in [0, 0.4]; \beta \in \{0, 1\}$	16x16	64x64
Efros and Freeman [12]	15 (s) up to several minutes	Block size not reported	Not reported	Not reported
Efros and Leung [11]	Not reported	Not reported	Not reported	Not reported
Eisenacher et al. [13]	about one minute	64, 16, and 4 size samples; $k = 3$ candidates	64x64 to 4096x4096	1024x1024
Heeger and Bergen [17]	Not reported	Not reported	Not reported	Not reported
Kwatra et al. [22]	5 (s) to 5 minutes	Not reported	Not reported	up to 1280x1024

Table 2.1 (cont'd)

Kwatra et al. [21]	2.5 up to 25; multi-level 1-3 minutes up to 7-10 minutes; time per frame: 20-60	32x32 to 8x8 neighborhood sizes; $k = 4$; 3-5 iterations	128x128, 256x256	64x64 up to 256x256
Lai et. al. [23]	1.6	5 singularities	Not reported	20,000 faces for a model
Lefebvre and Hoppe [24]	.0138 for synthesis; preprocessing time is one minute up to 12 minutes for exemplar	similarity sets of size $k = 2$, and $c = 2$ correction passes, each with $s^2 = 4$ subpasses	64x64	128x128
Lefebvre and Hoppe [25]	48.3 frames/sec 2D isometric mode	$c = 2$ correction passes, each with $s^2 = 4$ subpasses	64x64	256x256
Liang et. al. [26]	0.698	Not reported	128x128	200x200

Table 2.1 (cont'd)

Paget and Longstaff [30]	Not reported	Best results with high order neighborhood of N^{18}	128x128	256x256
Palacios and Zhang [31]	N=2 0.12, N=6 0.46 both with 12k mesh	Not reported	Not reported	512x512
Sanderson et al. [35]	less than a minute on GPU	15-25k iterations; cell size of 1.0 and step size=0.5	Not reported	Not reported
Turk [40]	Depending on Model and Mesh: from a few minutes to over an hour	3, 4 Levels Used in Gaussian Pyramid	64x64 to 256x256	Mesh with 256,000 vertices
Turk [39]	less than a minute to two minutes	Not reported	64x64 and 32x32	mesh with 64,000 points

Table 2.1 (cont'd)

Wei and Levoy [43]	Exhaustive Searching: 360; TSVQ Acceleration: 22 for Training and 7.5 for Synthesis	Not reported	128x128	200x200
Wu and Yu [46]	less than two minutes	Weight set to 0.5; patch size between 32x32 and 64x64	128x128	128x128 and 256x256
Zhang et al. [49]	Not reported	Not reported	Not reported	Not reported

2.2 Alignment Fields

In most texture patterns, there are salient feature directions. In applications such as synthesizing the patterns on a zebra starting from stripes, alignment to directions on the surfaces with controlled stripe width is desired. A continuous vector field can be used to provide both the directions and sizing. In patterns with rotational symmetry, i.e., those that remain invariant under certain rotations, alignment fields can be more flexible than continuous vector fields. These extensions to continuous vector fields are called N-Way Rotational Symmetry (N-RoSy) fields. For example, with square tile patterns, aligning them to vector fields with certain discontinuity (vectors are rotated by 90° for adjacent locations) can still create continuous patterns. In this section, we discuss related work first on vector fields and then on N-RoSy fields.

2.2.1 Vector Fields

Vector fields are ubiquitous in various mathematical and scientific areas. We focus on 2D vector fields, for example a vector-valued function $\mathbf{u}$ defined on a subset $S \subset \mathbb{R}^2$ of the plane,

$$\begin{aligned} \mathbf{u} : S &\rightarrow \mathbb{R}^2, \\ (x_1, x_2) &\mapsto (u_1(x_1, x_2), u_2(x_1, x_2)) . \end{aligned}$$

Here and throughout the thesis, we use bold font letters to denote vector values, regular lower case letters to denote scalar values, and subscript to index the components. In texture synthesis, the vector field in the output region denotes the salient feature direction. Assuming that one of the feature directions in the exemplar texture is along one of the coordinates,

a rotation that aligns the coordinate axis to the value of the vector field at a certain location should be applied when the texture pattern is copied from the input to the output. Furthermore, when the sizing of the pattern also needs to be modulated, the magnitude of the vector field can serve as a scaling factor.

In different applications, vector fields assume different representations. Vector fields in the plane are often discretized as 2D arrays of 2D vectors stored on a regular grid within the domain. It can also be stored as piecewise constant field on a triangle mesh (triangulated region). Coordinate independent representation can be obtained for triangle meshes by taking the line integral of the continuous vector field along each edge, and using piecewise linear interpolation basis functions with these scalar values to reconstruct the continuous vector field [9]. When only low frequencies are important, the number of degrees of freedom to represent a vector field can be greatly reduced. For instance, running a Fourier transformation or wavelet transformation for each component of the vector field treated as a separate scalar field and keeping only the low frequencies. Alternatively, singularities (locations where $\mathbf{u}=0$) can be used as parameters to determine the overall structure of the vector field. Throughout this thesis, we only use the non-reduced representations, since we wish to maintain the detailed alignment directions.

Various vector field visualization methods have been developed, since it is not easy to identify the global structure of a vector field with the traditional hedgehog plots, i.e. collections of arrows. In fact, some propose adapting existing texture synthesis algorithms to display the fields. For example, Sanderson et al. [35] proposed a reaction diffusion model to incorporate magnitude and direction information of any given vector field. However, it is slow to converge and is inefficient for interactive use. Cabral and Leedom [4] propose a texture synthesis method to visualize fluid movement called Line Integral Convolution (LIC),

one of the most commonly used methods to show the streamlines of vector field. Starting with a random noise texture, a convolution is applied to the pixel values along streamlines. Thus, colors become coherent along streamlines, while remaining distinct along the directions across the streamlines. Mathematically, this is represented for an output pixel as

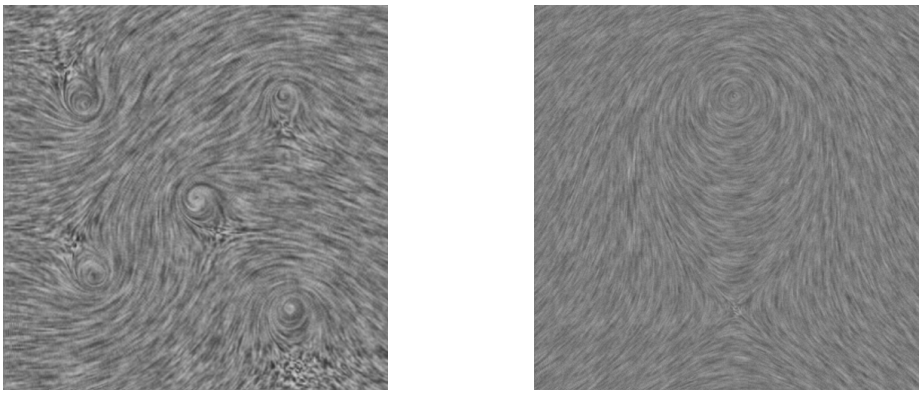
$$\bar{F}(\mathbf{p}_0) = \frac{\sum_{i=0}^{l} F(\lfloor \mathbf{p}_i \rfloor) h_i + \sum_{i=0}^{l'} F(\lfloor \mathbf{p}'_i \rfloor) h'_i}{\sum_{i=0}^{l} h_i + \sum_{i=0}^{l'} h'_i},$$

where $F(\mathbf{p})$ is the input pixel at position $\mathbf{p}$, $\lfloor \mathbf{p} \rfloor = (\lfloor p_1 \rfloor, \lfloor p_2 \rfloor)$, l (l') is the number of points when tracing forward (backward, resp.) along the streamline, and the weight h_i is computed using any chosen convolution kernel $k(\cdot)$. The streamline can be constructed by moving along the vector field $\mathbf{u}$,

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \mathbf{u}(\mathbf{p}_i)s,$$

where s is the step size. Unfortunately, the original LIC implementation is inefficient since 20-50 pixel points on the streamline are necessary to guarantee a high quality output. Wijk [41] proposed Image Based Flow Visualization (IBFV), a graphics hardware speed-up for visualizing vector field. Figure 2.8 shows examples of Wijk's implementation. It distorts the entire image repeatedly by moving pixels of the current image along the vector field, and blends the images to create the final LIC image. Compared to the original LIC, it is highly efficient with common graphics hardware, and can produce smooth animations mimicking a frame-by-frame capture of advective transport of decaying dye.

Zhang et al. [50] provide a framework for designing vector fields on curved surfaces with intuitive tools. In the surface case, a vector field $\mathbf{u}$ is defined as a mapping from a point $\mathbf{p}$ on a 2-manifold (locally $\mathbb{R}^2$-like) surface M to a tangent vector $\mathbf{u}(\mathbf{p})$ at $\mathbf{p}$. The streamlines on the surface can be defined as solutions to the ordinary differential equation $\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t} = \mathbf{u}(\mathbf{p})$.



(a) (b)

Figure 2.8: Vector fields designed from singularities, produced using Wijk's IBFV demo. (a) shows multiple vortices and (b) has a vortex and a saddle point.

The streamline passing through any point $\mathbf{p}_0 \in M$ have the property $\mathbf{p}(0) = \mathbf{p}_0$ when a solution $\mathbf{p} : \mathbb{R} \to M$ exists with that initial condition. A singularity $\mathbf{p}_s \in M$ is defined to be a point where $\mathbf{u}(\mathbf{p}_s) = 0$. These points can be further classified by the local linearization of the vector field. Given any local coordinate system, let J be the Jacobian of $\mathbf{u}$,

$$J = \frac{\partial \mathbf{u}}{\partial \mathbf{p}} = \begin{pmatrix} \frac{\partial u_1}{\partial p_1} & \frac{\partial u_1}{\partial p_2} \\ \frac{\partial u_2}{\partial p_1} & \frac{\partial u_2}{\partial p_2} \end{pmatrix}.$$

If the real parts of the two eigenvalues of $J(\mathbf{p}_s)$ are positive, $\mathbf{p}_s$ contains a source; if they are both negative, it contains a sink; if one is negative and the other positive, it contains a saddle point. The imaginary part contains information about whether there is a counterclockwise or clockwise vortex at $\mathbf{p}_s$. Other important definitions include separatrices and periodic orbits. Separatrix is a streamline where one of its ends (the limit as each point goes to negative or positive infinity of $p(t)$) forms a saddle [50]. Periodic orbits are streamlines that form closed loops. Vector field topology is then represented by the sets of singularities, separatrices, and periodic orbits that separate the domain. This setup on curved surfaces allows them to

use geodesic polar maps (one method to flatten the surface) and parallel transport in order to design radial basis functions for user controls of the vector field. Although we focus on planar texture synthesis, the notions are applicable to our cases.

One common goal of any vector field design tool is to provide intuitive control of the field and allow for a smooth boundary design. Turk [40] implemented a vector field design tool using a multi-resolution pull/push interpolation method from various user-specified directions. Chen et al. [6] extends static vector fields into time varying cases to show dynamic effects and study their design. A tangent vector field is described in [16], which produces decent results that preserve the properties of the continuous vector fields at interactive rates. The input vector fields in our tests are designed using a modified version of their tool. Implementation details are discussed in Chapter 3. This design tool uses the vector calculus in the coordinate independent representation from Discrete Exterior Calculus (DEC), where the vector field is represented as a discrete 1-form, i.e., the assignment of one scalar number per edge for a triangle mesh. It also allows differentiation and integral to be performed through linear algebra. The user interface allows the user to interactively design the vector field through placement of singularities, such as sources and vortices. A vortex can be created by forcing the curl of the vector field to be non-zero at the chosen location. The curl operator measures the local rotation of the vector field around a location. Mathematically, the curl operator $\nabla\times$ is defined as

$$\nabla\times \mathbf{u}(\mathbf{p}) = \begin{pmatrix} \frac{\partial}{\partial p_1} \\ \\ \frac{\partial}{\partial p_2} \end{pmatrix} \times \begin{pmatrix} u_1 \\ \\ u_2 \end{pmatrix} = \left(\frac{\partial u_2}{\partial p_1} - \frac{\partial u_1}{\partial p_2}\right).$$

Likewise, a source can be created by forcing a nonzero divergence. The divergence operator measures expansion of area when following the flow of the vector field around a user-specified

location, and is mathematically defined as

$$\nabla \cdot \mathbf{u}(\mathbf{p}) = \begin{pmatrix} \frac{\partial}{\partial p_1} \\ \frac{\partial}{\partial p_2} \end{pmatrix} \cdot \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \left(\frac{\partial u_1}{\partial p_1} + \frac{\partial u_2}{\partial p_2} \right).$$

Additionally, the user may sketch strokes, to indicate the desired flow directions.

2.2.2 Rotational Symmetry Fields

Real-world textures often contain N-way rotational symmetries, that is, the pattern would coincide with itself after rotating by an angle of $\frac{2\pi}{\mathrm{N}}$. These symmetries allow patterns to be aligned not only with smooth vector field, but direction fields that are not smooth in the traditional sense. More precisely, the target feature alignment directions at nearby locations can differ by a rotation angle of multiples of $\frac{2\pi}{\mathrm{N}}$ without creating discontinuous patterns. Mathematically, such feature directions can be defined as equivalence sets of vectors formed by rotating vectors by multiples of $\frac{2\pi}{\mathrm{N}}$, where N is determined by the desired symmetry [23, 32]. N-way rotational symmetry (N-RoSy) field is smooth in the sense that any vector from the equivalence set can always find one similar corresponding vector in the equivalence class at a nearby point. Figure 2.9 demonstrates the properties of RoSy fields. They can be stored as a (potentially discontinuous) vector field by choosing one representative from the equivalence class at each point, or alternatively as a special type of tensor fields in certain cases [23, 32]. Ray et al. [33] offers an equivalent definition for singularities of N-RoSy fields and introduced the important concept of turning numbers.

Being able to interact with N-RoSy fields as well as vector fields, is most important to various artistic and mathematical applications. In these programs, the end-user should be

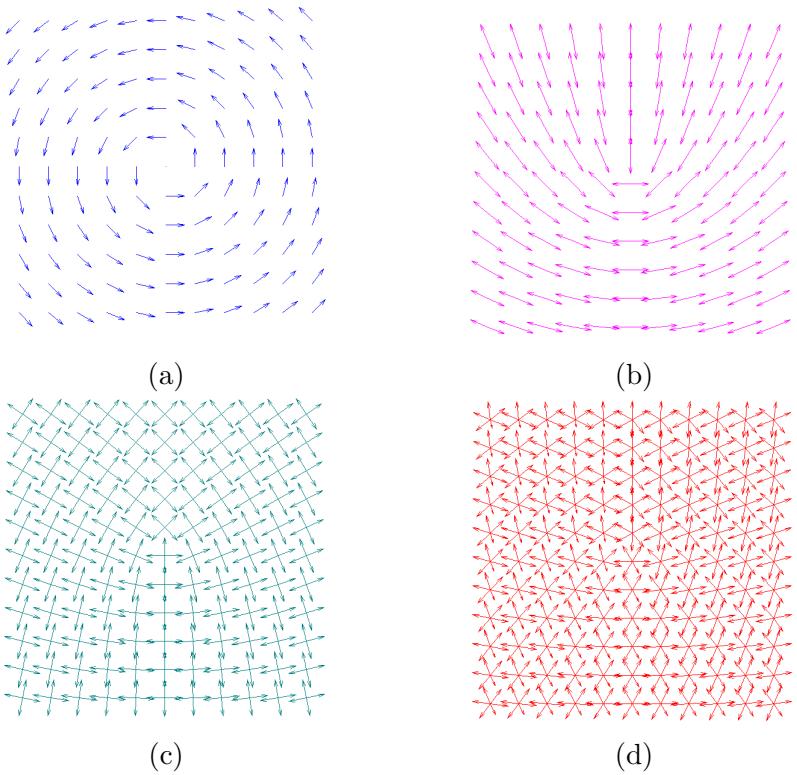


Figure 2.9: Image (a) demonstrates a vector field with one singularity. Images (b), (c), and (d) show (a)'s rotational symmetry fields for (b) 2-way, (c) 4-way, and (d) 6-way.

allowed to exert control over the field's topology, in particular, placement of singularities. For graphics purposes, Zhang et al. demonstrated this idea of using an interactive program for 2-RoSy [50]. Furthering this development, Palacios and Zhang extended the interactive tool to handle N-RoSy fields for $N \geq 3$ on surfaces [32]. They also adapted the LIC technique by decomposing an N-RoSy field into a set of N vector fields [31], however they must address the loss of contrast due to blending images. While these ideas are applicable for geometry processing, the approaches are also of importance to visualization, texture synthesis, rendering, and parameterization. Not all texture synthesis method can directly take the representative discontinuous vector field of an N-RoSy field as the target alignment direction without introducing artifacts [28], but N-RoSy fields are necessary to create the natural types of singularities, including cores and deltas in synthesizing fingerprints.

In particular, orientation fields, or 2-RoSy, contain inconsistent (nearly opposite) vectors in some neighborhoods. That is to say, the vectors may go in opposite directions from nearby sample points, causing potential problems in constructing consistent neighborhoods. One such example is the pattern of fingerprints. Fingerprint ridges and valleys do not have a specific forward or backward orientation [15, 47]. The singularities in such 2-RoSy fields include cores and deltas, which are more flexible than sources, sinks, vortices, and saddle points for smooth vector fields. Core, Figure 2.10c, is a U-shaped ridge pattern and delta is a Y-shaped ridge pattern [15, 47], shown towards the bottom right-hand corner in Figure 2.10a.

(a) (b) (c)

Figure 2.10: Examples for three main categories of fingerprints. (a) is a whorl pattern. (b) is an arch pattern. (c) is a loop pattern. Cores have been marked by a red circle. Deltas are marked by a green triangle.

Much research has been done in estimating ridge orientation from minutiae. Ross et al. [34] proposed an approach in which fingerprint orientation is estimated from the minutiae found in a template. In biometrics, a template describes the stored file in a fingerprint scanning product and is smaller than a fingerprint image to allow for faster processing time. They observe that the orientations of minutiae indicate the local ridge direction because fingerprints have a smoothly varying texture. A group of nearby minutiae can then be used to interpolate the ridge structure. They use three minutiae locations known as a triplet. Once good quality triplets are determined, the triplets are used in estimating the local orientation in triangular fingerprint areas. Finally, an averaging step is performed to obtain a smooth

result. This provides a good solution for estimating orientation which is then applied to an LIC algorithm to generate a fingerprint image. Unfortunately, the algorithm can generate spurious minutiae and reconstruct only a partial fingerprint. Cappelli et al. [5] adopt a model-based approach to estimate the orientation field from the minutiae directions. They employ Gabor filtering on the minutiae and grow the regions outward until they have a full fingerprint image. Again this algorithm generates spurious minutiae.

Jain et al. [18] create an orientation field from the fingerprint image by dividing it into blocks, 16x16 pixels, and assigning a single orientation to each block. They also introduce a quality measure for minutiae and blocks that can be either labeled as reliable, unreliable, or background. Reliable means that all minutiae and clear ridge structure are present in the block, whereas unreliable ones have poor quality; that is, unclear and/or partial ridge structure. Yoon et al. [47, 48] extend the block method to two different approaches: (1) obtain a coarse orientation field from a skeleton image and then fit an orientation model to it, and (2) a short-time Fourier transform per block to obtain orientation elements that is fed into a Random Sample Consensus (RANSAC) algorithm to build and evaluate the orientation field. RANSAC is an iterative approach to estimating parameters of a mathematical model from a set of data containing outliers. Both extended block methods use filtering to produce the texture. Feng and Jain [14] reconstruct a fingerprint based on a frequency modulated signal model. Minutiae are fed into the spiral phase and estimated orientation, both of which are used to calculate the gradient of continuous phase that produces the continuous phase. Spiral and continuous phase, are then used in the final reconstruction of the fingerprint. Their work produces good results with fewer spurious minutiae, but leaves room for improvement in making the fingerprints appear more realistic. Feng et al. [15] also research how latent fingerprint orientation fields can be improved by implementing a context-based correction

through a dictionary lookup, which is constructed from reference fingerprints. Their results show improvement in orientation field estimation over manual approaches.

We have discussed many approaches for texture synthesis and visualizing vector fields. One of the main problems in visualizing vector fields is that they produce spurious singularities. In the next chapter, we review our modifications to PatchMatch that allow us to have control over the placement of singularities.

Chapter 3

ORIENTATION GUIDED PATCHMATCH

In this section, we first discuss the original PatchMatch algorithm and orientation field design, before presenting our modifications to make PatchMatch applicable in cases where features are to be aligned with the chosen orientation field.

3.1 Original PatchMatch

PatchMatch is a patch-based texture synthesis method based on correlations between nearby locations, in which patches from the source are iteratively copied to a target image to refine the result by selecting an input patch with a minimal distance to the output patch to be replaced. The nearest-neighbor field (NNF) is defined as a mapping $\mathbf{f} : [0,1] \times [0,1] \rightarrow [0,1] \times [0,1]$ from the target region to the location of a patch center, assuming both source and target images are normalized in size. When it converges the patch, centered at $\mathbf{p}$ in the target domain, should be similar to the patch centered at the location $\mathbf{f}(\mathbf{p})$ in the source, for a given distance function measuring the difference between texture patches. For our implementation, we also use sum-of-squared differences (SSD). We discretize the mapping $\mathbf{f}$ as a 2D array with dimensions set according to the target image resolution. Patch size

can be specified by the user. Typical size is 7x7 pixels [2]. The three components for the algorithm are initialization, propagation, and random search, which we detail below.

The algorithm first initializes (Figure 3.1a) the NNF by either choosing prior information, or applying random offsets, both of which can be stored in the array. Random offsets are independently uniform samples taken from the source image. Prior information is used in a multi-resolution approach, where the output is synthesized in a coarse-to-fine manner. A multi-scale pyramid is constructed from downsampling of the original source, where each level of the target pyramid is constructed from synthesizing the source image in the corresponding level in the source pyramid. Prior information for level l is taken from the synthesized result at level-$l-1$ in the output pyramid. The process is repeated until the target image at the finest resolution is produced [2].

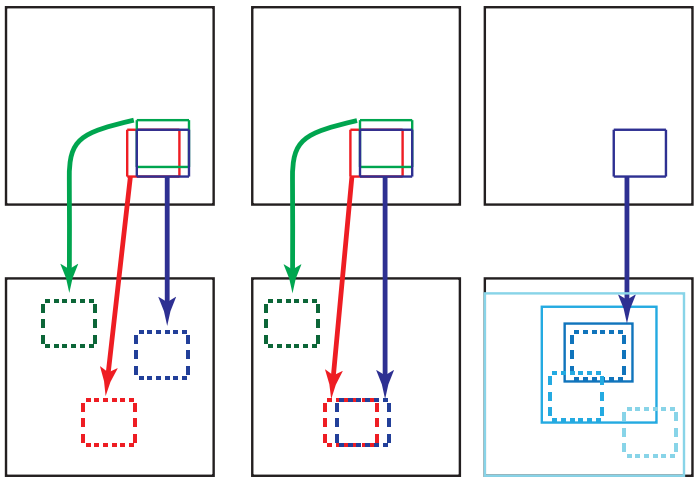


Figure 3.1: Algorithm steps: first image (a) initializes the NNF to have random assignments; second image (b) propagates matches by checking neighbors of the blue patch to see if they will improve the patch; third image (c) shows the current patch searching randomly, within a window, for improvements. Adapted from [2].

The next step, iteration, has two components: propagation and random search. Each component is interleaved at the patch level: let P_i and S_i be the propagation and random search at patch location i, then the order of operations is $P_1, S_1, P_2, S_2, P_3, S_3, ..., P_n, S_n$ where n is the total number of patches. Propagation (Figure 3.1b) improves the patch placement by propagating neighbors one pixel to right and down on odd iterations, and one pixel to left and up on even iterations. Propagation is defined mathematically as replacing $\mathbf{f}(x, y)$ by $arg\,min\{D(\mathbf{f}(x, y)), D(\mathbf{f}(x-1, y) + (1, 0)), D(\mathbf{f}(x, y-1) + (0, 1))\}$ for a patch at (x, y), where D is the distance measurement from the current patch to the candidate patch [2]. The process improves the coherence of the mapping, since if (x, y) is located in a region R and already has a good mapping, then additional correct mappings will be filled below R and right of (x, y). On odd iterations, we reverse the examining offsets such that $\mathbf{f}(x+1, y) + (-1, 0)$ and $\mathbf{f}(x, y+1) + (0, -1)$ are used as candidates.

Random search (Figure 3.1c) improves a patch by testing it against selected patches from within a given search radius that must be within the bounds of the source image. The search radius decreases in distance from the patch until the radius is below 1 pixel. Mathematically, we describe patch improvement candidate location at (x, y) as $\mathbf{u}_i = \mathbf{f}(x, y) + w\alpha^i\mathbf{R}_i$ where $\mathbf{R}_i$ is a random vector with uniform probability distribution in $[-1, 1] \times [-1, 1]$, w is the maximum search radius, α is the fixed ratio between search window sizes (we used $1/2$), and $i = 0, 1, 2, \ldots$ are the examined patches until the search radius $w\alpha^i < 1$ pixel [2]. This step helps avoid being trapped in a local minima.

Efficiency is improved by early termination in the propagation and random search phases, when $\mathbf{f}(\mathbf{v})$ is compared with a candidate mapped location $\mathbf{u}$, the partial sum for $D(\mathbf{u})$ is already larger than the current distance $D(\mathbf{f}(v))$ [2]. Each iteration is performed for a fixed number of times. On average, we found the target image converging in 5 iterations for image

retargeting applications as mentioned in [2].

3.2 Orientation Field Design with Boundary

We discuss the vector/orientation field design methods described in [16] and [28] since the alignment fields employed in our extended PatchMatch are constructed using these methods. For the terminology we used for the continuous alignment field setup, we refer the reader to Chapter 2.2.

Fisher et al. [16] describe a setup using tangent vector fields that are formulated as a weighted least squares problem for efficiency with user-placed design constraints. Such constraints are vortices and sources/sinks of the vector field, and direct constraints on the vector directions, at arbitrary chosen locations. Often, direct constraints are drawn in batches by user sketch strokes, though can be any constant vectors at selected locations. In this approach, user-specified vortices and sources/sinks are singularities with specified non-zero curl and divergence values respectively at given locations. Figure 3.2 exhibits the curl and divergence of a vector field.

The above mentioned weighted least squares problem involves a direct implementation using Discrete Exterior Calculus (DEC) when the appropriate fields are designed as discrete differential forms. A k-differential form is informally just an integrand, a field that can be integrated on a k-D domain. For instance, in 2D plane, a scalar field (function) $s(x, y)$ can be considered a 0-form (integrated at a point), $f(x, y)dx+g(x, y)dy$ in $\int f(x, y)dx+g(x, y)dy$ is considered a 1-form, and $h(x, y)dxdy$ in $\int\int h(x, y)dxdy$ is a 2-form [9]. Thus, in the plane, we can use a 1-form to represent a vector field $\mathbf{u}$ as an integrand $u_1(x, y)dx + u_2(x, y)dy$ that can be integrated along any curve segment. In a triangle mesh, the discrete version of

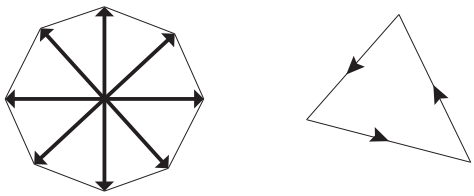
a k-form is stored as an array of its integrals on k-D regions in the mesh, i.e. 0-form as its evaluation per vertex, 1-form as its integral per edge, and 2-form as its integral per triangle.

With such a representation, vector field calculus can be turned into DEC through the generalized Stokes' theorem, and the usual differential operators (gradient, curl, and divergence) turned into sparse matrices applied to the discrete k-forms arrays treated as vectors. For example, we can consider a smooth function $s(x, y)$. The gradient of s represented as a 1-form is $ds = \frac{\partial s}{\partial x}dx + \frac{\partial s}{\partial y}dy$. In particular

$$\int_C ds = s(\mathbf{b}) - s(\mathbf{a}),$$

where C is a curve segment connecting point $\mathbf{a}$ with point $\mathbf{b}$. The above equation is called the Stokes' theorem, and when restricted to a line segment along x-axis, the Newton-Leibnitz formula or the first fundamental theorem of calculus. In the discrete representation, the gradient ds is stored on edges, and s is stored on vertices. Thus, it is a straightforward subtraction between the values on both ends of the edge to evaluate the gradient operator. Expressed with linear algebra, this turns the gradient operator into the edge-vertex incidence matrix with proper signs. Curl and divergence can be likewise constructed as sparse matrices. Thus, the resulting linear system for the vector field design problem turns out to be a discrete Poisson equation mixed with terms from flexible constraints.

In the following, we give the details on how the linear system is constructed, and present a natural boundary condition, which produces the smoothest vector field when the design is performed on a region with boundary.



(a) (b)

Figure 3.2: Image (a) A weighted average of the vector field integrated along the arrows gives the outgoing flux from the neighborhood region of the center vertex, which is what the divergence of a vector field measures. Image (b) The sum of integral of the vector field along the three arrows is the circulation of the vector field, which is precisely the integral of the curl of the vector field in the triangle.

3.2.1 Vector Field Setup

Fisher et al. [16] define their equations on a 2-manifold (2D surface) with boundary. It is represented by a triangle mesh M, with vertex set $v_i \in V$, edge set $e_k \in E$, and triangle set $t_j \in T$. We then store the vector field as a discrete 1-form, that is, one scalar value per oriented edge. The line integral of a vector field $\mathbf{u}$ along an oriented edge e_k is denoted by $c_k = \int_{e_k} \mathbf{u}$. The scalar field can either be saved as a 0-form, described by one value per vertex where $s_i = s(v_i)$, or as a 2-form with one value per triangle, $s_j = \int_{t_j} s_j$. It should be noted that while the divergence and the curl of a vector field are both scalar fields, typically, the divergence is expressed as a 0-form and curl expressed as a 2-form.

Two basic operators, the *differential* and *Hodge star*, in DEC can be used to derive the common differential operators in vector field analysis [16]. The *differential*, or exterior derivative d_k, simply maps k-forms to k+1-forms; whereas, the *Hodge star* $*_k$ is an one-to-one mapping of k-forms to n–k-forms in n dimensions. Since we work in 2D, we make additional interpretations (employing the above operators) as follows: d_0 is the *gradient* ∇, d_1 is the *curl* $\nabla\times$, $*_0$ is the multiplication by the area form, $*_1$ is a rotation by 90° on the tangent plane, and $*_2$ is the division by the area form. The discrete d, the transpose of the signed

incidence matrix, and the discrete $*$, the ratio between the size of dual mesh cell and the corresponding primal mesh cell, is described on M. The dual mesh (Voronoi diagram) can be described by forming a connection of centers from the primal (original) triangle mesh. Furthermore, all other differential operators can be fashioned from d and $*$, such as the *co-differential* $d_k^* = *_{k-1}^{-1} d_{k-1}^T *_k$, which maps $(k+1)$-forms to k-forms. Here $\cdot^T$ denotes adjoint operation, corresponding to transpose of matrices. The *divergence* $\nabla\cdot$, which turns a 1-form into a 0-form, can be accomplished through d_1^*.

(a) (b)

Figure 3.3: (a) and (b) show user strokes and demonstrate how Z changes as more light blue-colored strokes are added from (a) to (b). Nearby arrows may have nearly opposite directions as the user is designing an orientation field.

We now describe the weighted least squares approach in [16]; which, the discrete 1-form $\mathbf{U} = (u_1, u_2, \ldots, u_{n_1})^T$ denotes the vector field $\mathbf{u}$, where n_1 is the number of edges in the mesh, $u_i = \int_{e_k} \mathbf{u}$. Let $\mathbf{S} = (s_0, s_1, \ldots, s_{n_0})^T$ be 0-form representing the user-specified divergence field s, where n_0 is the number of vertices in the mesh, and $s_i = s(v_i)$. Likewise, 2-form $\mathbf{C} = (c_1, c_2, \ldots, c_{n_2})^T$ represents the user-specified curl field, where n_2 is the number of triangles. $\mathbf{U}_Z$ contains the user-specified constant values that $\mathbf{u}$ is constrained to be on user selected edges. Finally, Z is the matrix that projects an array denoting a 1-form onto the array of constrained edge values. With these discrete representations, the user-designed vector field can be formulated as the weighted least squares solution to the following equations

of $\mathbf{U}$,

$$d^*\mathbf{U} = \mathbf{S}, \quad d\mathbf{U} = \mathbf{C}, \quad Z\mathbf{U} = \mathbf{U}_Z,$$

which leads to

$$(*(dd^* + d^*d) + Z^T W Z)\mathbf{U} = *d\mathbf{S} + *d^*\mathbf{C} + Z^T W \mathbf{U}_Z,$$

where W is the weighting of the direct constraints on the vector directions, $*_0$ (area of each vertex) serves as the weighting for the constraint of divergence on vertices, and $*_2^{-1}$ (area of each triangle) serves as the weighting for the constraint of curl on triangles. Figure 3.3 demonstrates how Z updates when a user adds more stroke constraints. The resulting symmetric linear system is simply the vector field Poisson equation, setting aside the term induced by Z, with the Laplace-Beltrami operator $dd^* + d^*d$ being equivalent to ∇^2, leveraging the vector calculus identity below

$$\nabla^2 = \nabla\nabla\cdot - \nabla\times\nabla\times .$$

3.2.2 Natural Boundary Conditions

For Fisher et al.'s vector field design on regions with boundary, the *free* boundary condition is used when the vector field is not restricted to be at a certain angle with the boundary [16]. In their implementation, they introduce a term to properly evaluate the divergence operator for the vertices at the boundary, which has a neighborhood that is only partially within the domain. However, even with the additional term, the linear system is still under-determined with a large rank deficiency. As the operators for div and curl are discretized on different

mesh elements, vertices and faces, respectively, the resulting operator is still positive definite. However, the discrete Poisson equation with free boundary condition has a large number of near-zero eigenvalues, roughly as many as the number of boundary vertices. Thus, the vector fields produced may contain artificial singularities. With sufficient direct constraints, or a much denser bi-Laplacian, these spurious oscillations can be reduced at additional computational cost. However, the problem cannot be eliminated as their operator does not correspond to the actual vector field Dirichlet energy, which measures the variation of vector fields.

Liu et al.'s solution to the above problem is thus to minimize the Dirichlet energy $\int_M |\nabla \mathbf{u}|^2$, and thereby, obtain the smooth free boundary condition [28]. They first start by choosing a local orthonormal frame $\{\mathbf{e}_1, \mathbf{e}_2\}$ at each point $\mathbf{p}$, where the partial derivatives of the components of $\mathbf{u}$ are $u_{\alpha,\beta} = \frac{\partial u_\alpha}{\partial p_\beta}$. By focusing on the boundary's influence and considering a flattened region, they can ignore the curvature-related term and describe the boundary in the following equations:

$$\begin{aligned} & \int_M |\nabla \mathbf{u}|^2 = \int_M u_{1,1}^2 + u_{1,2}^2 + u_{2,1}^2 + u_{2,2}^2 \\ = \quad & \int_M (u_{1,1} + u_{2,2})^2 + (u_{2,1} - u_{1,2})^2 - 2(u_{1,1}u_{2,2} - u_{1,2}u_{2,1}) \\ = \quad & \int_M (\nabla \cdot \mathbf{u})^2 + (\nabla \times \mathbf{u})^2 - 2 \int_{\partial M} u_1 du_2 - u_2 du_1 \end{aligned}$$

The last equality turns the surface integral of the last term into the circulation boundary term $u_1 du_2 - u_2 du_1$ around the boundary curve of M. This is due to Stokes' theorem, which, in this case, states that the line integral of a vector field is equal to the surface integral of

its curl,

$$\nabla \times u_1 du_2 - u_2 du_1 = \nabla \times \begin{pmatrix} u_1 u_{2,1} + u_2 u_{1,1} \\ \\ u_1 u_{2,2} + u_2 u_{1,2} \end{pmatrix} = u_{1,1} u_{2,2} - u_{2,1} u_{1,1}$$

Then the term may be rewritten as

$$-\int_{\partial M} (\mathbf{u} \times d\mathbf{u}) \cdot \mathbf{n},$$

where $\mathbf{n}$ is the surface normal.

Liu et al. [28] describe the approach in a discrete setting with the resulting Dirichlet energy as a quadratic form of the vector $\mathbf{U}$ as $\mathbf{U}^T L \mathbf{U}$, and where $\mathbf{U}$ denotes the discrete 1-form representation of $\mathbf{u}$, and L is to be constructed as a Laplacian-like matrix. L is initialized as the sum of both the divergence and curl terms, and to which the boundary term is added. They then change the boundary integral into a summation over boundary edges in order to discretize the boundary term,

$$\sum_{e_i \in \partial M} (\mathbf{u}_i \times (\mathbf{u}_{i+1} - \mathbf{u}_i)) \cdot \mathbf{n}_i = \sum_{e_i \in \partial M} (\mathbf{u}_i \times \mathbf{u}_{i+1}) \cdot \mathbf{n}_i,$$

where they denote the boundary triangle associated with boundary edge i by triangle i , the edge that follows e_i along the boundary by e_{i+1} , and the surface normal at the shared vertex by $\mathbf{n}_i$.

By assuming that the discrete curl for the boundary triangles are near zero as in [16], a constant vector is located within each triangle, and thereby, allowing Liu et al. to plainly choose any point (more specifically, the barycenter) of the triangle for the computation of

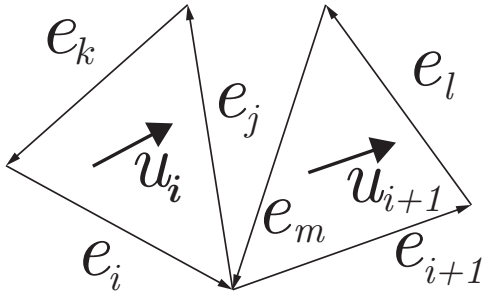


Figure 3.4: The image demonstrates two consecutive edges at the boundary.

$\mathbf{u}_i$. Recall that a discrete 1-form is described as one value per edge, $U = (u_1, u_2, \ldots, u_{n_1})$, where n_1 is number of edges. Figure 3.4 shows a pair of boundary triangles, which they mathematically define as

$$\begin{aligned} \mathbf{u}_i = & \quad u_i\boldsymbol{\phi}_i + u_j\boldsymbol{\phi}_j + u_k\boldsymbol{\phi}_k, \\ \mathbf{u}_{i+1} = & \quad u_{i+1}\boldsymbol{\phi}_{i+1} + u_l\boldsymbol{\phi}_l + u_m\boldsymbol{\phi}_m, \end{aligned}$$

where $\boldsymbol{\phi}_i = \frac{1}{3}(\nabla\phi_{v_2} - \nabla\phi_{v_1})$ is a constant vector equal to the basis function for edge i leading from v_1 to v_2, computed at the barycenter of the corresponding triangle, and ϕ_v denotes the linear basis function for vertex v. When updating L, it incorporates 18 additional terms in 9 pairs for each pair of consecutive boundary edges, as shown in

$$L_{jl} += (\boldsymbol{\phi}_j \times \boldsymbol{\phi}_l) \cdot \mathbf{n}_i, \; L_{lj} += (\boldsymbol{\phi}_j \times \boldsymbol{\phi}_l) \cdot \mathbf{n}_i.$$

The surface normal located at the vertex on a curved patch may be obtained from a reasonable weighted average.

Intuitively, one may choose an arbitrary global alignment direction, and the Dirichlet energy is minimized by a constant vector field, when there are no other constraints. This is reflected in that the final linear system has a kernel space of dimension two for simply connected domains. The modified matrix L remains symmetric, thus the linear system is still easy to solve. The effects of these conditions are shown in [28].

We concentrate on the algorithm for planar regions, so orientation fields can be represented by vector fields by doubling the angle of the vector in polar coordinates. Thus, the vector field design tool can be directly used as an orientation field design tool.

3.3 PatchMatch with Alignment Constraints

We have modified PatchMatch such that it is guided by a vector or orientation field as described in the previous section. The main difference from the original algorithm is that the patches (neighborhoods) used in the output images should be created by sample points on a local grid that is aligned to the given direction. This involves changes to a crucial step of the PatchMatch algorithm, i.e., the propagation step, which takes advantages of coherence within the output image to improve the overall efficiency. In addition, the initialization process for each level of the multi-resolution approach is also modified. One further change is required for handling orientation fields.

3.3.1 Patch Comparison

The output image is represented by the nearest neighbor field (NNF) $\mathbf{f}(\mathbf{p})$, which indicates where each point $\mathbf{p}$ in the output image should be mapped to in the source domain, such that the neighborhood patch of $\mathbf{p}$ in the target is similar to the neighborhood patch of $\mathbf{f}(\mathbf{p})$ in the source. With user-designed salient feature directions, the patch in the output image should no longer be a sub-grid centered at $\mathbf{p}$, but instead a patch created by resampling at the gridpoints of the subgrid transformed by an affine transformation matrix (see Figure 3.5). More precisely, if we assume the patch is a block of size $(2n+1) \times (2n+1)$ pixels, and $\mathbf{p} = (x, y)^T$, the original sub-grid is formed by the following grid points,

$$(x + ih, y + jh)^T, \quad -n \leq i, j \leq n,$$

where h is the grid spacing of the output image. The transformed sub-grid for the patch instead consists of

$$\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} u_1 & -u_2 \\ u_2 & u_1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}, \quad -n \leq i, j \leq n,$$

where $\mathbf{u} = (u_1, u_2)^2$ is the alignment target direction for the x-axis direction of the pattern in the exemplar. If we wish to ignore the influence of the sizing control in $\mathbf{u}$, we can normalize the vector field, or perform polar decomposition $\mathbf{u} = u(\cos\theta, \sin\theta)^T$, and construct the following subgrid with the original size,

$$\begin{pmatrix} x \\ y \end{pmatrix} + R \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix}, \quad -n \leq i, j \leq n,$$

where R is the rotation around origin by θ.

With the pixel values sampled on the transformed grid points through bilinear interpolation, the distance function measuring the differences between two patches, such as SSD, can be directly applied as in the original approach (see Algorithm 3). Patch construction in both propagation and random search should employ this transformation. This also influences the "patch-voting" step (see Algorithm 2), where pixel color $c(\mathbf{p})$ of a point $\mathbf{p}$ is determined by averaging the color of corresponding pixel with any patch copied from the source that covers the pixel,

$$c(\mathbf{f}(\mathbf{p} - (ih, jh)^T) + (ih, jh)^T), \quad -n \leq i, j \leq n.$$

With alignment field, each patch copied from the source/exemplar is rotated before voting for the final color of the original sample points that it covers. Assume that the affine transformation matrix (rotation and possible scaling) is A, the voting received from a patch retrieved from the source by a nearby output point $\mathbf{p}'$ is

$$c(\mathbf{f}(\mathbf{p}') + A^{-1}_{\mathbf{p}'}(\mathbf{p} - \mathbf{p}')).$$

3.3.2 Initialization in Multiresolution Approach

As often done in texture synthesis of patterns with multiscale features, we tile each level of the Gaussian pyramid of a user-supplied example texture in the matching level of the output image pyramid. Algorithm 1 and Figure 3.6 describe the setup of image pyramid for both the example and output textures. Note that, in practice, a frustum is often used instead of the pyramid, since it is not always necessary to get to the level with a single pixel. The

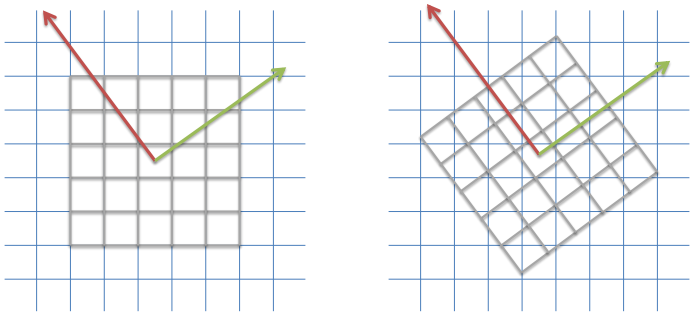


Figure 3.5: The altered algorithm takes the original patch (a) and transforms (b) it in the direction indicated by the vector field.

synthesis is performed in a top-down fashion. We start from the coarsest level, and then refine the result by stepping down the pyramid.

At each level l, the algorithm first initializes the nearest neighbor field using prior information from the example texture. The current level texture features remain roughly the same to the coarser level, by initializing $\mathbf{f}_l$ to an upsampling of $\mathbf{f}_{l-1}$ from the previously synthesized coarser level $(l-1)$ in the pyramid. In our implementation, we double the size from level $l-1$ to l. Thus, each pixel location in the previous level corresponds to 4 synthesized pixel locations in the current level.

3.3.3 Propagation in Rectified Directions

The key to the efficiency of PatchMatch algorithm is the coherence between the mapped source locations of nearby target patch centers. However, with local orientations, the displacement between the target points, for instance, $\mathbf{f}(\mathbf{p})$ and $\mathbf{f}(\mathbf{p}+(-1,0)^T)$, is not the same as the displacement $(-1,0)^T$ of the source points used to derive the candidate in the propa-

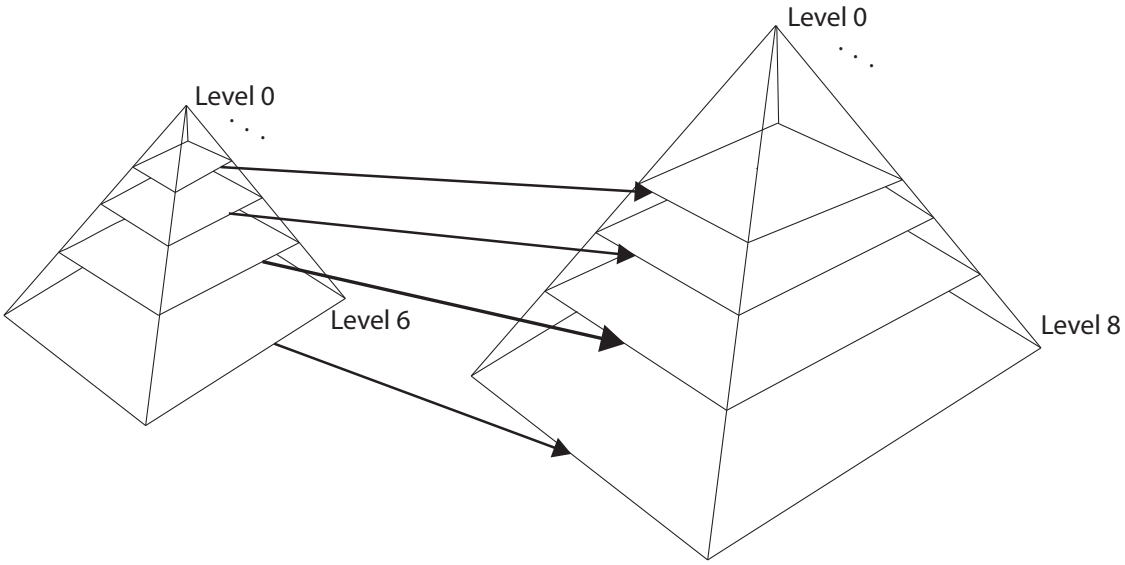


Figure 3.6: The smaller pyramid for a 64x64 pixel example texture and larger pyramid is for a 256x256 pixel output image. The arrows indicate how the example texture pyramid levels correspond to the output texture levels. For example, starting from the 3rd level of the example pyramid corresponds to tiling the level on the 6th level of the output pyramid.

gation process. More precisely, the neighboring patches in propagation must be aligned with the orientation field when refining the target patch at $\mathbf{p}$. Our solution is to add the inverse rotation used in the aforementioned alignment procedure directly to the displacement. For the above example of candidate set construction for even iteration, checking the location one pixel to the left, instead of using $\mathbf{f}(x-1,y)+(1,0)^T$ as a candidate, we use

$$\mathbf{f}(\begin{pmatrix} x-1 \\ \\ y \end{pmatrix}) + R^{-1}\begin{pmatrix} 1 \\ \\ 0 \end{pmatrix};$$

and for one pixel below we replace $\mathbf{f}(x,y-1)+(0,1)^T$ by

$$\mathbf{f}(\begin{pmatrix} x \\ \\ y-1 \end{pmatrix}) + R^{-1}\begin{pmatrix} 0 \\ \\ 1 \end{pmatrix}.$$

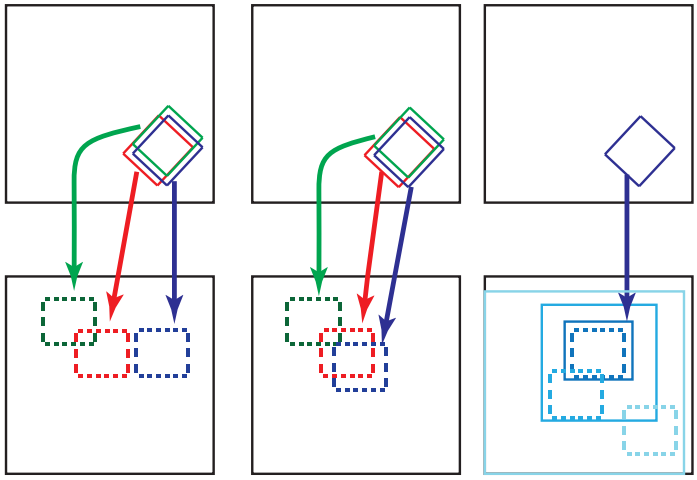


Figure 3.7: Modified algorithm steps: first image (a) initializes the NNF to have prior information from the Gaussian Pyramid; second image (b) propagates matches by checking neighbors of the blue patch, with direction indicated in the target texture, to see if they will improve the patch; third image (c) shows the current patch searching randomly, within a window, for improvements.

Note that we use real numbers for $\mathbf{f}$, which means that bilinear interpolation may have to performed, but it is necessary in order to shift the candidate to the correctly indicated source patch (see Figure 3.7b). A similar procedure is done for propagating patches from the right and up one pixel.

Alternative Solution Instead of modifying the displacement between the source patches, we can modify the displacement between the target patch instead. In the latter case, we predict candidate patches also by using the orientation R at the current target patch centered at (x, y), to find the neighbor patch that will be mapped to the given displacement. If the candidate patch is to be mapped to the source patch one pixel to the left of $\mathbf{f}(x, y)$ we

must then modify the target location from $(x-1,y)^T$ to $(x,y)^T + R(-1,0)^T$, and use

$$\mathbf{f}(\begin{pmatrix} x \\ y \end{pmatrix} + R \begin{pmatrix} -1 \\ 0 \end{pmatrix}) + \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

instead of $\mathbf{f}(x-1,y)+(1,0)^T$. This means we may have to bilinearly interpolate $\mathbf{f}$. In a similar manner, we modify $\mathbf{f}(x,y-1)+(0,1)^T$ to

$$\mathbf{f}(\begin{pmatrix} x \\ y \end{pmatrix} + R \begin{pmatrix} 0 \\ -1 \end{pmatrix}) + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

to find the candidate patch one pixel below the current patch. This is not a perfect solution for locations where $\mathbf{f}$ is not continuous, but may be preferable when large scaling is involved between the source and target, in which case the matrix R should be replaced by the affine transformation matrix.

3.3.4 N-RoSy Field Alignment

There are additional issues when N-RoSy fields are used instead of vector fields for textures with N-way rotational symmetry, one associated with the upsampling procedure in multiresolution approach, and the other with the propagation step.

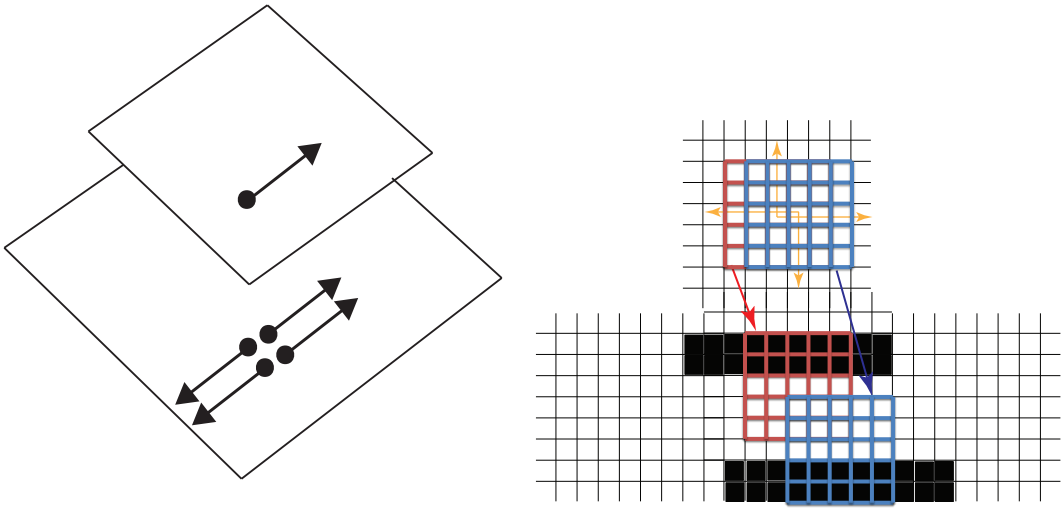
For multiresolution approach, we need to check the consistency of the orientation field between different scales since the coarse and fine levels may choose different alignments when upsampling from the original orientation fields as shown in Figure 3.8a. In order to handle this situation, we keep an array of integers, $S(\mathbf{p}) \in \{0,1,\ldots,N-1\}$, denoting the additional rotation alignment of the target patch by an angle of $S(\mathbf{p})\frac{2\pi}{N}$. This treatment

is more general than Liu et al.'s approach [28], where they use an array of boolean values $B(\mathbf{p})$ for each output point to indicate a rotation of π. They also have to store a π-rotated appearance vector since they use a compressed patch instead of a color at each location in the exemplar. While their method works for orientation fields, it does not handle the case for N-RoSy fields. N-RoSy fields could potentially have any of the N different rotations, and therefore, we must store the indicated rotation as an integer instead, in order to align the patch correctly at a discontinuous location. In the next chapter, we compare and discuss our method with Liu et al.'s Orientation Guided Parallel Texture Synthesis.

In addition to the inconsistency in upsampling for the initialization of level l from level $l-1$, within the same level, there can also be inconsistency during the propagation step. Figure 3.8b demonstrates such a case, where two neighboring locations have opposite directions. To construct a candidate location for the blue patch, we start from the red patch, however, the patch to the right of the red patch does not directly provide as good a candidate as the current mapping. However, if we take the relative rotation between $R(x-1, y)$ and $R(x, y)$ into account, we may be able to generate a competitive candidate. This additional rotation can also be handled by incorporating another rotation into the above-mentioned $S(\mathbf{p})$. This procedure is not necessary since the discontinuity is sporadic. However, it does speed up the convergence since propagation is likely to produce better candidates than the random search step.

3.3.5 Final Output and Parameters

The overall framework remains the same to the original PatchMatch. After our first "patch-voting" step to generate pixel values for the initial output texture image in the top level of the output pyramid (or frustum), we run our modified refinement iteration steps, including



(a) (b)

Figure 3.8: Image (a) shows the changes in the vector field when sampling patches from different pyramid levels. Image (b) demonstrates how to properly rotate the candidate patch given the discontinuous orientation in the target patches.

propagation and random search, taking the necessary additional rotation and possible scaling into account, and then again run the "patch-voting" procedure to generate an updated image from the NNF. To improve efficiency, we implemented the suggested early termination feature, should the partial summation exceed the best known distance as suggested in [2]. We then use the output from Orientation Guided PatchMatch in synthesizing the next level of the Gaussian pyramid, checking the consistency between the orientation (N-RoSy) field and its upsampling. We repeat the described procedure until we have reached the finest resolution image.

The number of iterations and patch size may be specified per level of the output pyramid. These parameters, plus the starting level of the exemplar, allow for improved control over the density of singularities in the output texture even though we may not know the exact location of the singularities, such as bifurcation and ending of ridge-like features. By starting from a finer resolution level in the exemplar, more singularities tend to appear. In this

scenario, we may need to increase the number of iterations, and/or, patch size to produce smoothly varying texture features. Alternatively, we can decrease the density of singularities by starting from a coarser pyramid level. Additionally, increasing the patch size will decrease the singularity count.

These effects can be interpreted by the amount of coherence influenced by each of the choices. Increasing the patch size forces larger regions to be similar to those in the exemplar, thus suppressing the spurious singularities. Starting from a coarser level has the same effect as it influences the initialization for all the finer levels. Finally, the number of iteration controls how refined the neighborhoods are, i.e., how close the patch centered at each location is to one of the patches in the exemplar. Thus, a large iteration number leads to fewer singularities if the exemplar does not contain any singularities.

In this chapter, we have presented our modifications to PatchMatch. Orientation (or N-RoSy) fields now guide the alignment of patches in synthesizing the output texture. Results of our implementation are shown and discussed in the next chapter.

Algorithm 1 Procedure for building a pyramid of a texture.

```
1: function BuildPyramid
2:     level ← 0
3:     Determine the total number of pyramid levels given the size of the texture
4:     for level → ImageSize >>= 1 do
5:         level + 1
6:     end for
7:     Coordinates ← ImageCoordinates                ▷ Image coordinates to be synthesized
8:     pyramid(level) ← Coordinates                  ▷ Finest resolution level contains the original texture coordinates
9:     while level > 0 do                            ▷ Assume height and width are the same
10:        DownSample(Coordinates)
11:        level − 1
12:        pyramid(level) ← Coordinates              ▷ Set next coarsest level coordinates of texture
13:    end while
14: end function
15:
16: function DownSample(Coordinates)
17:    width ← Coordinates.GetSize()/2               ▷ Resize the coordinate array by half
18:    height ← Coordinates.GetSize()/2
19:    temp(width, height) ← 0                       ▷ Temporary array
20:    for y = 0 → height; y + 1 do
21:        for x = 0 → width; x + 1 do
22:            temp(x, y) ← (Coordinates(2 ∗ x, 2 ∗ y) + Coordinates(2 ∗ x, 2 ∗ y + 1) + Coordinates(2 ∗ x + 1, 2 ∗ y) + Coordinates(2 ∗ x + 1, 2 ∗ y + 1))/4
23:            ▷ Each coordinate in the next coarsest level corresponds to the average of 4 coordinates from the current level.
24:        end for
25:    end for
26:    Coordinates ← temp
27: end function
```

Algorithm 2 Procedure for "Patch-Voting".

```
1: function VOTE(PatchSize, ExemplarLevel, R)                ▷ R is the rotation array.
2:     image ← 0                                  ▷ Holds color values for texture image.
3:     votes ← 0                                                  ▷ Count votes per pixel.
4:     for y = 0 → Image.GetHeight(); y + 1 do
5:         for x = 0 → Image.GetWidth(); x + 1 do
6:             patchLeftCorner ← PatchSize ∗ 0.5          ▷ Start from left corner of patch
7:             for dy = −patchLeftCorner → patchLeftCorner do
8:                 if y + dy < 0||y + dy ≥ height then          ▷ Check boundary of image
9:                     continue
10:                end if
11:                for dx = −patchLeftCorner → patchLeftCorner do
12:                    if x + dx < 0||x + dx ≥ height then      ▷ Check boundary of image
13:                        continue
14:                    end if
15:                    p ← (x + dx, y + dy)
16:                    pixel c ← c(f(p) + R(dx, dy)^T)
17:                    image(x, y)+ = c
18:                    votes(x, y) + 1
19:                end for
20:            end for
21:        end for
22:    end for
23:
24:    for y = 0 → height do                                          ▷ Averaging Step
25:        for x = 0 → width do
26:            outputimage(x, y) ← image(x, y)/votes(x, y)
27:        end for
28:    end for
29: end function
```

Algorithm 3 Procedure for Computing The Distance Between Two Patches.

function DISTANCE($PatchSize, x, y, ExemplarLevel, R, Cutoff = Max$)
    R is the rotation array.
    $offset \leftarrow 0$
    $patchLeftCorner \leftarrow PatchSize * 0.5$ ▷ Start from left corner of patch
    for $dy = -patchLeftCorner \rightarrow patchLeftCorner$ **do**
        if $y + dy < 0 || y + dy \geq height$ **then** ▷ Check boundary of image
            continue
        end if
        for $dx = -patchLeftCorner \rightarrow patchLeftCorner$ **do**
            if $x + dx < 0 || x + dx \geq height$ **then** ▷ Check boundary of image
                continue
            end if
            $\mathbf{p} \leftarrow (x + dx, y + dy)$
            pixel $o \leftarrow c(\mathbf{f}(\mathbf{p}))$ ▷ target pixel
            pixel $c \leftarrow c(\mathbf{f}(\mathbf{p}) + R(dx, dy)^T)$ ▷ candidate pixel
            $blue \leftarrow o(blue) - c(blue)$
            $green \leftarrow o(green) - c(green)$
            $red \leftarrow o(red) - c(red)$
            $offset+ = blue * blue + green * green + red * red$
            if $offset \geq cutoff$ **then**
                return $cutoff$
            end if
        end for
    end for
    return $offset$
end function

Chapter 4

RESULTS

We performed our tests on a normal laptop with AMD Athlon II P360 Dual-Core Processor with 4 GB of memory. Our default parameter setting is 7x7 pixels patch size and 5 iterations starting from the 3rd level of a pyramid for a 32x32 pixels example-based texture. We demonstrate this default parameter setting in Figure 4.1. For textures with larger features, we recommend using a larger patch size to capture all of the feature details; for example, a 14x14 pixels patch size. More iterations are required for larger patch sizes in order for the features to converge properly, and fewer iterations are needed for smaller patch sizes. This is due to the fact that we are making the pixels consistent within smaller local regions with a small patch size (e.g. 4x4 pixels), whereas we need to make them consistent within larger regions for a large patch size (e.g. 14x14 pixels).

The following figures demonstrate the necessity for tuning our approach's parameters. The choice of parameters is important in synthesizing quality textures. Should a wrong parameter be picked, the results will contain undesirable artifacts. Further, these examples explain what happens if we turn off and on a parameter.

Figure 4.2 exhibits effects of the multiresolution approach. If we turn off multi-resolution then the synthesized output does not have good coherence across patches. Whereas, turning it on produces a synthesized texture without artifacts around patches and binds together the texture features. As we downsample into the coarser levels of the exemplar, we are able to

(a) (b)

Figure 4.1: Image (a) is a 32x32 pixels example texture that was used to synthesize the 512x512 pixels texture shown in Image (b) using our default parameter settings of 7x7 pixels patch size and 5 iterations starting from the 3rd level of a pyramid.

capture a larger scale texture feature within the fixed patch size; thus, producing coherent regions in the output texture.

Figure 4.3 shows an experiment with turning on and off iteration. Of course, without iteration we are unable to propagate good matches across the texture. Instead "patch-voting" attempts to average the rotated patches in the output texture, producing a smoothed tiling effect. Turning on iteration allows for propagation of candidate patches, thereby, synthesizing a coherent texture.

The experiment in Figure 4.4 demonstrates the difference between a 3x3 pixels patch and a 7x7 pixels patch size. The result of a smaller patch size is similar to turning off iterations parameter. Neither "patch-voting" nor propagation can produce large coherent regions, although multiresolution does save the result from being completely uncorrelated pixels. In contrast, by setting the patch size parameter to 7x7 pixels, we show that we achieve good candidate patch placement in the output texture.

The following experiment in Figure 4.5 demonstrates turning off and on "patch-voting." Without "patch-voting," the synthesized texture contains many seams running along the

surface (see Figure 4.5b). However, by turning on "patch-voting," our approach produces a texture in Figure 4.5c that is seamless. In general, we observe that "patch-voting" always helps produce a smooth output texture.

In Figures 4.6, 4.8, and 4.7, we apply modified PatchMatch for 4-RoSy fields using different exemplars. Our results demonstrate that the method can handle 4-RoSy fields as well as 2-RoSy fields.

In Figure 4.9, we synthesize a latent fingerprint, G002l, from the NIST Special Database 27. The synthesized latent fingerprint is a reasonable match when compared to the template and latent fingerprints.

Table 4.1 shows the average runtimes using a 32x32 pixel example texture to synthesize 128x128 and 256x256 pixels output textures. We chose parameters of 5 and 10 iterations with patch sizes of 3x3, 7x7, and 14x14 pixels; additionally, we started from the 3rd and 5th levels of the example texture. Our approach is fast for output textures with smaller patch sizes, but efficiency can be further improved, for example, by replacing the bilinear interpolation by truncation. Speed-ups are noted when we start from a finer resolution of the pyramid and/or decrease the number of iterations, however, the output texture will contain more singularities as a result. Efficiency is lacking when setting the patch size parameter to 14x14 pixels. This may be related to the large amount of bilinear interpolation performed during the SSD evaluation.

Table 4.1: Average Runtime For a 32x32 Example Texture.

Patch Size	**Iterations**	**Example Pyramid Level**	**Output Size (pixels)**	**Processing Time (seconds)**
3	5	3	128	5.82
7	5	3	128	28.61
7	10	3	128	35.69
14	5	3	128	95.05
14	10	3	128	195.85
3	5	5	128	3.11
7	5	5	128	13.55
7	10	5	128	37.73
14	5	5	128	84.00
14	10	5	128	164.37
3	5	3	256	12.24
7	5	3	256	84.75
7	10	3	256	203.78
14	5	3	256	433.82
14	10	3	256	649.78
3	5	5	256	11.60
7	5	5	256	58.07
7	10	5	256	113.81
14	5	5	256	240.65

Table 4.1 (cont'd)

14	10	5	256	509.03

Next, we compare our results with Liu et al.'s Orientation Guided Parallel Texture Synthesis method [28] and Zhang and Palacios's N-RoSy fields approach [31]. Liu et al.'s proposed approach is similar to our approach in that both frameworks implement the same vector field design. The difference is the applied texture synthesis method. Zhang and Palacios have proposed a different vector-based representation for N-RoSy fields and implement a LIC method for visualizing them. We refer the reader back to Chapter 2 for more details on LIC and their vector-based representation.

Zhang and Palacios's proposed design is rendered on a mesh [31], so we cannot directly compare their method with ours. However we can still make inferences about their approach. It is very memory intensive, however, it does produce efficient results in 10 seconds or less. We found that artifacts do appear when singularities are placed close together. We show a couple of examples in Figures 4.10 and 4.11 using multiple singularities with 2-, 4-, and 6-RoSy fields.

Orientation Guided PatchMatch has similar efficiency results as Liu et al's approach. Their method produces spurious singularities and does not have direct control over singularity placement. Such an example is in Figure 4.12. However, both approaches have trouble converging in Figure 4.13 when synthesizing the metal exemplar. We show in Figure 4.14 that our approach produces less singularities, yet their approach is able to maintain better exemplar texture structure. In Figure 4.15, we have smoothed knot artifacts, yet we are still able to produce less singularities than Liu et al.'s method.

In this chapter, we demonstrated the effects of our parameters, and the necessity of the multiresolution approach and the propagation step. In these results, we show that singularity control is possible even without knowing their exact locations by synthesizing textures for 2- and 4-RoSy fields. Additionally, we presented the results of the application of orientation guided PatchMatch to enhancing latent fingerprints.

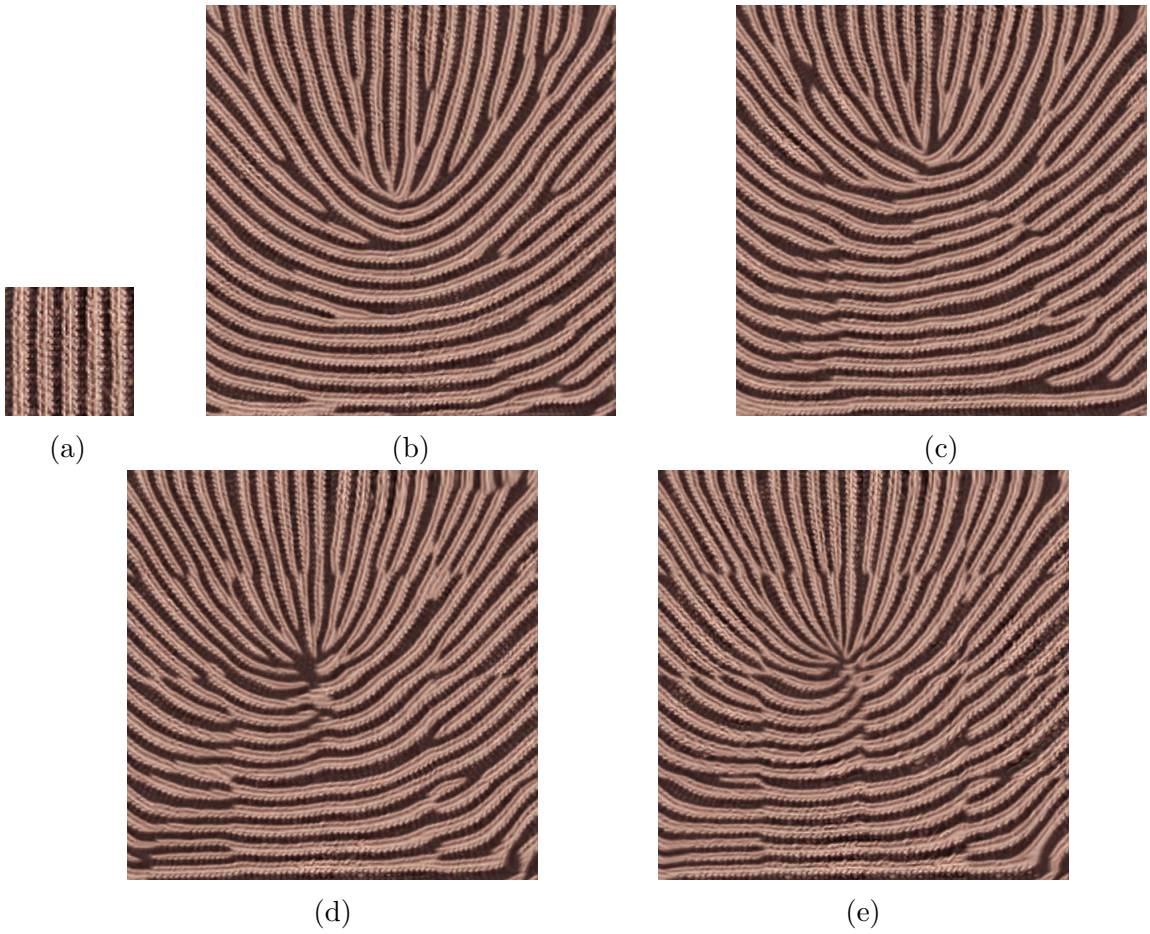
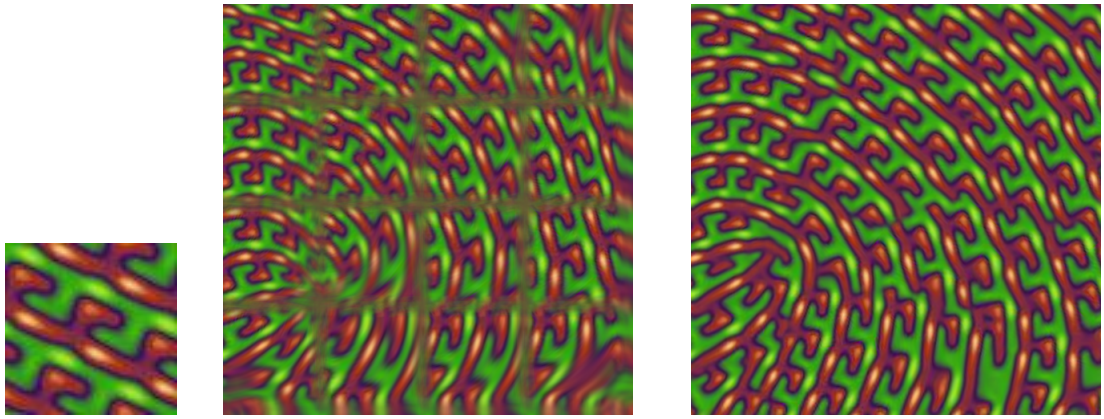
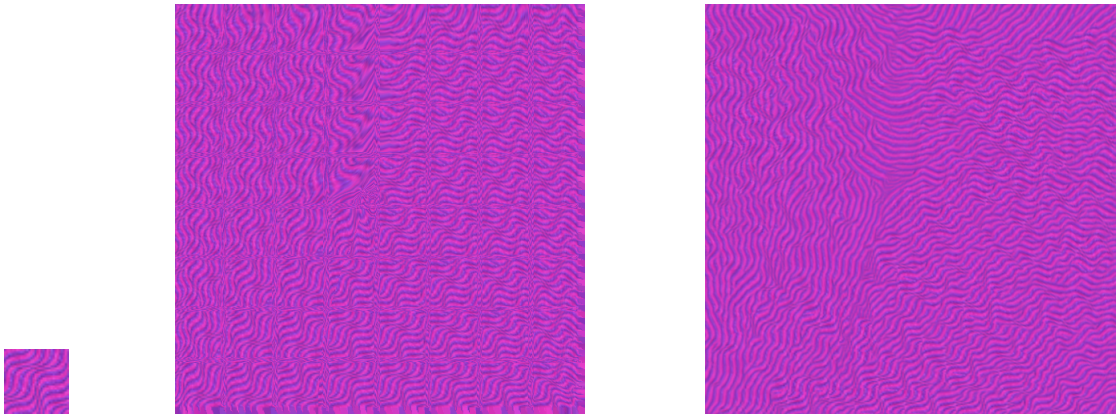


Figure 4.2: Images (b), (c), (d), and (e) are 512x512 pixels synthesized textures guided by orientation fields, with a patch size of 7x7 pixels and 5 iterations. (b) is starting from the 3rd level of 64x64 pixels example texture in (a). (c) is starting from the 4th level. (d) is starting from the 5th level. (e) does not use multi-resolution to synthesize the texture. (b) has better defined texture features as compared to (c), (d), and (e). (c) does show improve in retaining feature textures than (d). (e) contains visible seams along the tile boundaries, which result from the lack of large scale smoothing.



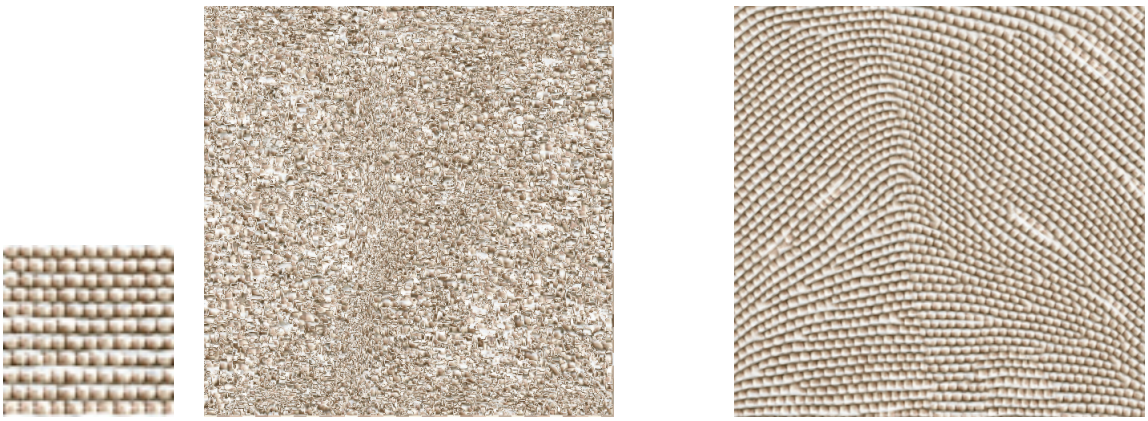
(a) (b) (c)

Figure 4.3: Image (b) and (c) are 512x512 pixels synthesized textures using orientation fields, demonstrate using a patch size of 14x14 pixels and 0 iterations for (b) and 10 iterations for (c), starting from the 3rd level of 64x64 pixels example texture in (a). (b) is an undesirable synthesized texture with boundary seams and many artifacts, whereas, (c) demonstrates good results.



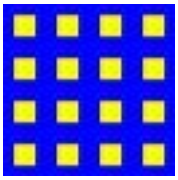
(a) (b) (c)

Figure 4.4: Image (b) and (c) are 512x512 pixels synthesized textures using orientation fields, demonstrate using a patch size of 3x3 pixels for (b) and 7x7 pixels patch size for (c) and 10 iterations, starting from the 3rd level of 32x32 pixels example texture in (a). (b) is a suboptimal synthesized texture with boundary seams and many artifacts, whereas, (c) demonstrates high-quality results.

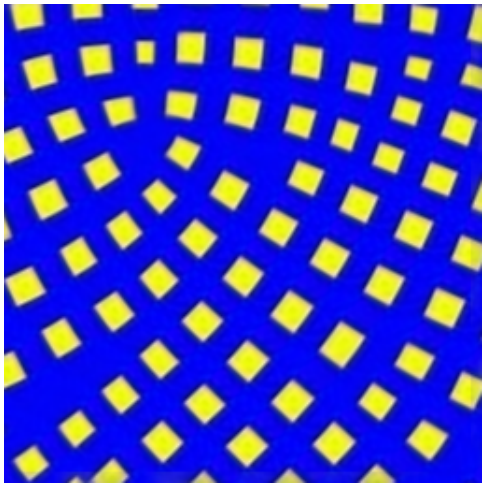


(a) (b) (c)

Figure 4.5: Image (b) and (c) are 512x512 pixels synthesized textures using orientation fields, demonstrate turning off and on patch-voting and using a patch size of 7x7 pixels and 5 iterations, starting from the 3rd level of 64x64 pixels example texture in (a). (b) is a noisy synthesized texture with boundary seams and many artifacts, whereas, (c) demonstrates structured results.

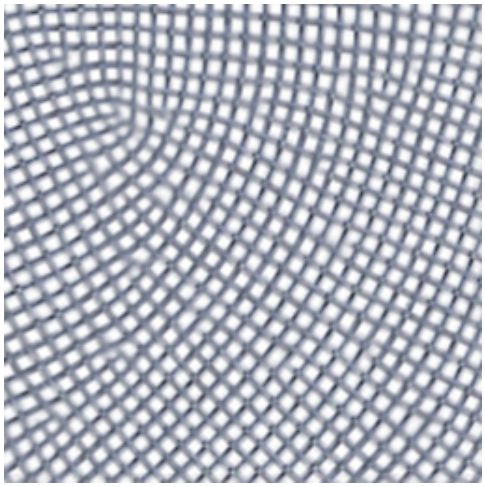


(a)



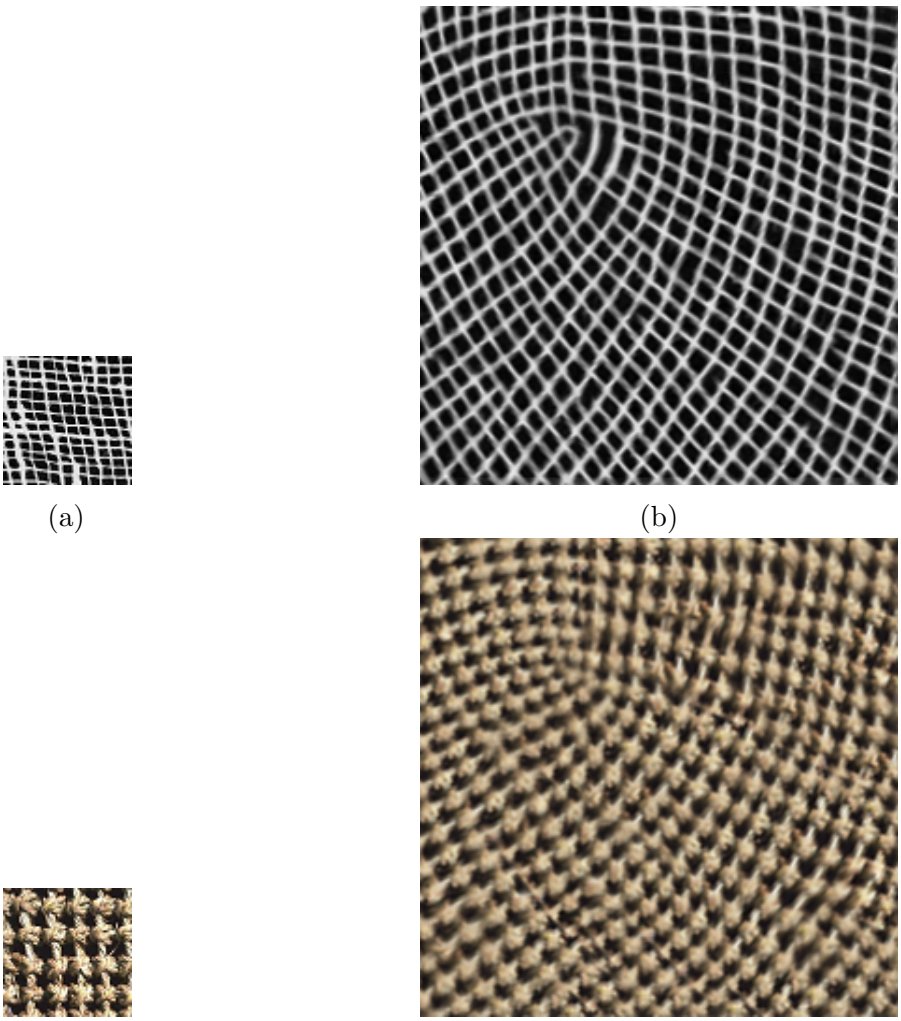
(b)

(c)



(d)

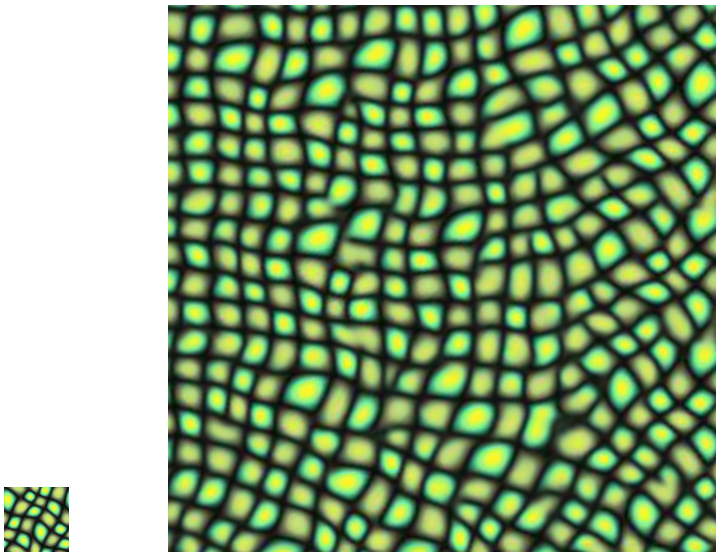
Figure 4.6: 4-RoSy field results. Image (a) and (b) are 64x64 and 32x32 pixels example textures. Image (b) was synthesized with a patch size of 10x10 pixels with 7 iterations, starting from the 3rd level of example texture. Image (c) was created using patch sizes of 32, 16, and 8 pixels with 10 iterations, starting from the 3rd level of example texture.



(a) (b)

(c) (d)

Figure 4.7: 4-RoSy field results. Image (a) and (b) are 64x64 pixels example textures. Image (b) was synthesized with a patch size of 7x7 pixels with 7 iterations, starting from the 3rd level of example texture. Image (c) was created using patch size 9x9 pixels with 10 iterations, starting from the 3rd level of example texture.

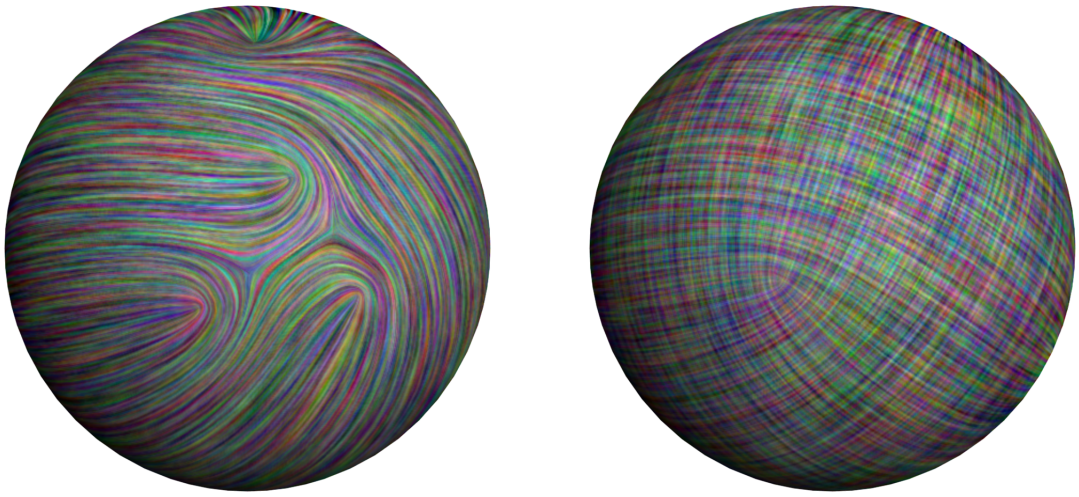


(a) (b)

Figure 4.8: 4-RoSy field results. Image (a) is a 64x64 pixels example texture. Image (b) is 512x512 pixels output that was synthesized with a patch size of 14x14 pixels with 5 iterations, starting from the 4th level of example texture.

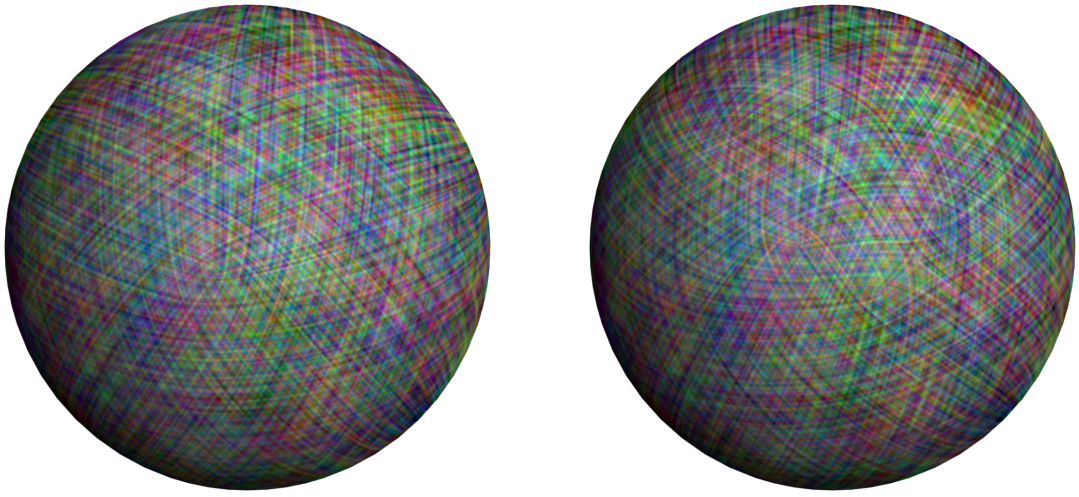
(a) (b) (c)

Figure 4.9: Latent fingerprint synthesis results. Image (a) latent fingerprint, G002l, (b) synthesized result with 7x7 pixel patch size for 10 iterations, starting from 3rd level of example stripe texture; (c) matching template fingerprint, G002t. Fingerprints are from the NIST Special Database 27.



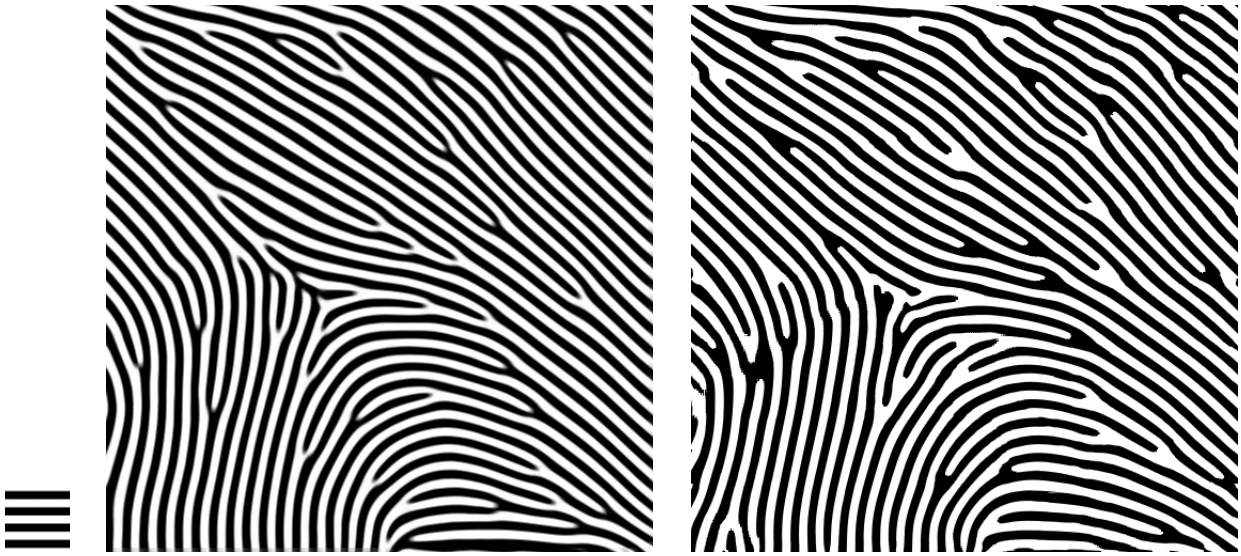
(a) (b)

Figure 4.10: Image (a) shows Zhang and Palacio's vector visualization approach for a 2-RoSy field with multiple singularities on a circle mesh. Image (b) uses a 4-RoSy field with one singularity.



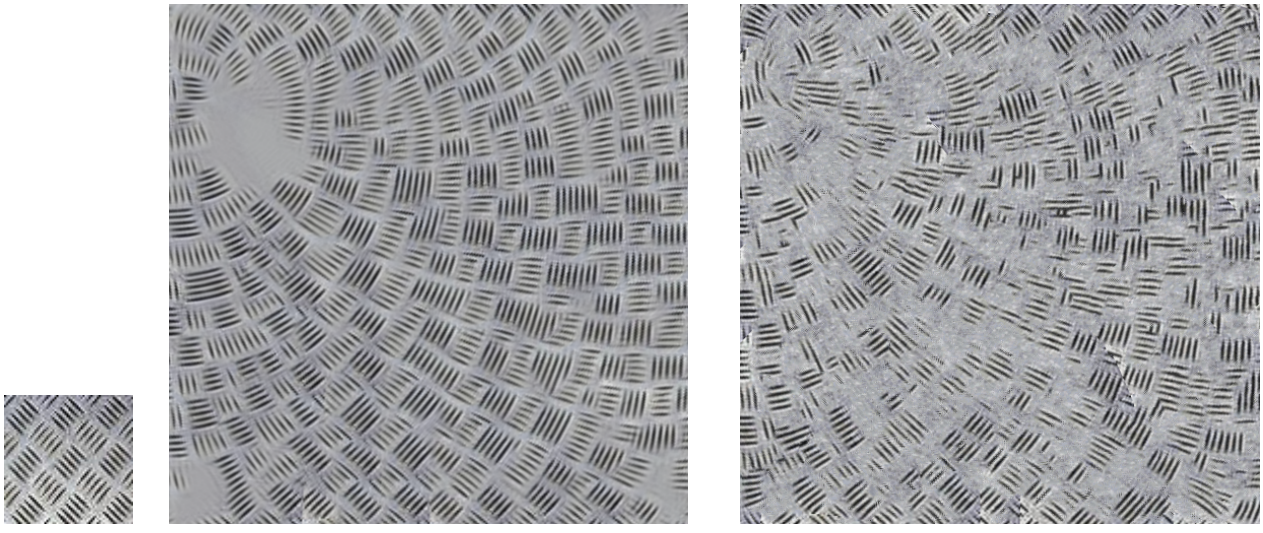
(a) (b)

Figure 4.11: Image (a) shows Zhang and Palacio's vector visualization approach for a 6-RoSy field with one singularity on a circle mesh. Image (b) uses a 6-RoSy field with multiple singularities. (b) shows some artifacts near the singularities.



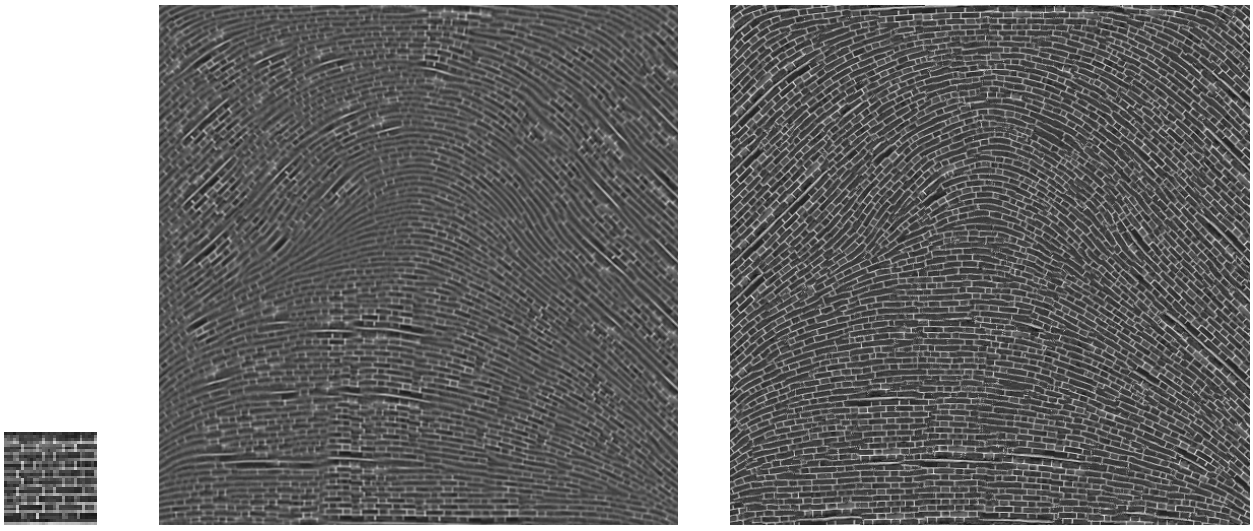
(a) (b) (c)

Figure 4.12: Image (a) is the exemplar used in synthesizing output images (b) and (c). Image (b) is produced by our method and image (c) is synthesized from Liu et al.'s method. Our output texture (b) contains less singularities than Liu et al.'s output texture in (c).



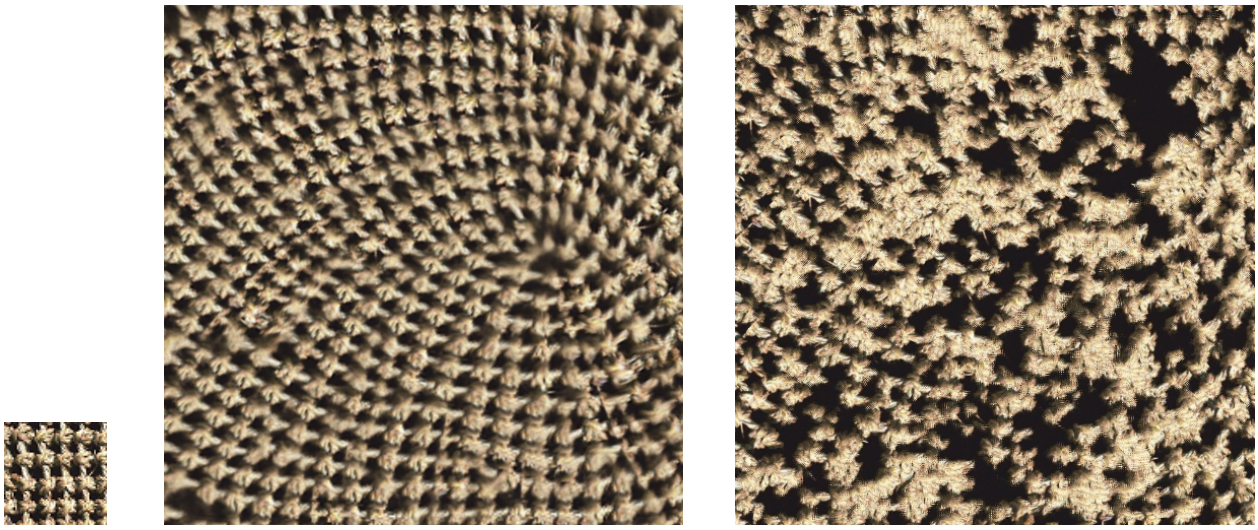
(a) (b) (c)

Figure 4.13: Image (a) is the exemplar used in synthesizing output images (b) and (c). Image (b) is produced by our method and image (c) is synthesized from Liu et al.'s method. Neither output texture is able to properly converge.



(a) (b) (c)

Figure 4.14: Image (a) is the exemplar used in synthesizing output images (b) and (c). Image (b) is produced by our method and image (c) is synthesized from Liu et al.'s method. Their method retrains better exemplar texture structure; however, we still produce fewer singularities.



(a) (b) (c)

Figure 4.15: Image (a) is the exemplar used in synthesizing output images (b) and (c). Image (b) is produced our method and image (c) is synthesized from Liu et al.'s method. Both algorithms produce artifacts: we have smoothed knots and theirs contains multiple holes that are not filled in.

Chapter 5

CONCLUSION

We presented Orientation Guided PatchMatch which guides the alignment of texture features, indicated by a direction in the x, y-plane, to synthesize new textures. We provide control over spurious singularities and demonstrated that we have more control over their placement. We also show that we can synthesize textures with two-and four-way symmetry. This is further extended to N-way symmetry by taking the angle, θ, and dividing it by N for creation of the RoSy field. We have compared our results with two other approaches, [28] and [31]. Finally, we show our results in enhancing a latent fingerprint.

Limitations As demonstrated in our results, our algorithm is not as efficient when compared to the timings in Barnes et al. [2]. During our experiments, we found the time and memory intensive part to be computing the distance between two patches. Additionally, the runtime for this procedure is even longer in the random search phase. Since PatchMatch is a sequential algorithm, the runtime should be $O(mMlogM)$ and memory usage $O(M)$ for image regions and patches of size M and m pixels [2]. In future work, we will explore how to improve the runtime and memory usage.

When enhancing a latent fingerprint, there is no guarantee that spurious singularities will not appear in our synthesized texture. Additional singularities, e.g., saddle points, may occur if we place two sources close to each other. When placing two singularities close

to each other, we have artifacts that appear in the synthesized texture. Only by tuning parameters, e.g., changing the hierarchical or final resolution size, can we reduce the extra singularities. However, it should be noted that we do not always know the exact location of the singularities. Such is the case for latent fingerprint enhancement.

Future Work Many possible applications exist for the extension of the work described in this thesis. The original PatchMatch algorithm was designed for implementation in hole-filling and constraints in image editing [2]. Additionally, texture synthesis algorithms have been extended onto 3-D models in order to demonstrate the properties of vector fields on the surface of an object. Flow, or time-varying, texture synthesis describes the movement of the vector field over time and is used to describe the flow of wind and water in various man-made structures. Finally, we wish to continue our work in latent fingerprint enhancement.

In the future, we will explore CPU and GPU implementation for orientation guided PatchMatch. Barnes et al. [2] describe both implementations; however, they do mention that the GPU implementation is known to be slower than CPU. Therefore, we should be able to extend Orientation Guided PatchMatch into either framework to improve efficiency.

One of the major contributions of PatchMatch is improved constrained texture synthesis. We will explore further how image editing — retargeting, hole filling, and reshuffling — can be used to improve the quality of latent fingerprints as well as N-RoSy fields. In our review of the literature, only [32] demonstrates how singularities can be canceled out and is done in more automatic fashion. It would be novel to allow users, especially latent examiners, to manually remove singularities, such as bifurcation, or reduce spacing while keeping certain directions in the synthesized latent fingerprint.

Design of flow simulations is an important topic in various scientific disciplines. Time-

varying vector fields, e.g., fields that change over time, can describe the evolving surface appearance, crowd movement, and artist brushstrokes. We can extend our current framework through temporal characteristics of the vector field by sequencing basis fields and intergrading key-frame design and field deformation as in [6].

We have only touched the surface of enhancing latent fingerprints. In the future, we will design how to keep the good original regions of the image while synthesizing the noisy regions of the latent fingerprint. This will require some major modifications to our framework, and therefore, we leave it as future work.

In summary, we have exhibited an approach for guiding texture synthesis using PatchMatch. We demonstrated control over placement of singularities and alignment of texture features using N-RoSy fields. Finally, many possible extensions exist for Orientation Guided PatchMatch, ranging from continuing our work in latent fingerprint enhancement to time-varying vector fields.

REFERENCES

REFERENCES

[1] Michael Ashikhmin. Synthesizing natural textures. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, pages 217–226, New York, NY, USA, 2001. ACM.

[2] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. Patchmatch: a randomized correspondence algorithm for structural image editing. In *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, pages 24:1–24:11, New York, NY, USA, 2009. ACM.

[3] Connelly Barnes, Eli Shechtman, Dan B. Goldman, and Adam Finkelstein. The generalized patchmatch correspondence algorithm. In *Proceedings of the 11th European conference on computer vision conference on Computer vision: Part III*, ECCV'10, pages 29–43, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 263–270, New York, NY, USA, 1993. ACM.

[5] R. Cappelli, A. Lumini, D. Maio, and D. Maltoni. Fingerprint image reconstruction from standard templates. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(9):1489–1503, September 2007.

[6] Guoning Chen, V. Kwatra, Li-Yi Wei, C.D. Hansen, and E. Zhang. Design of 2d time-varying vector fields. *Visualization and Computer Graphics, IEEE Transactions on*, 18(10):1717–1730, 2012.

[7] George R. Cross and Anil K. Jain. Markov random field texture models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-5(1):25–39, 1983.

[8] Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of the 24th annual conference on Computer graphics*

and interactive techniques, SIGGRAPH '97, pages 361–368, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[9] Mathieu Desbrun, Eva Kanso, and Yiying Tong. Discrete differential forms for computational modeling. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, pages 39–54, New York, NY, USA, 2006. ACM.

[10] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[11] A.A. Efros and T.K. Leung. Texture synthesis by non-parametric sampling. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1033–1038 vol.2, 1999.

[12] 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*, SIGGRAPH '01, pages 341–346, New York, NY, USA, 2001. ACM.

[13] Christian Eisenacher, Chuck Tappan, Brent Burley, Daniel Teece, and Arthur Shek. Example-based texture synthesis on disney's tangled. In *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10, pages 32:1–32:1, New York, NY, USA, 2010. ACM.

[14] Jianjiang Feng and Anil K. Jain. Fingerprint reconstruction: From minutiae to phase. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(2):209–223, 2011.

[15] Jianjiang Feng, Jie Zhou, and Anil K. Jain. Orientation field estimation for latent fingerprint enhancement. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(4):925–940, 2013.

[16] Matthew Fisher, Peter Schröder, Mathieu Desbrun, and Hugues Hoppe. Design of tangent vector fields. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[17] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 229–238, New York, NY, USA, 1995. ACM.

[18] A.K. Jain, Jianjiang Feng, A. Nagar, and K. Nandakumar. On matching latent fingerprints. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, 2008.

[19] Anil K. Jain and Jianjiang Feng. Latent fingerprint matching. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):88–100, 2011.

[20] Vladimir G. Kim, Yaron Lipman, and Thomas Funkhouser. Symmetry-guided texture synthesis and manipulation. *ACM Trans. Graph.*, 31(3):22:1–22:14, June 2012.

[21] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Trans. Graph.*, 24(3):795–802, July 2005.

[22] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: image and video synthesis using graph cuts. *ACM Trans. Graph.*, 22(3):277–286, July 2003.

[23] Yu-Kun Lai, Miao Jin, Xuexiang Xie, Ying He, J. Palacios, E. Zhang, Shi-Min Hu, and Xianfeng Gu. Metric-driven rosy field design and remeshing. *Visualization and Computer Graphics, IEEE Transactions on*, 16(1):95–108, 2010.

[24] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 777–786, New York, NY, USA, 2005. ACM.

[25] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 541–548, New York, NY, USA, 2006. ACM.

[26] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 20(3):127–150, July 2001.

[27] Wen-Chieh Lin, James Hays, Chenyu Wu, Vivek Kwatra, and Yanxi Liu. Quantitative evaluation of near regular texture synthesis algorithms. In *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–434, 2006.

[28] Beibei Liu, Yanlin Weng, Jiannan Wang, and Yiying Tong. Orientation field guided texture synthesis. *Journal of Computer Science and Technology (CVM2013)*, 2013.

[29] J. M. Ogden, E. H. Adelson, J R. Bergen, and P.J. Burt. Pyramid-based computer graphics, 1985.

[30] Rupert Paget and Dennis Longsta. Texture synthesis via a non-parametric markov random field. In *Proceedings of DICTA–95, Digital Image Computing: Techniques and Applications, Anthony Maeder and Brian*, page pp., 1995.

[31] J. Palacios and E. Zhang. Interactive visualization of rotational symmetry fields on surfaces. *Visualization and Computer Graphics, IEEE Transactions on*, 17(7):947–955, 2011.

[32] Jonathan Palacios and Eugene Zhang. Rotational symmetry field design on surfaces. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[33] Nicolas Ray, Bruno Vallet, Wan Chiu Li, and Bruno Lévy. N-symmetry direction field design. *ACM Trans. Graph.*, 27(2):10:1–10:13, May 2008.

[34] Arun Ross, Jidnya Shah, and Anil K. Jain. From template to image: Reconstructing fingerprints from minutiae points. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(4):544–560, April 2007.

[35] Allen R. Sanderson, Chris R. Johnson, and Robert M. Kirby. Display of vector fields using a reaction-diffusion model. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 115–122, Washington, DC, USA, 2004. IEEE Computer Society.

[36] George Stockman and Linda G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[37] Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. *ACM Trans. Graph.*, 21(3):665–672, July 2002.

[38] Alan Mathison Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London*, 237:B641, 1952.

[39] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. *SIGGRAPH Comput. Graph.*, 25(4):289–298, July 1991.

[40] Greg Turk. Texture synthesis on surfaces. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 347–354, New York, NY, USA, 2001. ACM.

[41] Jarke J. van Wijk. Image based flow visualization. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 745–754, New York, NY, USA, 2002. ACM.

[42] Warren Weckesser. A reaction-diffusion cellular automaton program. Online, January 2006.

[43] 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*, SIGGRAPH '00, pages 479–488, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[44] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 355–360, New York, NY, USA, 2001. ACM.

[45] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In *Computer Graphics*, pages 299–308, 1991.

[46] Qing Wu and Yizhou Yu. Feature matching and deformation for texture synthesis. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 364–367, New York, NY, USA, 2004. ACM.

[47] Soweon Yoon, Jianjiang Feng, and A.K. Jain. Latent fingerprint enhancement via robust orientation field estimation. In *Biometrics (IJCB), 2011 International Joint Conference on*, pages 1–8, 2011.

[48] Soweon Yoon, Jianjiang Feng, and Anil K. Jain. On latent fingerprint enhancement. *Proc. SPIE 7667, Biometric Technology for Human Identification*, VII:766707, 2010.

[49] Eugene Zhang, James Hays, and Greg Turk. Interactive tensor field design and visualization on surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 13(1):94–107, January 2007.

[50] Eugene Zhang, Konstantin Mischaikow, and Greg Turk. Vector field design on surfaces. *ACM Trans. Graph.*, 25(4):1294–1326, October 2006.