TEXTURE SYNTHESIS
BY
FIXED NEIGHBORHOOD SEARCHING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Li-Yi Wei
November 2001

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Marc Levoy
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Hanrahan

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Heeger

Approved for the University Committee on Graduate Studies:

Abstract

Textures can describe a wide variety of natural phenomena with random variations over repeating patterns. Examples of textures include images, motions, and surface geometry. Since reproducing the realism of the physical world is a major goal for computer graphics, textures are important for rendering synthetic images and animations. However, because textures are so diverse it is difficult to describe and reproduce them under a common framework.

In this thesis, we present new methods for synthesizing textures. The first part of the thesis is concerned with a basic algorithm for reproducing image textures. The algorithm is easy to use and requires only a sample texture as input. It generates textures with perceived quality equal to or better than those produced by previous techniques, but runs two orders of magnitude faster. The algorithm is derived from Markov Random Field texture models and generates textures through a deterministic searching process. Because of the use of this deterministic searching, our algorithm can avoid the computational demand of probability sampling and can be directly accelerated by a point searching algorithm such as tree-structured vector quantization.

The second part of the thesis concerns various extensions and applications of our texture synthesis algorithm. Texture synthesis can be used to remove undesirable artifacts in photographs and films such as scratches, wires, pops, or scrambled regions. We extend our algorithm for this purpose by replacing artifacts with textured backgrounds via constrained synthesis. In addition to 2D images, textures can also be used to model other physical phenomena such as 3D temporal textures such as fire, smoke, and ocean waves, as well as 1D articulated motion signals such as walking and running. Despite the diversity of the dimensionality and generation process of these textures, out algorithm is capable of modeling and

generating them under a common framework.

Texture mapping has become a ubiquitous tool for realistic image synthesis. However, it remains difficult to map image textures onto general manifold surfaces. Although algorithms exist for synthesizing a wide variety of textures over rectangular domains, it remains difficult to synthesize general textures over arbitrary manifold surfaces. In the third part of this thesis, we present a solution to this problem for surfaces defined by dense polygon meshes. Our solution extends our basic algorithm by generalizing the definition of search neighborhoods. For each mesh vertex, we establish a local parameterization surrounding the vertex, use this parameterization to create a small rectangular neighborhood with the vertex at its center, and search a sample texture for similar neighborhoods. Our algorithm requires as input only a sample texture and a target model. Notably, it does not require specification of a global tangent vector field; it computes one as it goes - either randomly or via a relaxation process. Despite this, the synthesized texture contains no discontinuities, exhibits low distortion, and is perceived to be similar to the sample texture. We demonstrate that our solution is robust and is applicable to a wide range of textures.

Most existing texture synthesis algorithms take a single texture as input and generate an output texture with similar visual appearance. Although the output texture can be made of arbitrary size and duration, those techniques can at best replicate the characteristics of the input texture. In the fourth part of this thesis, we present a new method that can create textures in interesting ways in addition to mimic existing ones. The algorithm takes multiple textures with probably different characteristics, and synthesizes new textures with combined visual appearance of all the inputs. We present two important applications of multiple-source synthesis: generating texture mixtures and solid textures from multiple 2D views.

In the fifth part of the thesis, we provide designs and extensions to target our algorithm for real-time graphics hardwares and applications. Unlike certain procedural texture synthesis algorithms which can evaluate each texel independently on the fly, our algorithm requires texels to be computed sequentially in order to maintain the consistency of the synthesis results. This limits the feasibility for applying our algorithm for real-time applications. We address this issue by presenting a new method that allows texels to be computed in any order while guarantees the invariance of the results, thus making it useful for real-time applications. We also present possible hardware designs for a real-time texture generator so that it can replace the traditional texture mapping hardwares.

In the last part of the thesis, we analyze our algorithm behavior and discuss potential future work.

Acknowledgements

Over the past several years I have been lucky enough to be a member of the best computer graphics lab on this planet. I have interacted most frequently with my colleagues Ravi Ramamoorthi, James Davis, Milton Chen, Szymon Rusinkiewicz, Ziyad Hakura, Kekoa Proudfoot, Bennett Wilburn, Li-Wei He (note: we are not the same person), Cindy Chen, our system guru John Gerth, and many others. I thank them for making the lab the more exciting and inspirational place, and I benefit greatly through the interactions with them.

I would also like to thank people outside Stanford who have helped me in various stages of my Ph.D. career: Kris Popat, Alyosha Efros, Aaron Hertzmann, Ken Perlin, Greg Turk, and others. I have benefited a lot from their codes, slides, thoughts, and discussions.

I would like to thank my committee, consisting of Marc Levoy, Pat Hanrahan, David Heeger, and Robert Gray, for their encouragement and inspirations about the project. Several of the key ideas in my algorithm, including image pyramids and tree-structured vector quantization, came from their courses, and I appreciate their excellent teachings that provide the necessary intuitions. I would like to thank professor Carlo Tomasi for his discussions on several computer vision issues. I would also like to thank my advisor Marc Levoy for his support during the years, for his insightful and tirelessly-long discussions we had on several projects, for his instructions on both the technical and non-technical aspects of my Ph.D. training, and for giving me the free space to explore different projects.

Finally, I would like to thank my family and friends for their support. In particular, I would like to thank my parents for their continuous support both mentally and financially, and my wife for her support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Texture is a ubiquitous experience. It can describe a variety of natural phenomena with repetition, such as sound (background noise in a machine room), motion (animal running), visual appearance (surface color and geometry), and human activities (our daily lives). S-ince reproducing the realism of the physical world is a major goal for computer graphics, textures are important for rendering synthetic images and animations. However, because textures are so diverse it is difficult to describe and reproduce them under a common framework.

In this thesis, we present new methods for synthesizing textures. The first part of the thesis is concerned with a basic algorithm for reproducing image textures. We show that limitations of traditional methods can be overcome by our approach based on search neighborhoods and tree-structured vector quantization. The rest of this thesis concerns with various extensions of the basic algorithm; the extensions concentrate on either reproducing textures of different physical phenomena such as motions, or creating textures in novel ways in addition to mimic existing ones.

This chapter is organized as follows. In Section 1.1, we pose the problem of texture synthesis. In Section 1.2, we describe some of the applications of texture synthesis. In Section 1.3, we describe the contributions of this thesis, and in Section 1.4 we outline the remainder of this thesis.

1.1 Problem Formulation

In this section, we describe the goal of texture synthesis. We begin with a brief discussion of the definition of textures.

1.1.1 What is a Texture?

Reproducing detailed surface appearance is important to achieve visual realism in computer rendered images. One way to model surface details is to use polygons or other geometric primitives. However, as details becomes finer and more complicated, explicit modeling with geometric primitives becomes less practical. An alternative is to map an image, either synthetic or digitized, onto the object surface, a technique called *texture mapping* [11, 6]. The mapped image, usually rectangular, is called a *texture map* or *texture*. A texture can be used to modulate various surface properties, including color, reflection, transparency, or displacements. In computer graphics the content of a texture can be very general; in mapping a color texture, for example, the texture can be an image containing arbitrary drawings or patterns.

Unfortunately, the meaning of texture in graphics is somehow abused from its usual meaning. The Webster's dictionary defines texture as follows:

> Texture, noun [1578]
>
> (a) something composed of closely interwoven elements; specifically a woven cloth
>
> (b) the structure formed by the threads of a fabric ...

In other words, textures are usually refered to as visual or tactile surfaces composed of repeating patterns, such as a fabric. This definition of texture is more restricted than the notion of texture in graphics. However, since a majority of natural surfaces consist of repeating elements, this narrower definition of texture is still powerful enough to describe many surface properties. This definition of texture is also widely adopted in computer vision and image processing communities.

In this thesis, we concentrate on the narrower definition of textures, i.e. images containing repeating patterns. Since natural textures may contain interesting variations or imperfections, we also allow a certain amount of randomness over the repeating patterns. For

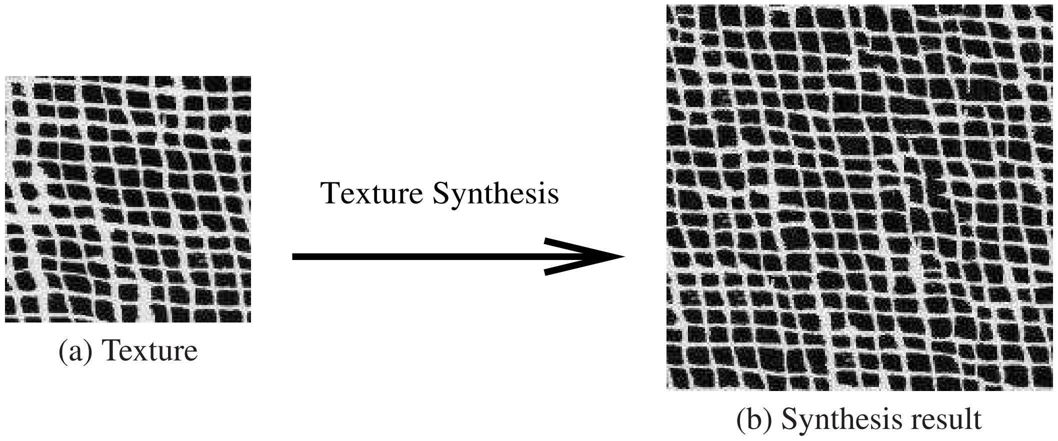


Figure 1.1: *Problem Formulation. Given a sample texture (a), our goal is to synthesize a new texture that looks like the input (b). The synthesized texture is tileable can be of arbitrary size specified by the user.*

example, a honeycomb texture is composed of hexagonal cells with slight variations of size and shape of each cell. The amount of randomness can vary for different textures, from s-tochastic (a sandbeach) to purely deterministic (a tiled floor). This definition of textures allows us to model textures under a unified framework. We also attempt to generalize the notion of textures beyond images to incorporate other physical phenomena such as animations and articulated motions.

1.1.2 What is Texture Synthesis?

Computer graphics applications often use textures to render synthetic images. These textures can be obtained from a variety of sources such as hand-drawn pictures or scanned photographs. Hand-drawn pictures can be aesthetically pleasing, but it is hard to make them photo-realistic. Most scanned images, however, are of inadequate size and can lead to visible seams or repetition if they are directly used for texture mapping.

Texture synthesis is an alternative way to create textures. Because synthetic textures can be made any size, visual repetition is avoided. Texture synthesis can also produce tileable images by properly handling the boundary conditions.

The goal of texture synthesis can be stated as follows: Given a texture sample, synthesize a new texture that, when perceived by a human observer, appears to be generated by the same underlying process (Figure 1.1). The major challenges are:

Modeling How to estimate the texture generation process from a given finite texture sample. The estimated process should be able to model both the structural and stochastic parts of the input texture. The success of modeling is determined by the visual fidelity of the synthesized textures with respect to the given samples.

Sampling How to develop an efficient sampling procedure to produce new textures from a given model. The efficiency of the sampling procedure will directly determine the computational cost of texture generation.

In this thesis, we present a very simple algorithm that can efficiently synthesize a wide variety of textures. We model textures by a set of spatial neighborhoods, and synthesize textures using a search procedure based on neighborhoods. We show that by proper acceleration, this search procedure can be executed in near real time. We also show the versatility of this approach through a series of generalizations and extensions.

1.2 Applications

Texture synthesis can be useful in a lot of applications in computer graphics, image processing, and computer vision.

1.2.1 Rendering

In rendering, textures can mimic the surface details of real objects, ranging from varying the surface's color, perturbing the surface normals (bump mapping), to actually deforming the surface geometry (displacement mapping). In pen and ink style illustrations, textures (hatches) can delineate the tone, shade, and pattern of objects [76, 77].

1.2.2 Animation

Computer generated animations often contain scripted events and random motions. Scripted events are non-repetitive actions such as opening a door or picking up an object, and are usually rendered under direct control. On the contrary, random motions are repetitive background movements such as ocean waves, rising smoke, or a burning fire. These kind of motions have indeterminate extent both in space and time, and are often refered as *temporal textures* [66]. These temporal textures are often difficult to render using traditional techniques based on physical modeling, since different textures are often generated by very different underlying physical processes. By treating them as textures, we can model and synthesize them using a single texture synthesis algorithm.

In addition to temporal textures, certain motions such as joint angles of articulated motions, could also be modeled as one dimensional textures. These textures can be synthesized on the fly to simulate delicate motions such as eye blinking or human walking.

1.2.3 Compression

Images depicting natural scenes often contain large textures regions, such as a grass land, a forest, or a sand beach. Because textures often contain significant high frequency information, they are not well compressed by transform-based techniques such as JPEG. By segmenting out these textured regions in a preprocessing step, they might be compressible by a texture synthesis technique. In addition to image compression, texture synthesis can also be employed for synthetic scenes containing large amounts of textures [4].

1.2.4 Restoration and Editing

Photographs, films and images often contain regions that are in some sense flawed. A flaw can be a scrambled region on a scanned photograph, scratches on an old film, wires or props in a movie film frame, or simply an undesirable object in an image. Since the processes causing these flaws are often irreversible, an algorithm that can fix these flaws is desirable. Often, the flawed portion is contained within a region of texture, and can be replaced by texture synthesis [19, 38].

1.2.5 Computer Vision

Several computer vision tasks use textures, such as segmentation, recognition, and classification. These tasks can benefit from a texture model, which could be derived from a successful texture synthesis algorithm.

1.3 Contributions

This thesis has two contributions. First, we present a new algorithm for synthesizing image textures. We show that this new method has several advantages over previous techniques:

Quality Textures generated by our approach has high visual quality; they are perceived to be very similar to the input sample textures. They are also tileable and can be of arbitrary size.

Generality Our algorithm can model a wide variety of textures, despite the versatility of their underlying physical generation process.

Simplicity Our approach is very simple and can be implemented using standard image processing operations.

Efficiency Unlike previous approaches, our algorithm is efficient. Typical textures take only seconds to minutes to generate.

Second, we present extensions and generalizations of the basic synthesis algorithm, as follows:

Constrained Texture Synthesis We modify our basic synthesis algorithm for image editing and restoration. The modified algorithm can remove flaws in a textured region by replacing them with a synthesized texture. The synthesized texture looks like the surrounding texture, and the boundaries between the new and old regions are invisible.

Temporal Texture Synthesis We generalize the notion of textures to 3D spatial-temporal volumes, refered to as temporal textures. We show that our technique can model different temporal textures such as fire, smoke, and ocean waves.

Surface Texture Synthesis Texture mapping often cause distortion and discontinuity over mapped surfaces. We address this problem by synthesizing textures directly over object surfaces. Our algorithm can grow a wide variety of textures over arbitrary manifold surfaces with minimum distortion and no discontinuity.

Multiple Source Texture Synthesis Most existing texture synthesis algorithms produce each new texture from a single source. Although useful in many applications, such techniques can at best mimic the characteristics of existing textures. A more interesting approach is to create new textures that do not previously exist. We achieve this by modifying our algorithm so that it can generate a new texture from multiple sources. We demonstrate two important applications of this new algorithm: generating solid textures from multiple 2D views (usually orthogonal) and producing texture mixtures that possess characteristics of several textures.

Real-time Texture Synthesis Unlike certain procedural texture synthesis algorithms which can evaluate each texel independently on the fly, our algorithm requires texels to be computed sequentially in order to maintain the consistency of the synthesis results. This limits the feasibility for applying our algorithm for real-time applications. We address this issue by presenting a new method that allows texels to be computed in any order while guarantees the invariance of the results, thus making it useful for real-time applications.

1.4 Organization

The rest of the thesis is organized as follows. In Chapter 2, we present our algorithm for synthesizing image textures. In Chapter 3, we accelerate our algorithm using tree-structured vector quantization. In the rest of the thesis we describe our extensions and generalizations. In Chapter 4, we develop our constrained synthesis technique for image editing. In Chapter 5, we generalize our algorithm for synthesizing temporal textures and articulated motion signals. In Chapter 6, we describe a technique for developing textures directly over manifold surfaces. In Chapter 7 we demonstrate how textures can be generated from multiple sources, and discuss two variations of the algorithm for generating texture mixtures and solid textures from multiple planar views. In Chapter 8 we present new methods that allow textures to be generated in any order while guarantee the invariance of the results, and we propose possible hardware designs for a texture generator. In Chapter 9 we analyze the algorithm behavior, and in Chapter 10 we conclude this thesis and describe future work.

Chapter 2

Image Texture Synthesis

Image texture synthesis has been an active area of research for many years. In computer vision, texture synthesis has been used to verity texture models for various tasks such as texture segmentation and classification. In computer graphics and image processing, texture synthesis has applications for rendering, compression, and image editing.

In this chapter, we present a very simple algorithm that can efficiently synthesize a wide variety of textures. The inputs consist of an example texture patch and a random noise image with size specified by the user (Figure 2.1). The algorithm modifies this random noise to make it look like the given example. This technique is flexible and easy to use, since only an example texture patch (usually a photograph) is required. New textures can be generated with little computation time, and their tileability is guaranteed. The algorithm is also easy to implement; the two major components are a multiresolution pyramid and a simple searching algorithm.

The rest of this chapter is organized as follows. In Section 2.1, we review previous work on image texture synthesis. In Section 2.2, we describe our algorithm. In Section 2.3, we demonstrate synthesis results and compare them with those generated by previous approaches.

2.1 Previous Work

Numerous approaches have been proposed for texture analysis and synthesis. In this section, we briefly review some recent and representative works. We refer the reader to [32, 71, 57, 62, 39] for more complete surveys.

2.1.1 Physical Simulation

One way to synthesize image textures is to directly simulate their physical generation processes. Biological patterns such as fur, scales, and skin can be modeled using reaction diffusion [78] and cellular texturing [81]. Some weathering and mineral phenomena can be faithfully reproduced by detailed simulations [16]. These techniques can produce textures directly on 3D meshes so the texture mapping distortion problem is avoided. However, different textures are usually generated by very different physical processes so these approaches are applicable to only limited classes of textures.

2.1.2 Markov Random Field and Gibbs Sampling

Many algorithms model textures by Markov Random Fields (or in a different mathematical form, Gibbs Sampling), and generate textures by probability sampling [19, 83, 53, 48]. Since Markov Random Fields have been proven to be a good approximation for a broad range of textures, these algorithms are general and some of them produce good results. A drawback of Markov Random Field sampling, though, is that it is computationally expensive: even small texture patches can take hours or days to generate.

2.1.3 Feature Matching

Some algorithms model textures as a set of features, and generate new images by matching the features in an example texture [33, 13, 59]. These algorithms are usually more efficient than Markov Random Field algorithms. Heeger and Bergen [33] model textures by matching marginal histograms of image pyramids. Their technique succeeds on highly stochastic textures but fails on more structured ones. De Bonet [13] synthesizes new images by randomizing an input texture sample while preserving the cross-scale dependencies.

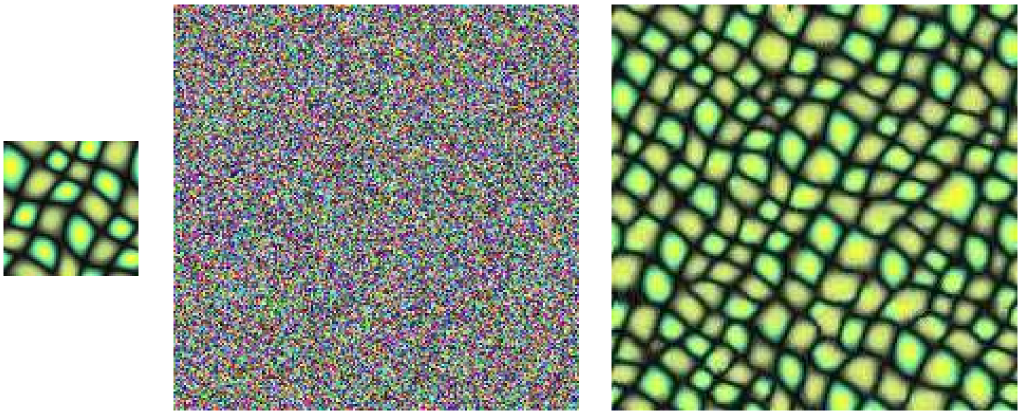


Figure 2.1: *Image Texture Synthesis. Our texture generation process takes an example texture patch (left) and a random noise (middle) as input, and modifies this random noise to make it look like the given example texture. The synthesized texture (right) can be of arbitrary size, and is perceived as very similar to the given example. Using our algorithm, textures can be generated within seconds, and the synthesized results are always tileable.*

This method works better than [33] on structured textures, but it can produce boundary artifacts if the input texture is not tileable. Simoncelli and Portilla [59] generate textures by matching the joint statistics of the image pyramids. Their method can successfully capture global textural structures but fails to preserve local patterns.

2.2 Algorithm

Our goal was to develop an algorithm that combines the advantages of previous approaches. We want it to be efficient, general, and able to produce high quality, tileable textures. It should also be user friendly; i.e., the number of tunable input parameters should be minimal. This can be achieved by a careful selection of the texture modeling and synthesis procedure. For the texture model, we use Markov Random Fields (MRF) since they have been proven to cover the widest variety of useful texture types. To avoid the usual computational expense of MRFs, we have developed a synthesis procedure which avoids explicit probability construction and sampling.

Markov Random Field methods model a texture as a realization of a *local* and *stationary* random process. That is, each pixel of a texture image is characterized by a small set

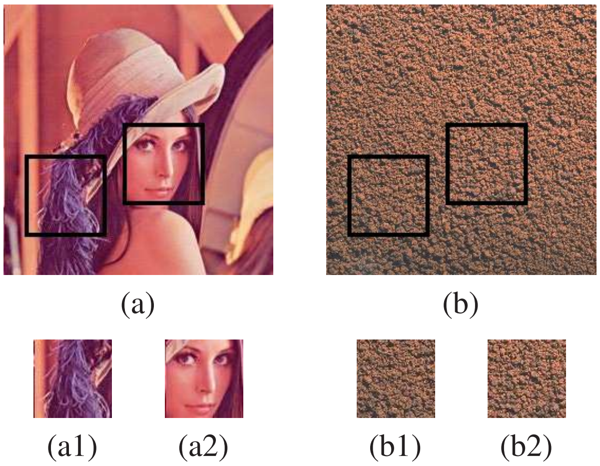


Figure 2.2: *How textures differ from images. (a) is a general image while (b) is a texture. A movable window with two different positions are drawn as black squares in (a) and (b), with the corresponding contents shown below. Different regions of a texture are always perceived to be similar (b1,b2), which is not the case for a general image (a1,a2). In addition, each pixel in (b) is only related to a small set of neighboring pixels. These two characteristics are called stationarity and locality, respectively.*

of spatially neighboring pixels, and this characterization is the same for all pixels. The intuition behind this model can be demonstrated by the following experiment (Figure 2.2). Imagine that a viewer is given an image, but only allowed to observe it through a small movable window. As the window is moved the viewer can observe different parts of the image. The image is stationary if, under a proper window size, the observable portion always appears similar. The image is local if each pixel is predictable from a small set of neighboring pixels and is independent of the rest of the image.

Based on these locality and stationarity assumptions, our algorithm synthesizes a new texture so that it is locally similar to an example texture patch. The new texture is generated pixel by pixel, and each pixel is determined so that local similarity is preserved between the example texture and the result image. This synthesis procedure, unlike most MRF based algorithms, is completely deterministic and no explicit probability distribution is constructed. As a result, it is efficient and amenable to further acceleration.

In the rest of this section, we first describe how the algorithm works in a single resolution. We then extend it using a multiresolution pyramid to obtain improvements in

Symbol	Meaning
I_a	Input texture sample
I_s	Output texture image
G_a	Gaussian pyramid built from I_a
G_s	Gaussian pyramid built from I_s
p_i	An input pixel in I_a or G_a
p	An output pixel in I_s or G_s
$N(p)$	Neighborhood around the pixel p
$G(L)$	Lth level of pyramid G
$G(L, x, y)$	Pixel at level L and position (x, y) of G
{RxC,k}	(2D) neighborhood containing k levels, with size RxC at the top level

Table 2.1: *Table of symbols.*

efficiency. For easy reference, we list the symbols used in Table 2.1 and summarize the algorithm in Table 2.2.

2.2.1 Single-resolution Algorithm

The algorithm starts with an input texture sample I_a and a white random noise I_s. We force the random noise I_s to look like I_a by transforming I_s pixel by pixel in a raster scan ordering, i.e. from top to bottom and left to right. Figure 2.3 shows a graphical illustration of the synthesis process.

To determine the pixel value p at I_s, its spatial neighborhood $N(p)$ (the L-shaped regions in Figure 2.3) is compared against all possible neighborhoods $N(p_i)$ from I_a. The input pixel p_i with the most similar $N(p_i)$ is assigned to p. We use a simple L_2 norm (sum of squared difference) to measure the similarity between the neighborhoods. The goal of this synthesis process is to ensure that the newly assigned pixel p will maintain as much local similarity between I_a and I_s as possible. The same process is repeated for each output pixel until all the pixels are determined. This is akin to putting together a jigsaw puzzle: the pieces are the individual pixels and the fitness between these pieces is determined by the colors of the surrounding neighborhood pixels.

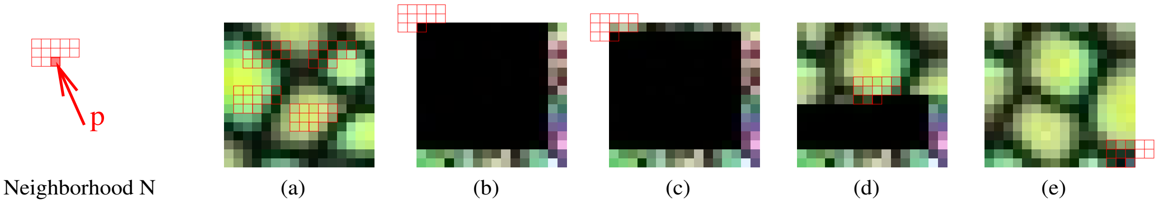


Figure 2.3: *Single resolution texture synthesis. (a) is the input texture and (b)-(e) show different synthesis stages of the output image. Pixels in the output image are assigned in a raster scan ordering. The value of each output pixel p is determined by comparing its spatial neighborhood $N(p)$ with all neighborhoods in the input texture. The input pixel with the most similar neighborhood will be assigned to the corresponding output pixel. Neighborhoods crossing the output image boundaries (shown in (b) and (e)) are handled toroidally, as discussed in Section 2.2.4. Although the output image starts as a random noise, only the last few rows and columns of the noise are actually used. For clarity, we present the unused noise pixels as black. (b) synthesizing the first pixel, (c) synthesizing the first pixel of the second row, (d) synthesizing the middle pixel, (e) synthesizing the last pixel.*

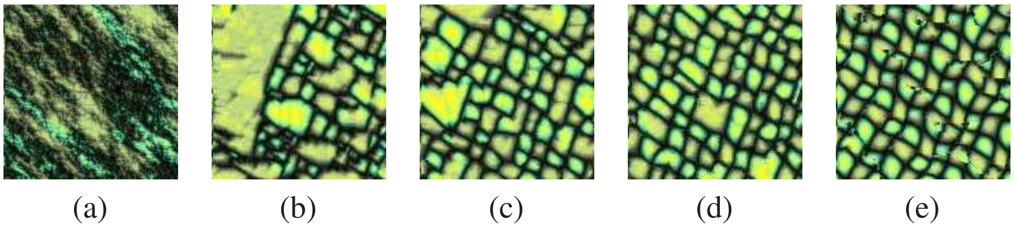


Figure 2.4: *Synthesis results with different neighborhood sizes. The neighborhood sizes are (a) 1x1, (b) 5x5, (c) 7x7, (d) 9x9, (e) 30x30, respectively. All images shown are of size 192x192. Note that as the neighborhood size increases the resulting texture quality gets better. However, the computation cost also increases.*

2.2.2 Neighborhood

Because the set of local neighborhoods $N(p_i)$ is used as the primary model for textures, the quality of the synthesized results will depend on its size and shape. Intuitively, the size of the neighborhoods should be on the scale of the largest regular texture structure; otherwise this structure may be lost and the result image will look too random. Figure 2.4 demonstrates the effect of the neighborhood size on the synthesis results.

The shape of the neighborhood will directly determine the quality of I_s. It must be causal, i.e. the neighborhood can only contain those pixels preceding the current output pixel in the raster scan ordering. The reason is to ensure that each output neighborhood

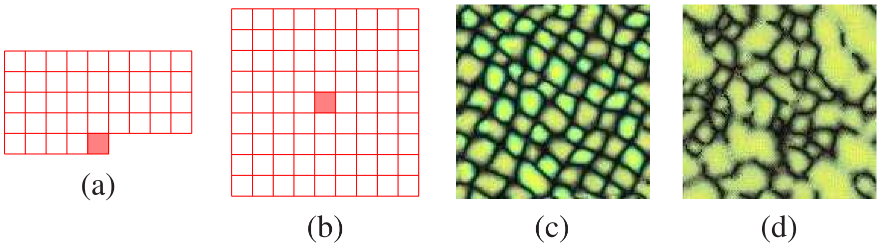


Figure 2.5: *Causality of the neighborhood. (a) A causal neighborhood (b) A noncausal neighborhood (c) synthesis result using the neighborhood in (a) (d) synthesis result using the neighborhood in (b). Both (c) and (d) are generated from the same random noise. As shown, a noncausal neighborhood is unable to generate valid results.*

$N(p)$ will include only already assigned pixels. For the first few rows and columns of I_s, $N(p)$ may contain unassigned (noise) pixels but as the algorithm progresses all the other $N(p)$ will be completely "valid" (containing only already assigned pixels). A noncausal $N(p)$, which always includes unassigned pixels, is unable to transform I_s to look like I_a (Figure 2.5). Thus, the noise image is only used when generating the first few rows and columns of the output image. After this, it is ignored.

2.2.3 Multi-resolution Algorithm

The single resolution algorithm captures the texture structures by using adequately sized neighborhoods. However, for textures containing large scale structures we have to use large neighborhoods, and large neighborhoods demand more computation. This problem can be solved by using a multiresolution image pyramid [10]; computation is saved because we can represent large scale structures more compactly by a few pixels in a certain lower resolution pyramid level.

The multiresolution synthesis algorithm proceeds as follows. Two Gaussian pyramids, G_a and G_s, are first built from I_a and I_s, respectively. The algorithm then transforms G_s from lower to higher resolutions, such that each higher resolution level is constructed from the already synthesized lower resolution levels. This is similar to the sequence in which a picture is painted: long and thick strokes are placed first, and details are then added. Within each output pyramid level $G_s(L)$, the pixels are synthesized in a way similar to the single resolution case where the pixels are assigned in a raster scan ordering. The only modification is that for the multiresoltion case, each neighborhood $N(p)$ contains pixels in the current resolution as well as those in the lower resolutions. The similarity between two multiresolution neighborhoods is measured by computing the sum of the squared distance of all pixels within them. These lower resolution pixels constrain the synthesis process so that the added high frequency details will be consistent with the already synthesized low frequency structures.

An example of a multiresolution neighborhood is shown in Figure 2.6. It consists of two levels, with sizes 5x5 and 3x3, respectively. Within a neighborhood, we choose the sizes of the lower levels so that they are about half the sizes of the previous higher resolution levels. For clarity, we use the symbol {RxC,k} to indicate multiresolution neighborhoods which contain k levels with size RxC at the top level.

Figure 2.7 shows results of multiresolution synthesis with different numbers of pyramid levels. Note that Figure 2.7 (c), although synthesized with a small {5x5,2} multiresolution neighborhood, looks comparable with Figure 2.4 (c), which was generated with a larger 9x9 single resolution neighborhood. This demonstrates a major advantage of multiresolution synthesis: moderately small neighborhoods can be used without sacrificing synthesis qualities.

2.2.4 Edge Handling

Proper edge handling for $N(p)$ near the image boundaries is very important. For the synthesis pyramid the edge is treated toroidally. In other words, if $G_s(L, x, y)$ denotes the pixel at level L and position (x, y) of pyramid G_s, then $G_s(L, x, y) \equiv G_s(L,\ x \ mod\ M,\ y \ mod\ N)$, where M and N are the number of rows and columns, respectively, of $G_s(L)$. Handling edges toroidally is essential to guarantee that the resulting synthetic texture will tile seamlessly. [1]

For the input pyramid G_a, toroidal neighborhoods typically contain discontinuities unless I_a is tileable. A reasonable edge handler for G_a is to pad it with a reflected copy of

[1]The multiresolution algorithm is also essential for tileability if a causal neighborhood is used. Since a single resolution causal neighborhood $N(p)$ contains only pixels above p in scanline order, the vertical tileability may not be enforced. A multiresolution neighborhood, which contains symmetric regions at lower resolution levels, avoids this problem.

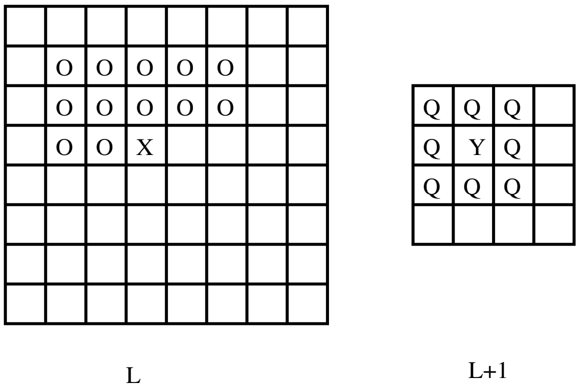


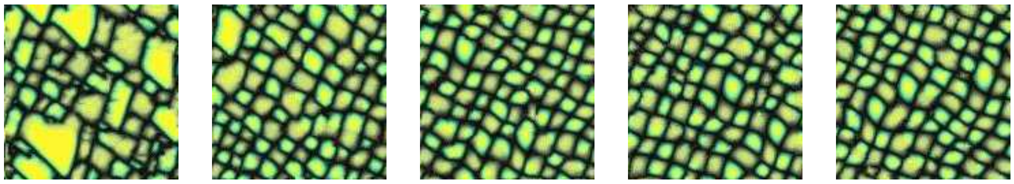
Figure 2.6: *A causal multiresolution neighborhood with size {5x5,2}. The current level of the pyramid is shown at left and the next lower resolution level is shown at right. The current output pixel* p*, marked as X, is located at* (L, x, y)*, where* L *is the current level number and* (x, y) *is its coordinate. At this level* L *of the pyramid the image is only partially complete. Thus, we must use the preceding pixels in the raster scan ordering (marked as O). The position of the parent of the current pixel, located at* $(L+1, \frac{x}{2}, \frac{y}{2})$*, is marked as Y. Since the parent's level is complete, the neighborhood can contain pixels around Y, marked by Q. When searching for a match for pixel X, the neighborhood vector is constructed that includes the O's, Q's, and Y, in scanline order.*

itself. Another solution is to use only those $N(p_i)$ completely inside G_a, and discard those crossing the boundaries. Because a reflective edge handler may introduce discontinuities in the derivative, we adopt the second solution which uses only interior blocks.

2.2.5 Initialization

Natural textures often contain recognizable structures as well as a certain amount of randomness. Since our goal is to reproduce realistic textures, it is essential that the algorithm capture the random aspect of the textures. This notion of randomness can sometimes be achieved by entropy maximization [83], but the computational cost is prohibitive. Instead, we initialize the output image I_s as a white random noise, and gradually modify this noise to look like the input texture I_a. This initialization step seeds the algorithm with sufficient entropy, and lets the rest of the synthesis process focus on the transformation of I_s towards I_a. To make this random noise a better initial guess, we also equalize the pyramid histogram of G_s with respect to G_a [33].

The initial noise affects the synthesis process in the following way. For the single



(a) (b) (c) (d) (e)

Figure 2.7: *Multiresolution synthesis with different number of pyramid levels. (a) 1 level, (b) 2 levels, (c) 3 levels, (d) 4 levels, (e) 5 levels. Except for the lowest resolution, which is synthesized with a 5x5 single resolution neighborhood, each pyramid level is synthesized using the multiresolution neighborhood shown in Figure 2.6. Note that as the number of pyramid levels increases, the image quality improves.*

resolution case, neighborhoods in the first few rows and columns of I_s contain noise pixels. These noise pixels introduce uncertainty in the neighborhood matching process, causing the boundary pixels to be assigned semi-stochastically (However, the searching process is still deterministic. The randomness is caused by the initial noise). The rest of the noise pixels are overwritten directly during synthesis. For the multiresolution case, however, more of the noise pixels contribute to the synthesis process, at least indirectly, since they determine the initial value of the lowest resolution level of G_s.

2.2.6 Summary

We summarize the algorithm as pseudocode in Table 2.2.

The architecture of this algorithm is flexible; it is composed from several orthogonal components. We list these components as follows and discuss the corresponding design choices.

Pyramid: The pyramids are built from and reconstructed to images using the standard routines **BuildImagePyramid** and **ReconImagePyramid**. Various pyramids can be used for texture synthesis; examples are Gaussian pyramids [53], Laplacian pyramids [33], steerable pyramids [33, 59], and feature-based pyramids [13]. A Gaussian pyramid, for example, is built by successive filtering and downsampling operations, and each pyramid

function $I_s \leftarrow$ ImageTextureSynthesis(I_a, I_s)
1 InitializeColors(I_s);
2 $G_a \leftarrow$ BuildImagePyramid(I_a);
3 $G_s \leftarrow$ BuildImagePyramid(I_s);
4 **foreach** level L from lower to higher resolutions of G_s
5   **loop** through all pixels p of $G_s(L)$
6     $C \leftarrow$ FindBestMatch(G_a, G_s, L, p);
7     $G_s(L, p) \leftarrow C$;
8 $I_s \leftarrow$ ReconImagePyramid(G_s);
9 **return** I_s;

function $C \leftarrow$ FindBestMatch(G_a, G_s, L, p)
10 $N_s \leftarrow$ BuildImageNeighborhood(G_s, L, p);
11 $N_a^{best} \leftarrow$ null; $C \leftarrow$ null;
12 **loop** through all pixels p_i of $G_a(L)$
13   $N_a \leftarrow$ BuildImageNeighborhood(G_a, L, p_i);
14   **if** Match(N_a, N_s) > Match(N_a^{best}, N_s)
15     $N_a^{best} \leftarrow N_a$; $C \leftarrow G_a(L, p_i)$;
16 **return** C;

Table 2.2: *Pseudocode of the algorithm.*

level, except for the highest resolution, is a blurred and decimated version of the original image. Reconstruction of Gaussian pyramids is trivial, since the image is available at the highest resolution pyramid level. These different pyramids give different trade-offs between spatial and frequency resolutions. In this chapter, we choose to use the Gaussian pyramid for its simplicity and greater spatial localization (a detailed discussion of this issue can be found in [52]). However, other kinds of pyramids can be used instead.

Neighborhood: The neighborhood can have arbitrary size and shape; the only requirement is that it contains only valid pixels. A noncausal/symmetric neighborhood, for example, can be used by extending the original algorithm with two passes (Chapter 4).

Synthesis Ordering: A raster scan ordering is used in line 5 of the function **TextureSynthesis**. This, however, can also be extended. For example, a spiral ordering can be used for constrained texture synthesis (Chapter 4), and a random order can be used for synthesizing textures over irregular meshes (Chapter 6). The synthesis ordering should cooperate with the **BuildImageNeighborhood** so that the output neighborhoods contain only valid pixels.

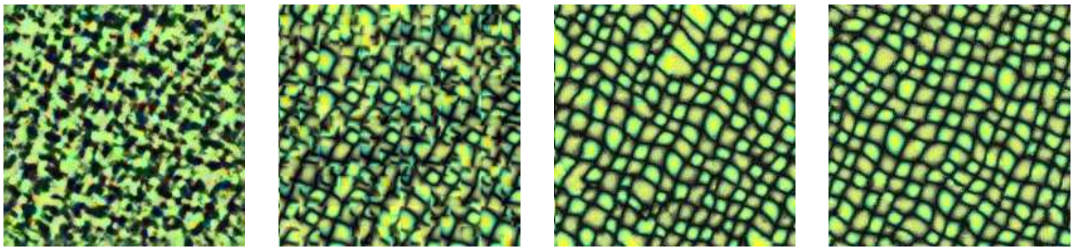
Searching: An exhaustive searching procedure **FindBestMatch** is employed to determine the output pixel values. Because this is a standard process, various point searching algorithms can be used for acceleration. This will be discussed in detail in Chapter 3.

2.3 Synthesis Results

To test the effectiveness of our approach, we have run the algorithm on many different images from standard texture sets. Figure 2.10 and Figure 2.11 show examples using the Brodatz texture album [8] and the MIT VisTex set [45], respectively. The Brodatz album is the most commonly used texture testing set and contains a broad range of grayscale images. Since most graphics applications require color textures, we also use the MIT VisTex set, which contains real world textures photographed under natural lighting conditions. All result textures are generated using a 4-level Gaussian pyramid, with neighborhood sizes {3x3,1}, {5x5,2}, {7x7,2}, {9x9,2}, respectively, from lower to higher resolutions. Additional texture synthesis results are available on our project website[2].

A visual comparison of our approach with several other algorithms is shown in Figure 2.8. Result (a) is generated by Heeger and Bergen's algorithm [33] using a steerable pyramid with 6 orientations. The algorithm captures certain random aspects of the texture but fails on the dominating grid-like structures. Result (b) is generated by De Bonet's approach [13] where we choose his randomness parameter to make the result look best. Though capable of capturing more structural patterns than (a), certain boundary artifacts are visible. This is because his approach characterizes textures by lower frequency pyramid levels only; therefore the lateral relationship between pixels at the same level is lost. Result (c) is generated by Efros and Leung's algorithm [19]. This technique is based on the Markov Random Field model and is capable of generating high quality textures. However,

[2] http://graphics.stanford.edu/projects/texture/



(a) (b) (c) (d)

Figure 2.8: *A comparison of texture synthesis results using different algorithms. (a) Heeger and Bergen's method [33] (b) De Bonet's method [13] (c) Efros and Leung's method [19] (d) Our method. Only Efros and Leung's algorithm produces results comparable with ours. However, our algorithm is two orders of magnitude faster than theirs after acceleration (Chapter 3). The sample texture patch has size 64x64, and all the result images are of size 192x192. A 9x9 neighborhood is used for (c), and (d) is synthesized using the same parameters as in Figure 2.11.*

a direct application of their approach can produce non-tileable results.[3]

Result (d) is synthesized using our approach. It is tileable and the image quality is comparable with those synthesized directly from MRFs. It took about 8 minutes to generate using a 195 MHz R10000 processor. However, this is not the maximum possible speed achievable with this algorithm. In the next chapter, we describe modifications that accelerate the algorithm greatly.

2.4 Discussion

Our algorithm presented in this chapter relates to an earlier work by Popat and Picard [53] in that a causal neighborhood and raster scan ordering are used for texture synthesis. However, instead of constructing explicit probability models, our algorithm uses deterministic searching. This approach shares the simplicity of Efros and Leung [19], but uses fix-sized neighborhoods which allow TSVQ acceleration. The fact that such a simple approach works well on many different textures implies that there may be computational redundancies in other texture synthesis techniques.

Although our algorithm is able to model a wide variety of textures, it still has several

[3]Though not stated in the original paper [19], we have found that it is possible to extend their approach using multiresolution pyramids and a toroidal neighborhood to make tileable textures.

Figure 2.9: *Limitations of our texture synthesis technique. The smaller patches (size 192x192) are the input textures, and to their right are synthesized results (size 200x200). Limitations of our approach include global features (grass in close up view), perspectivity (building exterior), 3D shape (pumping and beans), lighting and shadow (pumpkin), or textures composed of meaningful elements (beans).*

limitations as shown in Figure 2.9. Because we model textures as local and stationary phenomena, our algorithm cannot reproduce global features such as perspectivity, lighting and shadow. This is a fundamental limitation of most texture synthesis algorithms since textures are usually to be assumed to be characterized by local properties.

Because our algorithm models textures by spatial neighborhoods and measures the neighborhood similarities in L2 norm, it cannot distinguish between important and less important image information. In other words our algorithm cannot model high-level visual cues such as object silhouettes, boundaries, or semantic information that are easily picked up by the human visual system. One solution is to replace the Gaussian pyramid with a feature pyramid, containing semantic information such as filtered edges or importance maps about various regions of the input textures. However, this might require either human assistance or some computer vision algorithms to acquire these features.

For textures containing complicated textons, our algorithm may have difficulty finding good matches during the neighborhood search process. This, together with the use of Gaussian pyramids, might have caused the problem for synthesizing the beans texture in Figure 2.9 where the individual beans seem to be merged together. One quick fix is to generate textures in patches rather than individual pixels. As long as the patches are big enough to cover several textons, the output should look feasible since these textons are copied directly and kept intact. However since there is no guarantee that these patches will tile seamlessly, they might cause visual discontinuities in the output texture. For high-frequency textures the effect of visual masking may kick in and hide most of the discontinuities. For other textures we might have to do some registration, blending, or constrained synthesis at the patch boundaries. However, currently there seems to be no perfect solution that can solve this problem in general.

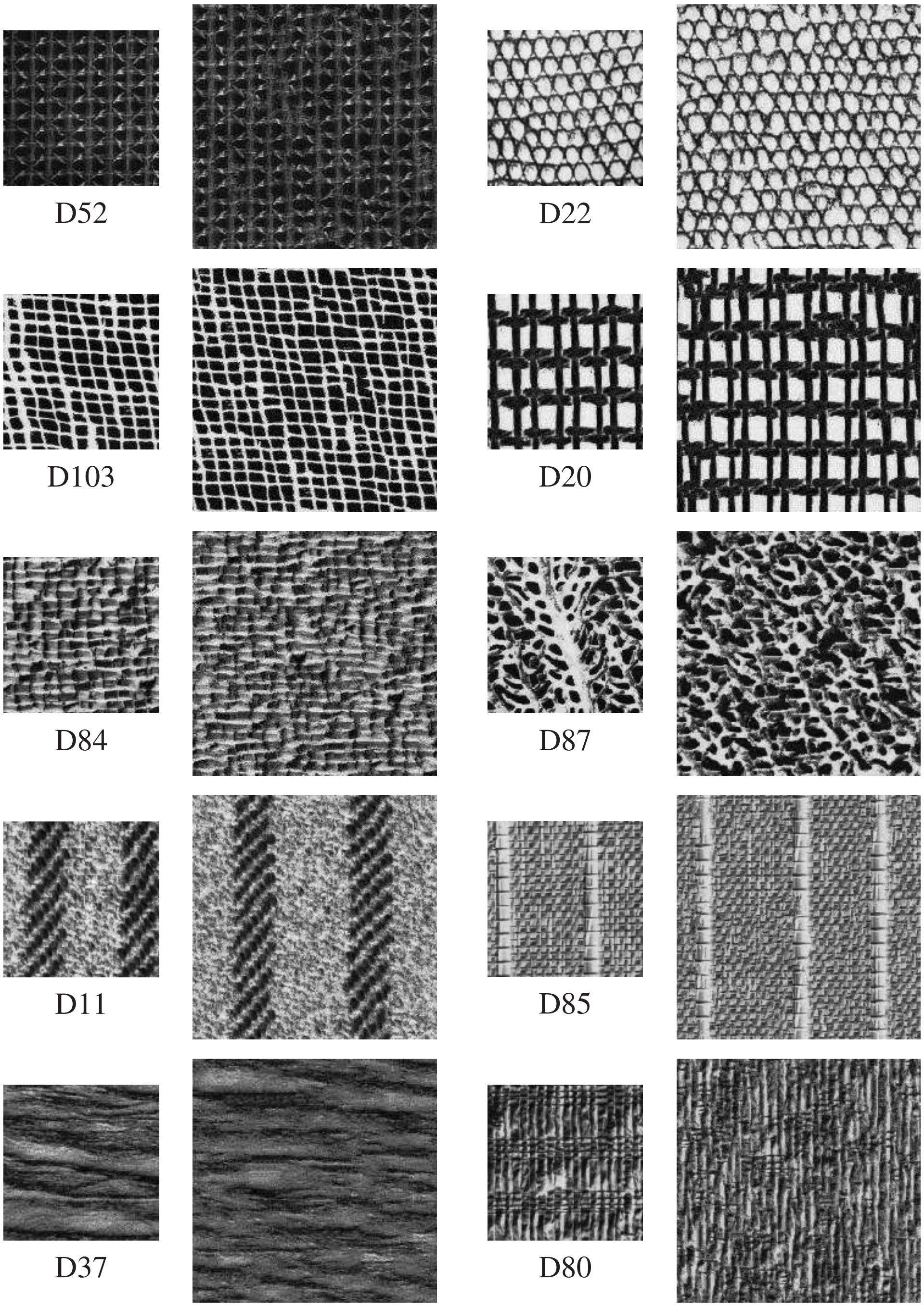


Figure 2.10: *Brodatz texture synthesis results. The smaller patches (size 128x128) are the input textures, and to their right are synthesized results (size 200x200).*

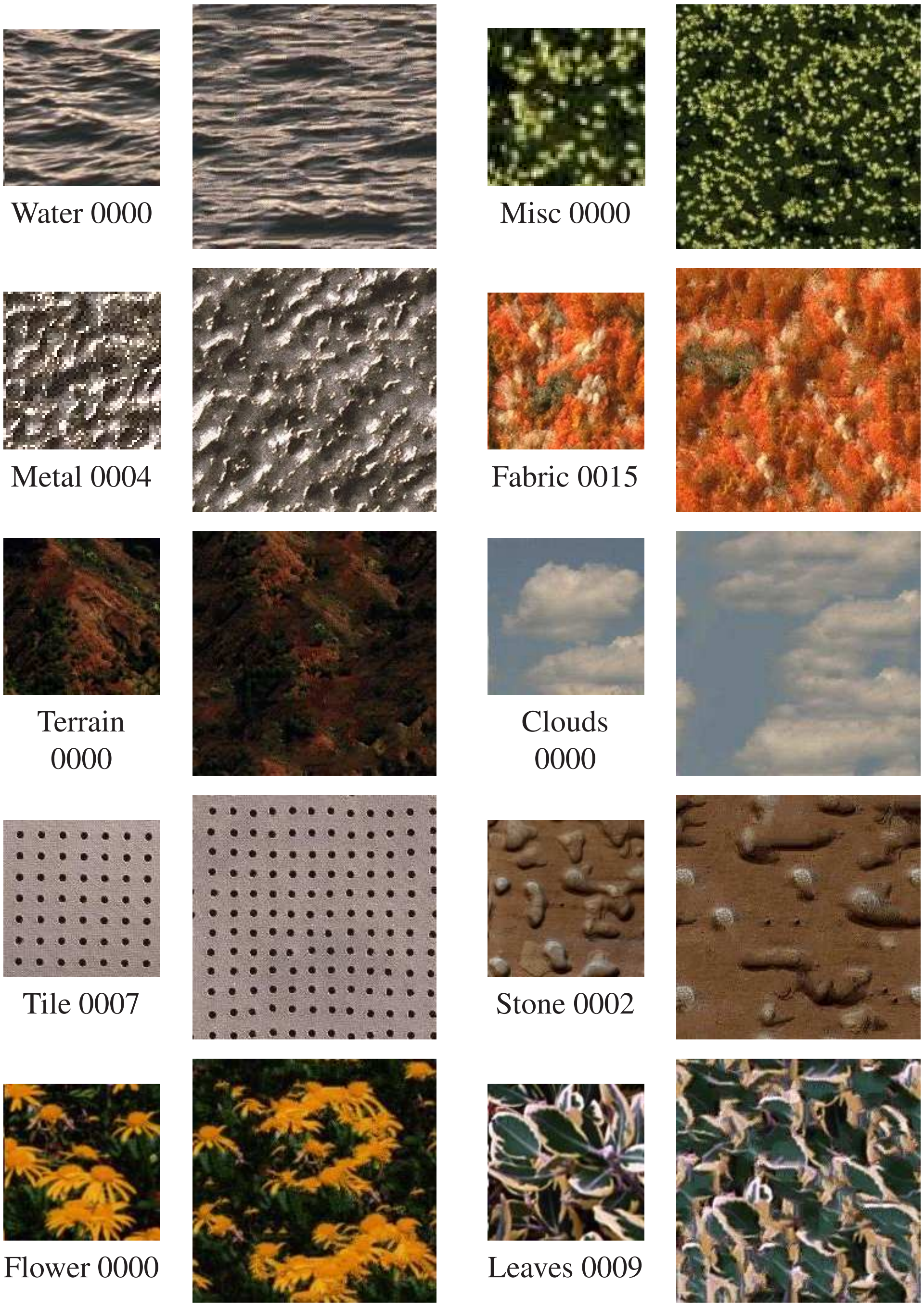


Figure 2.11: *VisTex texture synthesis results. The smaller patches (size 128x128) are the input textures, and to their right are synthesized results (size 200x200).*

Chapter 3

Acceleration

Our deterministic synthesis procedure introduced in the previous chapter avoids the usual computational requirement for sampling from a MRF. However, the algorithm as described employs exhaustive searching, which makes it slow. Fortunately, acceleration is possible. This is achieved by considering neighborhoods $N(p)$ as points in a multiple dimensional space, and casting the neighborhood matching process as a nearest-point searching problem [46]. We have found that one of the nearest-point searching techniques, known as tree-structured vector quantization (TSVQ, [23]), works particularly well for accelerating our texture synthesis algorithm.

In this chapter, we first introduce the nearest-point searching problem, and review previous works (Section 3.1). We then describe tree-structured vector quantization (Section 3.2). We show how tree-structured VQ can be applied to accelerate our synthesis process (Section 3.3), and compare synthesis results with and without acceleration (Section 3.4).

3.1 Nearest-Point Searching

Searching for nearest neighbors is an important problem in many fields of science and engineering, with applications ranging from pattern matching, object recognition, and database retrieval. We can formulate the nearest-point searching problem in multiple dimensions as follows: given a set S of n points and a novel query point Q in a d-dimensional space, find a point in the set such that its distance from Q is lesser than, or equal to, the distance of

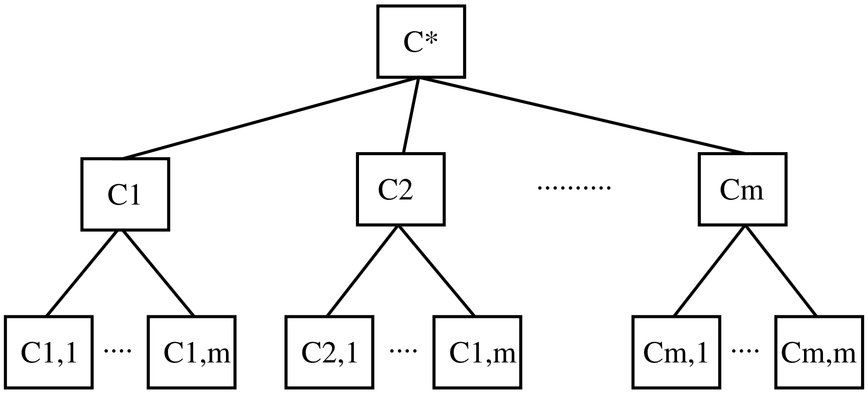


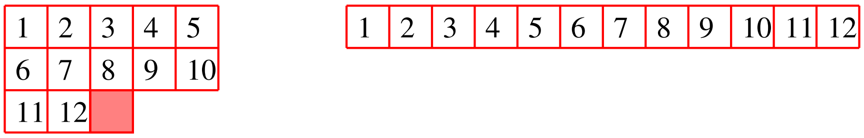
Figure 3.1: *Data structure of Tree-structured VQ.*

Q from any other point in the set. Because a large number of such queries may need to be conducted over the same data set S, the computational cost can be reduced if we preprocess S to create a data structure that allows fast nearest-point queries. Many such data structures have been proposed, and we refer the reader to [46] for a more complete reference.

Nearest-point searching in high dimensions is a hard problem. A lot of existing techniques have time complexity growing exponentially with the dimensionality d. However, most of these algorithms assume generic inputs and do not attempt to take advantage of any special structures they may have. Popat [53] observed that the set S of spatial neighborhoods from a texture can often be characterized well by a clustering probability model. Taking advantage of this clustering property, we propose to use tree-structured vector quantization [23] as the searching algorithm.

3.2 Tree-structured Vector Quantization

Tree-structured vector quantization (TSVQ) is a common technique for data compression. It takes a set of training vectors as input, and generates a binary-tree-structured codebook. The first step is to compute the centroid of the set of training vectors and use it as the root level codeword. To find the children of this root, the centroid and a perturbed centroid are chosen as initial child codewords. A generalized Lloyd algorithm [23], consisting of alternations between centroid computation and nearest centroid partition, is then used to find



(a) (b)

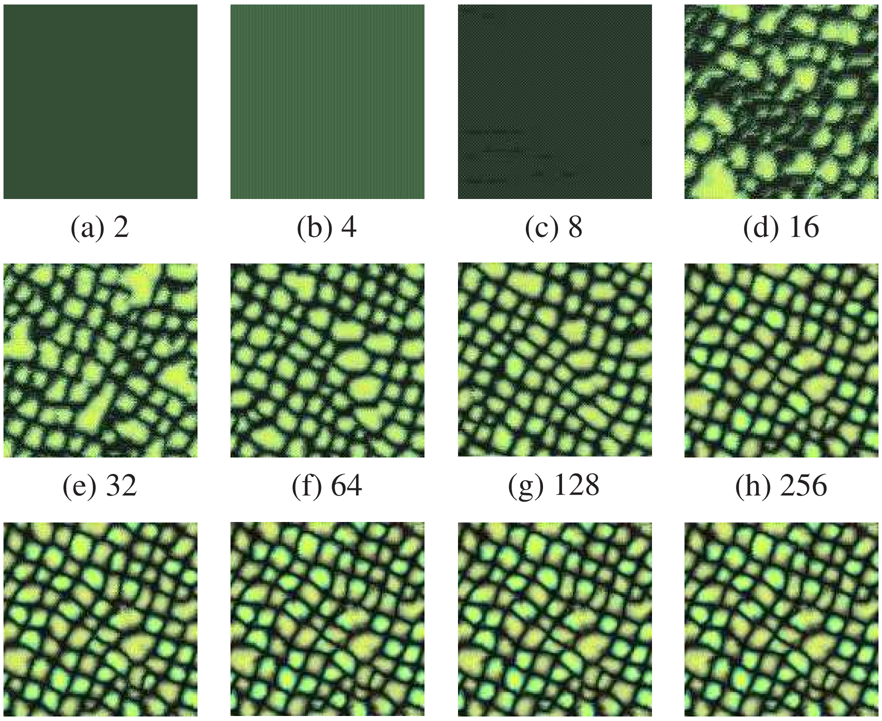
Figure 3.2: *Treat neighborhoods as high-dimensional points. (a) A causal neighborhood containing 12 pixels (b) The same neighborhood treated as a 12 dimensional point. The pixels are assumed to be monochromatic in this case. For RGB color textures the result will be a 36 dimensional point.*

the locally optimal codewords for the two children. The training vectors are divided into two groups based on these codewords and the algorithm recurses on each of the subtrees. This process terminates when the number of codewords exceeds a pre-selected size or the average coding error is below a certain threshold. The final codebook is the collection of the leaf level codewords.

The tree generated by TSVQ can be used as a data structure for efficient nearest-point queries. To find the nearest point of a given query vector, the tree is traversed from the root in a best-first ordering by comparing the query vector with the two children codewords, and then follows the one that has a closer codeword. This process is repeated for each visited node until a leaf node is reached. The best codeword is then returned as the codeword of that leaf node. Unlike full searching, the result codeword may not be the optimal one since only part of the tree is traversed. However, the result codeword is usually close to the optimal solution, and the computation is more efficient than full searching. If the tree is reasonably balanced (this can be enforced in the algorithm), a single search with codebook size $|S|$ can be achieved in time $O(log|S|)$, which is much faster than exhaustive searching with linear time complexity $O(|S|)$.

3.3 Tree-structured Vector Quantization for Texture Synthesis

To use TSVQ in our synthesis algorithm, we simply collect the set of neighborhood pixels $N(p_i)$ for each input pixel and treat them as a vector of size equal to the number of pixels



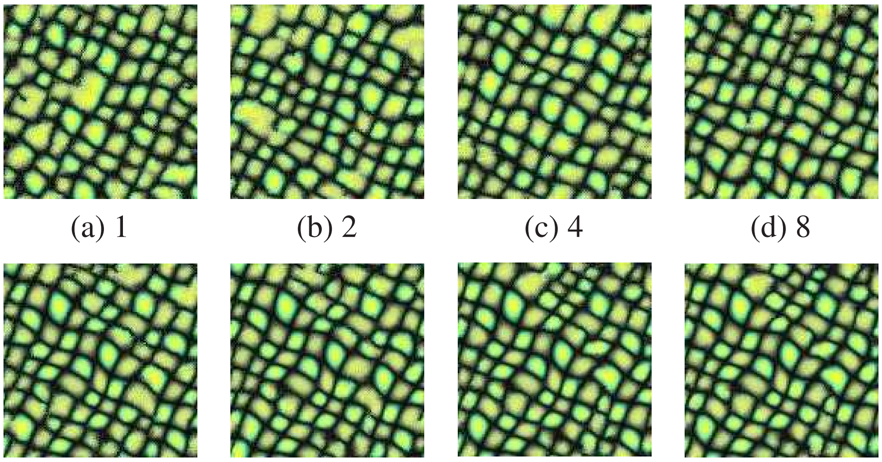
(a) 2 (b) 4 (c) 8 (d) 16

(e) 32 (f) 64 (g) 128 (h) 256

(i) 512 (j) 1024 (k) 2048 (l) 4096 (all)

Figure 3.3: *TSVQ acceleration with different codebook sizes. The original image size is 64x64 and all these synthesized results are of size 128x128. The number of codewords used are labeled below each image.*

in $N(p_i)$ (Figure 3.2). We use these vectors $\{N(p_i)\}$ from each $G_a(L)$ as the training data, and generate the corresponding tree structure codebooks $T(L)$. During the synthesis process, the (approximate) closest point for each $N(p)$ at $G_s(L)$ is found by doing a best-first traversal of $T(L)$. Because this tree traversal has time complexity $O(logN_L)$ (where N_L is the number of pixels of $G_a(L)$), the synthesis procedure can be executed very efficiently. Typical textures take seconds to generate; the exact timing depends on the input and output image sizes.



(a) 1 (b) 2 (c) 4 (d) 8

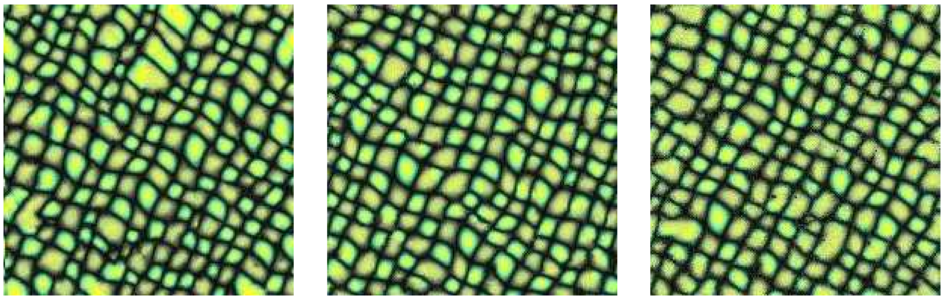
(e) 16 (f) 32 (g) 64 (h) 4096 (all)

Figure 3.4: *TSVQ acceleration with different number of visited leaf nodes. The original image size is 64x64 and all these synthesized results are of size 128x128. The number of visited leaf nodes are labeled below each image.*

3.4 Results

An example comparing the results of exhaustive searching and TSVQ is shown in Figure 3.6. The original image sizes are 128x128 and the resulting image sizes are 200x200. The average running time for exhaustive searching is 360 seconds. The average training time for TSVQ is 22 seconds and the average synthesis time is 7.5 seconds. The code is implemented in C++ and the timings are measured on a 195MHz R10000 processor. As shown in Figure 3.6, results generated with TSVQ acceleration are roughly comparable in quality to those generated from the unaccelerated approach. In some cases, TSVQ will generate more blurry images. We fix this by allowing limited backtracking in the tree traversal so that more than one leaf node can be visited. The amount of backtracking can be used as a parameter which trades off between image quality and computation time. When the number of visited leaf nodes is equal to the codebook size, the result will be the same as the exhaustive searching case. The effect of backtracking on synthesis quality is shown in Figure 3.4.

One disadvantage of TSVQ acceleration is the memory requirement. Because an input pixel can appear in multiple neighborhoods, a full-sized TSVQ tree can consume $O(d * N)$



(a) 1941 seconds (b) 503 seconds (c) 24 seconds

Figure 3.5: *A breakdown of running time for the textures shown in Figure 2.8. (a) timing of Efros and Leung's algorithm (b) timing of our algorithm using full search (c) timing of our algorithm using tree-structured VQ (12 seconds for training and 12 seconds for synthesis). All the timings were measured using a 195 MHz R10000 processor.*

memory where d is the neighborhood size and N is the number of input image pixels. Fortunately, textures usually contain repeating structures; therefore we can use codebooks with fewer codewords than the input training set. Figure 3.3 shows textures generated by TSVQ with different codebook sizes. As expected the image quality improves when the codebook size increases. However, results generated with fewer codewords (such as 512 codewords) look plausible compared with the full codebook result (4096 codewords). In our experience we can use codebooks less than 10 percent the size of the original training data without noticeable degradation of quality of the synthesis results. To further reduce the expense of training, we can also train on a subset rather than the entire collection of input neighborhood vectors.

Figure 3.5 shows a timing breakdown for generating the textures shown in Figure 2.8. Our unaccelerated algorithm took 503 seconds. The TSVQ accelerated algorithm took 12 seconds for training, and another 12 seconds for synthesis. In comparison, Efros and Leung's algorithm [19] took half an hour to generate the same texture (the time complexity of our approach over Efros and Leung's is $O(logN)/O(N)$ where N is the number of input image pixels). Because their algorithm uses a variable sized neighborhood it is difficult to accelerate. Our algorithm, on the other hand, uses a fixed neighborhood and can be directly accelerated by any point searching algorithm.

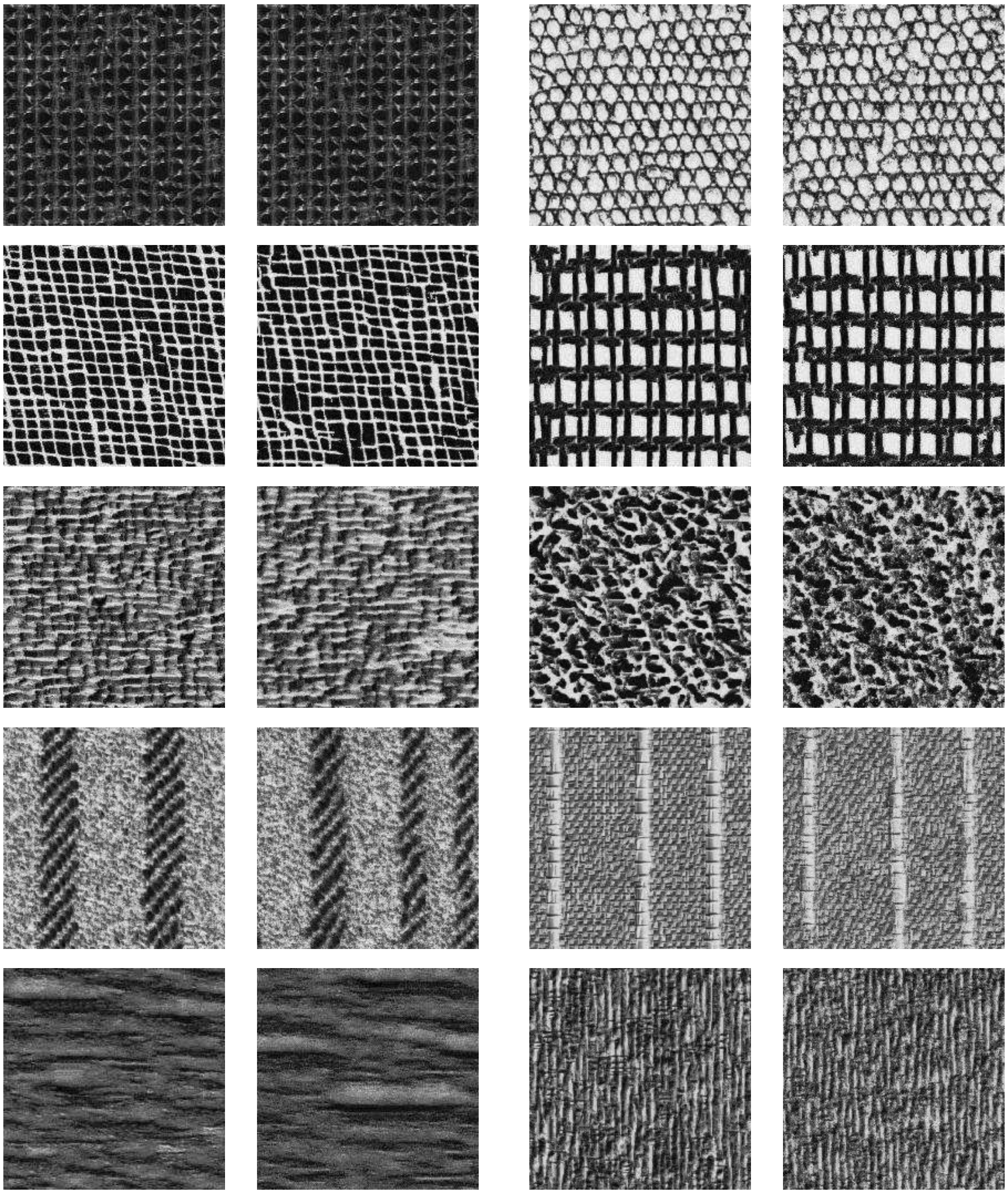


Figure 3.6: *Brodatz texture synthesis results with tree-structured VQ acceleration. For each pair of images, the result generated by exhaustive search is shown on the left (same as those in Figure 2.10), and the TSVQ acceleration result is shown on the right.*

Chapter 4

Constrained Texture Synthesis

Photographs, films and images often contain regions that are in some sense flawed. A flaw can be a scrambled region on a scanned photograph, scratches on an old film, wires or props in a movie film frame, or simply an undesirable object in an image. Since the processes causing these flaws are often irreversible, an algorithm that can fix these flaws is desirable. Often, the flawed portion is contained within a region of texture, and can be replaced by texture synthesis.

In this chapter, we extend the basic algorithm (Chapter 2 and Chapter 3) for replacing textured regions. We refer to this extended approach "constrained texture synthesis", since the synthesized texture region must blend seamlessly with the existing textures. In the rest of this chapter, we first give an overview of previous methods for image denoising and restoration (Section 4.1). We then describe our algorithm (Section 4.2), and demonstrate potential applications of this approach (Section 4.3).

4.1 Image Restoration

Image restoration/denoising techniques can be classified by what kind of "noise" they aim to remove, and what kind of information are available for removing noise:

4.1.1 Frequency Domain Techniques

Image noise is commonly assumed to occupy higher frequency bands. By low-pass filtering the images we can remove those high frequency noises. However, frequency domain techniques are not well suited to remove artifacts that are spatially localized.

4.1.2 Inter-Frames Techniques

For image sequences we can use motion estimation to interpolate losses in a single frame from adjacent frames. The basic idea is to find the right pixels from neighboring frames. However, this technique cannot be applied to still images or to films where the defects span many frames.

4.1.3 Block-based Techniques

A simple way to replace objects in images is to copy and paste blocks of image pixels. The new block can be blended smoothly with existing image regions using multiresolution spline [10]. However, this does not work for regions containing dominant structures such as textures (Figure 4.1 (b)).

4.1.4 Diffusion

Defects that are skinny can be removed by diffusion [5]. The basic idea is to gradually replacing these defects by colors diffused from surrounding regions. The advantage of this approach is that it is very easy to use; the user only needs to mark where the defects are. However this approach is cannot replace large textured regions.

4.1.5 Combining Frequency and Spatial Domain Information

Hirani and Totsuka [36] combine frequency and spatial domain information to fill a given region with artifacts. This technique requires the existence of a translationally similar region to the defected image area. Besides this restriction it works pretty well across a wide variety of situations.

4.1.6 Texture Replacement

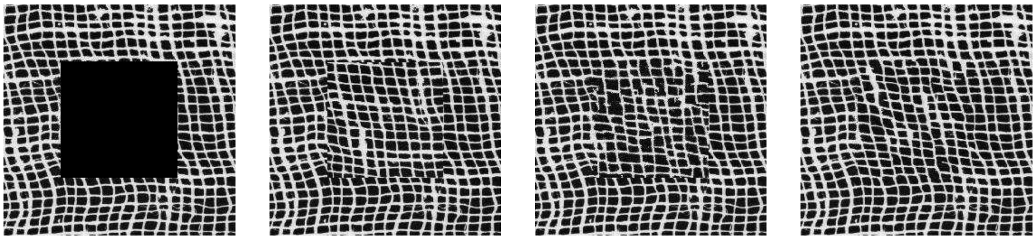
If the flawed region contains textures, it can be replaced by texture synthesis [19, 38, 44]. The challenge is to ensure that the synthesized new texture fits seamlessly with the existing region. Efros and Leung [19] achieve this by growing the textures from the hole boundaries. Because the boundary can have arbitrary shapes this technique cannot be accelerated by tree-structured VQ (Chapter 3).

4.1.7 Super-resolution Techniques

In addition to noise removal, there are several techniques that can fabricate new information, such as generating high frequency bands for super-resolution [21, 54]. Super-resolution is related to our constrained synthesis algorithm, as we shall see later.

4.2 Our Algorithm

Texture replacement by constrained synthesis must satisfy two requirements: the synthesized region must look like the surrounding texture, and the boundary between the new and old regions must be invisible. Multiresolution blending [10] with another similar texture, shown in Figure 4.1 (b), will produce visible boundaries for structured textures. Better results can be obtained by applying our algorithm in Chapter 2 over the flawed regions, but discontinuities still appear at the right and bottom boundaries as shown in Figure 4.1 (c). These artifacts are caused by the causal neighborhood as well as the raster scan synthesis ordering. Figure 4.3 explains how the discontinuities are caused. When synthesizing the first row and column (Figure 4.3 (a) (b)) of the hole, the causal neighborhood contains all the adjacent pixels (of the target pixel) at the boundary between the hole and the surrounding. Because these boundary pixels are part of the neighborhood, they enforce the search process to find candidates that fit smoothly with them. However, when synthesizing the bottom row and right column, some pixels adjacent to the target are not included in the neighborhood, such as those marked with “X” and “Y”. Because they are not part of the neighborhood, we can arbitrarily set the values without changing the result of the neighborhood search process. And this causes discontinuities.



(a) (b) (c) (d)

Figure 4.1: *Constrained texture synthesis. (a) a texture containing a black region that needs to be filled in. (b) multiresolution blending [10] with another texture region will produce boundary artifacts. (c) A direct application of the algorithm in Chapter 2 will produce visible discontinuities at the right and bottom boundaries. (d) A much better result can be generated by using a modification of the algorithm with 2 passes.*

To remove these boundary artifacts a noncausal (symmetric) neighborhood must be used. However, we have to modify the original algorithm so that only valid (already synthesized) pixels are contained within the symmetric neighborhoods; otherwise the algorithm will not generate valid results (Figure 2.5). This can be done with a two-pass extension of the original algorithm. Each pass is the same as the original multiresolution process, except that a different neighborhood is used. During the first pass, the neighborhood contains only pixels from the lower resolution pyramid levels. Because the synthesis progresses in a lower to higher resolution fashion, a symmetric neighborhood can be used without introducing invalid pixels. This pass uses the lower resolution information to "extrapolate" the higher resolution regions that need to be replaced. In the second pass, a symmetric neighborhood that contains pixels from both the current and lower resolutions is used. These two passes alternate for each level of the output pyramid. In the accelerated algorithm, the analysis phase is also modified so that two TSVQ trees corresponding to these two kinds of neighborhoods are built for each level of the input pyramid. Finally, we also modify the synthesis ordering in the following way: instead of the usual raster-scan ordering, pixels in the filled regions are assigned in a spiral fashion. For example, the hole in Figure 4.1 (a) is replaced from outside to inside from the surrounding region until every pixel is assigned (Figure 4.1 (d)). This spiral synthesis ordering removes the directional bias which causes the boundary discontinuities (as in Figure 4.1 (c)).

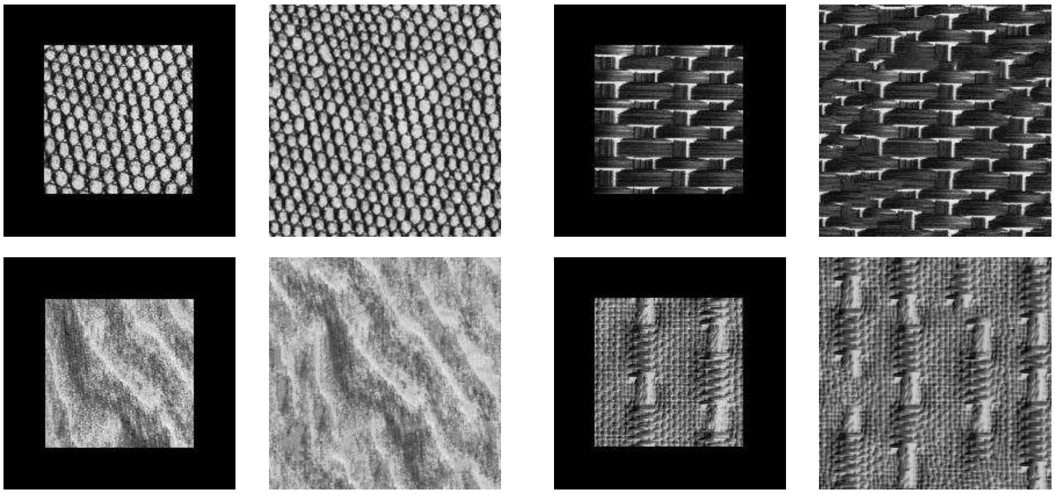


Figure 4.2: *Texture extrapolation. For each pair of images, we spatially extrapolate the texture on the left to the one on the right.*

4.3 Results and Applications

Figure 4.4 shows several examples of hole replacement. The algorithm is able to fill in the holes for a wide variety of textures. The newly synthesized textures look like the surrounding, and their structures connect smoothly with the surrounding textures. With a slight change of the synthesis ordering, the algorithm can be applied to other applications, such as the image extrapolation shown in Figure 4.2. The algorithm could also be used as a tool to remove undesirable objects in a photograph (Figure 4.5 (a), (b), and (c)), or to fabricate textures over dull regions in scanned images (Figure 4.5 (d)).

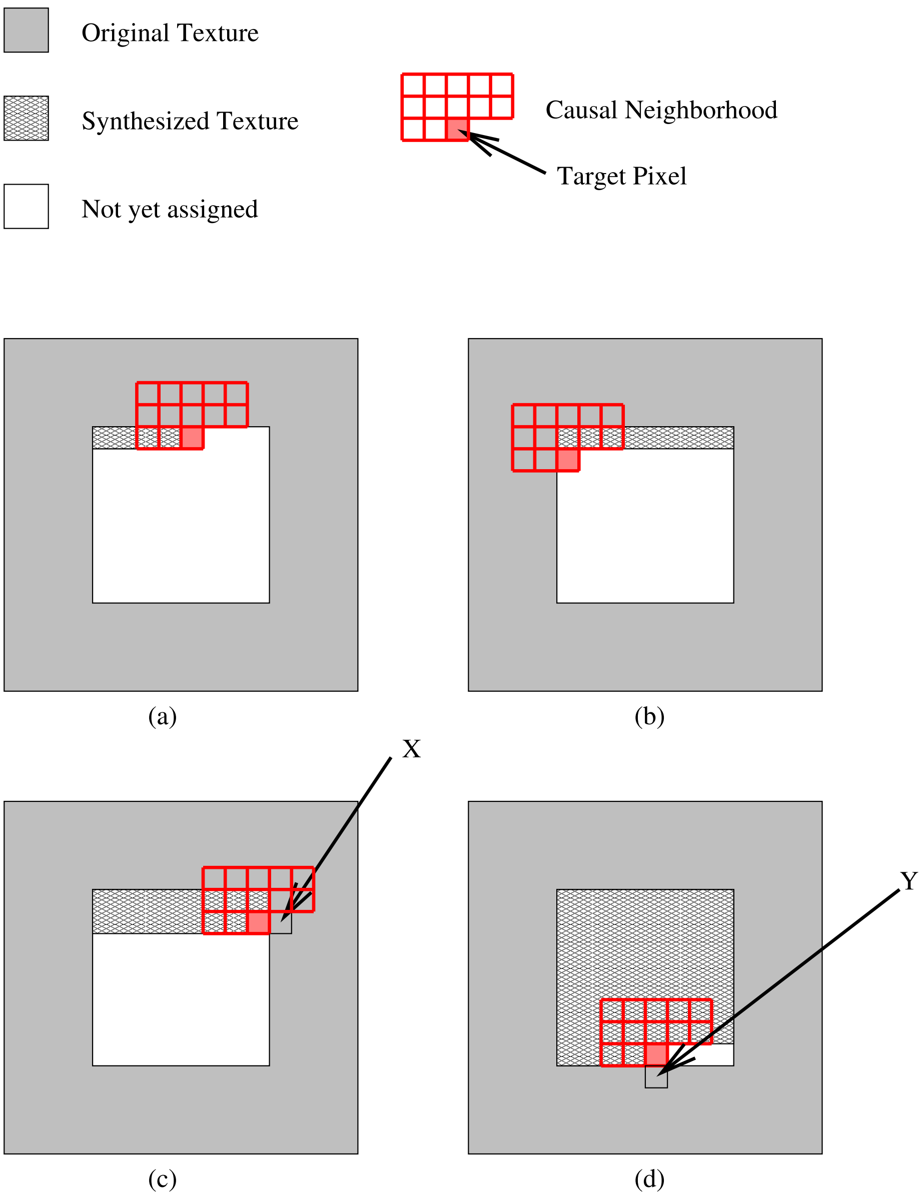


Figure 4.3: *Discontinuities caused by using a causal neighborhood in constrained synthesis. (a) synthesizing the top row (b) synthesizing the left column (c) synthesizing the right column (d) synthesizing the bottom row. In cases (c) and (d), there are pixels adjacent to the target pixel that are not included in its neighborhood (marked as X and Y). Because these pixels cannot effect the search process they will cause discontinuities.*

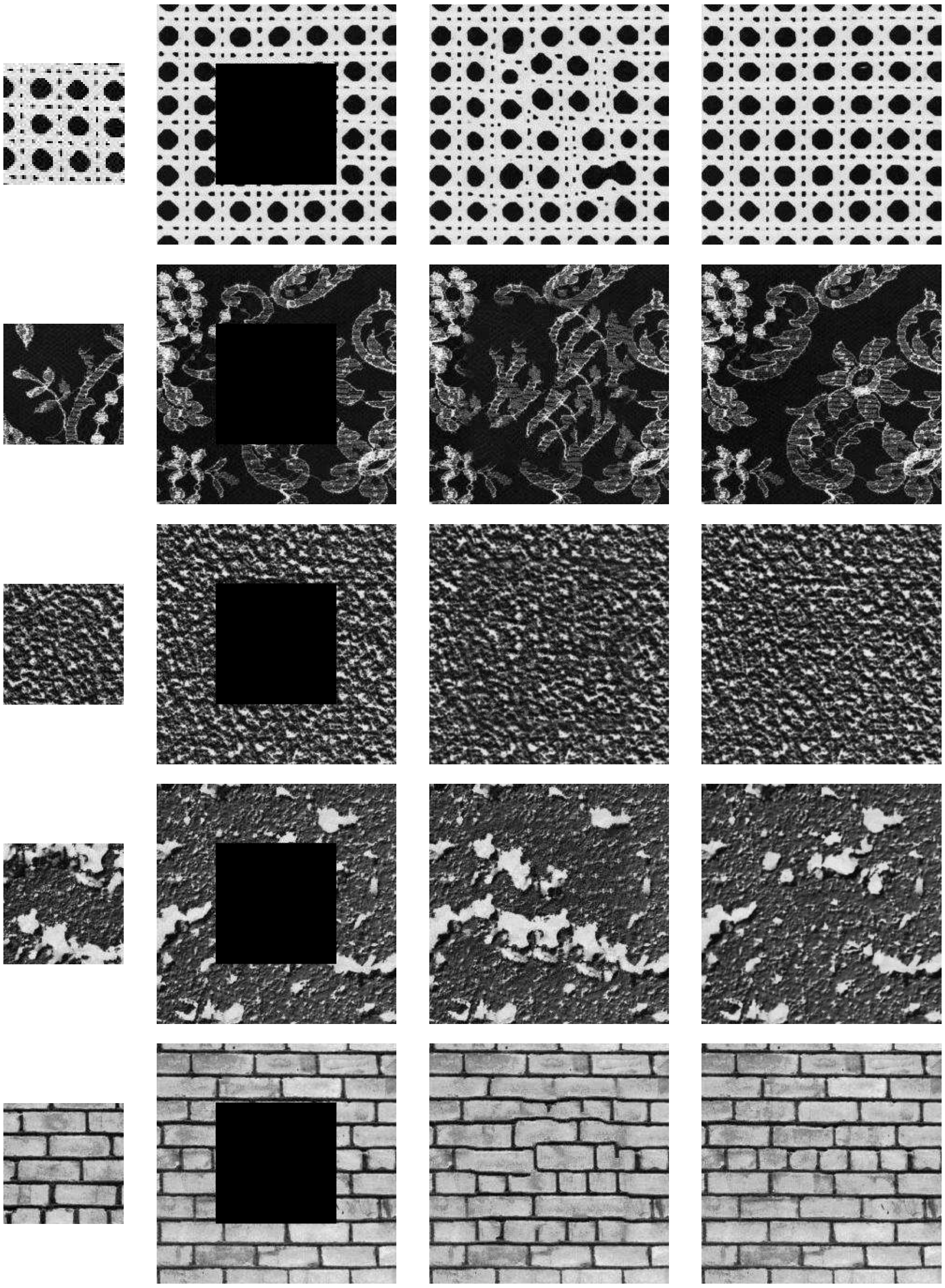


Figure 4.4: *Texture replacement. For each row of images, we show: (a) the sample image, (b) image with hole, (c) image with hole filled by our algorithm using the image in (a) as sample, and (d) the original image before the hole is introduced.*

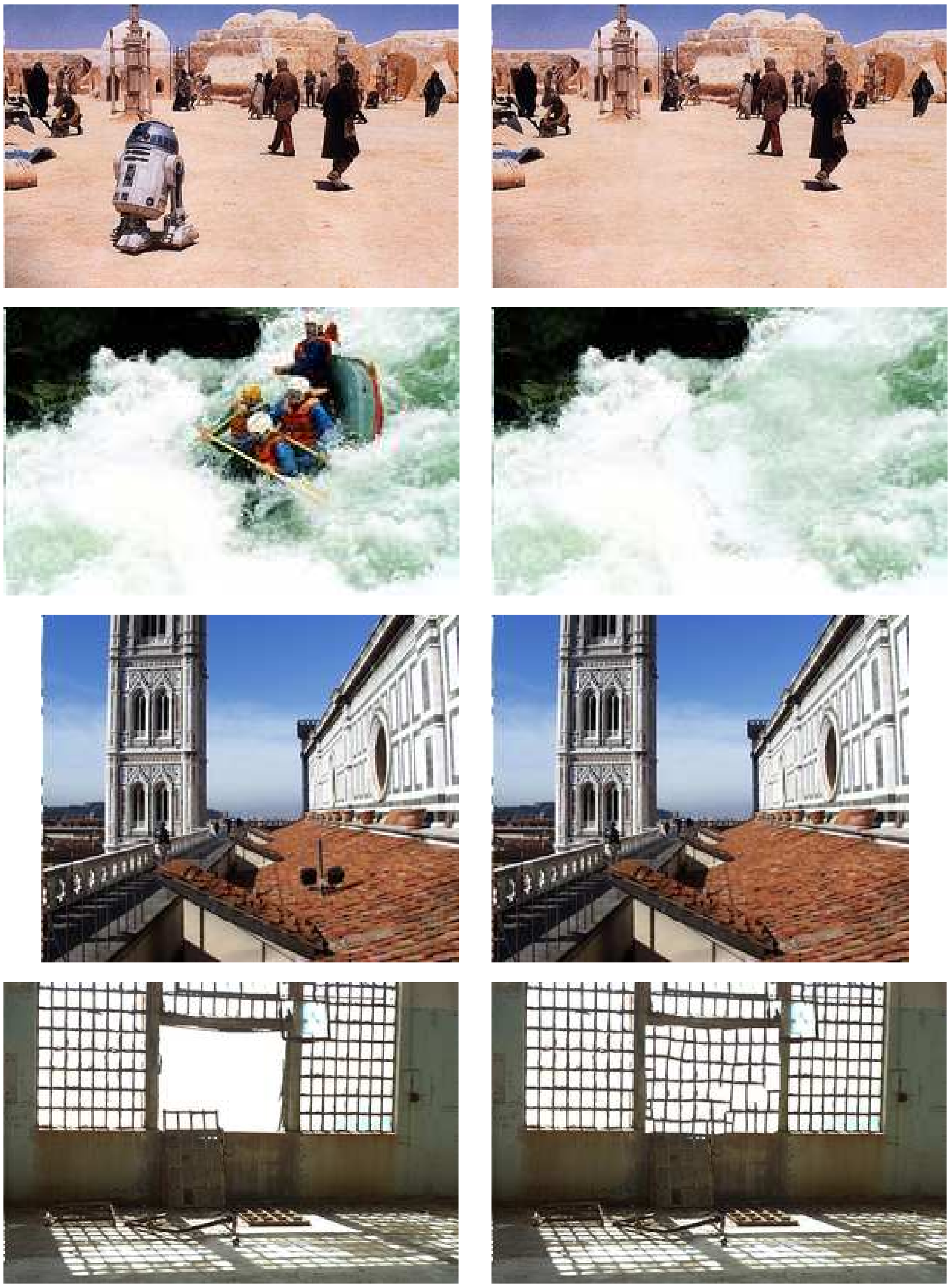


Figure 4.5: *Texture replacement for real scenes. For image pairs (a), (b), and (c), an object on the left image is replaced by the texture background, as shown on the right. Image pairs (d) show interesting applications of texture synthesis to fabricate scene details.*

Chapter 5

Motion Texture Synthesis

Rendering life-like animations is a major goal for computer graphics. However, compared to the maturity of photorealistic image rendering, computer generated motions are still at the stage of infancy. The difficulty to generate motions can be attributed to several reasons. Real motions usually involve complex physics, such as fluid flow, smoke rising or animal jumping, making them difficult to model. Real motions are diverse, therefore it is difficult to incorporate them under a single framework. In addition a lot of motions involve high-level human perception such as gestures or facial expressions, and this further aggregates the difficulty for realistic motion synthesis.

Real-life motions often contain repetitions. Examples are running, walking, fire burning, and ocean waves. These motions containing semi-regular repetitions are usually refered to as *motion textures*. Motion textures are important for realistic animations; without them motions will look stiff and robot like. Unfortunately, generating motion textures by hand can be very tedious; often animators only want explicit controls over high-level motion features such as opening a door or jumping over an obstacle without worrying about fine details such as tiny differences between each walking step. Fortunately, these kind of motion textures are very suitable for automatic generation; since motion textures contain semi-regular repetitions, they can be modeled as variations of image textures and synthesized by our algorithm.

In this chapter, we present modifications of our image texture synthesis algorithm

(Chapter 2 and Chapter 3) for synthesizing motions textures. In particular, we concentrate on two kinds of motions: 3D spatial-temporal volumes such as fire and smoke, and 1D articulated motion signals such as the joint angles of a human walking. The application of our algorithm to motion textures has two implications:

1. It provides a useful tool to synthesize motion textures and lets animators render realistic animations without worrying about tedious details.

2. It demonstrates the generality of our texture algorithm in the sense that a wide variety of different physical phenomena such as visual images, spatial-temporal volumes, and motion signals can all be modeled under a single framework.

5.1 Temporal Texture Synthesis

Temporal textures are motions with indeterminate extent both in space and time. They can describe a wide variety of natural phenomena such as fire, smoke, and fluid motions. Since realistic motion synthesis is one of the major goals of computer graphics, a technique that can synthesize temporal textures would be useful. Most existing algorithms model temporal textures by direct simulation; examples include fluid, gas, and fire [65, 63, 64, 75, 80]. Direct simulations, however, are often expensive and only suitable for specific kinds of textures; therefore an algorithm that can model general motion textures would be advantageous [66].

Temporal textures consist of 3D spatial-temporal volume of motion data. If the motion data is local and stationary both in space and time, the texture can be synthesized by a 3D extension of our algorithm. This extension can be simply done by replacing various 2D entities in the original algorithm, such as images, pyramids, and neighborhoods, with their 3D counterparts. For example, the two Gaussian pyramids are constructed by filtering and downsampling from 3D volumetric data; the neighborhoods contain local pixels in both the spatial and temporal dimension. The synthesis progresses from lower to higher resolutions, and within each resolution the output is synthesized slice by slice along the time domain. However, synthesizing 3D spatial-temporal volumes can be computationally expensive. Fortunately, the low cost of our accelerated algorithm enables us to consider synthesizing textures of dimension greater than two.

Figure 5.1 shows synthesis results of several typical temporal textures: fire, smoke, and ocean waves (animations available on our webpage). The resulting sequences capture the flavor of the original motions, and tile both spatially and temporally. This technique is also efficient. Accelerated by TSVQ, each result frame took about 20 seconds to synthesize. Currently all the textures are generated automatically; it is possible extend the algorithm to allow more explicit user controls (such as the distribution and intensity of the fire and smoke).

5.2 Synthesizing Articulated Motions

There are four main methods to create motions with articulated figures. *Key-frame interpolation* defines still poses at specific time instances, and generates the rest of the motion by interpolation. Although animators could have complete control over the style of motions, specifying poses for many time instances can be tedious. *Physical simulation* address this problem by emulating the dynamics of motions such as animal walking or running. By generating the motion automatically the animators are freed from explicitly specifying each keyframes. However, simulating articulated motions of complex creates (such as human) requires fancy physical models. As a result existing techniques for physically simulated animations are more successful in modeling simple motions such as cloth [3] and fluid [75] than in simulating articulated motions. In addition, physical simulations are often computationally expensive. *Machine learning* avoids some of the computation burdens of physical simulations by learning the salient features of motions [28]. However, because machine learning generates motions automatically, animators lose the ability to directly control the motions.

Motion capture addresses the limitations of other methods by capturing motions directly from life characters. With the recent advances of sensor technologies, motion capture is able to provide a wide variety of real motions in details. However, those motions are "canned" and usually require modifications before they can be reused for rendering animations.

5.2.1 Motion Signal Processing

Motion-captured signals can be modified for new purposes by signal processing operations. For example, traditional signal processing operations such as band-pass decomposition can be used for warping motion signals [79], depicting emotions [70], or adapting motions in general [9]. Existing motions can also be retargeted to new characters [26], or blended together to form motion mixtures [7]. The success of those techniques demonstrates the flexibility of applying signal processing operations to motion-captured data. However, none of these techniques directly target motion signals that exhibit texture-like behaviors.

We are interested in modeling a specific class of motion signals containing semi-regular repetitions such as walking, running, or eye blinking. We want to preserve not only the structures but also the random variations of the motions. For example, although a walking contains deterministic cycles, each walking step usually differs slightly from each other. These random variations are important for rendering realistic motions; without them animations may look predictable and machine like [51, 56].

5.2.2 Generating Motion Signals by Texture Synthesis

We generate semi-regular repetitive motions using our texture synthesis algorithm. Specifically, we synthesize a new motion one frame at the time, and the values of the motion signals in each frame are determined as follows. We construct a temporal neighborhood that covers several previous frames, and compare this neighborhood with similar ones in the input motion. The value of the best match is then returned as the new frame. This process is very similar to our image texture synthesis algorithm, except that we use temporal neighborhoods and motion signal values instead of spatial neighborhoods and RGB color channels.

Motion signals are often represented as raw marker positions (x, y, z, and sometimes orientations) or joint angles (degrees). In both cases we need to preprocess them before running our texture synthesis algorithm, since directly comparing motion signals may not make sense. In the next section we discuss details for preprocessing captured motion signals.

5.2.3 Data Preprocessing

5.2.3.1 Raw Marker Data

Raw marker data consists of marker positions (x, y, z) and optionally orientations (if motion capture is done in a magnetic field). In our application we only care about 3D marker positions x, y, z. Because marker positions of each skeleton pose can be shifted by global translations (such as a walking cycle), directly comparing marker positions during texture synthesis does not make sense. We address this issue by either subtracting marker positions of each frame with respect to the "pivot position" (usually the hip of a human skeleton), or using the differences of marker positions between adjacent frames. In the former case we synthesize global translations and intra-frame marker movements separately, and combine them to generate the final motion as a post-processing step after texture synthesis.

However, raw marker positions are not a good parameterization for generating motions, and we usually convert them into joint angle data before applying texture synthesis.

5.2.3.2 Joint Angle Data

With a rough knowledge of the skeleton geometry, raw marker data can be converted to joint angles via inverse kinematics. However, joint angles are still not suitable for texture synthesis [7]. We apply a sequence of transformation as follows:

Scaling Each joint angle causes different amount of pose changes. For example, with the same angular change, the shoulder joint angles will cause more pose changes than the finger joint angles. We can scale the joint angles according to their relative impacts on body movements using the technique in [25].

Trigonometry Because joint angles are by natural cyclic (ranging from 0 to 360 degrees), we cannot compare them by simply taking the sum of squared difference. One solution is to round all signals into the domain $[0, 360)$ degrees, and use cyclic arithmetic to correctly compute the difference. (For example, the difference between 1° and 359° is 2°, not 358°.) However, this approach requires changing our implementation since we need additional arithmetic and conditional instructions for each angle comparison. An easier approach that doesn't require changing our fundamental coding is to map the angles into an acyclic range and compare them directly like RGB colors. There are several possible mappings to achieve this [7]. In our current implementation we convert each angle θ into a $[sin(\theta), cos(\theta)]$ pair. This representation transforms cyclic angles into acyclic range $[0, 1]$, and the phase shifting between sin and cos provides good resolution for different angles.

Principle Component Analysis (PCA) We can optionally run PCA to the motion signals before applying texture synthesis. The advantage of PCA is that it could potentially reduce noise and unnecessary details. However, it may also remove intricate motion details. We usually skip PCA in our experiments.

Because the global body translation/rotation is lost after converting marker positions to joint angles, we synthesize the global motion separately and combine it with the joint angles as a post-processing step after texture synthesis.

5.2.4 Results

Figure 5.2 shows selected frames of synthesis results for a variety of walking and running styles. In each case, the synthesized motion signal is twice as long as the original motion signal. All motions are encoded as joint angles, and we preprocess them with only the trigonometric parameterization (without scaling and PCA). We have also tested our approach with over a hundred different articulated motions and results are available at our project webpage. In our experience texture synthesis is most suitable for cyclic motion such as walking or running, not acyclic motions such as falling down.

5.3 Discussion

Because image textures, temporal textures, and periodic motion signals are governed by different perceptual mechanism, our synthesis results can be improved by considering their special properties. For example, since the spatial and temporal dimensions of temporal textures are perceived differently, better results could be obtained by using different decomposition methods when building multi-resolution pyramids [2]. Different joint angles of a articulated motion can have also different perceptual importance. For example, although the finger joints usually incur smaller body movements than arm joints, they can be more important in contexts such as gesturing. Those high-level context-dependent knowledge can be incorporated into our synthesis pipeline by proper weighting/filtering the input data as a preprocessing step. For example, since finger motions are more important we can assign them heavier weights than leg motions.

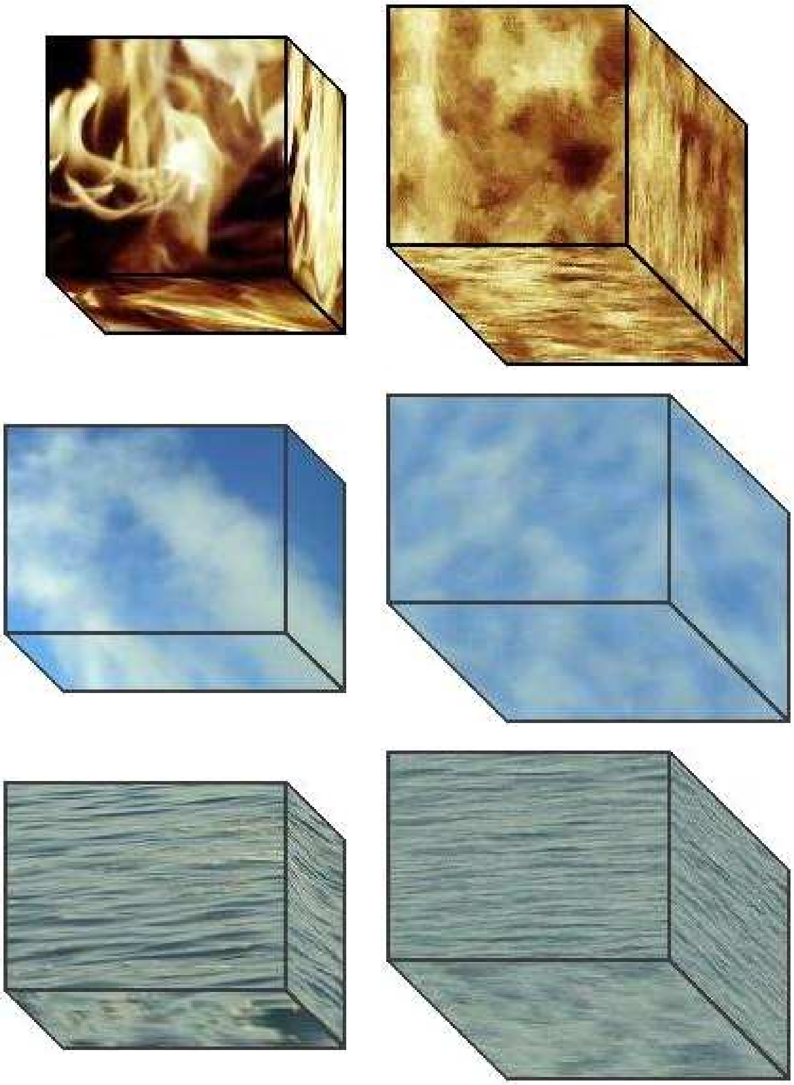


Figure 5.1: *Temporal texture synthesis results. (a) fire (b) smoke (c) ocean waves. In each pair of images, the spatial-temporal volume of the original motion sequence is shown on the left, and the corresponding synthesis result is shown on the right. A 3-level Gaussian pyramid, with neighborhood sizes {5x5x5,2}, {3x3x3,2}, {1x1x1,1}, are used for synthesis. The original motion sequences contain 32 frames, and the synthesis results contain 64 frames. The individual frame sizes are (a) 128x128 (b) 150x112 (c) 150x112. Accelerated by TSVQ, the training times are (a) 1875 (b) 2155 (c) 2131 seconds and the synthesis times per frame are (a) 19.78 (b) 18.78 (c) 20.08 seconds. To save memory, we use only a random 10 percent of the input neighborhood vectors to build the (full) codebooks.*

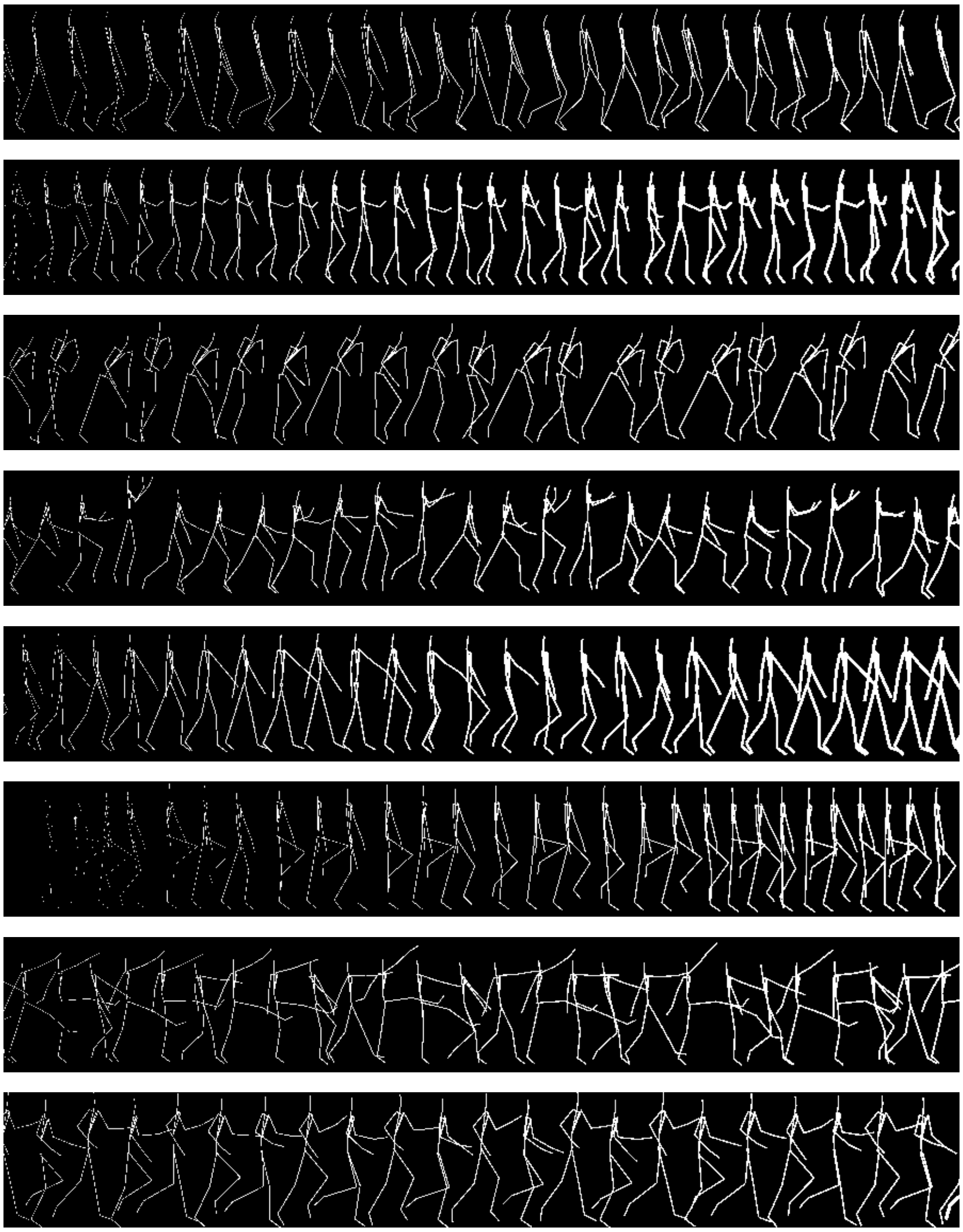


Figure 5.2: *Synthesis results of articulated motions. From top to bottom: walk, dainty walk, crippled walk, gallop walk, cowboy walk, march, goose march and run (available at our project webpage).*

Chapter 6

Surface Texture Synthesis

Computer graphics applications often use textures to decorate virtual objects without modeling geometric details. These textures can be generated from sample images using texture synthesis algorithms. However, most existing texture synthesis algorithms are designed for rectangular domains and can not be easily extended to general surfaces. One solution is to paste textures onto such surfaces using texture mapping. However, because general surfaces lack a continuous parameterization, this type of texture mapping usually causes distortions or discontinuities. An alternative approach that minimizes distortion is to generate textures directly over the surface. However, since we can not apply traditional image processing operations to surfaces, most existing methods for surface texture synthesis work only for limited classes of textures.

In this chapter, we present a method for synthesizing textures directly over 3D meshes.[1] Given a texture sample and a mesh model, our algorithm first uniformly distributes the mesh vertices using Turk's method [68]. It then assigns texture colors to individual mesh vertices so that the appearance of the surface appears to be the same as the input texture (Figure 6.1). It does this using a non-trivial extension of our algorithm for synthesizing planar textures as presented in previous chapters. Specifically, given a sample texture image, the planar algorithm synthesizes a new texture pixel by pixel in a scanline order. To determine the value of a particular output pixel, its spatial neighborhood is compared against all possible neighborhoods from the input image. The input pixel with the most similar neighborhood

[1]The majority of this chapter has been published in [74].

(a) Texture (b) Model (c) Synthesis result

Figure 6.1: *Surface texture synthesis. Given a texture sample (a) and a model (b), we synthesize a similar texture directly over the model surface (c).*

is then assigned to the output pixel. This neighborhood search process constitutes the core of our planar algorithm and is inspired by the pioneering work of Efros and Leung [19] and Popat and Picard [53]. The primary differences between our approach and [19, 53] are that our approach uses neighborhoods with fixed shapes and conducts the search deterministically; therefore it can be accelerated by tree-structured vector quantization.

Although our planar algorithm can synthesize a wide variety of textures, there are several difficulties in extending it to general meshes:

Connectivity Vertices on meshed surfaces are irregularly distributed, with varying inter-vertex distances and angles. As a result, the scanline order used in our planar algorithm cannot be applied.

Geometry Most surfaces are curved and cannot be flattened without cutting or distortion. This presents difficulties for defining the spatial neighborhoods that characterize textures.

Topology Because the surface of a general object cannot be mapped to a rectangle, it can not be parameterized using a rectangular grid. Most texture synthesis methods require the specification of a local texture orientation.

In this chapter, we present two modifications of our planar algorithm to address those challenges. First, we relax the scanline order, instead visiting vertices in random order, to allow texture synthesis over surfaces with arbitrary topology. Second, we replace the rectangular parameterization of the output domain that is implicit in our planar algorithm with tangent directions at each mesh vertex, coupled with a scale factor derived from the mesh vertex density. Based on this new parameterization we generalize the definition of search neighborhoods in our planar algorithm to meshes, and we show that this generalization works over a wide variety of textures. Specifically, for textures that are moderately isotropic, we use random tangent directions, and for anisotropic textures, we use tangent directions that are either user-specified or automatically assigned by our relaxation procedure.

The rest of the chapter is organized as follows. In Section 6.1, we review previous work. In Section 6.2, we present the algorithm. In Section 6.3, we demonstrate synthesis results. In Section 6.4, we conclude the chapter and discuss future work.

6.1 Previous Work

Texture Synthesis: Recent statistical texture synthesis algorithms [33, 59, 14, 73, 19] have achieved success in modeling image textures. Since these algorithms rely on planar grids, it is not clear how they can be extended to arbitrary surfaces. A different class of methods generate textures through specialized procedures [18]. These techniques produce textures directly over 3D surfaces, so the texture distortion problem is largely eliminated. However, procedural synthesis is capable of modeling only a limited class of textures.

There have been several attempts to extend statistical texture synthesis to surfaces [22] or 3D volumes [24, 33]. Based on second-order statistics, [22] relates pairs of mesh vertices via their geodesic curves. However, second-order statistics are unable to capture significant structures that occur in may textures [59]. Volumetric synthesis [24, 33] avoids this texture distortion. However, these algorithms begin from multiple 2D textures and require consistent statistics over these multiple views; therefore they can model only textures without large-scale structures.

Texture Mapping: Another body of related work is texture mapping algorithms. However, globally consistent texture mapping [42] is difficult. Often, either distortions or discontinuities, or both, will be introduced. [47] addressed this problem by patching the object with continuously textured triangles. However, this approach works only for isotropic textures, and it requires careful preparation of input texture triangles obeying specific boundary conditions. In addition, since it employs relatively large triangles, the approach is less effective for texturing narrow features. Our algorithm performs moderately well on semi-anisotropic textures, and it does not require extensive preparation. Another method that has been suggested is to cover a model with irregular overlapping patches [55]. This approach works well for some but not all kinds of textures. Also, the discontinuity between adjacent texture instances are evident if the textured model is seen close up. The local parameterization method used in [55] inspired the parameterization of the algorithm presented here.

Mesh Signal Processing: In principle, we could directly generalize our planar algorithm for meshes if there existed a toolkit of general mesh signal processing operations. Unfortunately, despite promising recent efforts [30, 58], mesh signal processing still remains largely an open problem; [58] works only for spheres and [30] is designed for filtering geometries and functions over meshes, not for general mesh signal processing operations such as convolution.

6.2 Algorithm

Our algorithm uses the same framework as our planar algorithm . To make the exposition clear, we first summarize that algorithm in Table 6.2. We then describe our extensions. The core of our planar algorithm uses spatial neighborhoods defined on rectangular grids to characterize image textures. In this chapter, we generalize the definition of spatial neighborhood so that it can be used for producing textures over general surfaces. We parameterize mesh surfaces using local coordinate orientations defined for each mesh vertex and a scale factor derived from vertex density. We also change the codes for building/reconstructing mesh pyramids, as well as the order for traversing output pixels. For clarity, we mark a $*$ at the beginning of each line in Table 6.2 that needs to be extended or replaced.

In the rest of this section, we present our extensions following the order in the pseudocode in Table 6.2. For easy comparison we also summarize our new algorithm in Table 6.3.

Symbol	Meaning
I_a	Input texture image
I_s	Output texture image
M_s	Output textured mesh
G_a	Gaussian pyramid built from I_a
G_s	Gaussian pyramid built from I_s or M_s
p_i	An input pixel in I_a or G_a
p	An output pixel/vertex in I_s/G_s
$P_s(p)$	Flattened patches around p
$N(p)$	Neighborhood around the pixel p
$G(L)$	Lth level of pyramid G
$G(L, p)$	Pixel p at level $G(L)$
$\vec{s}, \vec{t}, \vec{n}$	Local texture coordinate system: texture right, texture up, and surface normal
{RxC,k}	neighborhood containing k levels, with sample density RxC pixels at the top level

Table 6.1: *Table of symbols*

6.2.1 Preprocessing

The preprocessing stage consists of building multiresolution pyramids and initializing output texture colors (Table 6.2, line 1 to 3, and Table 6.3, line 1 to 5). For texturing a surface we add two more steps to this stage: retiling meshes and assigning a local texture orientation. Let us consider each step in this stage.

In Table 6.2, an image pyramid is built for both the input and output texture image. In the present algorithm, we build the image pyramid G_a via standard image processing routines, as in our planar algorithm . However, for output mesh M_s, we construct the corresponding pyramid G_s using mesh simplification algorithms [68]. Note that at this stage G_s only contains a sequence of simplifications of the geometry of M_s; the vertex colors are not yet assigned.

After building the mesh pyramid G_s, we retile the surfaces on each level using Turk's algorithm [68]. This retiling serves two purposes: 1) it uniformly distributes the mesh vertices, and 2) the retiling vertex density, a user-selectable parameter, determines the scale of the synthesized texture relative to the mesh geometry (Figure 6.2, see Section 6.2.3 for details). The retiling progresses from higher to lower resolutions, and we retile each

```
function Is ← ImageTextureSynthesis(Ia, Is)
*  InitializeColors(Is);
   Ga ← BuildImagePyramid(Ia);
*  Gs ← BuildImagePyramid(Is);
   foreach level L from lower to higher resolutions of Gs
*    loop through all pixels p of Gs(L) in scanline order
       C ← FindBestMatch(Ga, Gs, L, p);
       Gs(L, p) ← C;
*  Is ← ReconPyramid(Gs);
   return Is;

function C ← FindBestMatch(Ga, Gs, L, p)
*  Ns ← BuildImageNeighborhood(Gs, L, p);
   Na^best ← null;    C ← null;
   loop through all pixels pi of Ga(L)
     Na ← BuildImageNeighborhood(Ga, L, pi);
     if Match(Na, Ns) > Match(Na^best, Ns)
       Na^best ← Na;    C ← Ga(L, pi);
   return C;
```

Table 6.2: *Pseudocode of our planar algorithm (Chapter 2). Lines marked with a $*$ need to be replaced or extended for synthesizing surface textures.*

lower resolution mesh with one quarter of the number of vertices of the immediate higher resolution so that the relative sample densities of adjacent pyramid levels relative to one another are compatible between image pyramid G_a and mesh pyramid G_s.

After retiling, we initialize colors of each level of G_s by assigning random colors from the corresponding level in G_a. This initialization method naturally equalizes the color histograms between G_a and G_s, thereby improving the resulting texture.

The next step is to assign a local coordinate frame for each vertex in the mesh pyramid. This coordinate frame, which determines the texture orientation, consists of three orthogonal axes $\vec{s}$ (texture right), $\vec{t}$ (texture up), and $\vec{n}$ (surface normal). These three axes are tacitly assumed to be $\vec{x}$, $\vec{y}$, $\vec{z}$ for planar image grids. For general surfaces it is usually impossible to assign a globally consistent local orientation (e.g. a sphere). In other words, singularities are unavoidable.

Our solution to this problem is to assign the $\vec{s}$ vectors randomly, at least for isotropic

function $M_s \leftarrow$ SurfaceTextureSynthesis(I_a, M_s)

1 $G_a \leftarrow$ BuildImagePyramid(I_a);
2* $G_s \leftarrow$ BuildMeshPyramid(M_s);
3* RetileMeshes(G_s);
4* AssignTextureOrientation(G_s);
5* InitializeColor(G_s);
6 **foreach** level L from lower to higher resolutions of G_s
7*   **loop** through all pixels p of $G_s(L)$ in random order
8       $C \leftarrow$ FindBestMatch(G_a, G_s, L, p);
9       $G_s(L, p) \leftarrow C$;
10* $M_s \leftarrow$ ReconMeshPyramid(G_s);
11 **return** M_s;

function $C \leftarrow$ FindBestMatch(G_a, G_s, L, p)

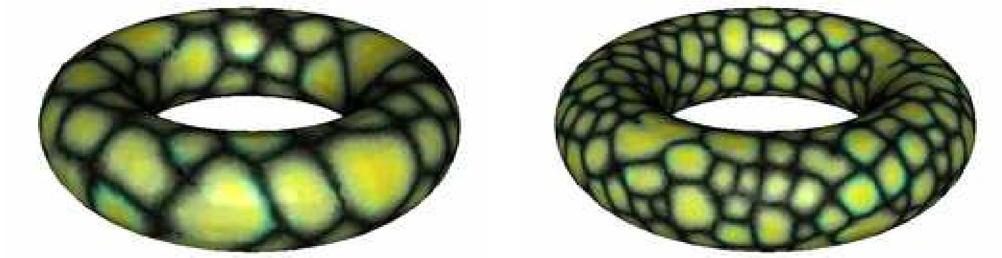
12* $N_s \leftarrow$ BuildMeshNeighborhood(G_s, L, p);
13 $N_a^{best} \leftarrow$ null; $C \leftarrow$ null;
14 **loop** through all pixels p_i of $G_a(L)$
15    $N_a \leftarrow$ BuildImageNeighborhood(G_a, L, p_i);
16   **if** Match(N_a, N_s) $>$ Match(N_a^{best}, N_s)
17       $N_a^{best} \leftarrow N_a$; $C \leftarrow G_a(L, p_i)$;
18 **return** C;

function $N_s \leftarrow$ BuildMeshNeighborhood(G_s, L, p)

19* $P_s(p) \leftarrow$ FlattenLocalPatch(G_s, L, p, $\vec{s}$, $\vec{t}$, $\vec{n}$);
20* $N_s \leftarrow$ ResampleNeighborhood($P_s(p)$);
21 **return** N_s;

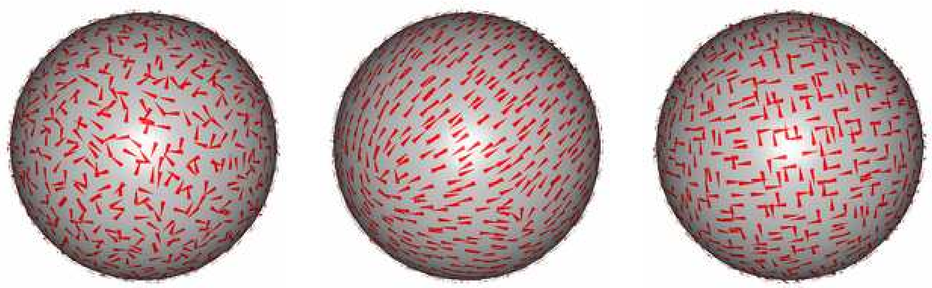
Table 6.3: *Pseudocode of our algorithm. Lines marked with a $*$ indicate our extensions from the algorithm in Table 6.2. Note that in our current implementation we only use Gaussian pyramids for meshes; therefore line 10 simply extracts the highest resolution from G_s.*

textures. One of the contributions of this chapter is the recognition that, in the context of a texture synthesis algorithm that searches a texture sample for matching neighborhoods, rotating the $\vec{s}$ and $\vec{t}$ between the searches conducted at adjacent mesh vertices does not significantly degrade the quality of the match found as long as the input texture is reasonably isotropic. (Although isotropic textures are by definition rotationally invariant, this does not immediately imply that we can generate isotropic textures by matching neighborhoods in a



(a) 24576 vertices (b) 73728 vertices

Figure 6.2: *The retiling vertex density determines the scale for texture synthesis. Textured torus with (a) 24576 vertices and (b) 73728 vertices.*



(a) (b) (c)

Figure 6.3: *Orienting textures via relaxation. The red arrows illustrate the $\vec{s}$ directions over the mesh vertices: (a) random (b) 2-way symmetry (c) 4-way symmetry.*

rotationally invariant way.)

For anisotropic textures this solution does not work. Therefore, we either let the user specify the texture direction as in [55], or we automatically assign $\vec{s}$ and $\vec{t}$ using a relaxation procedure. The goal of this relaxation procedure is to determine the local texture orientation from the directionality of the input texture. That is, given an n-way symmetric texture, we orient $\vec{s}$ vectors so that to the extent possible, adjacent $\vec{s}$ vectors form angles of integer multiples of $\frac{360}{n}$ degrees. The relaxation algorithm begins by assigning random orientations for the lowest resolution level of G_s. It then proceeds from lower to higher resolutions of G_s, and at each resolution it first initializes $\vec{s}$ vectors by interpolating from the immediate lower resolution. Each $\vec{s}$ is then aligned, iteratively, with respect to its spatial neighbors (at the current and lower resolutions) so that the sum of individual mis-registration is minimized. The amount of mis-registration for each $\vec{s}$ at vertex p is calculated by the following error function:

$$E = \sum_{q \text{ near } p} \left| \phi_{s_{qp}} - \text{round}(\frac{\phi_{s_{qp}}}{\frac{360}{n}}) \times \frac{360}{n} \right|^2,$$

where n is the degree of symmetry of the input texture, and $\phi_{s_{qp}}$ is the angle between $\vec{s_p}$ ($\vec{s}$ of vertex p) and the projection of $\vec{s_q}$ on the local coordinate system of vertex p. The idea of using energy minimization for assigning local directions is not new. A similar function is used in [49], with the following differences to our approach: (1) we set $\vec{s}$ and $\vec{t}$ to be always orthogonal to each other, and (2) we use modular arithmetic in the function so that it favors adjacent $\vec{s}$ vectors forming angles that are multiples of $\frac{360}{n}$ degrees. Our approach is also similar to [35], but we use a slightly different functional, and we do not require the direction fields to align with the principle surface curvatures. Examples of orienting 2-way and 4-way symmetric textures (e.g. stripes and grid) are shown in Figure 6.3 (b) and (c).

6.2.2 Synthesis Order

The scanline synthesis order in Table 6.2 (line 5) cannot be directly applied to mesh pyramid G_s since its vertices do not have rectangular connectivity. One solution might be to use the two-pass algorithm for constrained synthesis (Chapter 3), growing textures spirally outward from a seed point. However, there is no natural seed point for meshes of arbitrary topology. Surprisingly, we have found that our algorithm works even if we visit pixels of $G_s(L)$ in random order. Thus, we use a modified two-pass algorithm, as follows. During the first pass, we search the input texture using a neighborhood that contains only pixels from the lower resolution pyramid levels (except the lowest resolution where we randomly copy pixels from the input image). This pass uses the lower resolution information to "extrapolate" the higher resolution levels. In the second pass, we use a neighborhood containing pixels from both the current and lower resolution. In both passes, on each level, the neighborhoods used are symmetric (noncausal). We alternate these two passes for each level of the output pyramid, and within each pass we simply visit the vertices in a random order. In our experience this random order works as well as the spiral order used for constrained synthesis (Chapter 4), and it produces slightly worse textures than scanline order

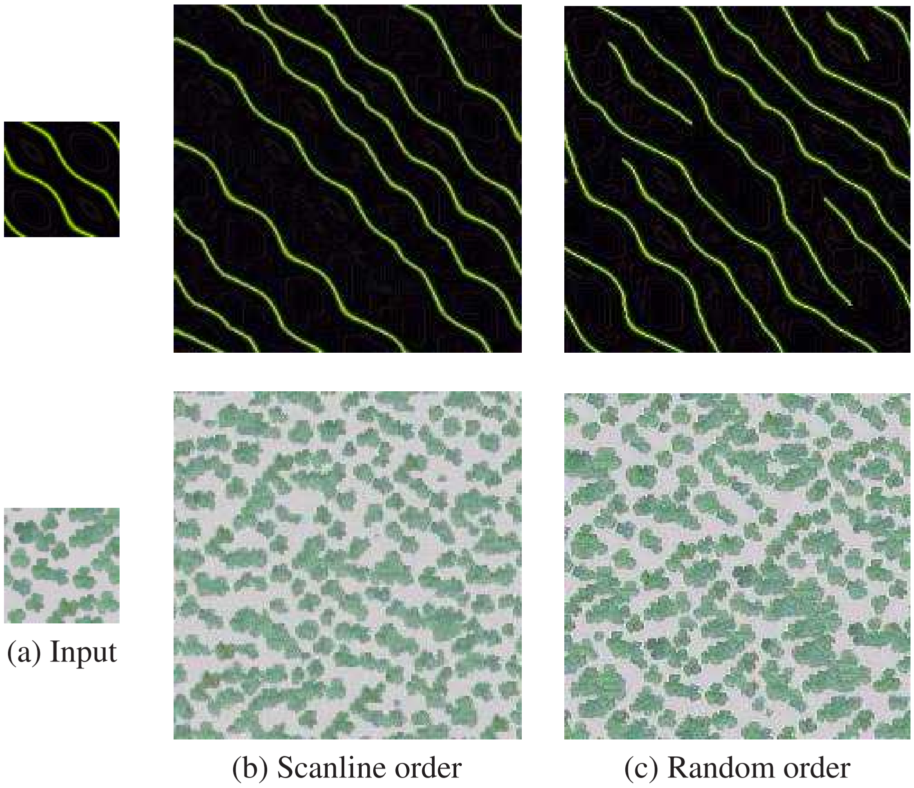


Figure 6.4: *Texture synthesis order. (a) Input textures (b) Results with scanline-order synthesis (c) Results with random-order synthesis. For textures without scanline dependencies, we have found that random-order works well.*

only for patterns with scanline dependencies. An example comparing different synthesis orders is shown in Figure 6.4.

6.2.3 Neighborhood Construction

Table 6.2 characterizes textures using spatial neighborhoods (line 10 and 13). These neighborhoods are planar and coincident with the pyramid grids. For meshes, however, we have to generalize neighborhoods so that they are defined over general surfaces having irregular vertex positions.

We build mesh neighborhoods by flattening and resampling the mesh locally (Figure 6.5). To build the neighborhood around an output vertex p, we first select and flatten a

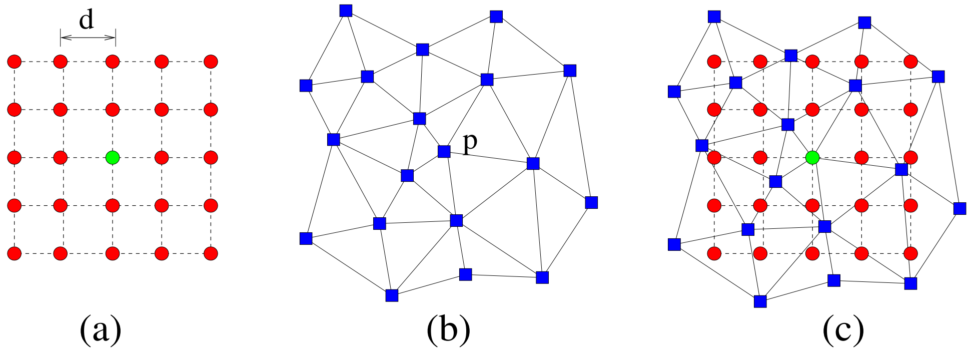
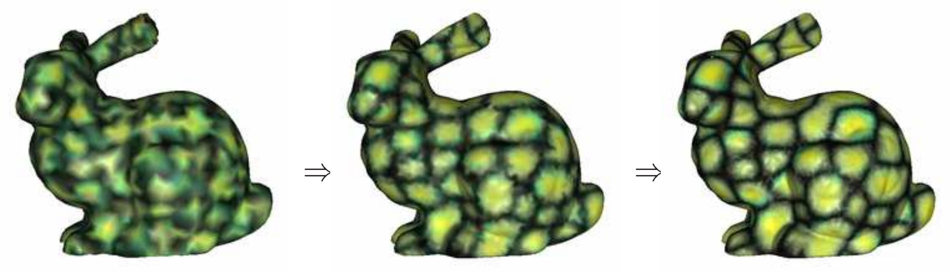


Figure 6.5: *Mesh neighborhood construction. (a) neighborhood template (b) flattened patch of the mesh (c) neighborhood template embedded in the flattened patch.*

set of nearby vertices, henceforth called a patch, so that they fully cover the given neighborhood template (Figure 6.5 (a,b)). We then resample the flattened patch (Figure 6.5 (c)) by interpolating the color of each neighborhood pixel (red circles) from the vertex colors of the patch triangle (blue squares) that contains that pixel. Before flattening, the neighborhood template is scaled with a constant $d = \sqrt{2 \times A}$, where $A =$ average triangle area of $G_s(L)$, so that the sampling density of the neighborhood and mesh vertices are roughly the same[2]. Leaving d much larger than $\sqrt{2 \times A}$ would either introduce aliasing during resampling or would waste mesh vertices by necessary filtering; if d were too small, the neighborhood would be poorly represented since most of its samples would come from the same triangle.

The method we use for flattening patches is taken from [55]. First, we orthographically project the triangles adjacent to p onto p's local texture coordinate system. Starting from these seed triangles, we grow the flattened patch by adding triangles one at a time until the neighborhood template is fully covered. Triangles are added in order of increasing distance from the seed triangles, and we determine the position of each newly added vertex using the heuristic in [43, Section 3.1.4]. Note that the flattening process can introduce flipped triangles. If this happens, we stop growing patches along the direction of flipping. This might in turn produce patches that only partially cover the neighborhood template. In this case, we assign a default color (the average of I_a) to the uncovered neighborhood pixels. Another solution might be to use smaller neighborhoods for highly curved areas. However,

[2]We choose this formula so that if the mesh is a regular planar grid, the neighborhood will be scaled to align exactly with the grid vertices.



(a) 2186 vertices, 4368 faces (b) 8715 vertices, 17426 faces (c) 34938 vertices, 69868 faces

Figure 6.6: *Multi-resolution surface texture synthesis. The synthesis progresses from lower to higher resolutions, and information at lower resolution meshes is used to constrain the growth of textures at higher resolutions.*

since a new neighborhood size would require a new VQ codebook (Chapter 3), this implies building multiple codebooks for tree-structured VQ acceleration. Fortunately, since we only use small neighborhoods, flipping rarely happens.

We construct multiresolution neighborhoods in a similar fashion. For each vertex p of pyramid G_s, we first find the corresponding parent faces at lower resolution pyramid levels by intersecting the normal $\vec{n}$ of p with the coarse meshes. We project each parent face orthographically with respect to p's $\vec{s}$, $\vec{t}$, $\vec{n}$, and we grow a flattened patch from the parent face as in the single-resolution case. The collection of flattened patches $P_s(p)$ is then resampled to obtain the multiresolution neighborhood $N(p)$[3].

6.3 Results

Our first example, illustrating the multiresolution synthesis pipeline, is shown in Figure 6.6. The synthesis progresses from lower to higher resolutions, and information at lower resolution meshes is used to constrain the growth of texture patterns at higher resolutions. All synthesis results shown in this chapter are generated with 4-level Gaussian pyramids, with neighborhood sizes $\{1x1,1\}$, $\{3x3,2\}$, $\{5x5,2\}$, $\{7x7,2\}$ (Table 6.1), respectively,

[3]If $\vec{n}$ of p does not intersect a particular coarse mesh (e.g. it lies on a crease), we simply skip flattening at that level. Instead we assign a default color to the neighborhood pixels that are not covered, as in the flipping case.

from lower to higher resolutions.

Texture Orientation: Figure 6.7 demonstrates the performance of our algorithm on textures with varying amounts of anisotropy. The model we use, a sphere, is the simplest non-developable object that has no consistent texture parameterization. Despite this, many textures are sufficiently isotropic that they can be synthesized using random texture orientations (columns (a) and (b)). For highly anisotropic textures (column (c)), a random parameterization may fail, depending on the nature of the textures (column (d)). We can retain the anisotropy by assigning consistent surface orientations either by hand (column (e) and (f)) or using our iterative relaxation procedure (column (g)).

Model Geometry & Topology: Several textured meshes with different topologies and geometries are shown in Figure 6.9. As shown, the algorithm generates textures without discontinuity across a variety of surface geometries and topologies, even across fine features such as the bunny ear (Figure 6.8). The algorithm can also be used to synthesize surface attributes other than colors such as displacement maps (the mannequin model in Figure 6.9).

Computation Time: By using an efficient data structure for meshes (we use the quad-edge data structure [29], although other approaches are possible), we achieve linear time complexity with respect to the neighborhood sizes for both the flattening and resampling operations. In our C++ implementation running on a 450 MHz Pentium II machine, the timing for texturing the sphere in Figure 6.7 is as follows: relaxation (30 iterations) - 85 seconds, synthesis with exhaustive search - 695 seconds, and synthesis with tree-structured VQ acceleration - 82 seconds.

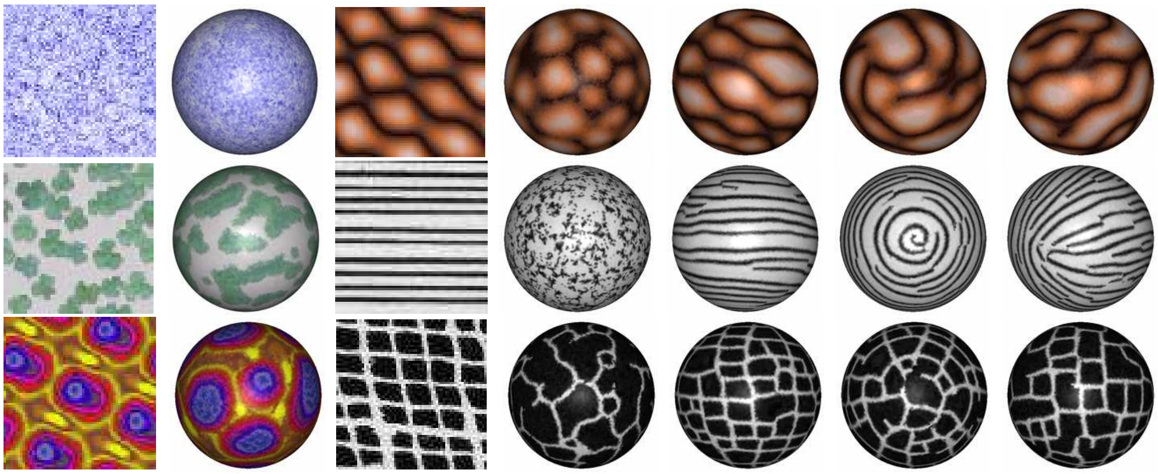
6.4 Conclusions and Future Work

We have presented extensions of our planar algorithm that permit us to synthesize textures over surfaces of arbitrary topology, beginning with a rectangular texture sample. The most significant of these extensions are that we traverse output vertices in a random order, thus allowing texture synthesis for general meshes, and we parameterize meshes with a user-selectable scale factor and local tangent directions at each mesh vertex. We define mesh neighborhoods based on this parameterization, and we show that this approach works over a variety of textures. Specifically, we synthesize isotropic textures with random local orientations, while generating anisotropic textures with local directions that are either hand-specified or automatically determined by our relaxation procedure.

Our approach has several limitations. Since it is an extension of our planar algorithm it only works for texture images; therefore it is not as general as [55] which can paste any image onto a mesh model. However for the class of textures that can be modeled by our planar algorithm , our approach usually produces continuous surface textures with less blocky repetitions. In addition, for textures that are not well modeled by our planar algorithm , we could generate better results by combining our surface-synthesis framework with other improved texture synthesis algorithms such as [1]. Finally, our representation of the output as a retiled polygonal mesh with vertex colors may not be desirable in cases where we would like to preserve the original mesh geometry. In such cases the output can be mapped back onto the original model in a post-process by resampling, such as in [12].

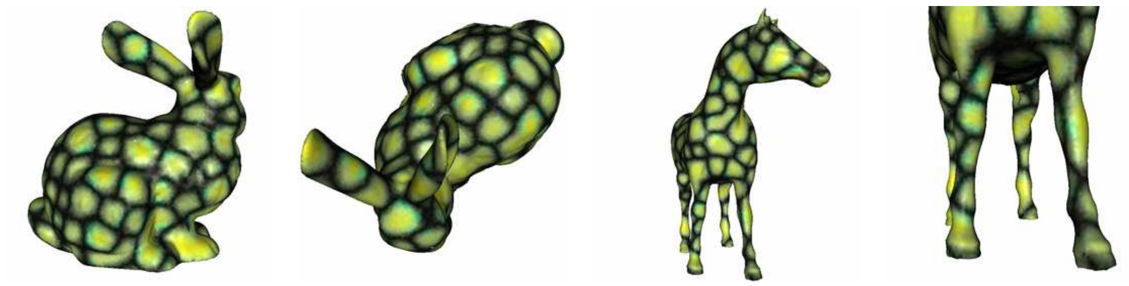
In concurrent work, Turk has developed a similar approach for synthesizing textures over surfaces [69]. The primary differences between [69] and our work are as follows: (1) we have used random as well as symmetric vector fields for certain textures, whereas [69] always creates a smooth vector field, (2) instead of a sweeping order, we visit mesh vertices in random order, (3) the two approaches use different methods for constructing mesh neighborhoods; [69] uses surface marching while we use flattening and resampling, and (4) we do not enforce a explicit parent-child relationship between mesh vertices at adjacent resolutions.

We envision several possible directions for future work. Although our relaxation procedure can assign reasonable local orientations for many anisotropic but symmetric textures, it remains an open problem for which symmetry classes local orientations can be assigned in this way. Another future direction is to use a variant of our algorithm to transfer textures (either colors or displacements) from one scanned model [41] to another mesh model. This could be done by replacing the input image I_a Table 6.3 with an input mesh model, and changing line 1 and 15 in Table 6.3 to **BuildMeshPyramid** and **BuildMeshNeighborhood**, respectively. Finally, our definition of mesh neighborhoods might be applicable to other signal processing operations over meshes such as convolution, filtering, and pattern matching.



(a) (b) (c) (d) (e) (f) (g)

Figure 6.7: *Texture synthesis over a sphere uniformly tesselated with 24576 vertices and 49148 faces. (a) Isotropic textures of size 64x64. (b) Synthesis with random orientations. (c) Anisotropic textures of size 64x64. (d) Synthesis with random orientations. (e) Synthesis with $\vec{s}$ and $\vec{t}$ vectors at each vertex parallel to longitude and altitude of the sphere. (f) The polar views of (e), showing the singularity. (g) Synthesis with orientation computed by our relaxation procedure (Section 6.2.1). The top two textures are generated using 2-way symmetry (Figure 6.3 (b)), while the bottom one is generated using 4-way symmetry (Figure 6.3 (c)).*



(a) (b) (c) (d)

Figure 6.8: *Different views of textured fine model features. (a) Bunny ears, back view. (b) Bunny ears, top view. (c) Horse legs. (d) Horse legs, close up view.*

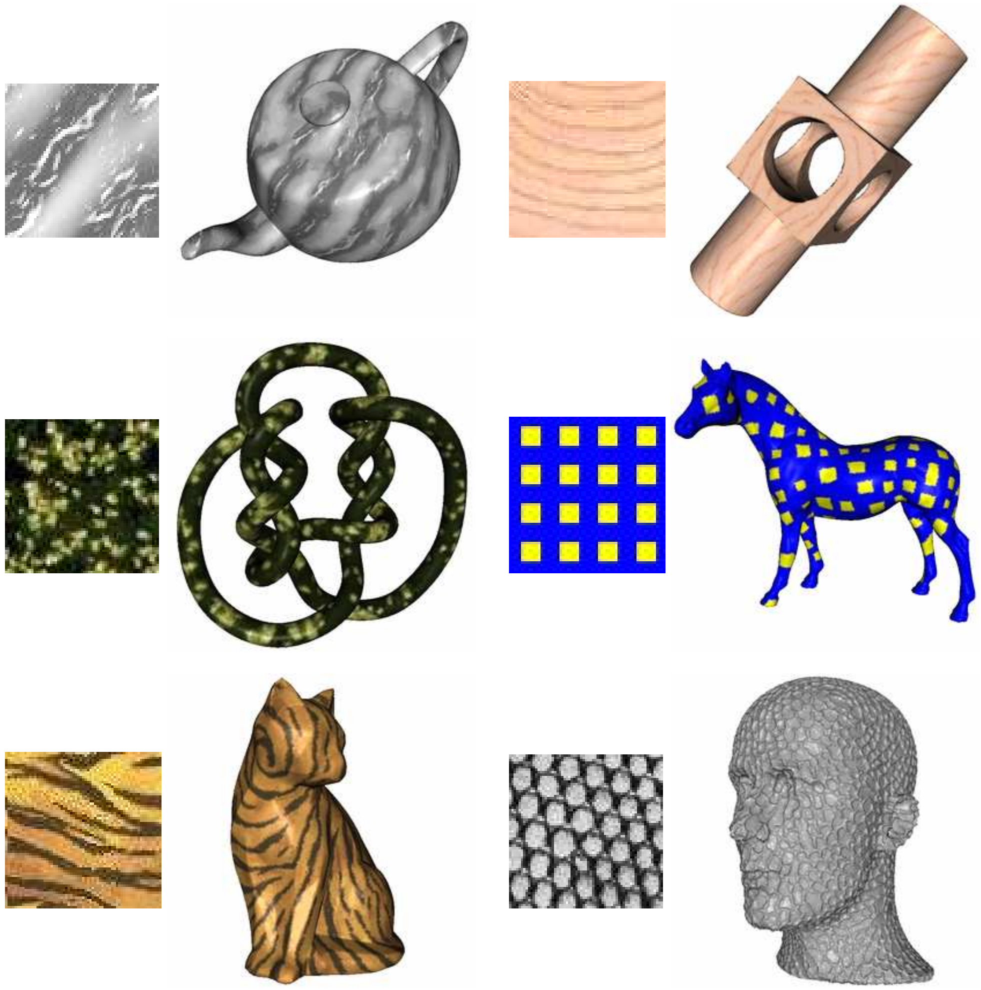


Figure 6.9: *Surface texture synthesis over different models. The small rectangular patches (size 64x64) are the input textures, and to their right are synthesis results. In all examples the textures are used to modulate the colors, except the last one where the texture is used for displacement mapping. Texture orientation and mesh sizes: teapot (no symmetry, 256155 vertices, 512279 faces), mechanical part (2-way symmetry, 49180 vertices, 98368 faces), knot (random, 49154 vertices, 98308 faces), horse (4-way symmetry, 48917 vertices, 97827 faces), cat (2-way symmetry, 50015 vertices, 100026 faces), and mannequin (no symmetry, 256003 vertices, 512002 faces).*

Chapter 7

Texture Synthesis from Multiple Sources

In previous chapters we have presented algorithms for synthesizing textures of different physical forms, including images, spatial temporal volumes, and articulated motions. However, all these algorithms take a single texture as input and generate an output texture with similar visual appearance. Although the output texture can be made of arbitrary size and duration, those techniques can at best replicate the characteristics of the input texture.

In this chapter, we would like to do something more interesting. We would like to have algorithms that can *create* new textures. We present approaches that take multiple textures with probably different characteristics, and synthesize new textures with combined visual appearance of all the inputs.

In the rest of this chapter, we first formulate the problem of synthesizing textures from multiple sources in Section 7.1. We define the problem in a general setting, and show that two specific variations of it are particularly useful: synthesizing solid textures from multiple 2D views and texture mixtures. We review previous work in Section 7.2. We present our algorithm in Section 7.3, and apply it to solid texture synthesis from 2D views (Section 7.4) and texture mixtures (Section 7.5). Although seemingly unrelated, these applications are actually slight variations of the same basic algorithm and can even share the same implementation.

7.1 Multi-Source Texture Synthesis

The goal of multi-source texture synthesis can be stated as follows: Given several sample textures, synthesize a new texture that has a combined appearance of the input samples. The meaning of "combined appearance" will depend on the specific applications. We are interested in two particular applications that are most useful for computer graphics:

7.1.1 Solid Texture Synthesis from Multiple 2D Views

Solid textures can be used to simulate the surface appearance of objects carved out of materials such as marble and wood. Since solid textures define colors for each 3D grid point, they can bypass the texture mapping process and avoids mapping distortion/discontinuity completely. However, solid textures are not easy to acquire. Unlike image textures, there is no easy way to "scan" an solid texture from real world materials; therefore most existing approaches use procedural synthesis for generating solid textures.

A more flexible approach is to generate a solid texture from 2D views (Figure 7.1). By allowing these 2D views to come from different image sources such as scanned photographs, we can synthesize arbitrary solid textures in a more general setting. For example, to produce a marble texture we can take three photographs of a marble surface, and then ask the algorithm to synthesize a cube of marble texture for us. The challenge for this approach is to design an algorithm that can generate a solid texture with matching visual appearance with respect to all the 2D views.

We consider the problem of generating solid textures as a special case of multi-source texture synthesis. In this specific application, the 2D views are the several image textures of a specific material (such as marble), and the synthesized solid texture will have a "combined appearance" of the inputs in the sense that the solid texture looks similar to each 2D view from the corresponding viewing directions. In the most common case the views consist of three images describing the solid texture from orthogonal directions.

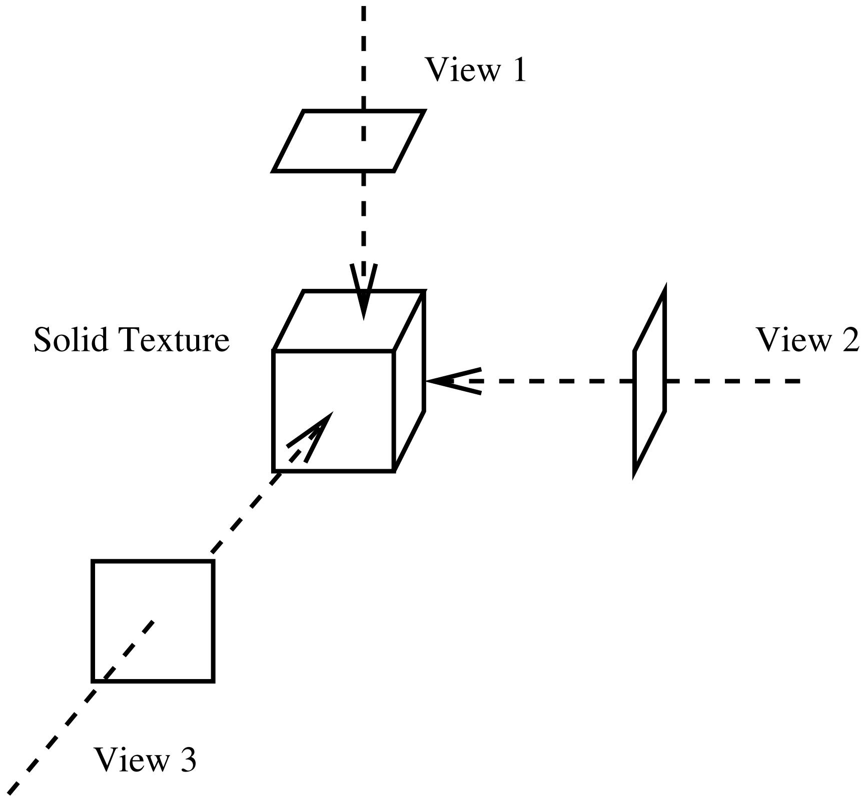


Figure 7.1: *Generating a solid texture from multiple 2D views. Given multiple 2D images, the goal is to generate a solid texture that has consistent characteristics with every image viewed at the corresponding directions.*

7.1.2 Texture Mixture

A texture mixture simultaneously captures the characteristics of several different input textures. This can be useful for creating textures that do not previously exist. For example, given an image of animal fur and belly, can we generate a new texture that has a combined appearance and looks like the transition region between animal back and belly?

Texture mixture can be considered as a special case of multi-source texture synthesis. However, in this situation the notion of "combined appearance" is less well defined than the solid texture synthesis application, since the definition of "combined appearance" is highly case dependent. For example, one possible way to generate a texture mixture is to blend smoothly from one texture to another such as the animal fur and belly example. However, one can also imagine other kinds of mixtures, such as patterns from one texture and colors from another.

7.2 Previous Work

7.2.1 Solid Texture Synthesis

Most previous approaches for generating solid textures use specialized procedures [50, 81, 18]. These approaches work well for specific textures such as marble, cloud, and wood. In addition they can be executed very efficiently by careful software coding or hardware acceleration. However, procedural texture synthesis cannot model general textures and for those textures it could model, it usually requires some effort to get the parameters right.

An alternative is to generate solid textures automatically from examples. Although there are many 2D image texture synthesis algorithms that can do this, the analogy cannot be easily carried over to 3D since it is usually impossible to "scan" a real solid texture (such as a piece of marble or wood). A more feasible option is to provide multiple 2D views of the texture, and require the algorithm to generate a new texture that has consistent statistics for all those 2D views. [33] synthesizes solid textures by equalizing histograms between the input 2D views and the output 3D volume. This approach can simulate only homogeneous textures. [15] improves the algorithm in [33] by taking into account the Fourier transform coefficients. Although it can reproduce some anisotropy of textures (such as wood grain), it still cannot generate textures with dominating structures.

7.2.2 Texture Mixture

There has been only a few methods related to generating texture mixtures. They are primarily distinguished by the parameter space where different textures are mixed together. For example, [33, 54] synthesizes texture mixtures in the domain of steerable pyramids, and [2] uses statistical learning tree to mix textures. However, none of those approaches generate compelling texture mixtures that truly convey a sense of capturing multiple input texture characteristics.

7.3 Algorithm

Our algorithm for synthesizing textures from multiple sources is extended from the basic algorithm presented in Chapter 2 and Chapter 3. To make the exposition clear, we summarize the algorithm in Table 7.1 where we mark a $*$ at the beginning of each line which is our new contribution. In the rest of this section we present our new contributions following the order of the pseudo-code in Table 7.1.

7.3.1 Input

The input of the algorithm consists of several texture sources $\{I_a\}$. Those sources can be different views of a solid texture, or different textures for generating a texture mixture. Each source I_a is associated with a weight I_w, which is of the same size as I_s. Those weights $\{I_w\}$ are user selected parameters specifying how the input sources should be mixed together. For example, if the input consist of two sources and we would like our mixture result to look more similar to the first one, we can assign heavier weight to the first texture. We let each I_w to have the same size as I_s so that they can indicate spatially varying properties, such as a texture mixture that transforms gradually from one texture to another horizontally.

In addition, each source I_a is also associates with a neighborhood parameter, specifying the size and shape of the neighborhood used during the search process for this specific source. This can be useful for situations such as different sources of a texture mixture have different element sizes (so we would like to use larger neighborhoods for some textures), or solid texture synthesis where each view has different neighborhoods oriented with the corresponding viewing directions.

7.3.2 Initialization

The initialization phase consists of building pyramids and initializing colors (Table 7.1, lines 1 to 4). Since we have a set of texture samples $\{G_a\}$, we build separate pyramids for each sample texture (line 1). The output pyramid G_s is constructed as in Chapter 2. Note that we build all the pyramids with the same number of levels so that they are compatible

function $I_s \leftarrow$ MultiSourceTextureSynthesis($\{I_a\}$, $\{I_w\}$, I_s)
1* $\{G_a\} \leftarrow$ BuildPyramid($\{I_a\}$);
2* $\{G_w\} \leftarrow$ BuildPyramid($\{I_w\}$);
3 $G_s \leftarrow$ BuildPyramid(I_s);
4* InitializeColors($\{G_a\}$, $\{G_w\}$, G_s);
5 **foreach** level L from lower to higher resolutions of G_s
6   **loop** through all pixels p of $G_s(L)$
7*   **iterate** several times
8* $C \leftarrow$ FindBestMatch($\{G_a\}$, $\{G_w\}$, G_s, L, p);
9          $G_s(L,p) \leftarrow C$;
10 $I_s \leftarrow$ ReconPyramid(G_s);
11 **return** I_s;

function InitializeColors($\{G_a\}$, $\{G_w\}$, G_s)
12* i $\leftarrow$ 0;
13* **foreach** G_a in $\{G_a\}$
14*   i $\leftarrow$ i + 1;
15* $G_{s_i} \leftarrow G_s$;
16*   InitializeColors(G_a, G_{s_i}); *% same as in Table 2.2*
17* $G_s \leftarrow \sum_{i=1}^{N} G_{w_i} \times G_{s_i}$; *% N is the number of elements in* $\{G_a\}$

function $C \leftarrow$ FindBestMatch($\{G_a\}$, $\{G_w\}$, G_s, L, p)
18* i $\leftarrow$ 0;;
19* **foreach** G_a in $\{G_a\}$
20*   i $\leftarrow$ i + 1;
21* $C_i \leftarrow$ FindBestMatch(G_a, G_s, L, p); *% same as in Table 2.2*
22* $C \leftarrow \sum_i G_{w_i}(L,p) \times C_i$;
23 **return** C;

Table 7.1: *Pseudocode of the multi-source texture synthesis algorithm. For clarity, we mark a $*$ at the beginning of each line that is different from the basic algorithm (Table 2.2). The symbol $\{\}$ indicates sets, such as $\{G_a\}$ and $\{C_i\}$.*

with each other.

Since we have multiple texture sources, the color initialization step also needs change (line 4). In our current implementation, we simply equalize the histogram of G_s with the weighted average of the histograms of $\{G_a\}$. That is, the histogram of L^{th} level of G_s

is equalized with respect to the weighted average of the histograms at L^{th} levels of $\{G_a\}$ using the L^{th} levels of weights $\{G_w\}$.

7.3.3 Synthesizing One Pixel

We now discuss how to determine each output pixel/voxel value (Table 7.1, line 7 to 9). Recall that in our synthesis algorithm from a single source (Chapter 2), each output pixel is determined so that the local similarity between the input and output textures is preserved as much as possible. We would like to achieve the same goal for multi-source texture synthesis. However, since we now have more than one input textures we have to pick the output pixel value so that it preserves local similarity simultaneously with all the input sources.

Mathematically, for each output sample p we would like to find a set of input pixels $\{p_i\}$ so that the following error function is minimized:

$$E(p, \{p_i\}) = \sum_i w_i \times (\|p - p_i\|^2 + \|N(p) - N(p_i)\|^2) \tag{7.1}$$

where i runs through all the input textures, p and each p_i are the output and matching input pixel colors, and $N(p)$, $N(p_i)$ are their neighborhoods (defined as in Chapter 2). The error function is computed as a weighted sum of the L_2 norm between $\{p, N(p)\}$ and $\{p_i, N(p_i)\}$, and the weights $\{w_i\}$ specify the relative importance of the input textures. Specifically, each w_i is equal to $G_{w_i}(L, p)$ where G_{w_i} is the i^{th} input weight pyramid and L is the level at which p resides.

To minimize the error function $E(p, \{p_i\})$, we need to determine the values p and $\{p_i\}$ so that the sum on the right hand side of Equation 7.1 is minimized. We first present a solution when there is only one source, and propose a general solution when multiple sources are present.

7.3.3.1 One Source

When only one source is present, we can directly minimize Equation 7.1 as follows: simply choose the p_i such that $\|N(p) - N(p_i)\|^2$ is minimal, and set p to be equal to p_i. This is exactly our algorithm presented in Chapter 2.

7.3.3.2 Multiple Sources

When multiple sources are present, we cannot solve Equation 7.1 directly. Instead we use an iterative procedure, alternatively setting the values of $\{p_i\}$ and p while gradually decreasing $E(p, \{p_i\})$ (Figure 7.2). At the beginning of each iteration, we fix the value p and choose $\{p_i\}$ so that each individual error term $\|p - p_i\|^2 + \|N(p) - N(p_i)\|^2$ is minimized[1]. We then keep $\{p_i\}$ fixed, and set p as the weighted mean of $\{p_i\}$ (Table 7.1, line 22). It can be easily proven that both of these steps either decrease $E(p, \{p_i\})$ or keep it the same. The iteration can be stopped when $E(p, \{p_i\})$ remains the same after two consecutive iterations, but experimentally we have found that 1 to 4 iterations are sufficient.

7.4 Solid Texture Synthesis

To synthesize solid textures from multiple views, simply specify several images consisting of different views of a hypothesized solid texture. Each view is associated with a neighborhood oriented with respect to the specific viewing direction. In the most common situation where the views are orthogonal to each other, each view is associated with a neighborhood perpendicular to one of the major axis.

To make the algorithm successful, it is important that the input views are consistent with each other. Otherwise there might be no solid texture that can satisfy the requirements of all the views. We illustrate the importance of specifying views in Figure 7.3. Given a sample texture with stripes, there are several ways to specify the views. One way is to specify the texture as 3 views with patterns oriented orthogonal to each other. Unfortunately, no solid texture can satisfy those inconsistent requirements and the synthesized result contains garbage (Figure 7.3(b)). If we specify only two views horizontally with patterns parallel to each other, then we have a consistent specification and the result solid texture will contain horizontal slices (Figure 7.3 (c)). Another way to specify views consistently is shown in Figure 7.3 (d), where the result consists of vertical bars.

[1] For the first iteration where the value p is not yet determined, we only consider the term $\|N(p) - N(p_i)\|^2$ in choosing the values of $\{p_i\}$.

Figure 7.4 shows more synthesis results. The algorithm performs reasonably well for a wide variety of textures, including stochastic textures (Figure 7.4 (a) to (f)), textures with dominating orientations (Figure 7.4 (h)), and textures with large scale structures (Figure 7.4 (i) to (l)). However, in general the synthesized texture qualities are not as good as those generated by surface texture synthesis, such as Figure 7.4 (h), (k) and (l). This is because solid texture synthesis is inherently a more difficult problem than surface synthesis, where generated solid texture needs to be consistent with multiple views. Nevertheless, to our knowledge, our approach is the first algorithm that can generate structured solid textures from 2D views.

Compared to generating textures directly over object surfaces (Chapter 6), solid texture synthesis has both advantages and disadvantages:

Surface Texture Synthesis:

- \- Textures generated for one surface cannot be reused for other surfaces.
- \- When surface curvature is large, the generated texture can suffer from distortion.
- \+ Because the texture only needs to look good at the surface, the synthesis problem is easier since essentially we care about only one viewing direction per surface location.
- \+ The algorithm usually consumes less computation time since texture colors only need to be determined at selected surface locations.
- \+ For textures with dominating orientation such as stripes, we can freely tune the direction on the surface.

Solid Texture Synthesis from 2D Views:

- \+ Once a solid texture is synthesized, it can be reused repeatedly for different objects.
- \+ The synthesized solid texture contains no distortion.
- \- Because solid textures need to look consistently with all the views, it is a more difficult problem than generating textures over a surface, especially when the number of views is large (greater than or equal to three).

- The algorithm is usually more computationally expensive since we need to generate a whole solid texture in addition to those which actually touch the final object surfaces. However, this can be overcome by generating solid textures at only those voxels close to the final surface.

- For textures with dominating orientation such as stripes, the orientation is fixed for the whole solid texture. Therefore we cannot locally tune it (for example the horse model in Figure 7.5.)

7.5 Texture Mixture

We generate texture mixtures by supplying several input textures to the algorithm. Each texture is associated with a weight image I_w, specifying how each source is going to effect the mixture result. For example, by assigning equal weighting to all the inputs, the result will be a uniform mixture of the input textures. However, by letting the weight images to be spatially varying, we can achieve special effects such as one texture gradually transforming to another (Figure 7.6).

To generate meaningful results, the input textures should have comparable colors and patterns. Figure 7.7 shows the effect of input colors on the mixture results. In case (a) we use two textures of the same kind of generate a "mixture" result. (Although this is not a real mixture since the two sources are of the same kind, we can use this as a good example to demonstrate the importance of colors.) In case (b), we invert the colors of one of the input textures. Look how drastically the result in (b) looks different from the result in (a). By inverting the color of the second input, we are essentially matching the foreground patterns (white grid) on the first texture to the background patterns on the second texture. The situation is even worse if the two inputs have disjoint color spaces, such as the red and green textures shown in (c).

To avoid this problem, we should compare the textures in a common color space. We achieve this by adding an extra color channel to the input RGB images, and use this channel with the existing RGB channels in the neighborhood search process. This extra channel servers as a common space for comparing different input textures, and it can be specified by the user as a way to control how the inputs should be compared. We usually scale this extra channel so that it has more impact in the neighborhood search process. For example, by using the intensity of the input textures as the extra channel, we could better match patterns such as lines and grids since human visual systems are more sensitive to intensity variations. The effect of this modified approach is shown in Figure 7.7 (d), where the result is generated by searching neighborhoods in the intensity space (with scale factor 100). The result shows the expected texture mixture, both in terms of color and pattern. The effect of this approach is even more evident in the linear texture transformation result shown in Figure 7.6.

Figure 7.8 and Figure 7.9 show mixture results with gray-scale textures. We use gray-scale textures so that we can focus on the effect of mixing various patterns. When the inputs contain similar patterns (Figure 7.8 (a) and (b)), the algorithm is able to generate mixtures that resemble each input. In addition, the mixture is able to connect the patterns from one texture to another, as shown on the right-most images of each row. Even for inputs with dissimilar patterns (Figure 7.8 (c) to (f) and Figure 7.9 (a) to (b)), the algorithm still performs reasonably well. Note that a slight change in one of the input pattern can have dramatic effect on the output, such as rotating the textons of one of the input (Figure 7.8 (e) and (f)). In addition to structures, we can also mix other texture properties such as randomness (Figure 7.9 (c)) or orientation (Figure 7.9 (f) and (g)). The algorithm performs less effective only when the two inputs have very different patterns (Figure 7.9 (f)).

Figure 7.10 shows mixture results with natural color textures. For cases (b) to (f) we use the intensity image as the extra color channel for neighborhood search, and for case (a) we use the red channel as the extra channel. In all cases the extra channel is scaled by a factor of 100. As shown, our algorithm is able generate interesting mixtures for a variety of natural textures even with a simple method of using the intensity image as the extra channel. Better results could be obtained by providing better information in the extra channel, such as asking a human to specify which features on the input images should be treated as similar and should be blended together. However, in general texture mixture suffers the same fundamental restrictions of the basic algorithm and might require extensive user-intervention or solving difficult computer vision problems for feature extraction.

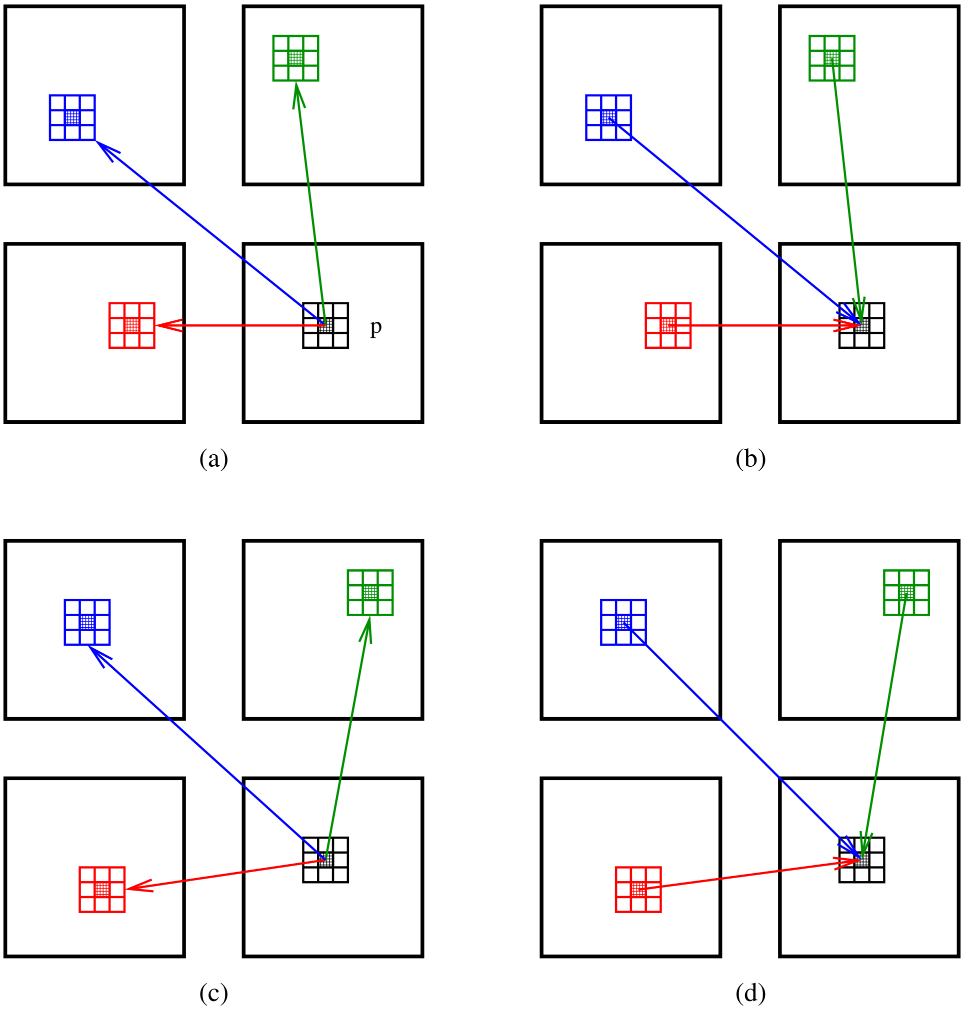


Figure 7.2: *Iterative algorithm for multi-source synthesis. Figures (a) through (d) show the iterative process for determining the value of a single output pixel p. In each figure the three input textures are shown on top and left, and the output texture is shown on the lower-right corner. Shown here are two iterations with the two phases for each iteration: iteration 1 (a,b) and iteration 2 (c, d). At the beginning of each iteration, we fix the value of p and search for best neighborhood matches from all the inputs (a). After finding the best matches, we re-compute the value of p by taking the weighted averages from the centers of the best matches (b). This process is then repeated for (c,d). Note that due to the change of value p, the locations of the best matches in (c) might be different from (a,b).*

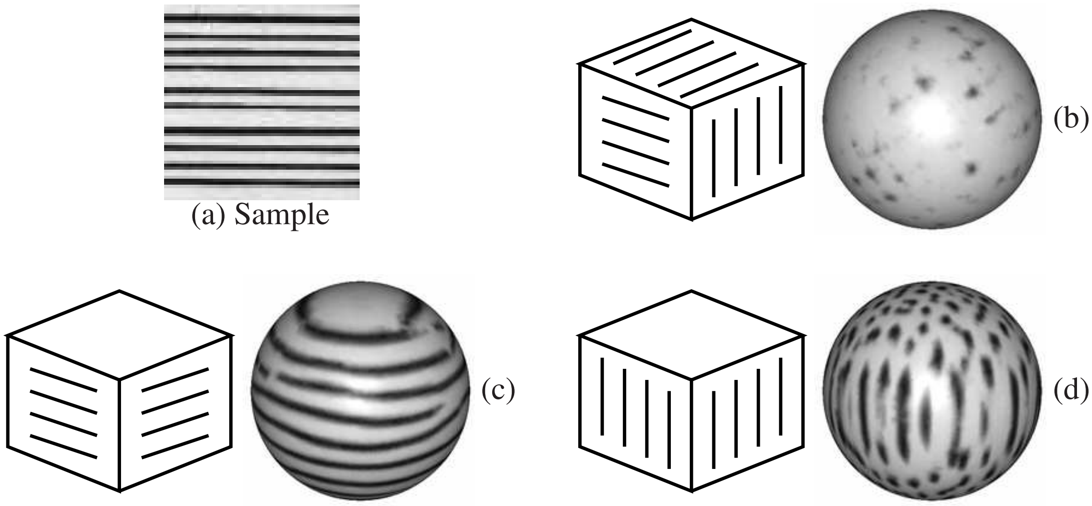


Figure 7.3: *Specifying views for synthesizing a solid texture from multiple 2D views. (a) A single texture sample used to specify the views. (b) Solid texture generated by specifying the views inconsistently. (c) Solid texture generated by specifying two views so that the result form horizontal slices. (d) Solid texture generated by specifying two views so that the result form vertical bars.*

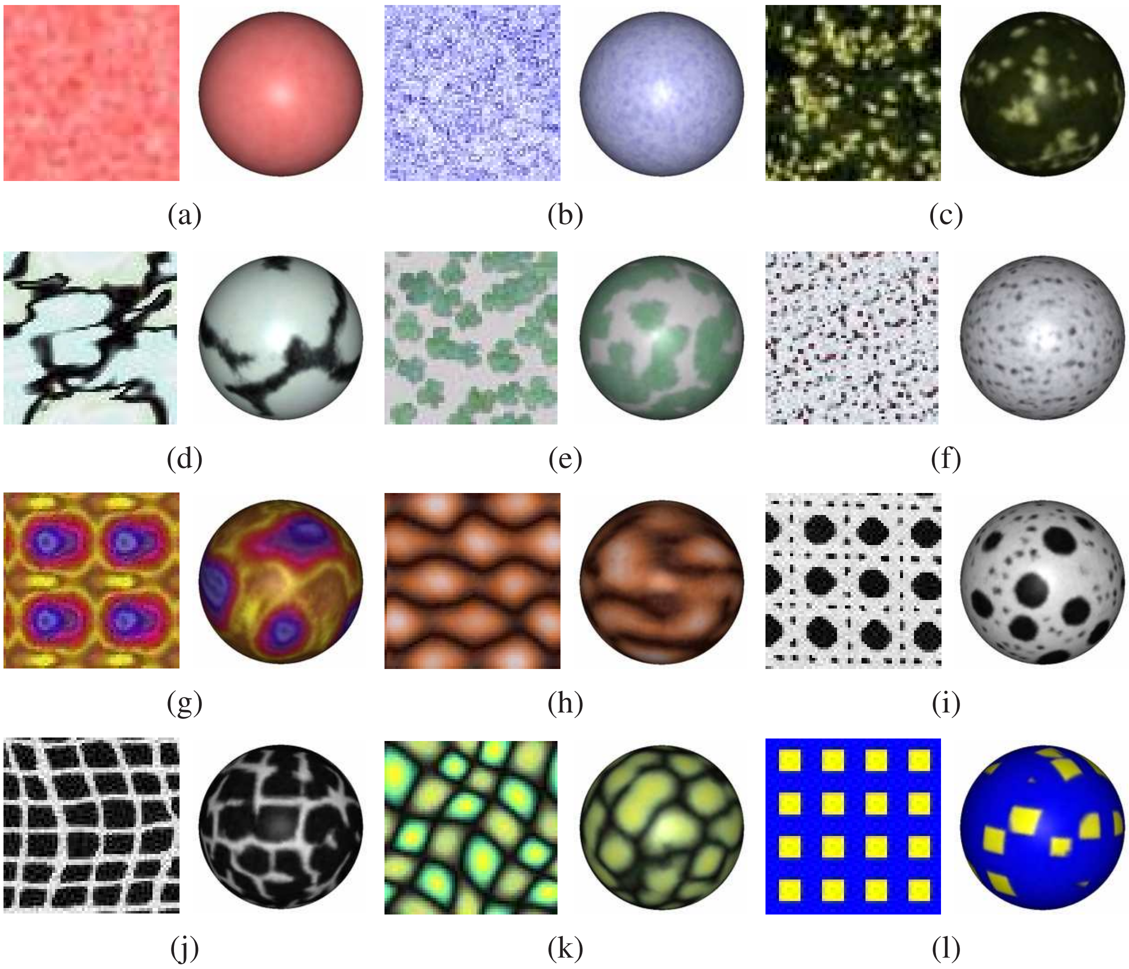


Figure 7.4: *Solid texture synthesis results. For each pair of images, the original texture is shown on the left, and the corresponding synthesis result is shown on the right. Each result is generated by carving a 3D model out of a cube of synthesized solid texture. The original images have size 64x64, and the synthesized solid textures have size 64x64x64. Except (h), which is generated by two views as in Figure 7.3 (d), all solid textures are generated from three views as in Figure 7.3 (b). There is a slight scale change between original textures and synthesis results; this is caused by displaying the results over the spheres.*

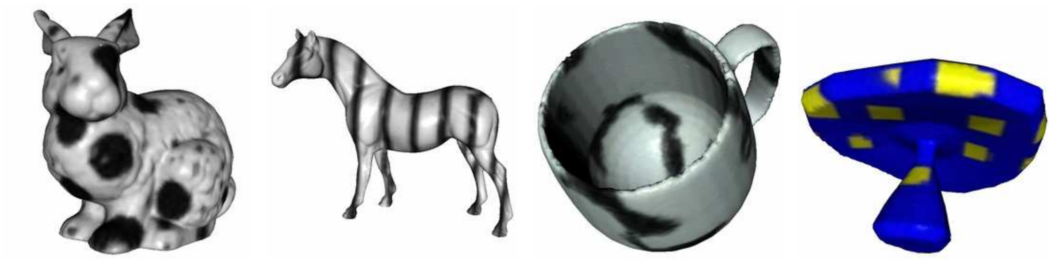
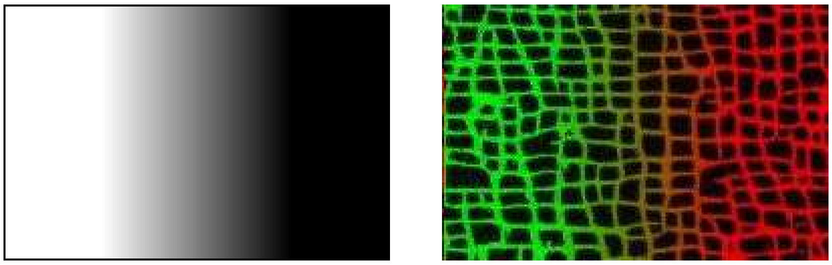


Figure 7.5: *Solid texture synthesis results mapped to different models. Each model is carved out from a synthesized solid texture, shown in Figure 7.3 and Figure 7.4.*



(a) (b)

Figure 7.6: *Generating texture mixture with weighted blending. Our algorithm is flexible with how the input textures are mixed together. In this example, the two textures shown in Figure 7.7 are weighted according to the ramp image shown in (a). The corresponding synthesis result is shown in (b). Note the transition region in the mixture result.*

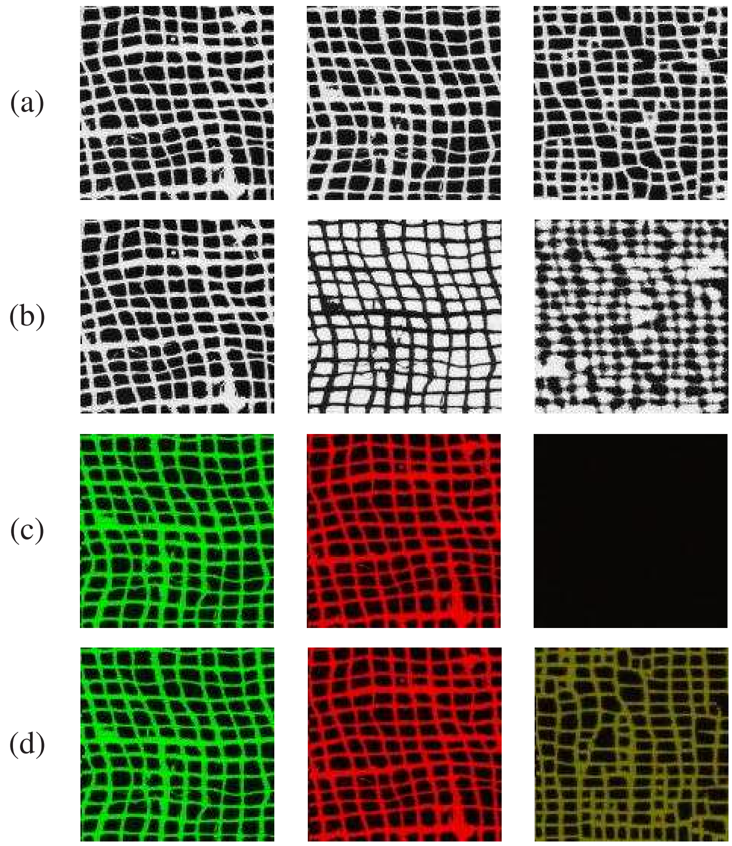


Figure 7.7: *The effect of colors on the texture mixture results. For each row of images, the left two are samples and the right image is a texture mixture generated with equal weighting from the two sources. (a) two sources have the same color space. (b) two sources have the same color space, but their patterns have opposite colors. (c) two sources have disjoint color spaces, and the result is generated by direct color matching. (d) two sources have disjoint color spaces, but the result is generated by matching the intensities rather than RGB colors of the sources.*

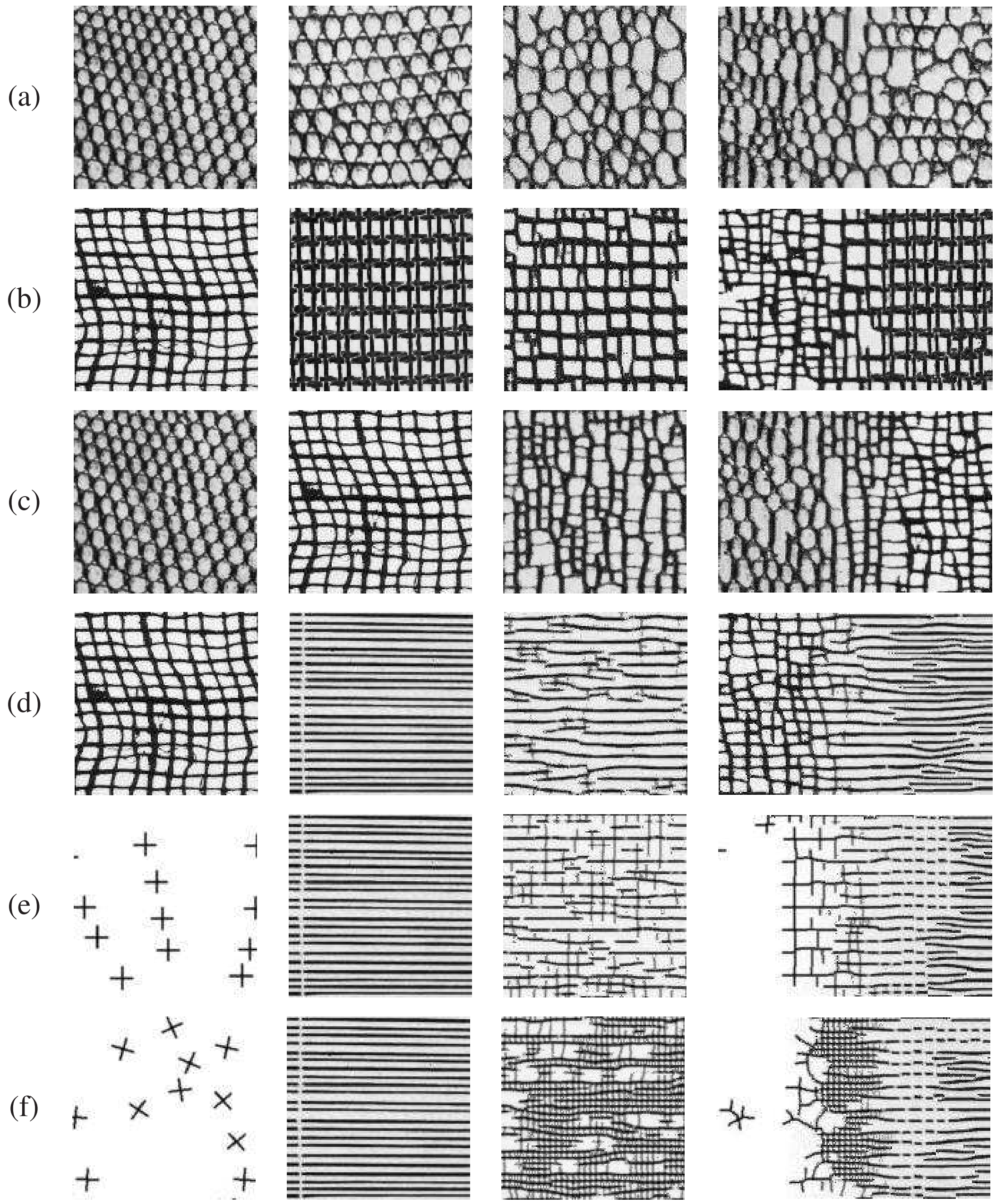


Figure 7.8: *Texture mixture results. For each row of figures, the two input textures are shown on the left, and the corresponding synthesis result, with equal weighting and ramp weighting, are shown on the right. (a) Brodatz D36 and D22 (b) Brodatz D103 and D20 (c) Brodatz D36 and D103 (d) Brodatz D103 and D49 (e) Brodatz D49 and a artificial texture with plus signs (f) Brodatz D49 and a artificial texture with rotated plus signs.*

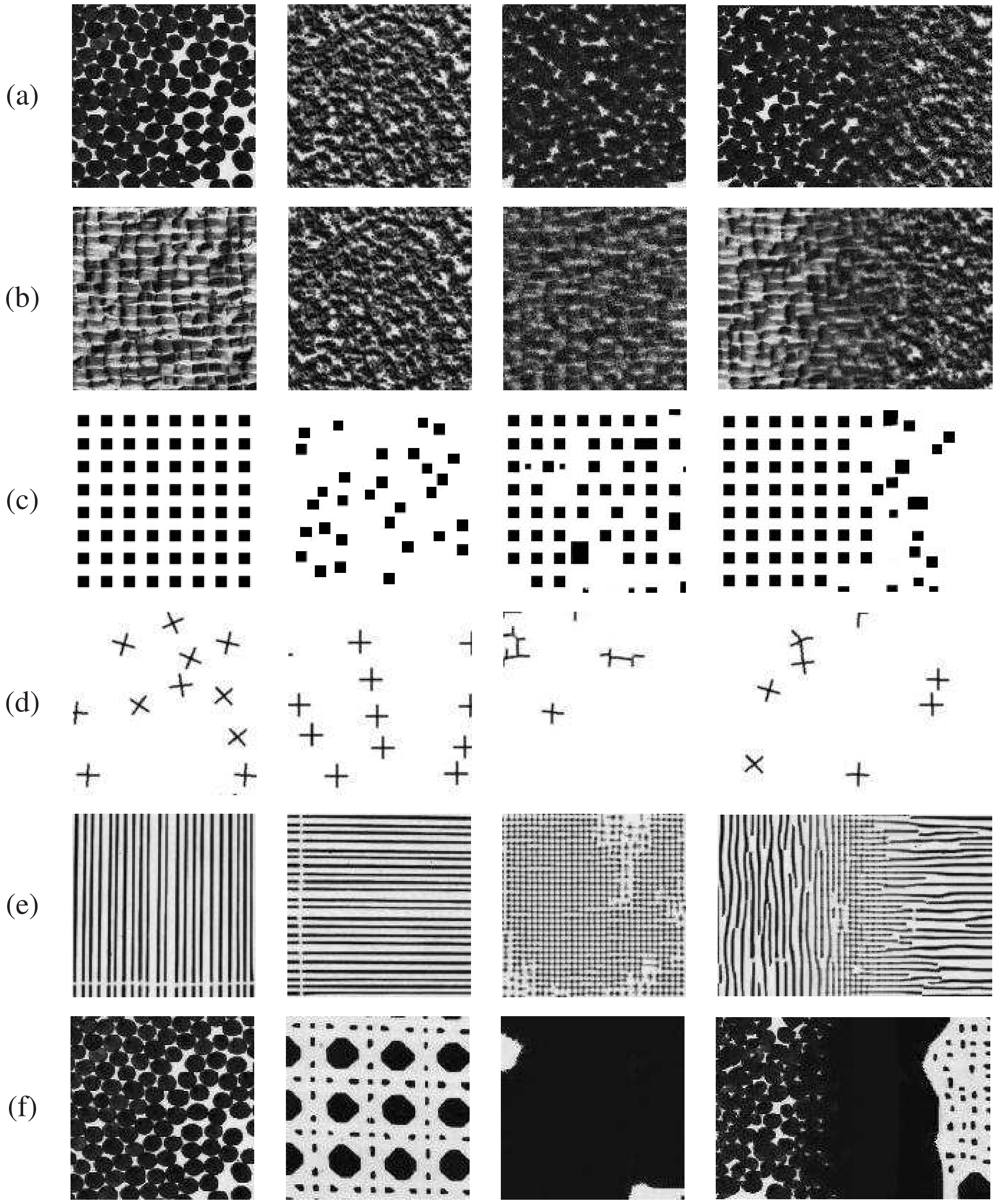


Figure 7.9: *More texture mixture results. (a) Brodatz D67 and D57 (b) Brodatz D84 and D57 (c) Square textures with regular and random placements (d) Artificial textures with up-right and rotated plus signs (e) Brodatz D49 texture with different orientations (f) Brodatz D67 and D101.*

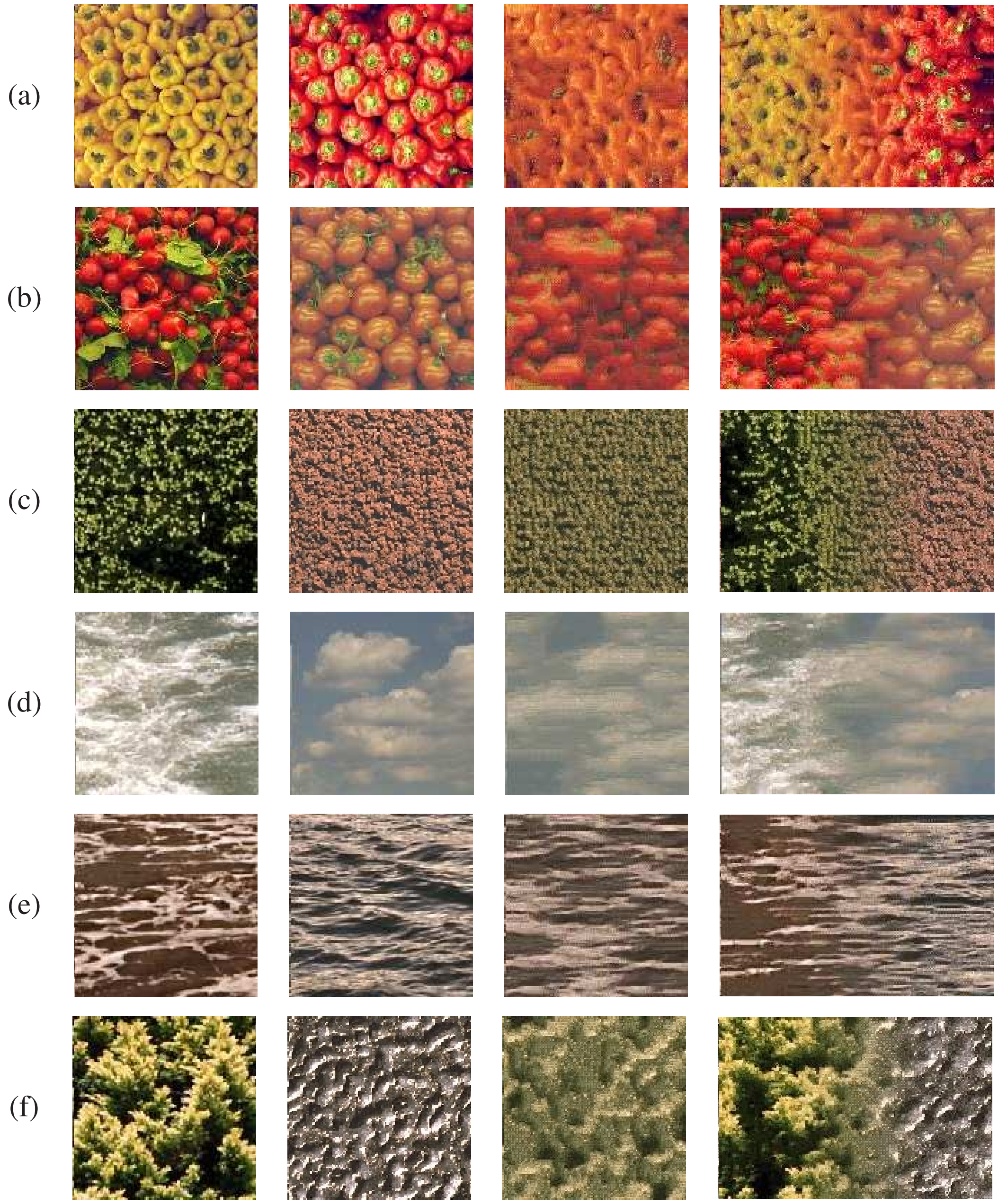


Figure 7.10: *Color texture mixture results. (a) yellow and red peppers (b) radishes and tomatoes. VisTex textures: (c) Misc.0000 and Food.0005 (d) Water.0007 and Clouds.0000 (e) Water.0004 and Water.0000 (f) Metal.0004 and Leaves.0006.*

Chapter 8

Real-time Texture Synthesis

Texture mapping has become ubiquitous for real time rendering. However, texture mapping is expensive both in computation and memory access. Recent progresses in hardware development have made the computation part relatively inexpensive; however, texture memory access remains a bottleneck. Methods to address the memory access problem (both bandwidth and latency) include compression [4], texture patching [82, 55, 20], and texture caching and prefetching [31, 37].

An alternative solution to reduce memory access bottleneck is to synthesize textures on the fly rather than storing them in the memory hierarchy. To achieve the same rendering speed as traditional texture mapping hardware, we need to be able to synthesize textures fast enough. Unfortunately, most existing texture synthesis techniques (both procedural and statistical) are too slow for real time applications. With recent developments in hardware, it has become feasible to start talking about real-time texture synthesis. As an example, Ken Perlin has mapped his procedural noise technique into gate design (see http://www.noisemachine.com/ for more details). We are interested in making statistical texture synthesis real-time. Although our acceleration presented in Chapter 3 allows textures to be generated reasonably fast, it is still not fast enough for real time applications.

In this chapter, we present extensions of our algorithm for real time texture synthesis. We first describe methods that allow output pixels to be generated in any sequence while maintaining the same final result. This technique, henceforth called *order-independent* texture synthesis, allows our algorithm to be evaluated in parallel and called much like a procedural texture synthesis routine. Based on this algorithm, we then project possible software and hardware implementations.

8.1 Explicit v.s. Implicit Texture Synthesis

Texture synthesis techniques can be classified as either *explicit* or *implicit* [18, Chapter 2]; an explicit algorithms generates a whole texture directly while an implicit algorithm answers a query about a particular point (much like scan-converting polygons versus ray-tracing implicit surfaces). Most existing statistical texture synthesis algorithms are explicit; because the value of each texture pixel is related to other pixels (such as spatial neighboring ones in Markov Random Field approaches) it is impossible to determine their values separately. On the other hand, most procedural texture synthesis techniques are implicit since they allow texels to be evaluated independently (such as Perlin noise).

Implicit texture synthesis offers several advantages over explicit texture synthesis. Because only those texels that are actually used need to be evaluated, implicit methods are usually computationally cheaper than the explicit ones. Implicit methods often consume less memory since they don't need to store the whole texture (especially for high dimensional textures). Implicit methods are also more flexible since they allow texture samples to be evaluated independently and in any order. Unfortunately, implicit methods are usually less general than explicit ones. Because of the requirement of independent texel evaluation, implicit methods cannot use general statistical texture modeling based on inter-pixel dependencies.

We would like to combine the advantages of both implicit and explicit texture synthesis. The ideal algorithm should be at least as general as our approach while allows texture samples to be evaluated independently. However, since inter-pixel dependencies have been shown to be important to model textures, fully implicit methods are unlikely to achieve the same generality as current statistical texture synthesis algorithms.

An alternative is to relax the independence requirement slightly by allowing textures to be evaluated *pseudo-implicitly*. The idea is to design an synthesis traversal order which lets each texel depend on only a constant number other texels. Though not as efficient as fully-implicit techniques, a pseudo-implicit method is at least more flexible than traversing the whole output texture in scanline order.

A pseudo-implicit texture synthesis order must satisfy the following requirements:

Quality The synthesis order should preserve the synthesis quality as well as those produced by the scanline order. This is not difficult to meet; we have shown in Chapter 6 that a random order with two passes works as well as scanline order for general situations. Therefore any synthesis order that uses similar two passes should satisfy this requirement.

Flexibility The synthesis order should allow textures to be evaluated non-sequentially. More specifically, we should minimize the depth of the dependency-graph that determines the traversal order for synthesizing output pixels. The scanline order is the worst case since its dependency graph reduces to a chain.

Consistency Given the same initial condition, the synthesis order should always produce the same result. Otherwise we can see popping artifacts if the result texture is used in animations. This is usually termed the "internal consistency" requirement [82].

We now describe a new synthesis order that satisfies all the above requirements. This traversal order, termed *order-independent* texture synthesis, has constant time complexity for evaluating each output pixel where the constant depends only on the neighborhood sizes. It can be considered as an extension of our two pass random traversal (Chapter 6) with an additional pyramid buffer.

8.2 Order-Independent Texture Synthesis

To achieve constant time complexity, we need to design a traversal order that has no dependency between pixels at the same pyramid level. This can be achieved by modifying our random traversal order (Chapter 6) with a buffer pyramid, as shown in Figure 8.1. The basic idea is to alternate the destination for storing output pixels between the buffer and the output pyramid so that the neighborhood built for each output pixel contains only buffered pixels. Figure 8.1 (a) shows a buffer and an output pyramid, with the lower resolution already synthesized. To generate the next higher resolution, we use a two pass method similar to the random traversal order in Chapter 6. In the first pass (Figure 8.1 (b)), we use the lower resolution level to extrapolate the higher resolution level by running the synthesis algorithm with a neighborhood containing pixels only from the lower resolution. But instead of writing new pixels directly to the output higher resolution, we write pixels into the buffer. In the second pass (Figure 8.1 (c)), we use a symmetric neighborhood containing both the lower resolution and the current resolution at the buffer to generate new pixels into the output pyramid. Because the neighborhoods used in this pass only contain pixels in the buffer or in the lower resolution, different traversal orders will yield the same result. After this, we can re-iterate the algorithm several times by swapping the roles of the buffer and output pyramids at each iteration, as shown in Figure 8.1 (d). In our experiments, this traversal order works as well as the random order in Chapter 6 for most textures we have tried.[1] In addition, it can be easily shown that this traversal order allows each output pixel to be evaluated depending on a set of spatially neighboring pixels, where the number of dependent pixels is determined by both the neighborhood sizes (used for each pyramid level) and the number of iterations. Specifically, this number is independent of the output image size.

Because each output pixel only depends on a small set of neighboring pixels, we can further refine the algorithm so that no explicit output pyramids and buffering are required. This refined algorithm is shown in Table 8.1. Instead of pyramids, the algorithm stores computed pixel values in a cache. The cache consists of entries (L, p, m, C), where L and p are the pixel level and location, respectively, m is the iteration number, and C is the pixel color. The portion (L, p, m) is the cache tag and C is the cache value. To synthesize a specific pixel (L, p, m) (function **SynthesizePixel**), it first checks if it is in the cache. If so, no computation is required and the cache entry is returned. Otherwise, we build the neighborhood around (L, p, m) and search for the best match from the input pyramid G_a. The code for neighborhood searching (lines 4 through 9 in Table 8.1) is very similar to Table 2.2, except that we use different ways to build input and output neighborhoods. The input neighborhood is built as before (**BuildInputNeighborhood**), but the output neighborhood is built from the cache rather than an output pyramid (**BuildOutputNeighborhood**).

[1]Note that this traversal order will reduce to the random order in Chapter 6 if we do not use the extra buffer pyramid.

function $C \leftarrow$ SynthesizePixel(G_a, L, p, m)
if CacheHit(L, p, m)
  return CacheEntry(L, p, m);
else
  $N_s \leftarrow$ BuildOutputNeighborhood(L, p, m);
  $N_a^{best} \leftarrow$ null; $C \leftarrow$ null;
  loop through all pixels p_i of $G_a(L)$
    $N_a \leftarrow$ BuildInputNeighborhood(G_a, L, p_i);
    if Match(N_a, N_s) $>$ Match(N_a^{best}, N_s)
      $N_a^{best} \leftarrow N_a$; $C \leftarrow G_a(L, p_i)$;
  AddCacheEntry(L, p, m, C);
  return CacheEntry(L, p, m);

function $N_s \leftarrow$ BuildOutputNeighborhood(L, p, m)
$N_s \leftarrow$ null;
foreach $(L_n, p_n, m_n) \in$ Neighborhood(L, p, m)
  % note: $(L_n, m_n) \prec (L, m)$
  if CacheHit(L_n, p_n, m_n)
    $N_s \leftarrow N_s \oplus$ CacheEntry(L_n, p_n, m_n);
  else
    $C \leftarrow$ SynthesizePixel(G_a, L_n, p_n, m_n);
    $N_s \leftarrow N_s \oplus C$;
return N_s;

Table 8.1: *Pseudocode of order-independent texture synthesis.*

The function **BuildOutputNeighborhood** works as follows. For each pixel (L_n, p_n, m_n) in the neighborhood of (L, p, m), we first check if it is in the cache. If so, we add it directly to the output neighborhood N_s. Otherwise, we call **SynthesizePixel** recursively to compute its value and add the computed value to N_s. Note that we require each (L_n, m_n) to be lexically smaller than (L, m), meaning that the Neighborhood(L, p, m) can contain only pixels from lower resolutions, as well as pixels from the same resolution which are generated in earlier iterations. Because of this, the dependencies of the pixels form an acyclic graph and the mutual recursive calls between **SynthesizePixel** and **BuildOutputNeighborhood** are guaranteed to terminate, unless the cache is too small to simultaneously hold all pixels in Neighborhood(L, p, m).

Initialization: We initialize the lowest resolution of the cache by copying pixels randomly from the lowest resolution of the input pyramid. In other words, each output pixel at the lowest resolution/iteration (L_{min}, m_{min}) is assigned a random value from the input. This initialization completely determines the synthesis result since each output pixel is determined only by these pixels at (L_{max}, m_{min}). (This can be shown by recursively expanding the dependency graph for each pixel, following the mutual calls between **SynthesizePixel** and **BuildOutputNeighborhood**.) This initialization can be implemented by either permanently storing pixels (L_{max}, m_{min}) in the cache, or by using a pseudo-random number table to choose the random values on the fly (as implemented in Perlin noise).

8.3 Results

In Figure 8.2, we compare the results generated by order-independent synthesis and our earlier algorithm using random ordering (Chapter 6). We use the same parameters for both versions of algorithms: Gaussian pyramid with 4 levels, a neighborhood of size $\{5,2\}$, and three passes with the first pass using lower resolution information only and two subsequent passes using both current and the lower resolution. As shown, our order-independent algorithm generates results with comparable quality with our earlier methods.

Figure 8.3 through Figure 8.6 illustrate the cache access footprints with different request patterns. In each figure, we show the content of the cache for pixels at different levels and iterations (black pixels indicate pixels not in cache). We use 4 pyramid levels and 3 iterations except the lowest resolution where only one iteration (the initial value) is used, and a spherical neighborhood with size $\{5,2\}$. Figure 8.3 shows the cache footprint for synthesizing one pixel. We can see that the footprint size increases as we move toward lower levels and iterations. In fact, the footprint shapes can be estimated by computing the convolution of the neighborhood shapes with respect to footprints at larger levels/iterations. As an example, the footprint shape at (L_3, m_1) is the shape of the neighborhood at the current level, and the footprint shape at (L_3, m_2) is the convolution of the neighborhood shape with the footprint at (L_3, m_1). We can also see that many cache pixels may be touched in order to synthesize one pixel. However, the cost of touching multiple cache pixels can be amortized for synthesizing multiple pixels, as shown in Figure 8.5, Figure 8.6 and Figure 8.4. Figure 8.4 has an S-shaped pattern. Both Figure 8.5 and Figure 8.6 synthesize the same number of output pixels. However, since Figure 8.6 has a spherical request pattern, its cache footprint is more coherent and much fewer pixels are touched in the cache compared to Figure 8.5.

Since each output pixel may request evaluating multiple cache pixels, it will be interesting to know what is the relationship between the number of touched cache pixels and number of request pixels in different patterns. For a given number of request pixels, we argue that the spherical pattern will involve minimum cache footprints since it has the maximum coherence. On the contrary, random pattern will have the worst coherence and will require touching more cache pixels. In Figure 8.7, we show the percentages of touched cache pixels with respect to percentage of requested input pixels using spherical and random patterns. We use a reasonably large texture of size 512x512 to minimize the effect of neighborhood sizes. As shown, the spherical pattern offers near optimum behavior and is almost linear, whereas the random pattern has worse performance and requests more cache pixels. Renderings of real scenes should have caching behavior between these two curves since their mipmap footprints should be bounded between random and spherical patterns. In fact, we believe most real renderings will have curves close to linear since their mipmap footprints are mainly coherent.

8.4 Architecture Design

A conceptual design of our order-independent texture synthesis architecture is shown in Figure 8.8. The design consists of a set of read-only input neighborhoods, a read/write output pixel cache, an array of pixel synthesis units (PSU), and a comparator network that connects the outputs of the pixel synthesis units. To synthesize a new pixel, each PSU collects the output neighborhood from the cache, searches for the best match from the set of input neighborhoods, and outputs the best-matched pixel error and color. The search can be conducted either sequentially or in small amount parallelism, depending on the particular implementation. Further parallelism can be achieved by using multiple PSUs and let them search for disjoint subsets of the input neighborhoods, and comparing these results in the comparator network.

The comparator network chooses the best match among the set of results generated by PSUs through an inter-connection of two-way comparators. Each comparator compares two sets of inputs of color and error, and outputs the set with smaller error. The comparator network can be implemented in a variety of topologies. To achieve maximum flexibility, we can implement it as a complete binary tree, shown in Figure 8.9. A complete-binary-tree comparator with P input units can find the best match in $log_2 P$ time steps.[2]

The bank of input neighborhoods must allow concurrent read from all the PSUs. However, since these PSUs read disjoint subsets of the input neighborhoods, we can conceptually store the set of input neighborhoods in a big shift register, as shown in Figure 8.10. The set of PSUs are equally spaced in the address space of the shift register. At every time step the content of the shift register is shifted one cell to the right, and concurrently each PSU compares its output neighborhood with the corresponding input neighborhood. After $O(\frac{MN}{P})$ shifts where M is the number of training neighborhoods, N is the neighborhood size, and P is the number of PSUs, the outputs from the PSUs will be ready to be forwarded to the comparator network.

The cache holds synthesized pixel values. There are several parameters for the design of the cache, such as number of cache entries, cache line size, associativity, replacement policy, and if we should use different parameters for pixels at different pyramid levels. At initialization, the cache needs to hold the values for the lowest resolution (permanently) to guarantee the consistency of the subsequent synthesis results.[3] Since pixels at lower resolutions tend to be used for frequency, it might be useful to give them higher priorities for being retained during cache replacement. These cache parameters need to be determined via a set of benchmarks (such as in [31]).

---

[2]each time step is the amount of time required to advance the results from one layer of comparators to another.

[3]Another option is to use a noise table like Perlin noise.

8.5 Discussion and Future Work

We are currently in the conceptual stage in designing a hardware for real-time texture synthesis. We have achieved the first step towards real time texture synthesis by reducing the time complexity of our texture synthesis algorithm so that pixels can be evaluated in constant time while maintaining the consistency of the synthesis results. There are several advantages of our design:

Parallelism Our order-independent synthesis algorithm can be evaluated completely in parallel. In fact, since the neighborhood search can also be conducted in parallel, our algorithm is parallel for both the input space (where the neighborhood search is conducted) and the output space (where the output texture is synthesized pixel-by-pixel).

Reconfigurability Our hardware design is highly configurable. The hardware can consist of any number of PSUs (input space units), and we can connect them to the comparator network (output space units) in any topology. Example topologies include a fully-connected network, where each output unit can utilize any number of input units, or a singularly-connected network where each output unit can utilize only a single input unit. This network should be programmable for different applications where the user might want to control the amount of parallelism for both the input and output spaces. We can also configure the hardware with a variety amount and configuration of cache/memories for both the input training set and output pixel caches.

Shading Language Since our algorithm can now be called much like a procedural texturing routine, it could be easily integrated into an existing shading language such as RenderMan. Combining with a texture synthesis hardware, it can also be integrated into a real-time shading language. Since professional animators are more used to procedural texturing/shading, this might provide an easy interface for them to utilize our algorithm.

Here is a list of problems to be addressed:

Writable Cache To evaluate each output pixel, our algorithm needs to store temporary values for the necessary neighborhood pixels. This requires a fast writable cache or memory and complicates the hardware design. In contrast, most commercial graphics hardware chips do not require any writable cache.

Tree Search or Parallel Comparison The neighborhood search process in our algorithm can be conducted by either tree-search (TSVQ) or parallel-search. Tree-search is primarily a software acceleration and might have scattered memory access patterns; therefore it is not a perfect candidate for hardware implementation. It is not clear if there are alternative hardware search strategies other than parallel comparison as we described before.

Combining with Patch-based Techniques Patch-based texture synthesis techniques [20, 82, 55] provide a fast way to generate textures by simply copying texture patches. Although these techniques are more efficient, they can suffer from discontinuity artifacts at patch boundaries. We could combine patch-based sampling with our pixel-synthesis technique to achieve real-time synthesis by simply using our algorithm to fill in the gaps between patches. The width of the gap can be used as a parameter to tune between image quality and computation speed.

For future work we plan to implement our design in a hardware simulator, run a set of texture-mapping benchmarks, and study the performance over different design parameters, especially for the output cache. In particular, we plan to repeat several of the experiments in [31, 37]. We believe that many of the observations made in these papers are applicable to our design, since it is very similar to traditional mipmapping except that during cache misses we synthesize texels rather than fetching them from the memory hierarchy. This difference causes our algorithm to have slightly larger cache footprints than mipmapping and may require different cache parameters from [31, 37].

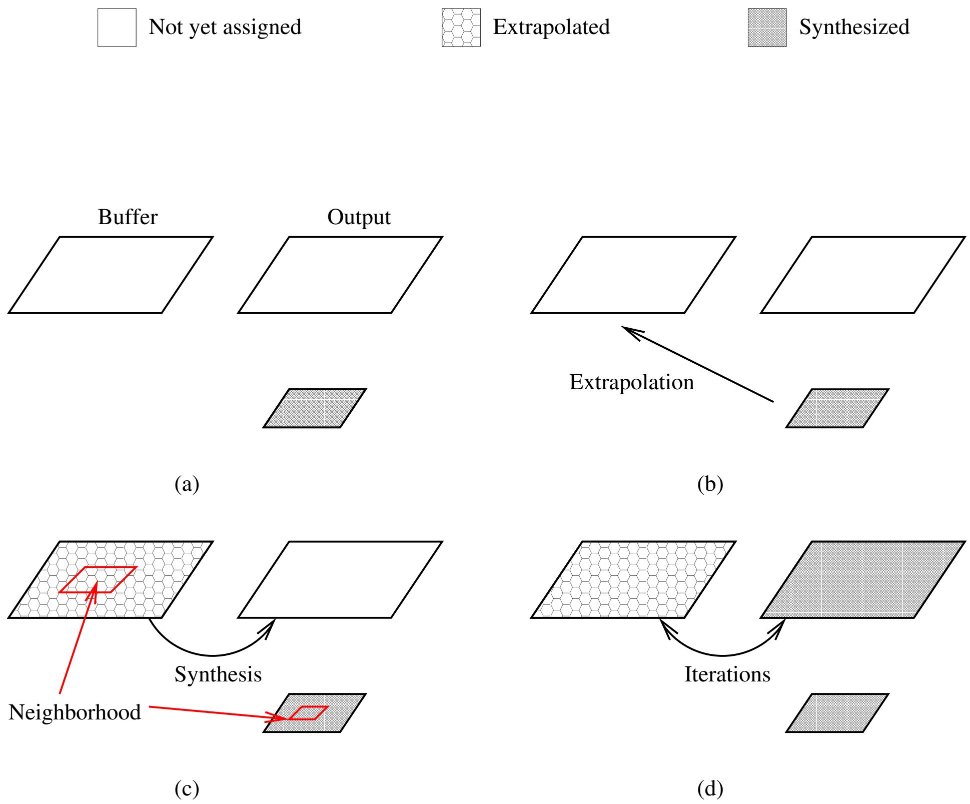


Figure 8.1: *Order-independent texture synthesis. (a) A buffer and an output pyramid with the lower level already synthesized. (b) The lower level is extrapolated to the buffer, generating an initial guess. This extrapolation is done by texture synthesis using a neighborhood containing lower resolution pixels only. (c) The lower resolution, together with the extrapolated resolution (in the buffer), are used to synthesize textures in the higher resolution at the output pyramid. This is done by using a neighborhood containing pixels from the buffer and lower resolution. (d) We can iterate this synthesis process by swapping the roles of buffer and output pyramid.*

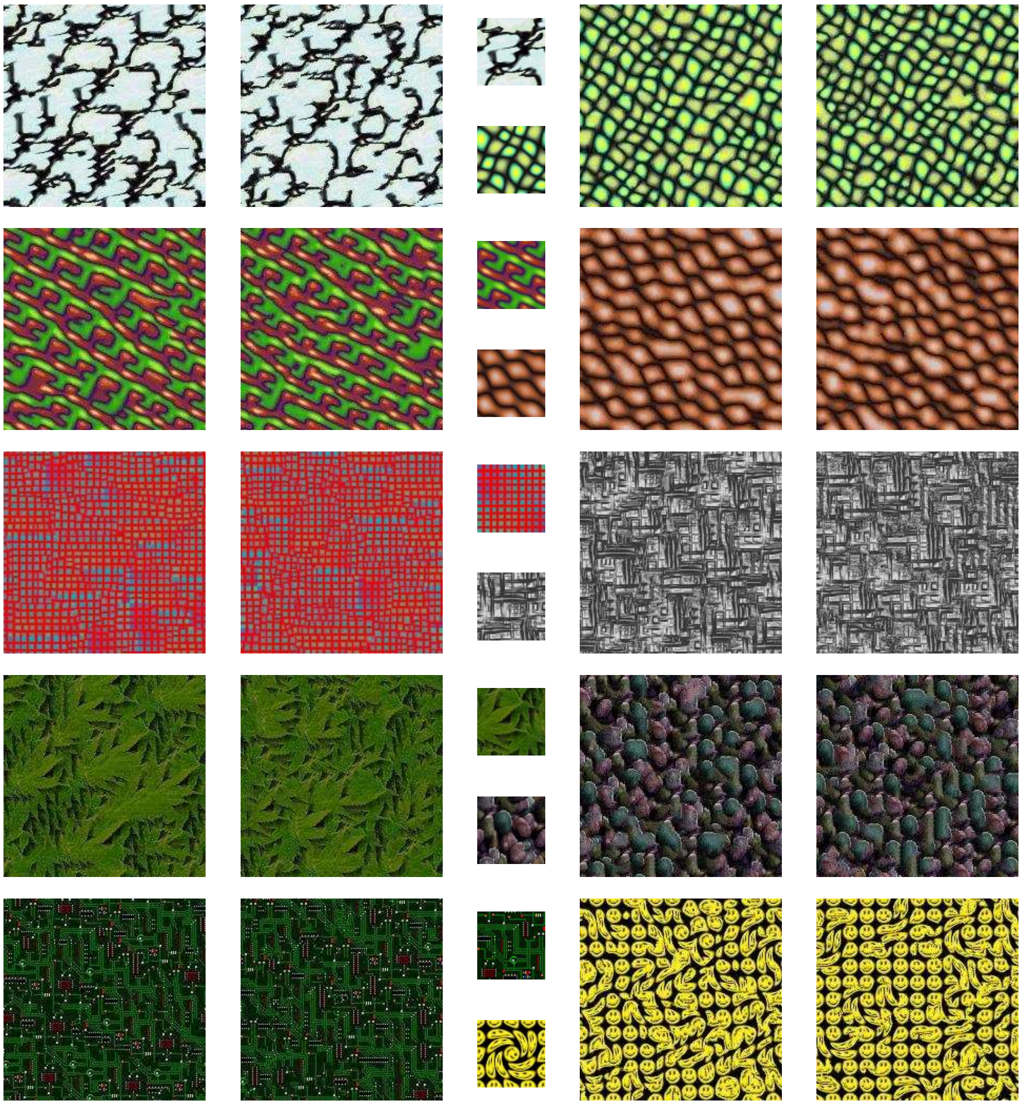


Figure 8.2: *Quality comparison between order-independent synthesis and our earlier methods. The original textures are shown in the middle, and the synthesis results, with bigger sizes, are shown on the sides. For each pair of synthesis results, the left one is generated by random ordering (Chapter 6), and the right one is generated by order-independent synthesis.*

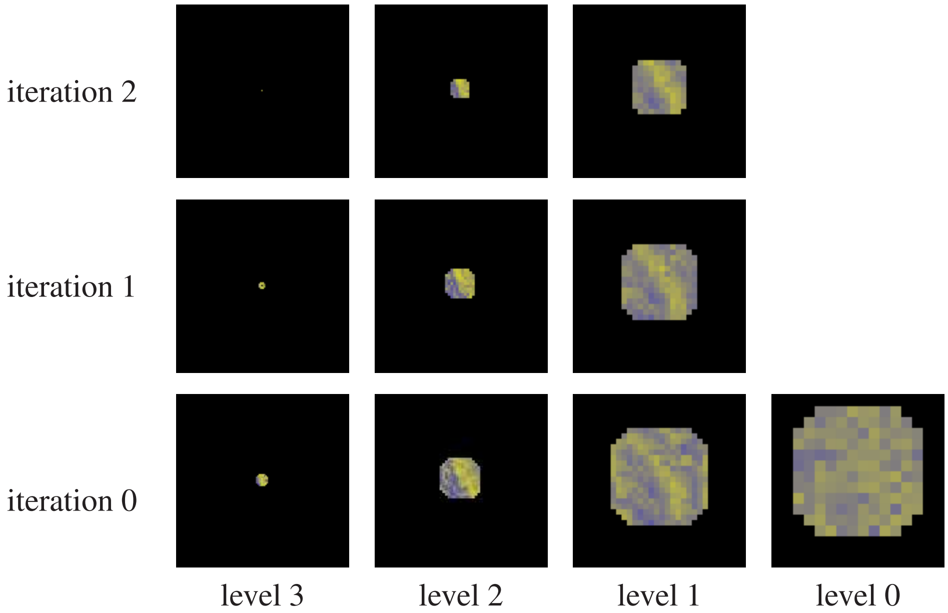


Figure 8.3: *Cache access footprint for a single pixel. The images show the contents of the cache at different iterations and different levels, with higher resolution on the left and later iteration on the top. The neighborhood size is {5,2}. Image sizes are 128x128, 64x64, 32x32, and 16x16, respectively, from left to right.*

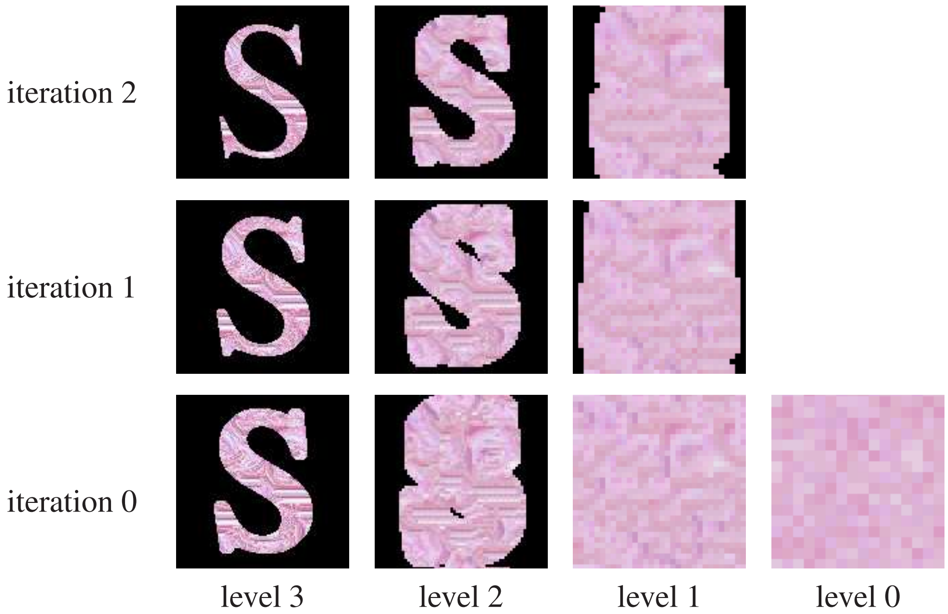


Figure 8.4: *Cache access footprint for an S-shaped request pattern.*

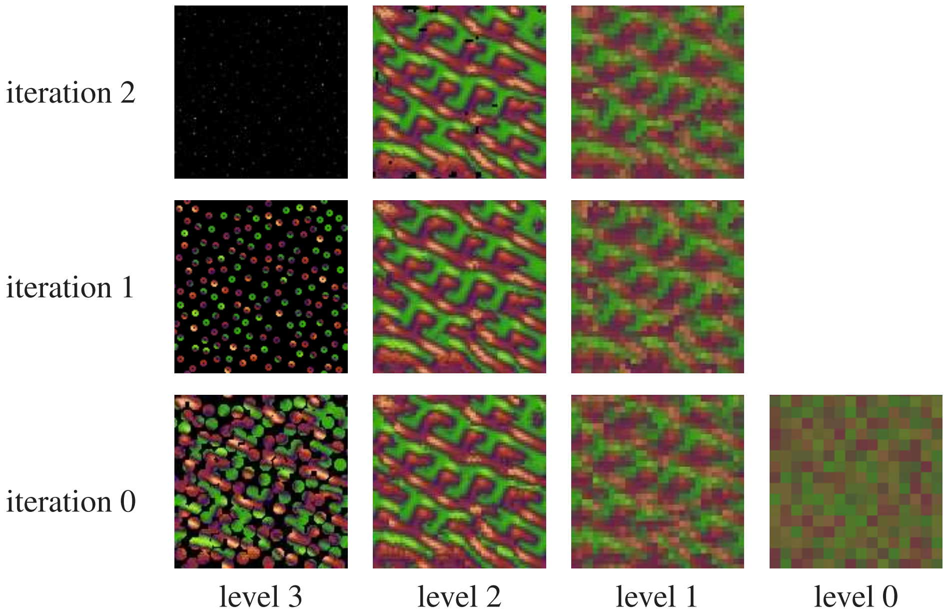


Figure 8.5: *Cache access footprint for a set of uniform random pixels generated by a poisson disk process.*

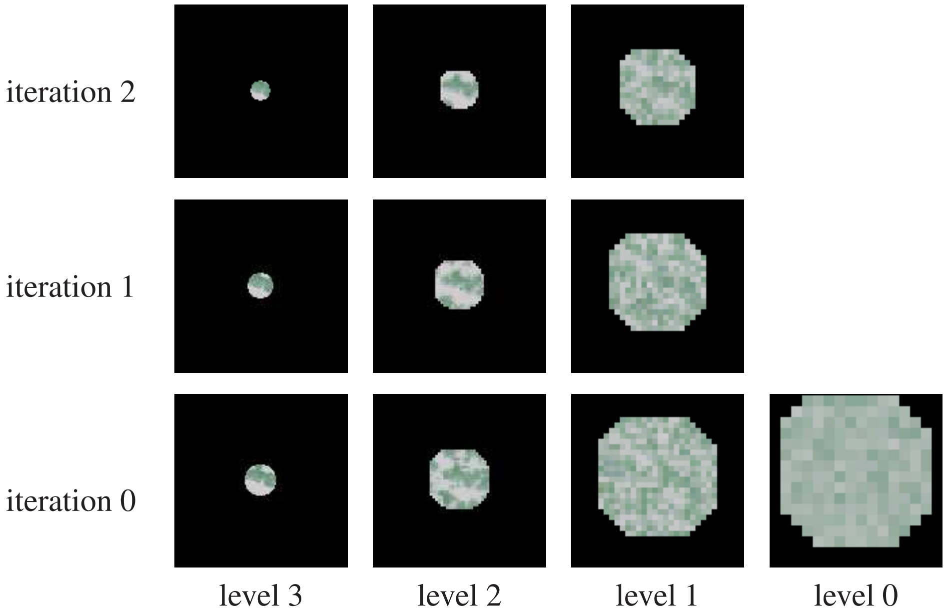


Figure 8.6: *Cache access footprint for a spherical request pattern.*

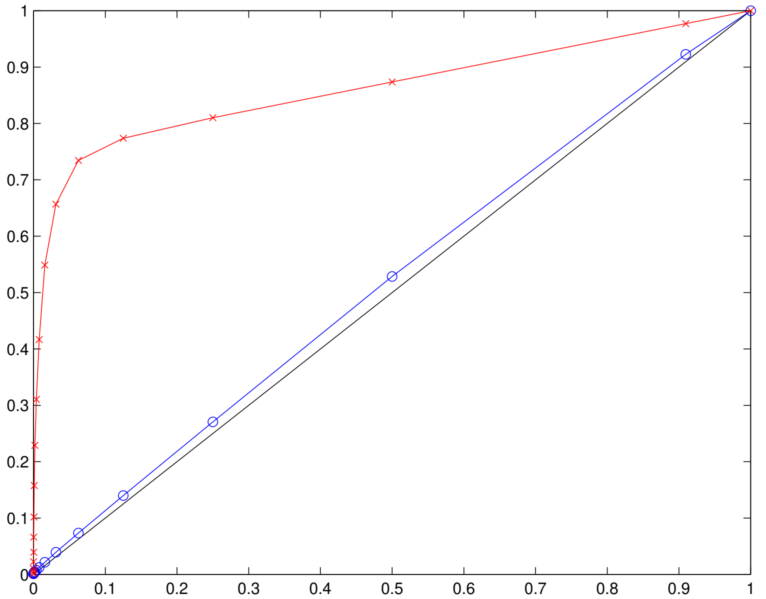


Figure 8.7: *Texture cache usage. This diagram illustrates how much texture cache will be touched with respect to different amount of input requests. The horizontal axis indicates the percentage of pixels requested for a 512x512 texture, and the vertical axis indicates the percentage of pixels touched in the corresponding texture cache. Two different request patterns are shown. The blue curve (with circles) indicates spherical access patterns (Figure 8.6) and the red curve (with crosses) indicates random access patterns (Figure 8.5). The black line indicates the ideal linear behavior. The vertical intercept of the red and blue curves at the left is about 0.0011 (1148 pixels), indicating the large (although constant) footprint for synthesizing one pixel.*

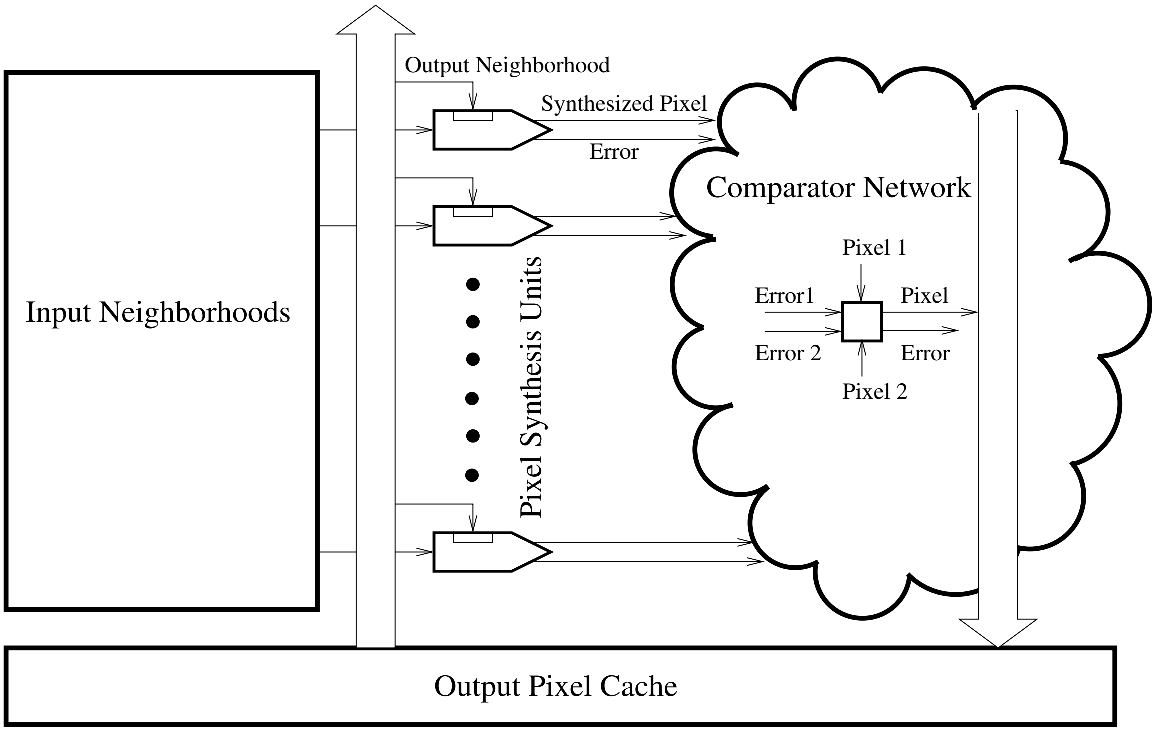


Figure 8.8: *Our texture synthesis architecture.*

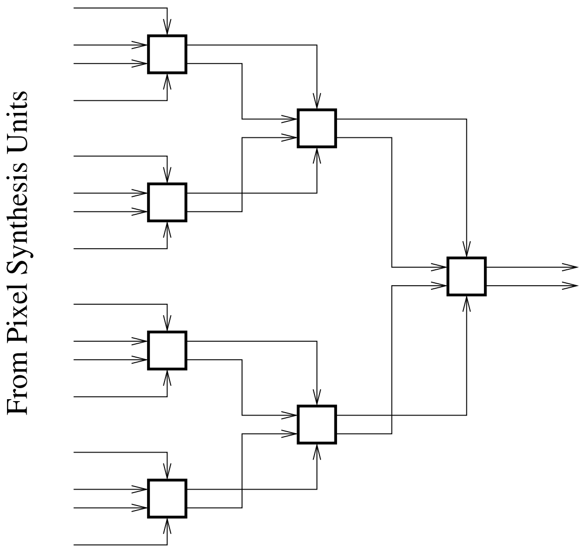


Figure 8.9: *A comparator network implemented in complete binary tree. The number of first level units (leftmost level in this figure) is equal to the number of pixel synthesis units (PSU).*

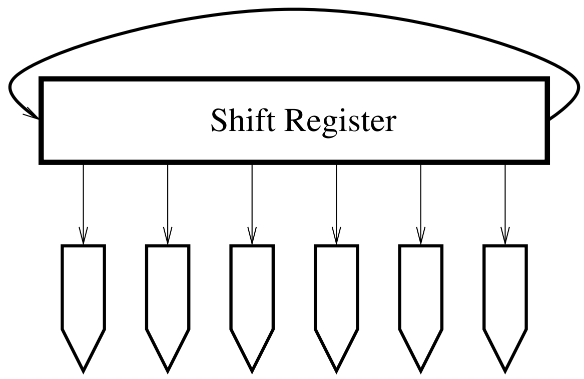


Figure 8.10: *A shift register holding the set of input neighborhoods. The PSUs are equally spaced along the address space of the shift register.*

Chapter 9

Algorithm Analysis

In this chapter, we analyze the algorithm behavior both analytically and experimentally. We begin by discussing the neighborhood searching process, and argue that due to the special properties of texture neighborhoods, nearest neighbor searching is a good approximation to more expensive approaches such as Markov Random Fields. We then discuss the convergence of our algorithm, and demonstrate that convergence is not related to image quality. In fact, achieving convergence by adding excessive iterations can only degrade synthesis quality and demand unnecessary computation. We end this chapter by discussing the relationship between different versions of our algorithm.

9.1 Neighborhood Searching

The core of our algorithm consists of searching neighborhoods. Because we keep the shape of the neighborhoods fixed, we can transform the problem of searching neighborhoods as nearest neighbor searching in a high dimensional space, and lead to our acceleration based on tree-structured VQ. However, these neighborhood vectors are not merely samples in a high dimensional space; they possess some special coherence since they are sampled from the same input texture.

Based on this observation, we argue that our algorithm is an efficient implementation of MRF texture synthesis algorithms. In particular, given limited samples (neighborhoods from input texture) in a high-dimensional space (the dimensionality of the neighborhoods),

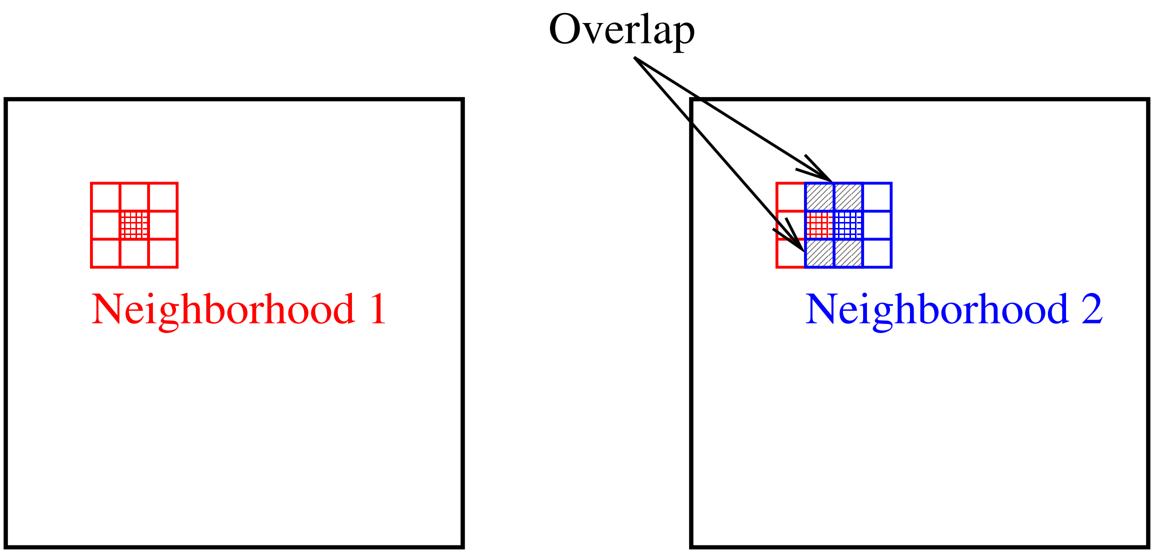


Figure 9.1: *Neighborhood Coherence. The left figure shows a neighborhood (in red), and the right figure shows the neighborhood shifted one pixel right (in blue). Note that they overlap significantly as shown in the shaded portion.*

nearest neighbor search is a reasonable approximation to maximum likelihood sampling. We also show that this correlation leads to the patching behavior, and this offers chances for further acceleration.

9.1.1 Texture Neighborhoods

Neighborhoods collected from a texture image are not merely samples of a high dimensional space; they are correlated with each other due to their spatial arrangement. For example, a neighborhood at pixel location (x_i, y_i) has significant overlap with the adjacent neighborhood at location $(x_i + 1, y_i)$ (Figure 9.1). This implies that the coherence of input locations can be carried over to the output locations. Specifically, if the best match for output location (x_o, y_o) is $N(x_i, y_i)$, then it is very likely that the best match for location $(x_o + 1, y_o)$ is $N(x_i + 1, y_i)$. This fact has been noted by Michael Ashikhmin and he showed that, for some textures, taking this coherence into account can improve the texture synthesis quality [1]. The coherence also exists for any offset (x_s, y_s) smaller than the neighborhood size. That is, if the best match for output location (x_o, y_o) is $N(x_i, y_i)$, then it is also likely that the best match for location (x_o+x_s, y_o+y_s) is $N(x_i+x_s, y_i+y_s)$. In general this likelihood decreases with the increase of (x_s, y_s) due to the decreasing of overlapped portions.

In addition to be coherent, texture neighborhoods usually have high dimensionality. For example, a 2-level symmetric neighborhood with sizes 9x9 and 5x5 at each resolution will have 106 samples.[1] This high dimensionality along with the fact that we don't have much samples imply that the probability space is only very sparsely sampled. In fact, due to the curse of dimensionality, the neighborhood vectors tend to be equally far away from each other. Here is a little math to convince this. Consider a sphere with radius r centered at any given texture neighborhood vector, and a thin layer of thickness ϵ of this sphere. The relative volume of this slice with respect to the whole sphere is $1-(\frac{r-\epsilon}{r})^N$, where N is the dimensionality. Given a fixed ϵ, no matter how small, it is easily seen that $\lim_{N\to\infty} 1-(\frac{r-\epsilon}{r})^N = 1$. In other words, most of the volumes of an N-dimensional sphere will be near the surface of it, and for a given texture neighborhood vector, most of the other neighborhood vectors will lie within this thin layer and therefore are almost equally far away.

We can argue that these two properties of texture neighborhoods, coherence and high dimensionality, make nearest-neighbor search a good approximation for more general sampling strategies such as maximum-likelihood. Given a query neighborhood from the output at location (x_o, y_o), its distance to incompatible input neighborhoods will be almost equally far away. However, the distance from $N(x_o, y_o)$ will be closer to the set of overlapped neighborhoods $\{N((x_s, y_s) + M(x_o - x_s, y_o - y_s))|(x_s, y_s)\text{within the neighborhood size}\}$, where $N(x, y)$ is the neighborhood at location (x, y) and $M(x_o, y_o)$ is the input location of the best match for output location (x_o, y_o). In particular, it will be very close to those $\{N((x_s, y_s)+M(x_o-x_s, y_o-y_s))\}$ where (x_s, y_s) are small. If we visualize this phenomena in the high dimensional space, we can see that the query will be at the vicinity of one (or very few) input neighborhoods and at the same time far away from other irrelevant input neighborhoods. Because the query is effected by primarily one input neighborhood, doing a nearest-neighbor searching is not much different from a maximum likelihood sampling.

[1] Each sample can contain more than one real number. For example, if the texture is RGB and we use a Gaussian pyramid, then each sample contains 3 real numbers. However if we use a steerable pyramid with 4 orientations, then each sample might contain up to 12 real values.

For example, if we use Gaussian radial basis functions centered at those input neighborhoods as the probability model, then only the nearest Gaussian has significant effect on the output query.

9.1.2 Experiments

We have conducted a series of experiments to observe the neighborhood matching process. In each experiment, we record the position of each input pixel when they are copied to the output pyramid. By doing so we can know how the input pixels are re-arranged to form the output texture. An example is shown in Figure 9.2. Figure 9.2 (a) shows the synthesis result, and (b) shows the input location. We use the image red and green channels to encode the row and column positions, respectively. Notice that in this case the location image consists of several patches. To make this more visible, Figure 9.2 (c) shows a high-pass filtered image of (b). We can clearly see the patch boundaries as indicated by brighter edges. We use a small 7x7 neighborhood for this experiment. We have conducted a similar experiment with much larger neighborhood size 41x41 as shown in Figure 9.3. The results show that the patching behavior is much stronger, and there are discontinuities at the color result between adjacent patch boundaries. This patching behavior fits well with our previous discussions about the coherence property of the texture neighborhoods. And due to the curse of dimensionality, the patching behavior becomes more obvious as we increase the neighborhood sizes.

We also measured the neighborhood matching error (the L_2 distance between the query neighborhood and the best matching neighborhood) for the synthesis process, and show the results in Figure 9.2 and Figure 9.3. As expected, we can observe that the matching error correlates to the patch boundaries well. In particular, the matching error images look like a low-pass filtered version of the patch boundaries with filter extent determined by the neighborhood sizes. We also record the matching error prior to, during, and after the synthesis process, and for each case we record the errors for the best and second best matches. In both Figure 9.2 and Figure 9.3, the error prior to synthesis is large and homogeneous, confirming our previous observation that due to the curse of dimensionality, the irrelevant neighborhood vectors tend to be equally far away from each other. The errors during and after synthesis are much smaller due to the neighborhood coherence. Even though, we can still see that the second best matches have larger and more homogeneous errors than the best match.

We have also run the synthesis algorithm over a white random noise as shown in Figure 9.4. According to our previous discussions, white noise is the best candidate for testing our patching theory; neighborhood vectors located far away are unlikely to correlate to each other, and overlapping neighborhood vectors possess strong coherence. The results shown in Figure 9.4 confirm our estimation: there are obvious patching behavior; the nearest errors roughly match the patch boundaries; and the second-best errors are homogeneous and have roughly the same magnitude as those errors measured before synthesis.

This patching behavior can happen more or less depending on the input texture type, as shown in Figure 9.5. Here are a few observations:

- The patching is less obvious for stochastic textures or textures containing small patterns, such as (b).
- For structured textures such as (c), patterns are better preserved within patches. In places where no obvious patch is formed, the large scale structures can be lost. However, in general the matching error only weakly correlates to the perceived image quality, indicating that a simple L_2 norm is not a perfect perceptual metric (at least in this neighborhood size).
- For some textures the patch boundaries will be effected by the dominant texture structure, such as the diagonal patterns in (d).

9.2 Relationship to Previous Work

As discussed earlier, our algorithm can be treated as an efficient approximation to MRF texture synthesis algorithms. It is interesting to see that most of the previous MRF approaches do not take into account the coherence of the texture neighborhoods, therefore a lot of their computations are wasted (since for each pixel, they start a new probability sampling from scratch). Our approach is particularly relevant to Kris Popat's algorithm

[53, 52]. [53, 52] used a collection of Gaussian blobs to model textures (similar to radial basis functions), and the parameters of those Gaussian blobs are derived by clustering texture neighborhoods. The tree-structured VQ training of our algorithm is very similar to his clustering phase, and our nearest neighbor approximation can be thought as replacing his Gaussian blobs with piecewise constant Voronoi regions.

The patching behavior of our algorithm shows that there is some redundancy in the computation. Ashikhmin noted this and extended our algorithm so that it can better handle some textures (but performs worse for others) [1]. Efros and others extended this idea further by using patches instead of pixels as the basic building blocks for textures [20, 82, 55]. Those patch-based texture synthesis algorithm can be treated as further accelerations of our approach; since our algorithm forms patches often, it saves computation by copying them directly. Nevertheless, these algorithms are also less flexible than pixel-based approaches since they use only large patches, and this can sometimes introduce discontinuities into adjacent patches. These artifacts can be observed in [20, 82, 55] even after patch registration and blending. This problem with large patches can also be observed by comparing Figure 9.2 and Figure 9.3. Figure 9.3 contains only large patches and cannot avoid patch discontinuities. Figure 9.2, on the other hand, has patches with varying sizes and allows the use of small patches to maintain continuity between larger ones.

9.3 Convergence

In our experiments, we have found that our algorithm doesn't require a lot of iterations to generate good results. However, it is unclear how the algorithm will behave if we add more iterations. For example, if we iterate the algorithm many times, will it eventually converge, or will it oscillate between multiple states? Since we use nearest neighbor instead of probability sampling, the oscillation can happen. It's not difficult to construct pathological combinations of a input texture, an initial output value, and a synthesis order that will make the synthesis oscillating forever. The simplest case will be a periodic input texture (say horizontal stripes), and an output texture with size which is not an integer multiple of the periodicity. If we use a scanline order, it can be seen easily that the result will have the periodic patterns shift forever.

Despite those pathological cases, we are interested in the more general behaviors of the algorithm. We would like to explore the convergence of the algorithm under different inputs, synthesis order, and neighborhood shapes/sizes. A series of such experiments are shown in Figure 9.6 and Figure 9.7. In Figure 9.6, we generate textures via a scanline order, but iterate it 100 times for each resolution. We record the synthesis result as well as time sequence plots for both the matching error and percentage of pixels changed. We can observe several interesting things. First, the scanline order cannot converge all textures (at all resolutions). For the 161 texture, it converges quickly at the highest two resolutions but not the two lower ones; for the 726 texture it doesn't converge for all levels; for the 654 texture it shows some definite oscillations at the 3rd level. Furthermore, the convergence is not well correlated with the decreasing of synthesis errors. For textures 161, 726 and 654 the matching errors remains roughly the same throughout the whole simulation. Even when the algorithm converges as in texture 759, it may not be a good thing. In this case, the many iterations converge the output texture to a dull repetition. This degrading of image quality can also be observed in 726 and 654.

In Figure 9.7, we run a similar set of experiments using a random instead of scanline order. We can notice that in most cases the random order converges the result better than the scanline order. The error seems to be more stably decreasing with less fluctuation. However, random order can still generate bad results such as 654 and 759. It can even increase the matching error as shown in the highest resolution of texture 654. Note that for higher resolutions the percentage of pixels changed at the first iteration is lower than 100 percent; this shows that the extrapolation pass (using lower-level information only) can generate a good initial value and therefore only partial modifications are required.

We have conducted several experiments to investigate the reasons for the slower convergence of Figure 9.6 than Figure 9.7 by isolating the effects of the three fundamental differences between the two experiments: scanline v.s. random order, causal v.s. noncausal neighborhood, and the extra lower-level-extrapolation pass. From these experiments we can conclude that the causal neighborhood is the major reason for the slower convergence, caused by the asymmetry of causal neighborhoods. Consider a newly generated output pixel (x_o, y_o). If we use a symmetric neighborhood, we are sure that all the local neighbors of (x_o, y_o) are happy with it since they are contained in $N(x_o, y_o)$ during the search process. However, a causal neighborhood will omit the half of the local neighbors at the same resolution; therefore it introduces unstability into the synthesis process since newly added pixels can deteriorate the balances established earlier.

We have experimented with many other textures. In general, adding more iterations does not help the synthesis result; in most cases it may even hurt. We attribute this to the deterministic natural of our pixel synthesis process; excessive iterations will make the result more repetitive and losing the natural randomness of input textures. In these experiments, we can see that the highest resolution of most textures take less than 10 iterations to settle down. In addition, the neighborhood matching error doesn't seem to decrease much with adding iterations. These facts demonstrate that it is sufficient to run the algorithm with a few iterations; in our experience running two iterations works reasonably well.

We might be able to achieve complete convergence by using a weighted version of our algorithm. Instead of copying pixels directly from input to output, we copy by weighted blending. In Figure 9.8, we have shown results of such an experiment. The blending weight is chosen so that after 100 iterations the "validity" of each pixel is at least 0.999. Since we have noted before that the oscillations only happen for a small fraction of the pixels (using a random order) and complete convergence does not imply good image quality (such as texture 759 in Figure 9.7), this weighted blending does not seem to pay off. A possible future research direction is to figure out how to choose the blending weight properly; however, since we believe that adding more iterations will not help the image quality it doesn't seem to be a very useful research direction.

9.4 Algorithm Evolution

Throughout this thesis we have shown several generations of the algorithm, involving different synthesis orders and neighborhood shapes. The first generation of the algorithm, using causal neighborhoods and scanline ordering, is inspired by traditional MRF approaches. One crucial observation we made is that it is important to restrain the containment of noise pixels in the output neighborhoods; if you use a scanline order and a noncausal neighborhood the synthesis result will not be good, since a symmetric neighborhood will always contain noise pixels, at least during the first pass of the algorithm. (Adding more passes doesn't seem to help.) However, if we use a causal neighborhood with scanline order, then the noise will only effect the first few rows and columns of the output. In addition, since the amount of noise pixels decreases as we generate more and more valid output pixels, their effect can be further constrained using the coherence of the neighborhood vectors as argued in the previous sections. Specifically, assuming that we just finish synthesizing pixel (x_o, y_o) located somewhere in the upper or left portion of the output texture (so that its neighborhood contains noise). Now we move on to the next pixel $(x_o + 1, y_o)$. Due to the natural of scanline order, we know that $N(x_o + 1, y_o)$ contains no more noise than $N(x_o, y_o)$, and they are coherent. In previous sections we argued that a random noise vector is likely to be equally far away from its neighbors in a high dimensional space. We can argue similarly that the noise portions of $N(x_o + 1, y_o)$ will equally favor the input neighborhoods, and the valid portion of it will dominate the determination of the best match. Due to the coherence of the valid portion of $N(x_o, y_o)$ and $N(x_o + 1, y_o)$, it is likely that the match found for $(x_o + 1, y_o)$, $M(x_o + 1, y_o)$, will be $M(x_o, y_o) + (1, 0)$. And since the amount of noise diminishes as we continue the synthesis process, we will have more and more coherence between adjacent output neighborhoods.

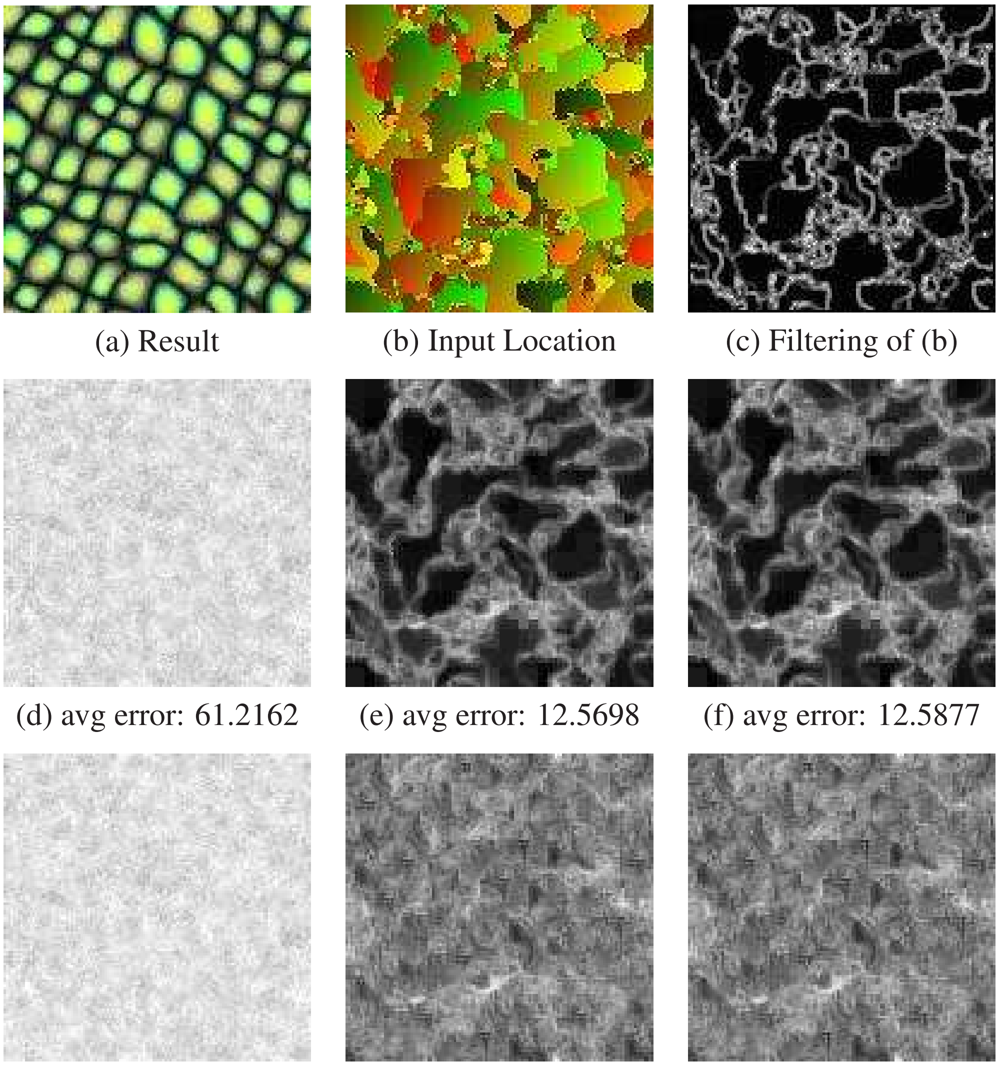
The scanline order, although adequate for synthesis, is too limited for other applications such as hole-filling or implicit texture evaluation. We have relaxed this scanline order requirement by carefully constructing the combination of synthesis order and neighborhood shapes so that we don't have the noise problem mentioned earlier. We use a general approach to take care of the noise in the multi-resolution framework. Before synthesizing each resolution, we use the information from the (already synthesized) lower resolution to generate an initial guess to the current resolution. This is done by running the texture synthesis algorithm using neighborhoods containing lower resolution information only, as described in Chapter 6. This process can be considered as doing image extrapolation (or super-resolution) using our texture synthesis framework. The extrapolated information, although not completely valid, usually preserves the rough feature of the texture pattern and is at least much better than noise. After this, we are free to use symmetric neighborhood with random synthesis order to produce valid output textures. In general, it suffices to use a random order for this two pass algorithm, and we have found no major degradation to the image quality. However, we restrain the synthesis order to be a diffusion process when dealing with constrained synthesis such as hole filling; it is always better to fill the holes by growing from the boundary of the hole toward the middle.

These extensions, although flexible, still have one major drawback. Even given the same initial noise, visiting output pixels in different orders will produce different results. This prohibits us from evaluating texture implicitly while guarantee the consistency of the result. We address this issue by further relaxing the algorithm, allowing pixels to be visited in an arbitrary order while maintaining the same result. In other words, now we can evaluate our texture algorithm much like an implicit procedural texture synthesis code. Conceptually, the price we paid is an extra buffer; however, in actual implementation this can done using clever caching as described in Chapter 8.

Figure 9.9 show the relationship between different versions of our algorithm. Basically, generations on the outside are more general than those inside. However, we have to note that more restricted versions sometimes give better results, so there is a trade-off between generality and quality. You should choose the generation depending on your application: use scanline order for ordinary synthesis, use diffusion order for hole filling, and use independent-order for implicit evaluation. The only common feature between those generations are the neighborhood search framework, and the criteria of restraining noise in the output neighborhoods during synthesis.

9.5 Conclusions

We have analyzed several aspects of our algorithm. We have shown that textures generated by our approach can form patches. This patching behavior provides insights into why our algorithm works, connects our algorithm with several previous methods, and provides insights on potential acceleration techniques. We have also explored the convergence property of our algorithm. In general, convergence does not seem to relate to synthesis quality, and adding excessive iterations to achieve convergence can only degrade image quality and incur unnecessary computation.

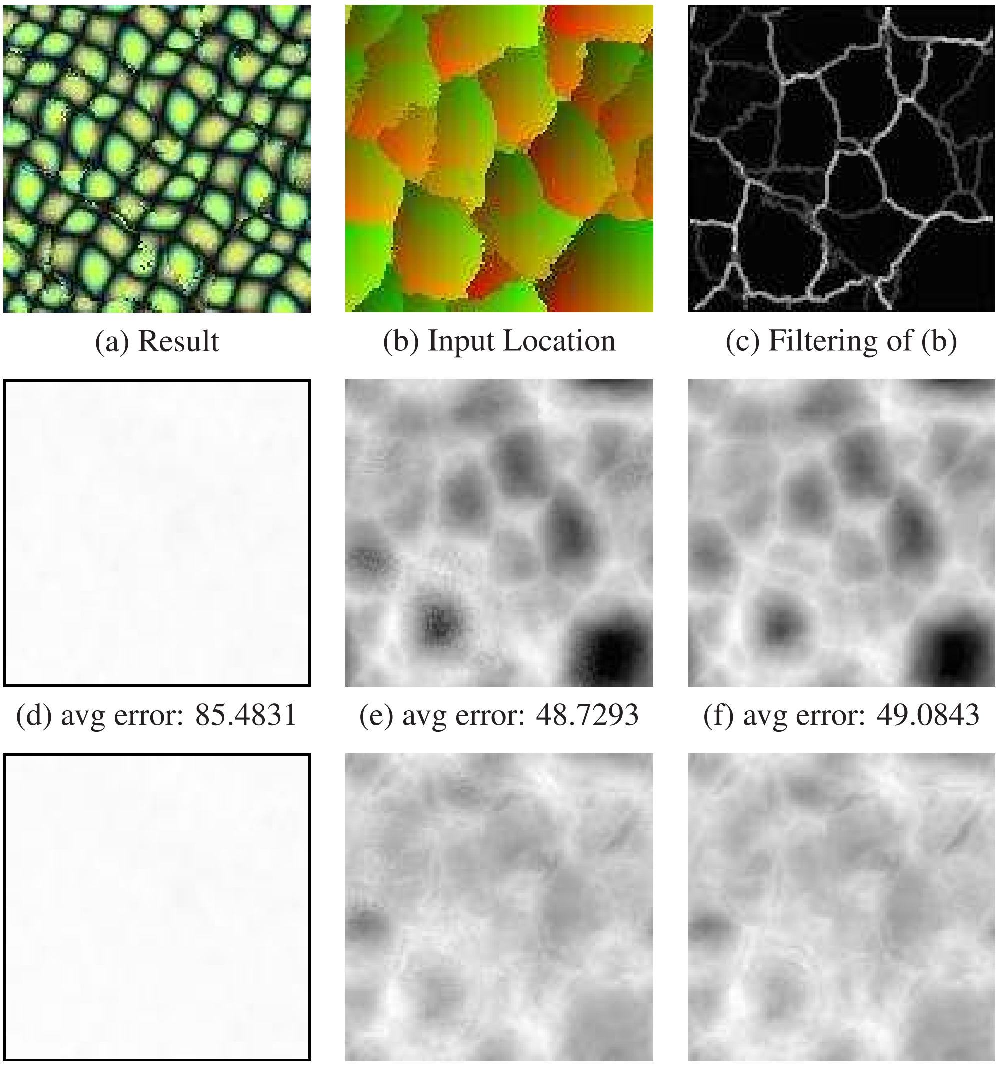


(a) Result (b) Input Location (c) Filtering of (b)

(d) avg error: 61.2162 (e) avg error: 12.5698 (f) avg error: 12.5877

(g) avg error: 62.1729 (h) avg error: 20.6567 (i) avg error: 20.6931

Figure 9.2: *Patching behavior of texture synthesis. (a) Synthesis result. (b) Location image, with row, column positions color-encoded by red and green, respectively. (c) High-pass filtered version of (b). (d)-(i) neighborhood matching errors. The first column (d, g) shows the error before synthesis (i.e. the noise), the second column (e, h) shows the error measured during synthesis, and the last column (f, i) shows the error measured after synthesis. The first row (d, e, f) shows the error of the best match, while the second row (g, h, i) shows the second best match. The average error, measured per pixel per color channel, is indicated below each figure. The color images have pixel range in [0, 255]. The synthesis parameters are as follows: 4-level Gaussian pyramids with neighborhood sizes 9x9x2 at each resolution; random synthesis order with 2 passes.*

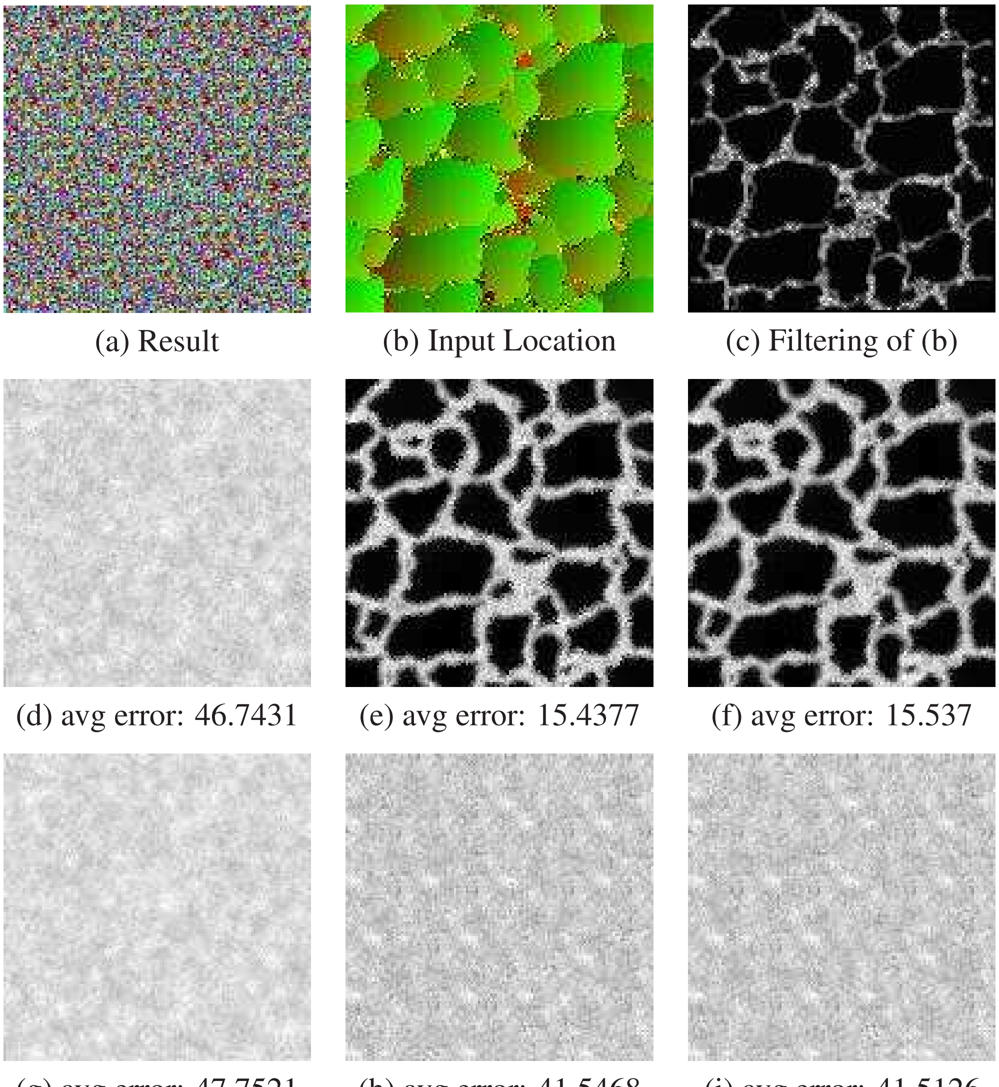


(a) Result (b) Input Location (c) Filtering of (b)

(d) avg error: 85.4831 (e) avg error: 48.7293 (f) avg error: 49.0843

(g) avg error: 85.6761 (h) avg error: 56.8515 (i) avg error: 57.0463

Figure 9.3: *Patching behavior of texture synthesis. A similar experiment with Figure 9.2, but a much larger neighborhood is used. The synthesis parameters are as follows: 2-level Gaussian pyramids with neighborhood sizes 41x41x2 at each resolution; random synthesis order with 2 passes.*



(a) Result (b) Input Location (c) Filtering of (b)

(d) avg error: 46.7431 (e) avg error: 15.4377 (f) avg error: 15.537

(g) avg error: 47.7521 (h) avg error: 41.5468 (i) avg error: 41.5126

Figure 9.4: *Patching behavior of texture synthesis. We use a white random noise as input to test the patching behavior. Synthesis parameters are the same as in Figure 9.2.*

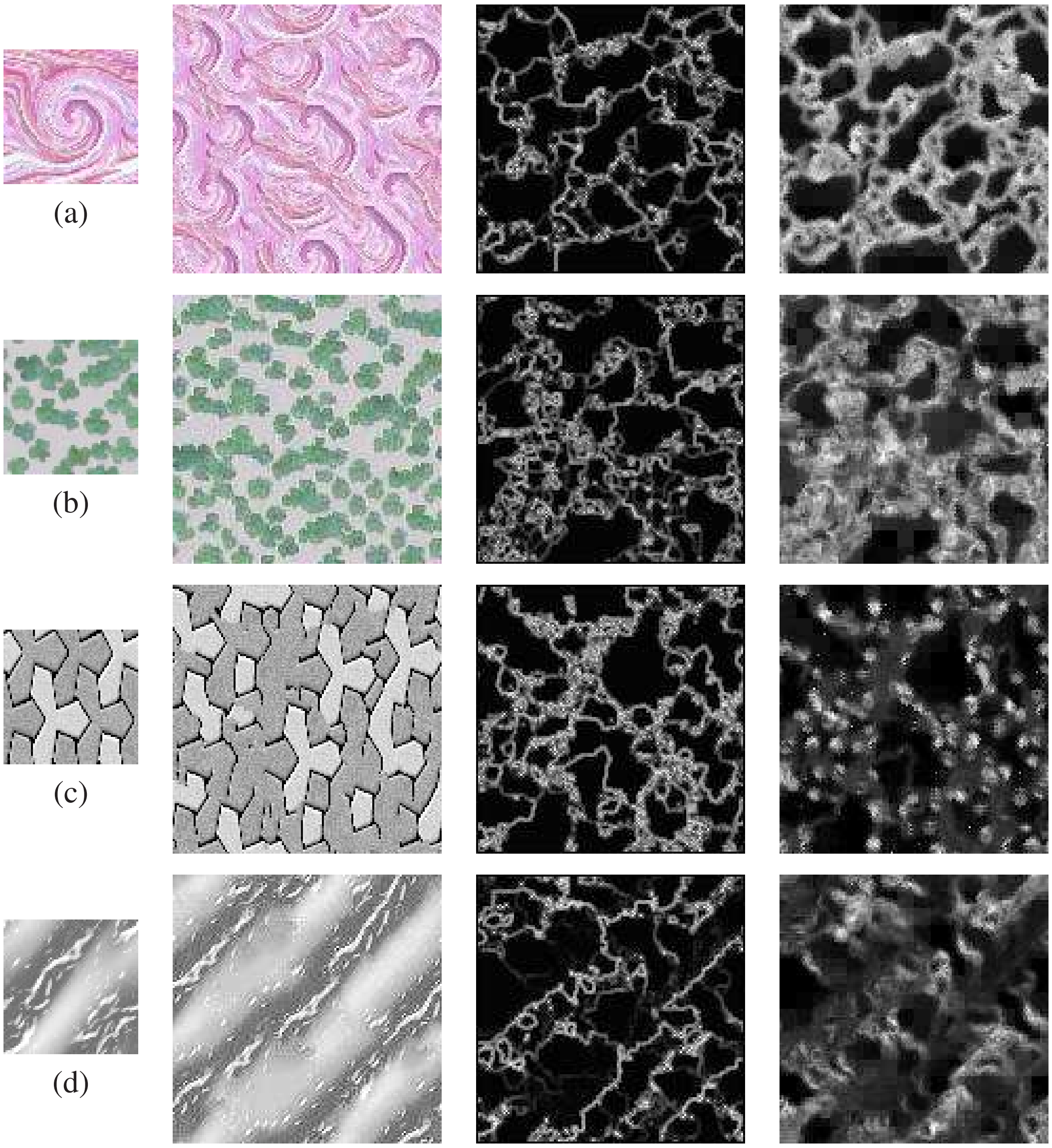


Figure 9.5: *Patching behavior for different artificial textures.* 1^{st} *column: sample texture.* 2^{nd} *column: synthesis result.* 3^{rd} *column: patch boundary for the result.* 4^{th} *column: average synthesis error. The average neighborhood matching errors for each case are: (a) 9.072 (b) 8.7323 (c) 11.1446 (d) 7.6536. Synthesis parameters are the same as Figure 9.2.*

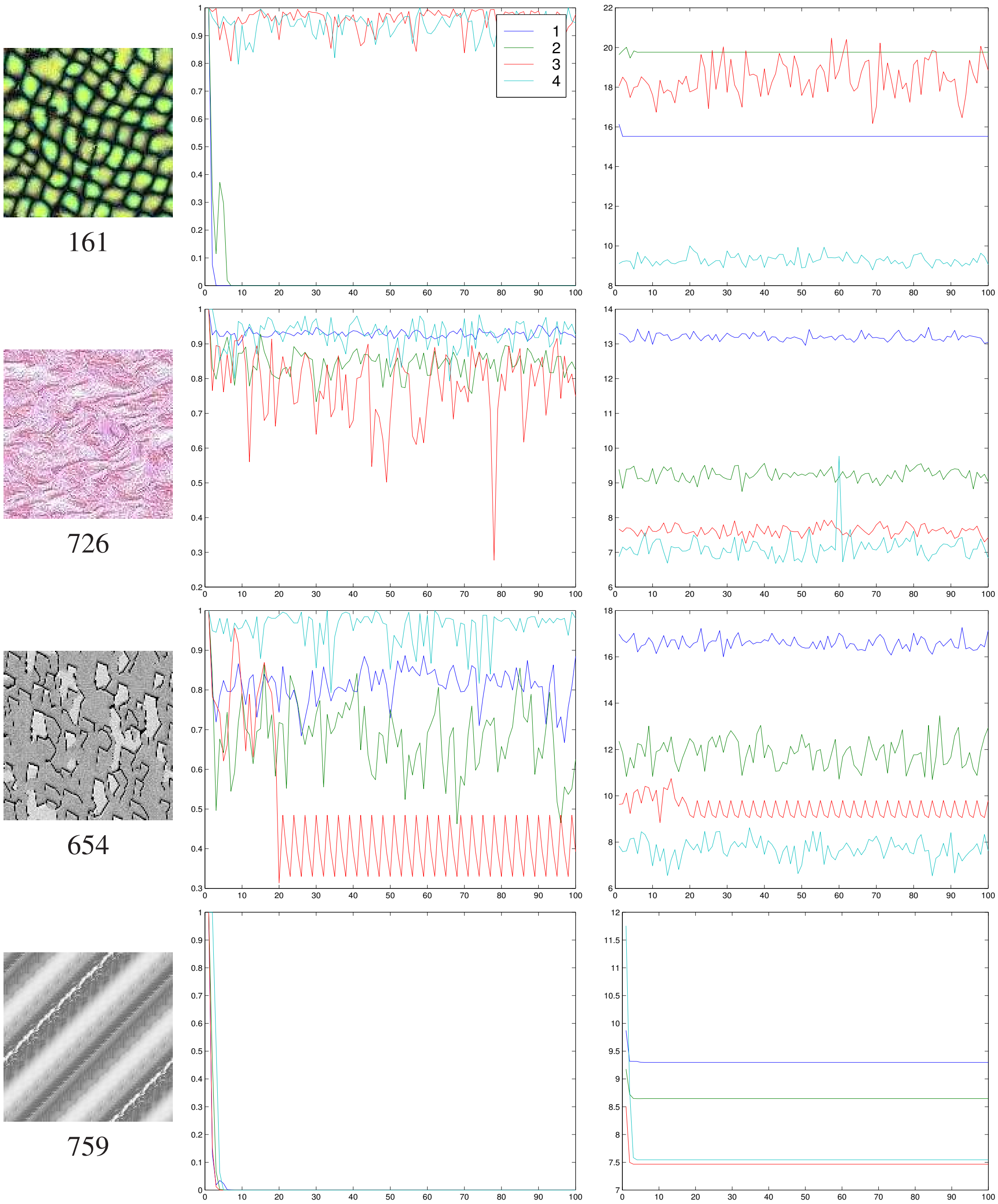


Figure 9.6: *The convergence of texture synthesis. First column: synthesis results. Second column: percentage of pixels changed for each iteration. Each resolution is plotted with a separate curve, and level 1 is the highest resolution. Third column: normalized neighborhood matching errors for each iteration. A scanline order with 4-level Gaussian pyramids are used for all cases.*

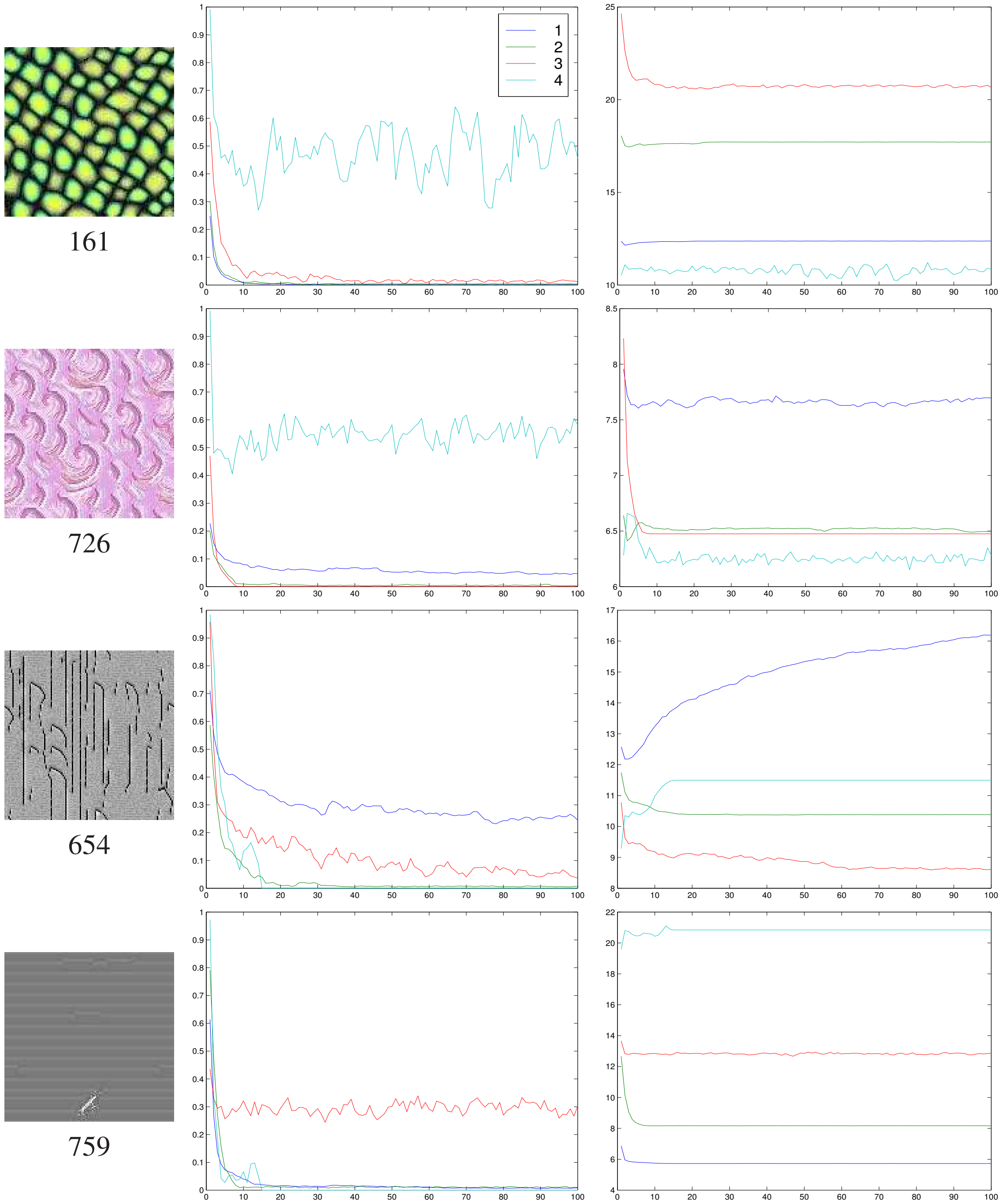


Figure 9.7: *The convergence of texture synthesis. Similar to Figure 9.6, but we use a random synthesis order in this case.*

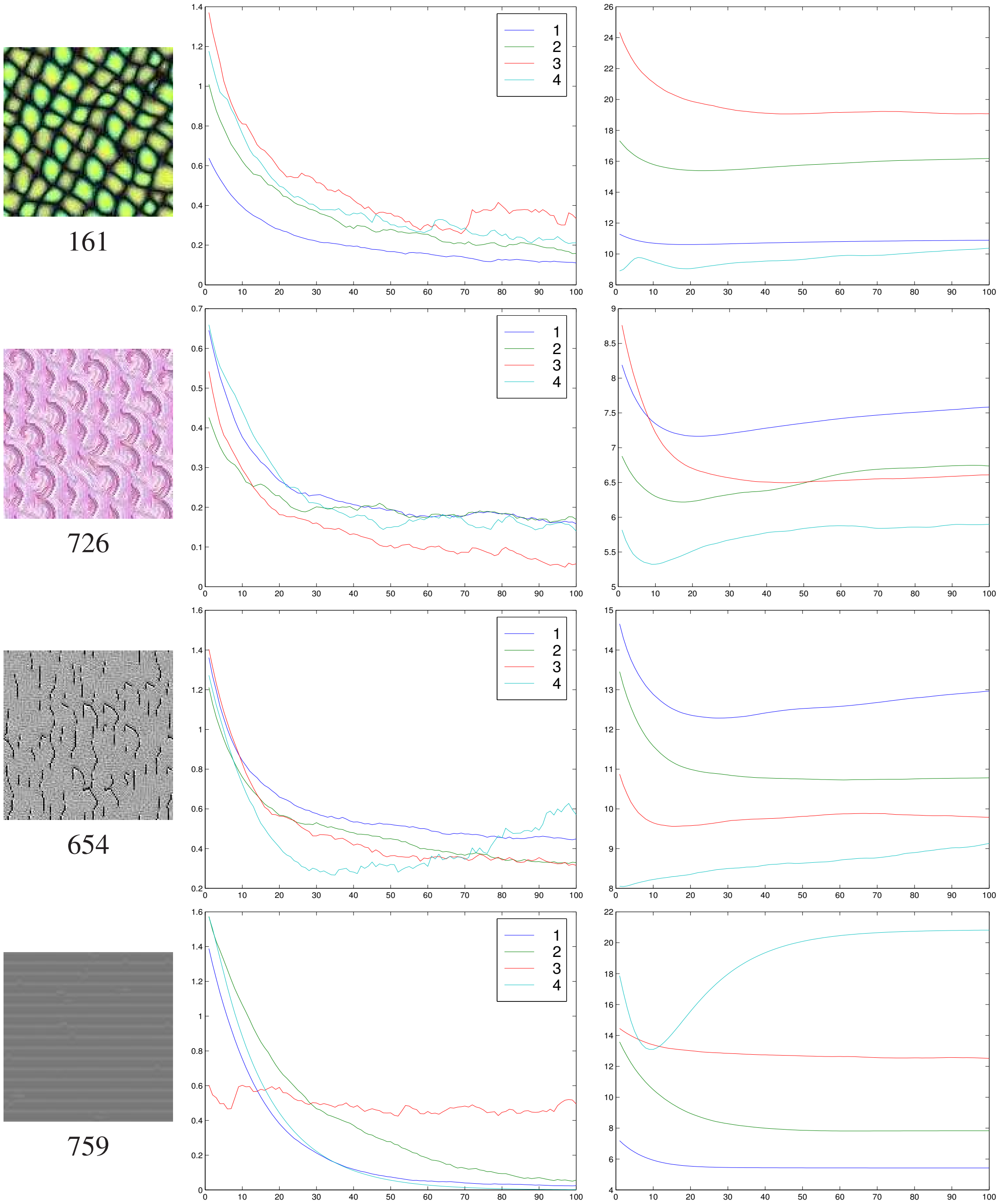


Figure 9.8: *The convergence of texture synthesis. Similar to Figure 9.7, but we use a weighted blending with weight 0.066746 in this case. In the middle column, we plot the mean pixel change instead of percentage of pixels changed (since almost all pixels will be touched).*

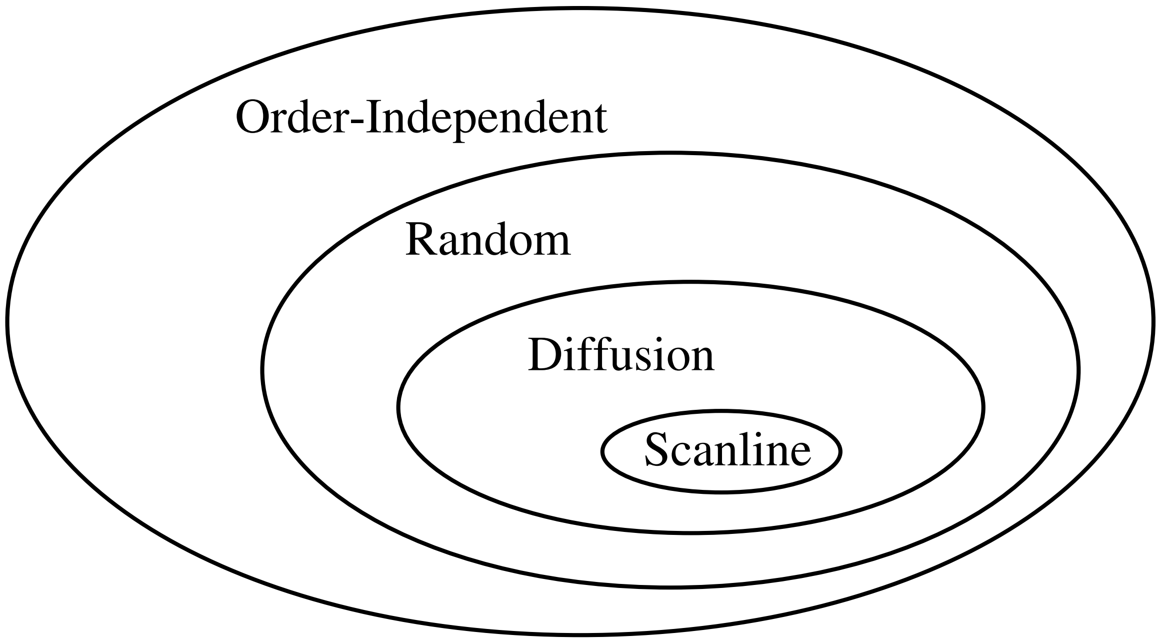


Figure 9.9: *Relationship between different generations of our algorithm. From inside to outside are different generations of our algorithm : scanline order + causal neighborhood, diffusion order (for constrained synthesis) + non-causal neighborhood, random order + non-causal neighborhood, and independent order + non-causal neighborhood. Both diffusion and random orders require a two pass algorithm with the first pass as the extrapolation. Order-independent requires two passes as well as an extra buffer.*

Chapter 10

Conclusions and Future Work

Textures are important for a wide variety of applications in computer graphics and image processing. On the other hand, they are hard to synthesize. The goal of this thesis is to provide a practical tool for efficiently synthesizing a broad range of textures. Inspired by Markov Random Field methods, our algorithm is general: a wide variety of textures can be synthesized without any knowledge of their physical formation processes. The algorithm is also efficient: by a proper acceleration using TSVQ, typical image textures can be generated within seconds on current PCs and workstations. The algorithm is also easy to use: only an example texture patch is required.

The simplicity of generality of our algorithm allow us to extend it in various ways. We have modified our approach for constrained synthesis, or for synthesizing textures directly over manifold meshes. We have extended the notion of textures over other physical phenomena including temporal textures, articulated motion signals, and geometric details. We have also presented new methods to create textures that previously do not exist, by producing texture mixtures from multiple sources and generating solid textures from several planar views.

We envision several possible directions for future work:

10.1 Modeling Geometric Details by Displacement Maps

Models scanned from real world objects often contain texture-like geometric details, making the models expensive to store, transmit or manipulate. These geometric details can be represented as displacement maps over a smoother surface representation [40]. The resulting displacement maps should be compressible/decomposable as 2D textures using our technique. Taking this idea further, missing geometric details, a common problem in many scanning situations [41], could be filled in using our constrained texture synthesis technique.

We can generate displacement maps directly over manifold meshes using our surface texture synthesis algorithm (Chapter 6). We can also combine our approach with volumetric diffusion to fill large and complex holes in scanned models.

10.2 Multi-dimensional Texture

The notion of texture extends naturally to multi-dimensional data. Examples presented in this thesis include temporal textures, articulated motion sequences, and solid textures (Chapter 7). The same technique can also be directly applied to generate animated solid textures or light fields textures.

10.3 Texture Compression/Decompression

Textures usually contain repeating patterns and high frequency information; therefore they are not well compressed by transform-based techniques such as JPEG. However, codebook-based compression techniques work well on textures [4]. This suggests that textures might be compressible by our synthesis technique. Compression would consist of building a codebook, but unlike [4], no code indices would be generated; only the codebook would be transmitted and the compression ratio is controlled by the number of codewords. Decompression would consist of texture synthesis. This decompression step, if accelerated one more order of magnitude over our current software implementation, could be usable for real time texture mapping. The advantage of this approach over [4] is much greater compression, since only the codebook is transmitted.

10.4 Super-resolution

Our constrained synthesis algorithm (Chapter 4) can also be applied to fabricate high detailed images from low resolution ones [34, 21]. An example is shown in Figure 10.1. Given a low resolution image (Figure 10.1 (a)), we would like to enlarge it while at the same time adding high frequency details. One naive approach is to use extrapolation (Figure 10.1 (b)). Because no high-frequency information is added, extrapolation can only generate blurry results. What will happen if we are given some training set from a higher resolution image (red squares in Figure 10.1 (c))? Given such information, we can then run our constrained synthesis algorithm to fabricate the high frequency information as follows. We use the low-resolution image as "constraints", and generate the extra high-frequency levels using the given training set. We treat those training set as textures, and simply run our constrained synthesis algorithm over those missing high-resolution levels. The synthesis result is shown in Figure 10.1 (d). It has the high frequency structures better preserved than simple extrapolation (Figure 10.1 (e) and (f)).

10.5 Texture-based Rendering

The basic principle of searching neighborhoods in our texture synthesis algorithm can be applied to other domains such as rendering images, motions, and videos. For example, Hertzmann et al [34] has extended texture synthesis algorithms to *Image Analogies* and applied it to a wide variety of applications such as artistic filters, texture-transfer, and texture-by-numbers. Similar ideas for transferring textures from one object to another have also been presented by Ashikhmin [1] and Efros [20]. The success of these approaches shows the promise of using simple local statistics to model complicated natural phenomena. One could imagine continuing this direction and apply texture synthesis to other rendering problems such as images, motions, or animations.

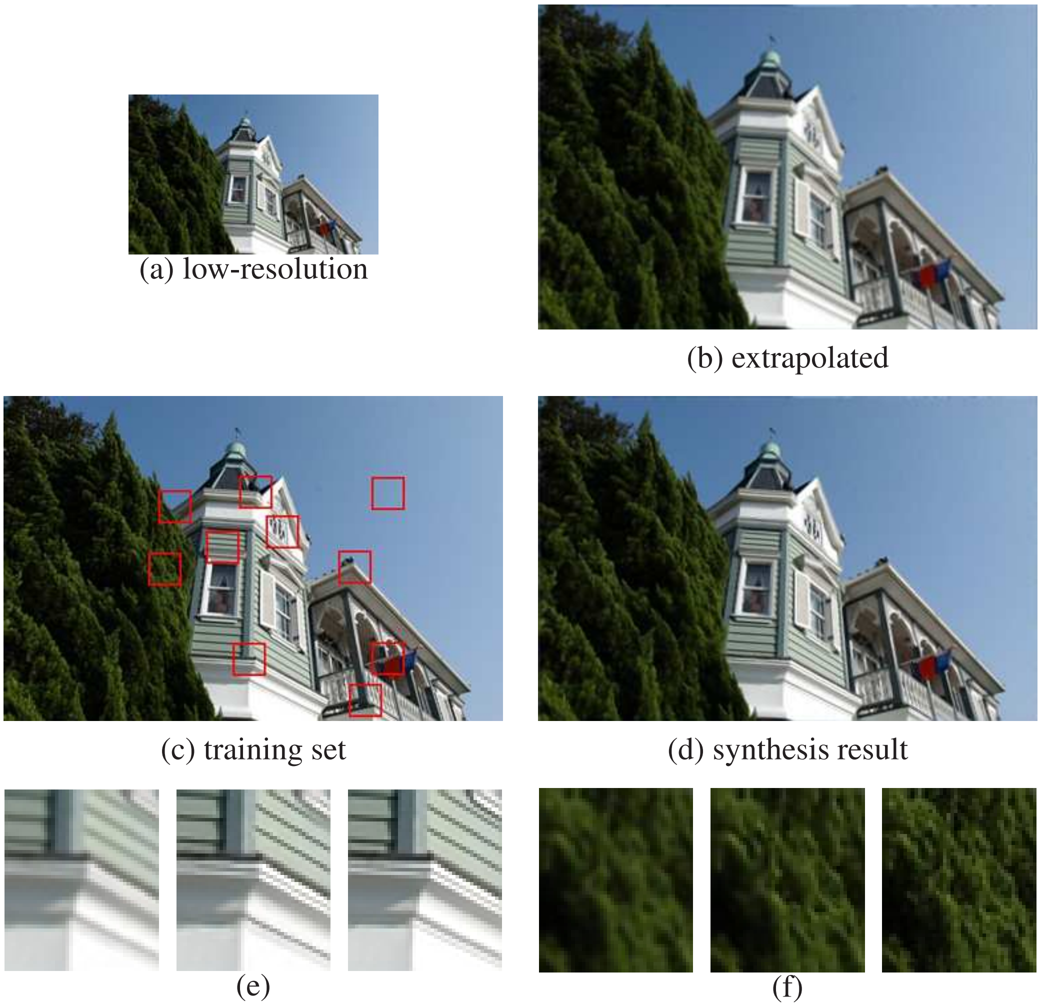


Figure 10.1: *Super-resolution by constrained synthesis. (a) low-resolution image, (b) enlarged image by simple extrapolation, (c) training image with high resolution, (d) super-resolution result by constrained synthesis, (e) and (f) enlarged versions comparing (b) and (d). For each group of images the one on the left is cropped from (b), the one on the middle is cropped at the same location from (d), and the one on the right is the original high resolution image.*

Bibliography

[1] Michael Ashikhmin. Synthesizing natural textures. *2001 ACM Symposium on Interactive 3D Graphics*, pages 217–226, March 2001. ISBN 1-58113-292-1.

[2] Ziv Bar-Joseph, Ran El-Yaniv, Dani Lischinski, and Mike Werman. Texture mixing and texture movie synthesis using statistical learning. *SIGGRAPH 2000 Sketch*, pages 266–266, July 2000.

[3] David Baraff and Andrew Witkin. Large steps in cloth simulation. *Proceedings of SIGGRAPH 98*, pages 43–54, July 1998.

[4] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. *Proceedings of SIGGRAPH 96*, pages 373–378, August 1996.

[5] Marcelo Bertalmio, Guillermo Sapiro, Vicent Caselles, and Coloma Ballester. Image inpainting. *Proceedings of SIGGRAPH 2000*, pages 417–424, July 2000.

[6] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542—546, 1976.

[7] Matthew Brand and Aaron Hertzmann. Style machines. *Proceedings of SIGGRAPH 2000*, pages 183–192, July 2000. ISBN 1-58113-208-5.

[8] P. Brodatz. *Textures: A Photographic Album for Artists and Designers*. Dover, New York, 1966.

[9] Armin Bruderlin and Lance Williams. Motion signal processing. *Proceedings of SIGGRAPH 95*, pages 97–104, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[10] P. J. Burt and E. H. Adelson. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics*, 2(4):217–236, October 1983.

[11] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Computer Science Department, University of Utal, Salt Lake City, Utah, 1974.

[12] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini. Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer*, 15(10):519–539, 1999. ISSN 0178-2789.

[13] Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 361–368. ACM SIGGRAPH, Addison Wesley, August 1997.

[14] Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 361–368. ACM SIGGRAPH, Addison Wesley, August 1997.

[15] J. M. Dischler, D. Ghazanfarpour, and R. Freydier. Anisotropic solid texture synthesis using orthogonal 2D views. *Computer Graphics Forum*, 17(3):87–96, 1998.

[16] Julie Dorsey, Alan Edelman, Justin Legakis, Henrik Wann Jensen, and Hans Khling Pedersen. Modeling and rendering of weathered stone. *Proceedings of SIGGRAPH 99*, pages 225–234, August 1999.

[17] Dr. Strous's Answer Book. Mass, size, and density of the universe. http://louis.lmsal.com/PR/answerbook/universe.html#q70.

[18] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers, 1998.

[19] Alexei Efros and Thomas Leung. Texture synthesis by non-parametric sampling. In *International Conference on Computer Vision*, volume 2, pages 1033–8, Sep 1999.

[20] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. *Proceedings of SIGGRAPH 2001*, Auguest 2001.

[21] William Freeman and Egon Pasztor. Markov networks for super-resolution. Technical Report TR-2000-08, Mitsubishi Electric Research Laboratory, March 2000.

[22] A. Gagalowicz and Song-Di-Ma. Model driven synthesis of natural textures for 3-D scenes. *Computers and Graphics*, 10(2):161–170, 1986.

[23] Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.

[24] Djamchid Ghazanfarpour and Jean-Michel Dischler. Generation of 3D texture using multiple 2D models analysis. *Computer Graphics Forum*, 15(3):311–324, August 1996. Proceedings of Eurographics '96. ISSN 1067-7055.

[25] Michael Gleicher. Motion editing with spacetime constraints. *1997 Symposium on Interactive 3D Graphics*, pages 139–148, April 1997. ISBN 0-89791-884-3.

[26] Michael Gleicher. Retargeting motion to new characters. *Proceedings of SIGGRAPH 98*, pages 33–42, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[27] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, 1989.

[28] Radek Grzeszczuk, Demetri Terzopoulos, and Geoffrey Hinton. Neuroanimator: Fast neural network emulation and control of physics-based models. *Proceedings of SIGGRAPH 98*, pages 9–20, July 1998. ISBN 0-89791-999-8. Held in Orlando, Florida.

[29] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[30] Igor Guskov, Wim Sweldens, and Peter Schröder. Multiresolution signal processing for meshes. *Proceedings of SIGGRAPH 99*, pages 325–334, August 1999.

[31] Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. *24th International Symposium on Computer Architecture*, 1997.

[32] R.M. Haralick. Statistical image texture analysis. In *Handbook of Pattern Recognition and Image Processing*, volume 86, pages 247–279. Academic Press, 1986.

[33] David J. Heeger and James R. Bergen. Pyramid-Based texture analysis/synthesis. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 229–238. ACM SIGGRAPH, Addison Wesley, August 1995.

[34] Aaron Hertzmann, Charles Jacobs, Nuria Oliver, Brian Curless, and David Salesin. Image analogies. *Proceedings of SIGGRAPH 2001*, Auguest 2001.

[35] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. *Proceedings of SIGGRAPH 2000*, pages 517–526, July 2000. ISBN 1-58113-208-5.

[36] A. N. Hirani and T. Totsuka. Combining frequency and spatial domain information for fast interactive image noise removal. *Computer Graphics*, 30(Annual Conference Series):269–276, 1996.

[37] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a texture cache architecture. *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 133–142, August 1998. Held in Lisbon, Portugal.

[38] Homan Igehy and Lucas Pereira. Image replacement through texture synthesis. In *International Conference on Image Processing*, volume 3, pages 186–189, Oct 1997.

[39] H. Iversen and T. Lonnestad. An evaluation of stochastic models for analysis and synthesis of gray scale texture. *Pattern Recognition Letters*, 15:575–585, 1994.

[40] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. *Proceedings of SIGGRAPH 96*, pages 313–324, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[41] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan

Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. *Proceedings of SIGGRAPH 2000*, pages 131–144, 2000.

[42] Bruno Lévy and Jean-Laurent Mallet. Non-distorted texture mapping for sheared triangulated meshes. *Proceedings of SIGGRAPH 98*, pages 343–352, July 1998.

[43] Jérôme Maillot, Hussein Yahia, and Anne Verroust. Interactive texture mapping. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 27–34, August 1993.

[44] T. Malzbender and S. Spach. A context sensitive texture nib. In *Proceedings of Computer Graphics International*, pages 151–163, June 1993.

[45] MIT Media Lab. Vision texture. http://www-white.media.mit.edu/vismod/imagery/-VisionTexture/vistex.html.

[46] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:989–1003, 1997.

[47] Fabrice Neyret and Marie-Paule Cani. Pattern-based texturing revisited. *Proceedings of SIGGRAPH 99*, pages 235–242, August 1999.

[48] R. Paget and I.D. Longstaff. Texture synthesis via a noncausal nonparametric multiscale Markov random field. *IEEE Transactions on Image Processing*, 7(6):925–931, June 1998.

[49] Hans Køhling Pedersen. Decorating implicit surfaces. *Proceedings of SIGGRAPH 95*, pages 291–300, August 1995.

[50] Ken Perlin. An image synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):287–296, July 1985. Held in San Francisco, California.

[51] Ken Perlin and Athomas Goldberg. Improv: A system for scripting interactive actors in virtual worlds. *Proceedings of SIGGRAPH 96*, pages 205–216, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[52] Ashok C. Popat. *Conjoint Probabilistic Subband Modeling*. PhD thesis, Massachusetts Institute of Technology, 1997.

[53] K. Popat and R.W. Picard. Novel cluster-based probability model for texture synthesis, classification, and compression. In *Visual Communications and Image Processing*, pages 756–68, 1993.

[54] Javier Portilla and Eero P Simoncelli. Texture modeling and synthesis using joint statistics of complex wavelet coefficients. In *IEEE Workshop on Statistical and Computational Theories of Vision*, June 1999.

[55] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. *Proceedings of SIGGRAPH 2000*, pages 465–470, July 2000.

[56] Katheline Pullen and Chris Bregler. Animating by multi-level sampling. *IEEE Computer Animation Conference 2000*, May 2000.

[57] A.R. Rao. *A Taxonomy for Texture Description and Identification*. Springer-Verlag, 1990.

[58] Peter Schröder and Wim Sweldens. Spherical wavelets: Efficiently representing functions on the sphere. *Proceedings of SIGGRAPH 95*, pages 161–172, August 1995.

[59] E. Simoncelli and J. Portilla. Texture characterization via joint statistics of wavelet coefficient magnitudes. In *Fifth International Conference on Image Processing*, volume 1, pages 62–66, October 1998.

[60] Karl Sims. Artificial evolution for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):319–328, July 1991. ISBN 0-201-56291-X. Held in Las Vegas, Nevada.

[61] Karl Sims. Evolving virtual creatures. *Proceedings of SIGGRAPH 94*, pages 15–22, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[62] G. Smith. *Image Texture Analysis using Zero Crossings Information*. PhD thesis, Department of Computer Science and Electrical Engineering, University of Queensland, St Lucia 4072, Australia, 1998.

[63] Jos Stam. Stochastic dynamics: Simulating the effects of turbulence on flexible structures. *Computer Graphics Forum*, 16(3):159–164, August 1997.

[64] Jos Stam. Stable fluids. *Proceedings of SIGGRAPH 99*, pages 121–128, August 1999.

[65] Jos Stam and Eugene Fiume. Depicting fire and other gaseous phenomena using diffusion processes. *Proceedings of SIGGRAPH 95*, pages 129–136, August 1995.

[66] Martin Szummer and Rosalind W. Picard. Temporal texture modeling. In *International Conference on Image Processing*, volume 3, pages 823–6, Sep 1996.

[67] S. Todd and W. Latham. *Evolutionary Art and Computers*. Boston, MA: Acadamiec Press, 1993.

[68] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):55–64, July 1992.

[69] Greg Turk. Texture synthesis on surfaces. *Proceedings of SIGGRAPH 2001*, pages 347–354, August 2001. ISBN 1-58113-292-1.

[70] Munetoshi Unuma, Ken Anjyo, and Ryozo Takeuchi. Fourier principles for emotion-based human figure animation. *Proceedings of SIGGRAPH 95*, pages 91–96, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[71] L. VanGool, P. Dewaele, and A. Oosterlinck. Texture analysis anno 1983. *Computer Vision, Graphics, and Image Processing*, 29(3):336–357, March 1985.

[72] Li-Yi Wei. Deterministic texture analysis and synthesis using tree structure vector quantization. In *XII Brazilian Symposium on Computer Graphics and Image Processing*, pages 207–213, October 1999.

[73] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. *Proceedings of SIGGRAPH 2000*, pages 479–488, July 2000.

[74] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. *Proceedings of SIGGRAPH 2001*, pages 355–360, August 2001. ISBN 1-58113-292-1.

[75] Henrik Weimer and Joe Warren. Subdivision schemes for fluid flow. *Proceedings of SIGGRAPH 99*, pages 111–120, August 1999.

[76] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. *Proceedings of SIGGRAPH 94*, pages 91–100, July 1994. ISBN 0-89791-667-0. Held in Orlando, Florida.

[77] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. *Proceedings of SIGGRAPH 96*, pages 469–476, August 1996. ISBN 0-201-94800-1. Held in New Orleans, Louisiana.

[78] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 299–308, July 1991.

[79] Andrew Witkin and Zoran Popovic. Motion warping. *Proceedings of SIGGRAPH 95*, pages 105–108, August 1995. ISBN 0-201-84776-0. Held in Los Angeles, California.

[80] Patrick Witting. Computational fluid dynamics in a traditional animation environment. *Proceedings of SIGGRAPH 99*, pages 129–136, August 1999.

[81] Steven P. Worley. A cellular texture basis function. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 291–294. ACM SIGGRAPH, Addison Wesley, August 1996.

[82] Ying-Qing Xu, Baining Guo, and Harry Shum. Chaos mosaic: Fast and memory efficient texture synthesis. Technical Report MSR-TR-2000-32, Microsoft Research, 2000.

[83] S. Zhu, Y. Wu, and D. Mumford. Filters, random fields and maximun entropy (FRAME) - towards a unified theory for texture modeling. *International Journal of Computer Vision*, 27(2):107–126, 1998.