

# Fast Texture Synthesis using Tree-structured Vector Quantization

Category: Research  
Paper number: 0015

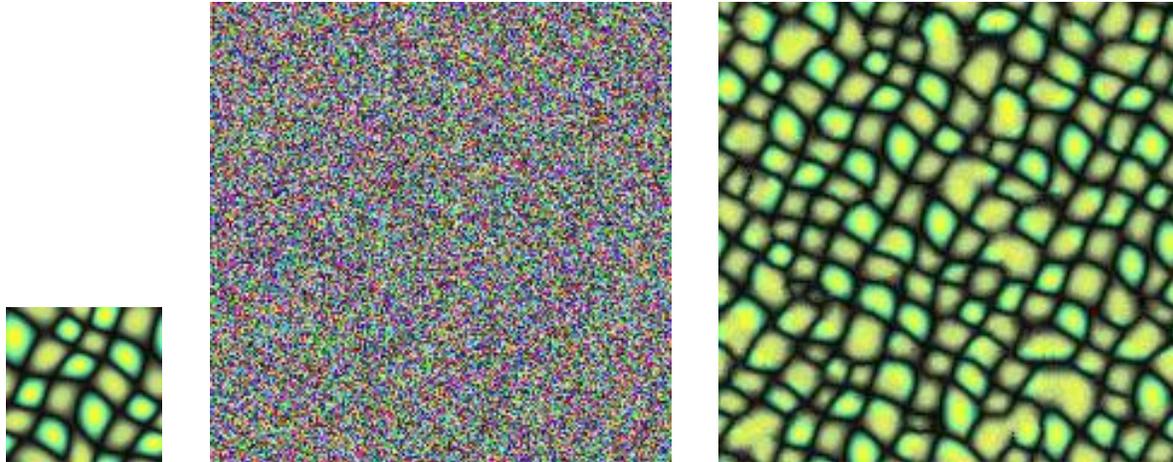


Figure 1: Our texture generation process takes an example texture patch (left) and a random noise (middle) as input, and modifies this random noise to make it look like the given example texture. The synthesized texture (right) can be of arbitrary size, and is perceived as very similar to the given example. Using our algorithm, textures can be generated within seconds, and the synthesized results are always tileable.

## Abstract

Texture synthesis is important for many applications in computer graphics, vision, and image processing. However, it remains difficult to design an algorithm that is both efficient and capable of generating high quality results. In this paper, we present an efficient algorithm for realistic texture synthesis. 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. This permits us to apply texture synthesis to problems where it has traditionally been considered impractical. In particular, we have applied it to constrained synthesis for image editing and temporal texture generation. Our algorithm is derived from Markov Random Field texture models, and generates textures through a deterministic searching process. We accelerate this synthesis process using tree-structured vector quantization.

**Keywords:** Texture Synthesis, Compression Algorithms, Image Processing

## 1 Introduction

Texture is a ubiquitous visual experience. It can describe a wide variety of surface characteristics such as terrain, plants, minerals, fur and skin. Since reproducing the visual realism of the real world is a major goal for computer graphics, textures are commonly employed when rendering 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 of any size, visual repetition is avoided. Texture synthesis can also produce tileable images by properly handling the boundary conditions. Potential applications of texture synthesis are also broad; some examples are image denoising, occlusion fill-in, and compression.

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 stochastic process. The major challenges are 1) Modeling- how to estimate the stochastic process from a given finite texture sample and 2) Sampling- how to develop an efficient sampling procedure to produce new textures from a given model. Both the modeling and sampling parts are essential for the success of texture synthesis; the visual fidelity of generated textures will depend primarily on the accuracy of the modeling, while the efficiency of the sampling procedure will directly determine the computational cost of texture generation.

In this paper, 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 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 key advantages of this algorithm are quality and speed; the quality of the synthesized textures are equal to or better than those generated by previous techniques, while the computation speed is 2 orders of magnitude faster than those approaches that generate comparable results to our algorithm. This permits us to apply our

algorithm in areas where texture synthesis has traditionally been considered too expensive. In particular, we have extended the algorithm to constrained synthesis for image editing and motion texture synthesis.

## 1.1 Previous Work

Numerous approaches have been proposed for texture analysis and synthesis, and an exhaustive survey is beyond the scope of this paper. We briefly review some recent and representative works and refer the reader to [9] and [13] for more complete surveys.

**Physical Simulation:** It is possible to synthesize certain surface textures by directly simulating their physical generation process. Biological patterns such as fur, scales, and skin can be modeled using reaction diffusion ([28]) and cellular texturing ([29]). Some weathering and mineral phenomenon can be faithfully reproduced by detailed simulations ([6]). 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 process so these approaches are applicable to only limited classes of textures.

**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 ([7], [30], [21], [19]). 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.

**Feature Matching:** Some algorithms model textures as a set of features, and generate new images by matching the features in an example texture ([10], [5], [23]). These algorithms are usually more efficient than Markov Random Field algorithms. Heeger and Bergen ([10]) model textures by matching marginal histograms of image pyramids. Their technique succeeds on highly stochastic textures but fails on more structured ones. De Bonet ([5]) synthesizes new images by randomizing an input texture sample while preserving the cross-scale dependencies. This method works better than [10] on structured textures, but it can produce boundary artifacts if the input texture is not tileable. Simoncelli and Portilla ([23]) 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.

## 1.2 Overview

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 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). 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

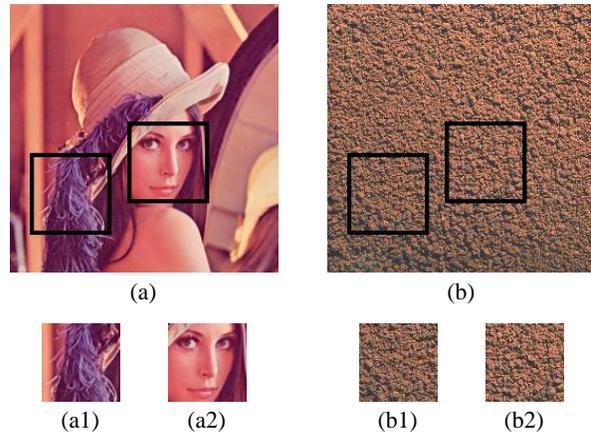


Figure 2: How textures differ from images. (a) is a general image while (b) is a texture. A movable window with two different positions are drawn as black squares in (a) and (b), with the corresponding contents shown below. Different regions of a texture are always perceived to be similar (b1,b2), which 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.

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.

The remainder of the paper is organized as follows. In section 2, we present the algorithm. In section 3, we demonstrate synthesis results and compare them with those generated by previous approaches. In section 4, we propose acceleration techniques. In sections 5 and 6, we discuss applications, limitations, and extensions.

## 2 Algorithm

Using Markov Random Fields as the texture model, the goal of the synthesis algorithm is to generate a new texture so that each local region of it is similar to another region from the input texture. We first describe how the algorithm works in a single resolution, and then we extend it using a multiresolution pyramid to obtain improvements in efficiency. For easy reference, we list the symbols used in Table 1 and summarize the algorithm in Table 2.

### 2.1 Single Resolution Synthesis

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 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 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

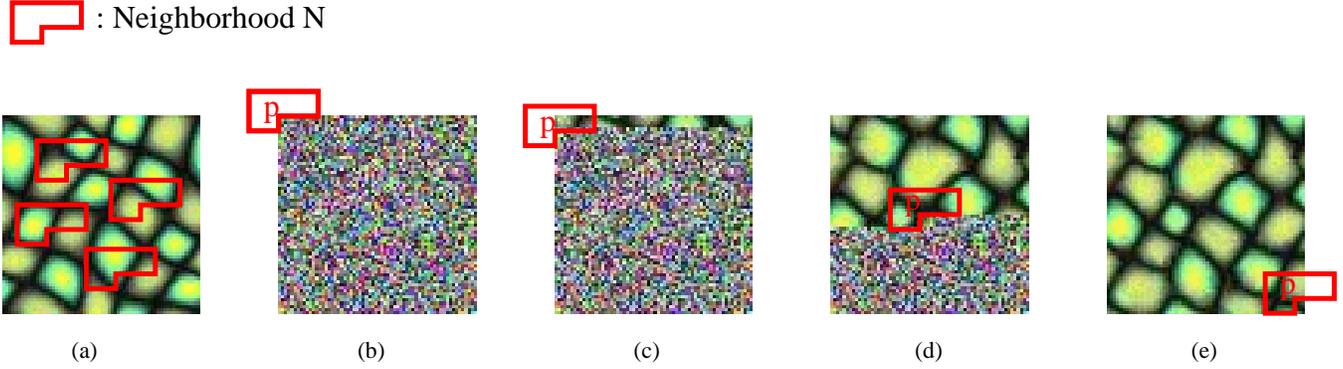


Figure 3: Single resolution texture synthesis. (a) is the input texture and (b)-(d) show different synthesis stages of the output image. Pixels in the output image are assigned in a raster scan ordering. The value of each output pixel  $p$  is determined by comparing its spatial neighborhood  $N(p)$  with all neighborhoods in the input texture. The input pixel with most similar neighborhood will be assigned to the corresponding output pixel. Neighborhoods crossing the output image boundaries (shown in (b) and (c)) are handled toroidally, as discussed in Section 2.4.

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 $p$
$G(L)$	$L$ th level of pyramid $G$
$G(L, x, y)$	pixel at level $L$ and position $(x, y)$ of $G$

Table 1: Table of symbols

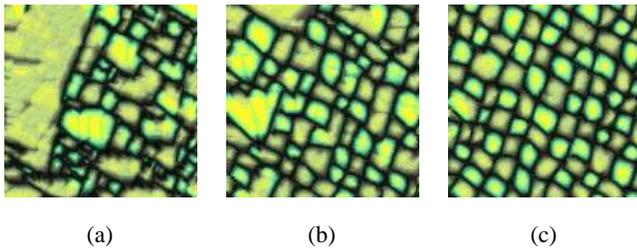


Figure 4: Synthesis results with different neighborhood sizes. The neighborhood sizes are (a) 5x5, (b) 7x7, (c) 9x9, respectively. All images shown are of size 128x128. Note that as the neighborhood size increases the resulting texture quality gets better. However, the computation cost also increases.

(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.

## 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

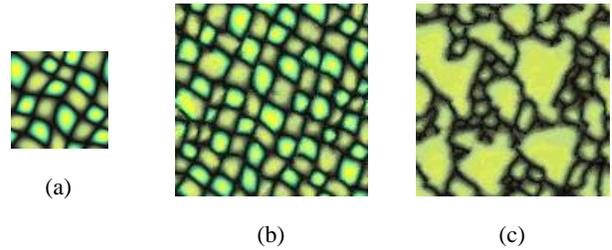


Figure 5: Causality of the neighborhood. (a) sample texture (b) synthesis result using a causal neighborhood (c) synthesis result using a noncausal neighborhood. Both (b) and (c) are generated from the same random noise using a 9x9 neighborhood. As shown, a noncausal neighborhood is unable to generate valid results.

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 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  $N(p)$  will contain 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” (contains only already assigned pixels). A noncausal  $N(p)$ , which always contains unassigned pixels, is unable to transform  $I_s$  to look like  $I_a$  (Figure 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.

## 2.3 Multiresolution Synthesis

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 ([4]); computation is saved because we can represent large scale struc-

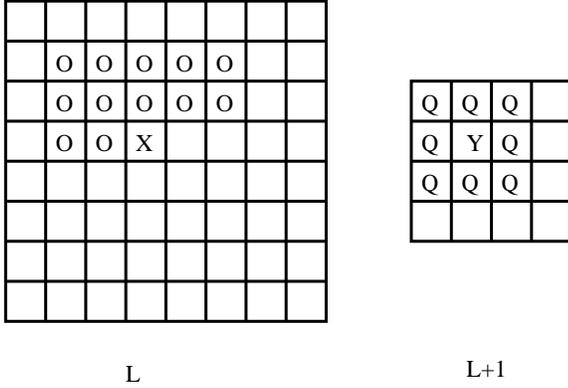


Figure 6: A causal neighborhood containing two levels of pyramid pixels. The current level of the pyramid is shown at left and the next lower resolution level is shown at right. The current output pixel  $p$ , marked as  $X$ , is located at  $(L, x, y)$ , where  $L$  is the current level number and  $(x, y)$  is its coordinate. At this level of the pyramid  $L$  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  $Y$ .

tures more compactly by a few pixels in a certain lower resolution pyramid level.

The multiresolution synthesis algorithm proceeds as follows. Two Gaussian pyramids ([4]),  $G_a$  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 multiresolution case, each neighborhood  $N(p)$  contains pixels in the current resolution as well as those in the lower resolutions. An example of multiresolution neighborhood with two levels is shown in Figure 6. The similarity between two multiresolution neighborhoods is measured by computing the sum of 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.

## 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  $G_s(L, x, y) \equiv G_s(L, x \bmod M, y \bmod N)$ , 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.

For the input pyramid  $G_a$ , toroidal neighborhoods will typically contain discontinuities unless  $I_a$  is tileable. A reasonable edge handler for  $G_a$  is to pad it with a reflected copy of itself. Another solution is to use only those  $N(p_i)$  completely inside  $G_a$ , and discard those crossing the boundaries. We use a reflective edge handler for all examples shown in this paper.

## 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 captures the random aspect of the textures. This notion of randomness can sometimes be achieved by entropy maximization ([30]), 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  $G_a$  ([10]).

The initial noise effects the synthesis process in the following way. For the single 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 over-written 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  $G_s$ .

## 2.6 Summary of Algorithm

We summarize the algorithm in the following pseudocode.

```

function  $I_r \leftarrow \text{TextureSynthesis}(I_a, I_s)$ 
1  Initialize( $I_s$ );
2   $G_a \leftarrow \text{BuildPyramid}(I_a)$ ;
3   $G_s \leftarrow \text{BuildPyramid}(I_s)$ ;
4  foreach level  $L$  from lower to higher resolutions of  $G_s$ 
5    loop through all pixels  $(x_s, y_s)$  of  $G_s(L)$ 
6       $C \leftarrow \text{FindBestMatch}(G_a, G_s, L, x_s, y_s)$ ;
7       $G_s(L, x_s, y_s) \leftarrow C$ ;
8   $I_r \leftarrow \text{ReconPyramid}(G_s)$ ;
9  return  $I_r$ ;

function  $C \leftarrow \text{FindBestMatch}(G_a, G_s, L, x_s, y_s)$ 
1   $N_s \leftarrow \text{BuildNeighborhood}(G_s, L, x_s, y_s)$ ;
2   $N_a^{best} \leftarrow \text{null}$ ;  $C \leftarrow \text{null}$ ;
3  loop through all pixels  $(x_a, y_a)$  of  $G_a(L)$ 
4     $N_a \leftarrow \text{BuildNeighborhood}(G_a, L, x_a, y_a)$ ;
5    if  $\text{Match}(N_a, N_s) > \text{Match}(N_a^{best}, N_s)$ 
6       $N_a^{best} \leftarrow N_a$ ;  $C \leftarrow G_a(L, x_a, y_a)$ ;
7  return  $C$ ;

```

Table 2: Pseudocode of the Algorithm

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 **BuildPyramid** and **ReconPyramid**. Various pyramids can be used for texture synthesis; examples are Gaussian pyramid ([21]), Laplacian pyramid ([10]), steerable pyramid ([10], [23]), and feature-based pyramids ([5]). Different pyramids will give different trade-offs between spatial and frequency resolutions. In this paper, we choose to use the Gaussian pyramid for its simplicity and greater spatial localization (a detailed discussion of this issue can be found in [20]). 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 (Section 5.1).

**Synthesis Ordering:** A raster scan ordering is used in line 5 of function `TextureSynthesis`. This, however, can also be extended. For example, a spiral ordering can be used for constrained texture synthesis (Section 5.1). The synthesis ordering should cooperate with the `BuildNeighborhood` so that the neighborhood contains 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 Section 4.

### 3 Synthesis Results

To test the effectiveness of our approach, we have run the algorithm on many different images from standard texture sets. Figure 7 shows examples using the Brodatz texture album ([3]) and the MIT VisTex set ([17]). The Brodatz album is the most commonly used texture testing suite 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.

A visual comparison of our approach with several other algorithms is shown in Figure 8. Result (a) is generated by Heeger and Bergen’s algorithm ([10]) 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 ([5]) 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 very 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 ([7]). This technique is based on the Markov Random Field model and is capable of generating high quality textures. However, a direct application of his approach can produce non-tileable results.<sup>1</sup>

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 section, we describe modifications that accelerate the algorithm greatly.

### 4 Acceleration

Our deterministic synthesis procedure 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 ([18]).

The nearest point searching problem in multiple dimensions is stated as follows: given a set  $S$  of  $n$  points and a novel query point

<sup>1</sup>We have found that it is possible to extend their approach using multiresolution pyramids and a toroidal neighborhood to make tileable textures. However this is not stated in the original paper ([7]).

$Q$  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  $Q$  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 [18] for a more complete reference. However, most of these algorithms assume generic inputs and do not attempt to take advantage of any special structures they may have. Popat ([21]) 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 (TSVQ, [8]) as the searching algorithm ([27]).

#### 4.1 TSVQ Acceleration

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 ([8]), consisting of alternations between centroid computation and nearest centroid partition, is then used to find 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 query. 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  $O(|S|)$ .

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 in  $N(p_i)$ . 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  $O(\log N_L)$  (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.

#### 4.2 Acceleration Results

An example comparing the results of exhaustive searching and TSVQ is shown in Figure 9. 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

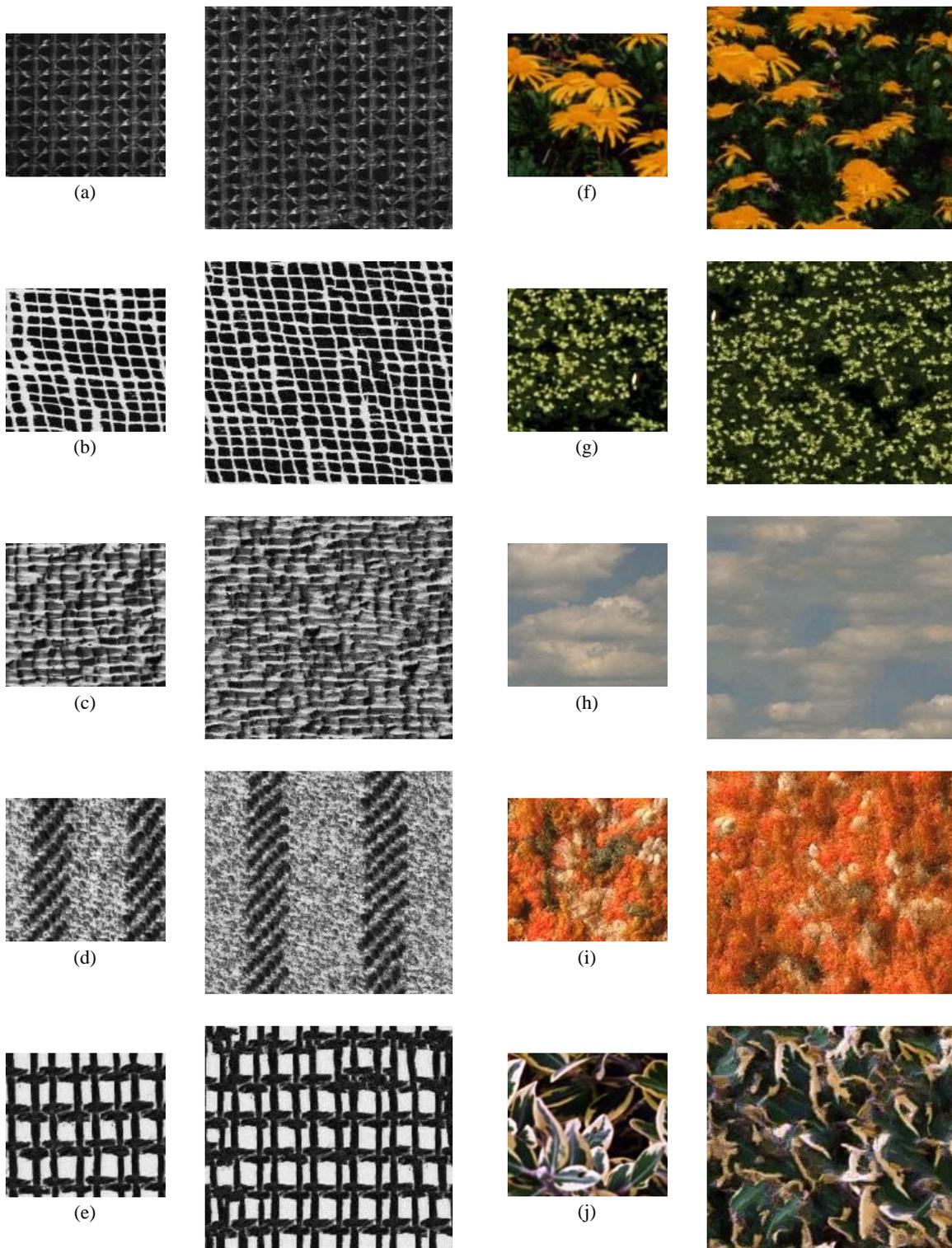


Figure 7: Texture synthesis results. The smaller patches are the input textures, and to their right are synthesized results. A  $9 \times 9$  neighborhood is used for all cases. Brodatz textures: (a) D52 (b) D103 (c) D84 (d) D11 (e) D20. VisTex textures: (f) Flowers 0000 (g) Misc 0000 (h) Clouds 0000 (i) Fabric 0015 (j) Leaves 0009.

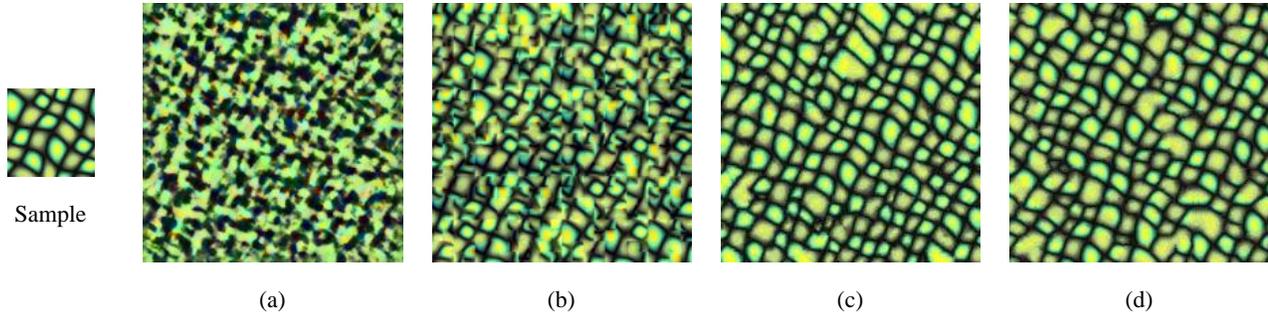


Figure 8: A comparison of texture synthesis results using different algorithms: (a) Heeger and Bergen’s method ([10]) (b) De Bonet’s method ([5]) (c) Efros and Leung’s method ([7]) (d) Our method. Only Efros and Leung’s algorithm produces results comparable with ours. However, our algorithm is 2 orders of magnitude faster than theirs (Section 4). 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).

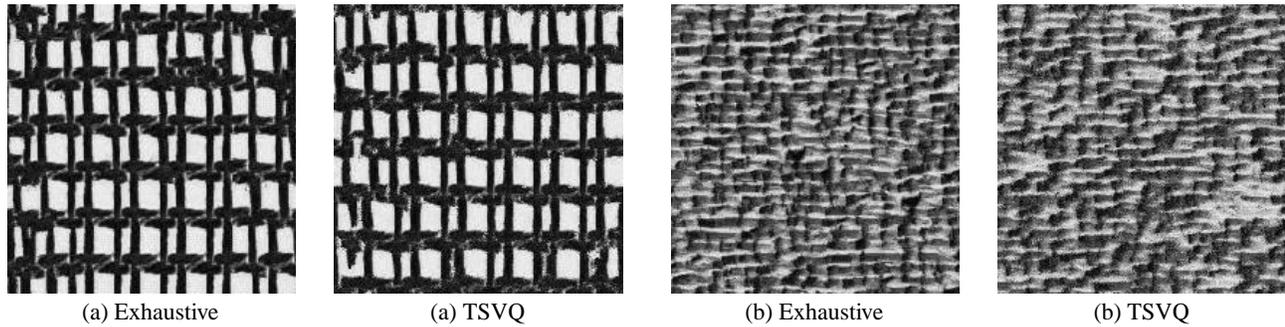


Figure 9: Accelerated synthesis using TSVQ. In each pair of figures, the result generated by exhaustive searching is on the left and the TSVQ accelerated result is on the right. The original images are shown in Figure 7. All generated images are of size 200x200. The average running time for exhaustive searching is 360 seconds. The average training time for TSVQ is 22 seconds and the average synthesis time is 7.5 seconds.

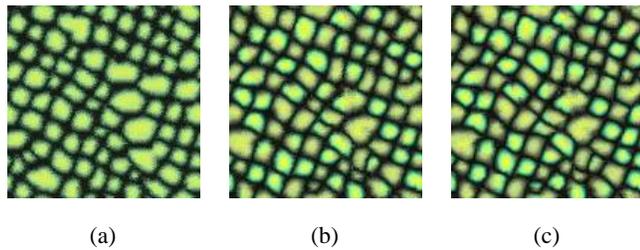


Figure 10: TSVQ acceleration with different codebook sizes. The original image size is 64x64 and all these synthesized results are of size 128x128. The number of codewords in each case are (a) 64 (b) 512 (c) 4096 (all).

measured on a 195MHz R10000 processor. As shown in Figure 9, results generated with TSVQ acceleration are roughly comparable in quality to those generated from the unaccelerated approach. In a few cases, TSVQ will generate more blurry images (such as Figure 9 (b)). We fix this by allowing limited backtracking in the tree traversal so that more than one leaf node can be visited. When the number of visited leaf nodes is the same as the codebook size, the result will be exactly the same as the exhaustive searching case.

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  $O(d * N)$  memory where  $d$  is the neighborhood size and  $N$  is the number of input image pixels. Fortunately, textures usually contain repeating structures; therefore

Algorithm	Training Time	Synthesis Time
Efros and Leung	none	1941 seconds
Exhaustive Searching	none	503 seconds
TSVQ acceleration	12 seconds	12 seconds

Table 3: A breakdown of running time for the textures shown in Figure 8. The first row shows the timing of Efros and Leung’s algorithm. The second and third rows show the timing of our algorithm, using exhaustive searching and TSVQ acceleration, respectively. All the timings were measured using a 195 MHz R10000 processor.

we can use codebooks with fewer number of codewords than the input training set. Figure 10 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 number of codewords such as (b) look plausible compared with the full codebook result (c). In our experience we can use codebooks with less than 10 percent 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.

Table 3 shows a timing breakdown for generating the textures shown in Figure 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 ([7]) took half an hour to generate the same texture <sup>2</sup>. 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.

## 5 Applications

One of the chief advantages of our texture synthesis method is its low computational cost. This permits us to explore a variety of applications in addition to the usual texture mapping for graphics that were previously impractical. Presented here are constrained synthesis for image editing and temporal texture generation.

### 5.1 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. For example, Hirani and Totsuka ([11]) developed an interactive algorithm that finds translationally similar regions for noise removal. Often, the flawed portion is contained within a region of texture, and can be replaced by constrained texture synthesis ([7],[12]).

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 ([4]) with another similar texture, shown in Figure 11 (b), will produce visible boundaries for structured textures. Better results can be obtained by applying our algorithm in Section 2 over the flawed regions, but discontinuities still appear at the right and bottom boundaries as shown in Figure 11 (c). These artifacts are caused by the causal neighborhood as well as the raster scan synthesis ordering.

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 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 11 (a) is replaced from outside to inside from the surrounding region until every pixel is assigned (Figure 11 (d)). This spiral synthesis ordering removes

<sup>2</sup>In this timing comparison we choose a very small input patch (with size 64x64). Because the time complexity of our approach over Efros and Leung’s is  $O(\log N)/O(N)$  where  $N$  is the number of input image pixels, our approach performs even better for larger input textures.

the directional bias which causes the boundary discontinuities (as in Figure 11 (c)).

Examples of constrained synthesis for hole filling and image extrapolation are shown in Figure 12. Within each pair of images, the black region is filled in using informations available in the rest of the image. The synthesized regions can blend smoothly with the original parts even for structured textures. Because of its efficiency, this approach may be useful as an interactive tool for image editing or denoising ([16]).

### 5.2 Temporal Texture Synthesis

The low cost of our accelerated algorithm enables us to consider synthesizing textures of dimension greater than two. An example of 3D texture is temporal texture. 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 ([24]). 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 ([26]).

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 original 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.

Figure 13 shows synthesis results of several typical temporal textures: fire, smoke, and ocean waves (shown in the accompanying video tape). 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; we plan to extend the algorithm to allow more explicit user controls (such as the distribution and intensity of the fire and smoke).

## 6 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 paper 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 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.

One drawback of the Markov Random Field approach is that only local and stationary phenomena can be modeled. Other visual cues such as 3D shape, depth, lighting, or reflection can not be captured by this approach. One possible solution would be to incorporate this information in a preprocessing step, or to impose certain constraints during the synthesis process. For example, to synthesize a perspectively viewed brick wall, we could use a shape

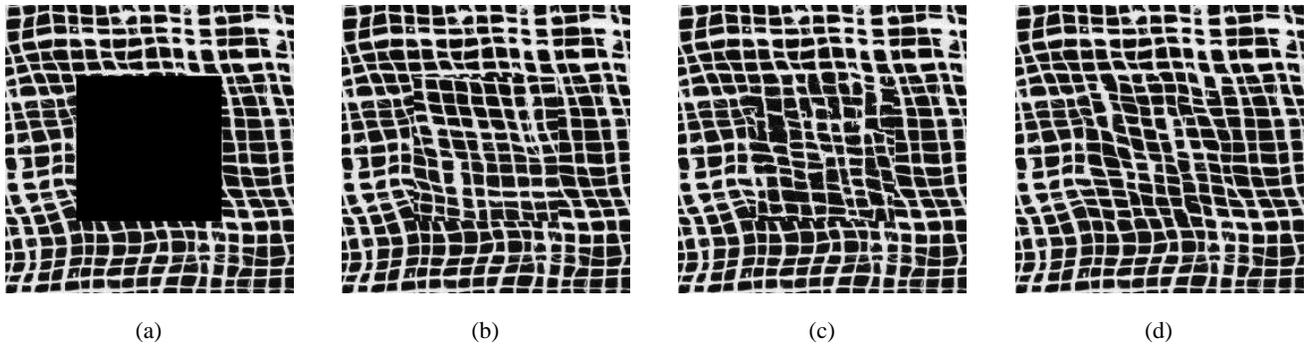


Figure 11: Constrained texture synthesis. (a) a texture containing a black region that needs to be filled in. (b) multiresolution blending ([4]) with another texture region will produce boundary artifacts. (c) A direct application of the algorithm in Section 2 will produce visible discontinuities at the right and bottom boundaries. (d) A much better result can be generated by using a modification of the algorithm with 2 passes.

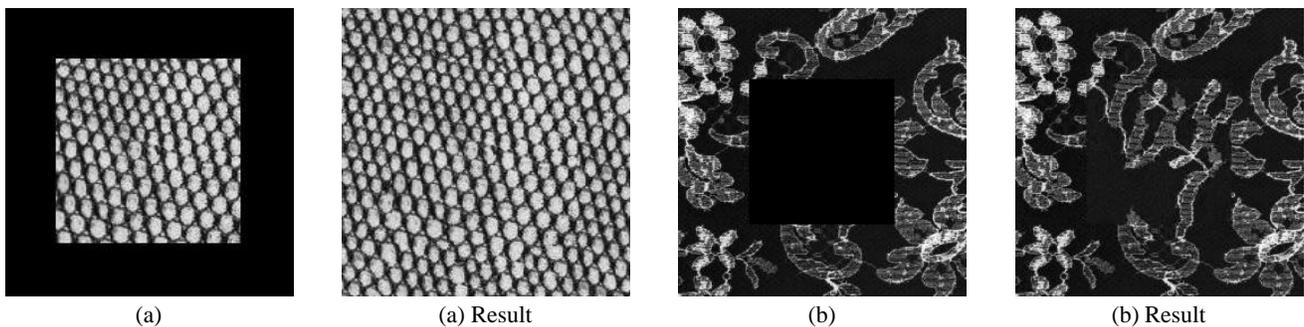


Figure 12: Constrained synthesis examples. In each pair of figures, the original image is on the left and the synthesized result is on the right. The goal is to fill in the black regions without changing the rest of the image. Examples shown are Brodatz textures with image extrapolation (a) D36 and hole filling (b) D40.

from texture technique ([15]) to determine the sizes and orientation of individual bricks in the original image. Then, during the synthesis process, a global constraint is enforced over the output image so that the pattern is generated according to the relative position and orientation between the wall and the eye point.

Aside from constrained synthesis and temporal textures, numerous applications of our approach are possible. Other potential applications/extensions are:

**Multidimensional texture:** The notion of texture extends naturally to multi-dimensional data. One example was presented in this paper - motion sequences. The same technique can also be directly applied to solid textures or animated solid texture synthesis. We are also trying to extend our algorithm for generating structured solid textures from 2D views ([10]).

**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 ([2]). This suggests that textures might be compressible by our synthesis technique. Compression would consist of building a codebook, but unlike [2], 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 [2] is much

greater compression, since only the codebook is transmitted.

**Motion synthesis/editing:** Some motions can be efficiently modeled as spatial-temporal textures. Others, such as animal or human motion, are too highly structured for such a direct approach. However, it might be possible to encode their motion as joint angles, then apply texture analysis-synthesis to the resulting 1D temporal motion signals.

**Modeling geometric details:** 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 ([14]). The resulting displacement maps should be compressible/decompressible as 2D textures using our technique. Taking this idea further, missing geometric details, a common problem in many scanning situations ([1]), could be filled in using our constrained texture synthesis technique.

**Direct synthesis over meshes:** Mapping textures onto irregular 3D meshes by projection often cause distortions ([22]). These distortions can sometimes be fixed by establishing suitable parameterization of the mesh, but a more direct approach would be to synthesize texture directly over the mesh. In principle, this can be done using our technique. However, this will require extending ordinary signal processing operations such as filtering and downsampling to irregular 3D meshes.

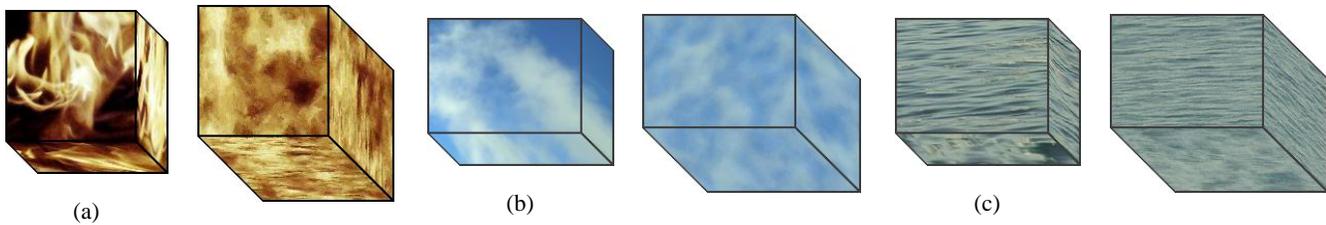


Figure 13: Temporal texture synthesis results. (a) fire (b) smoke (c) ocean waves. In each pair of images, the spatial-temporal volume of the original motion sequence is shown on the left, and the corresponding synthesis result is shown on the right. A  $5 \times 5$  causal neighborhood is used for synthesis. The original motion sequences contain 32 frames, and the synthesis results contain 64 frames. The individual frame sizes are (a)  $128 \times 128$  (b)  $150 \times 112$  (c)  $150 \times 112$ . Accelerated by TSVQ, the training time are (a) 1875 (b) 2155 (c) 2131 seconds and the synthesis time 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. The original fire sequence is acquired from [25].

## Acknowledgement

[Removed for paper submission]

## References

- [1] Anonymous. The Digital Michelangelo Project: 3D scanning of large statues. submitted for publication.
- [2] A. C. Beers, M. Agrawala, and N. Chaddha. Rendering from compressed textures. *Proceedings of SIGGRAPH 96*, pages 373–378, August 1996.
- [3] P. Brodatz. *Textures: A Photographic Album for Artists and Designers*. Dover, New York, 1966.
- [4] P. J. Burt and E. H. Adelson. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics*, 2(4):217–236, Oct. 1983.
- [5] J. S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In T. Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 361–368. ACM SIGGRAPH, Addison Wesley, Aug. 1997.
- [6] J. Dorsey, A. Edelman, J. Legakis, H. W. Jensen, and H. K. Pedersen. Modeling and rendering of weathered stone. *Proceedings of SIGGRAPH 99*, pages 225–234, August 1999.
- [7] A. Efros and T. Leung. Texture synthesis by non-parametric sampling. In *International Conference on Computer Vision*, volume 2, pages 1033–8, Sep 1999.
- [8] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.
- [9] R. Haralick. Statistical image texture analysis. In *Handbook of Pattern Recognition and Image Processing*, volume 86, pages 247–279. Academic Press, 1986.
- [10] D. J. Heeger and J. R. Bergen. Pyramid-Based texture analysis/synthesis. In R. Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 229–238. ACM SIGGRAPH, Addison Wesley, Aug. 1995.
- [11] 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.
- [12] H. Igehy and L. Pereira. Image replacement through texture synthesis. In *International Conference on Image Processing*, volume 3, pages 186–189, Oct 1997.
- [13] 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.
- [14] V. Krishnamurthy and M. 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.
- [15] J. Malik and R. Rosenholtz. Computing local surface orientation and shape from texture for curved surfaces. *International Journal of Computer Vision*, 23(2):149–168, 1997.
- [16] T. Malzbender and S. Spach. A context sensitive texture nib. In *Proceedings of Computer Graphics International*, pages 151–163, June 1993.
- [17] MIT Media Lab. Vision texture. <http://www-white.media.mit.edu/vismod/imagery/VisionTexture/vistex.html>.
- [18] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEETPAMI: IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:989–1003, 1997.
- [19] R. Paget and I. Longstaff. Texture synthesis via a noncausal nonparametric multiscale Markov random field. *IEEE Transactions on Image Processing*, 7(6):925–931, June 1998.
- [20] A. C. Popat. *Conjoint Probabilistic Subband Modeling*. PhD thesis, Massachusetts Institute of Technology, 1997.
- [21] K. Popat and R. Picard. Novel cluster-based probability model for texture synthesis, classification, and compression. In *Visual Communications and Image Processing*, pages 756–68, 1993.
- [22] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. E. Haeberli. Fast shadows and lighting effects using texture mapping. *Computer Graphics (Proceedings of SIGGRAPH 92)*, 26(2):249–252, July 1992.
- [23] 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, Oct. 1998.
- [24] J. Stam and E. Fiume. Depicting fire and other gaseous phenomena using diffusion processes. *Proceedings of SIGGRAPH 95*, pages 129–136, August 1995.
- [25] M. Szumner. Temporal texture database. <ftp://whitechapel.media.mit.edu/pub/-szumner/temporal-texture/>.
- [26] M. Szumner and R. W. Picard. Temporal texture modeling. In *International Conference on Image Processing*, volume 3, pages 823–6, Sep 1996.
- [27] L. 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.
- [28] A. Witkin and M. Kass. Reaction-diffusion textures. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 299–308, July 1991.
- [29] S. P. Worley. A cellular texture basis function. In H. Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 291–294. ACM SIGGRAPH, Addison Wesley, Aug. 1996.
- [30] S. Zhu, Y. Wu, and D. Mumford. Filters, random fields and maximum entropy (FRAME) - towards a unified theory for texture modeling. *International Journal of Computer Vision*, 27(2):107–126, 1998.

# Fast Texture Synthesis using Tree-structured Vector Quantization

Li-Yi Wei

Marc Levoy

Stanford University

Category: Research

Format: Print

Contact: Li-Yi Wei  
Room 386, Gates Computer Science Building  
Stanford, CA 94309, U.S.A.  
phone: (650)725-3733  
fax: (650)723-0011  
email: liyiwei@graphics.stanford.edu

Paper number: 0015

Estimated # of pages: 10

Keywords: Texture Synthesis, Compression Algorithms, Image Processing

Texture synthesis is important for many applications in computer graphics, vision, and image processing. However, it remains difficult to design an algorithm that is both efficient and capable of generating high quality results. In this paper, we present an efficient algorithm for realistic texture synthesis. 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. This permits us to apply texture synthesis to problems where it has traditionally been considered impractical. In particular, we have applied it to constrained synthesis for image editing and temporal texture generation. Our algorithm is derived from Markov Random Field texture models, and generates textures through a deterministic searching process. We accelerate this synthesis process using tree-structured vector quantization.