Real-Time Texture Synthesis By Patch-Based Sampling

Lin Liang, Ce Liu, Yingqing Xu,
Baining Guo, and Heung-Yeung Shum
bainguo@microsoft.com

March 2001

Technical Report
MSR-TR-2001-40

We present a patch-based sampling algorithm for synthesizing textures from an input sample texture. The patch-based sampling algorithm is fast. Using patches of the sample texture as building blocks for texture synthesis, this algorithm makes high-quality texture synthesis a real-time process. For generating textures of the same size and comparable (or better) quality, patch-based sampling is orders of magnitude faster than existing texture synthesis algorithms. The patch-based sampling algorithm synthesizes high-quality textures for a wide variety of textures ranging from regular to stochastic. By sampling patches according to a non-parametric estimation of the local conditional MRF density, we avoid mismatching features across patch boundaries. Moreover, the patch-based sampling algorithm remains effective when pixel-based non-parametric sampling algorithms fail to produce good results. For natural textures, the results of the patch-based sampling look subjectively better.

1 Introduction

Texture synthesis has a variety of applications in computer vision, graphics, and image processing. An important motivation for texture synthesis come from texture mapping. Texture images usually come from scanned photographs, and the available photographs may be too small to cover the entire object surface. In this situation, a simple tiling will introduce unacceptable artifacts in the forms of visible repetition and seams. Texture synthesis solves this problem by generating textures of the desired sizes. Other applications of texture synthesis include various image processing tasks such as occlusion fill-in and image/video compression.

The texture synthesis problem may be stated as follows: Given an input sample texture I_{in}, synthesize a texture I_{out} that is sufficiently different from the given sample texture, yet appears perceptually to be generated by the same underlying stochastic process. In this work, we use the Markov Random Field (MRF) as our texture model and assume that the underlying stochastic process is both local and stationary. We choose MRF because it is known to accurately model a wide range of textures. Other successful but more specialized models include reaction-diffusion [17, 19], frequency domain [10], and fractals [5, 20].

In recent years, a number of successful texture synthesis algorithms have been proposed in graphics and vision. Motivated by psychology studies, Heeger and Bergen developed a pyramid-based texture synthesis algorithm that approximately matches marginal histograms of filter responses [7]. Zhu et al. introduced a mathematical model called FRAME, which integrates filters and histograms into MRF models and uses a minimax entropy principle to select feature statistics [24, 25]. Several texture synthesis algorithms are based on matching joint statistics of filter responses. De Bonet's algorithm matches the joint histogram of a long vector of filter responses [3]. Portilla and Simoncelli developed an iterative projection method for matching the correlations of certain filter responses [14]. These methods, along with many others in the literature [8, 21], represent two different approaches to texture synthesis. The first is to compute global statistics in feature space and sample images from the texture ensemble [23] directly [7, 3, 14, 23]. The second approach is to estimate the local conditional probability density function (PDF) and synthesize pixels incrementally [24].

The texture synthesis algorithm we propose follows the second approach. In some earlier work, Zhu et al. explored this approach using the analytical FRAME model and an accurate but expensive Markov chain Monte Carlo method [24]. More recently, Efros and Leung demonstrated the power of sampling from a local PDF by presenting a non-parametric sampling algorithm that synthesizes high-quality textures for a wide variety of textures ranging from regular to stochastic [4]. Efros and Leung's algorithm, while much faster than [24], is still too slow. In-

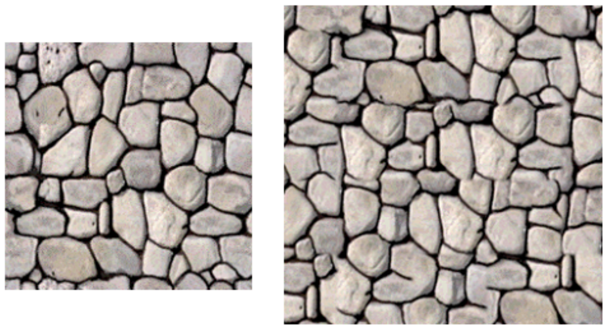
Figure 1: Texture synthesis example. Left: 192×192 input sample texture. Right: 256×256 texture synthesized by patch-based sampling. The synthesis takes 0.02 seconds on a 667 MHz PC.

spired by a cluster-based texture model [13], Wei and Levoy significantly accelerated Efros and Leung's algorithm using tree-structured vector quantization (TSVQ) [18]. However, this TSVQ-accelerated non-parametric sampling algorithm is still not real-time. Another problem with [4] and [18] is that, for some textures, [4] is a greedy algorithm that has a tendency to "slip" into a wrong part of the search space and start to grow garbage. The TSVQ acceleration [18] further aggravates this problem.

In this paper we show that high-quality texture can be synthesized in *real-time*. A key ingredient of the algorithm we propose is a patch-based sampling scheme that uses texture patches of the sample texture as building blocks for texture synthesis. The advantages of patch-based sampling include

speed For synthesizing textures of the same size and comparable (or better) quality, our algorithm is orders of magnitude faster than existing texture synthesis algorithms, including TSVQ-accelerated non-parametric sampling [18]. As a result, high-quality texture synthesis is now a real-time process on a mid-level PC.

quality The patch-based sampling algorithm synthesizes high-quality textures for a wide variety of textures ranging from regular to stochastic. Like [4, 18], ours is also a greedy algorithm for non-parametric sampling. However, the patches in our sampling scheme implicitly provide constraints for avoiding garbage. For this reason, our algorithm continues to synthesize high-quality textures even when [4] and [18] cease to be effective. For natural textures, the results of patch-based sampling look subjectively better.

Figure 1 shows an example produced by our algorithm. After spending 0.6 seconds

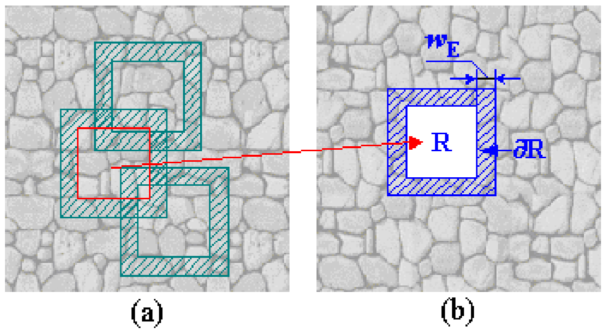

Figure 2: A patch-based sampling strategy. In the synthesized texture shown in (b), the hatched area is the boundary zone. In the input sample texture shown in (a), three patches have boundary zones matching the texture patch I_R in (b) and the red patch is selected.

analyzing the input sample, our algorithm took 0.02 seconds to synthesize this texture on a 667 MHz PC.

The idea of using texture patches for texture synthesis has been used before. Xu et al. have proposed a texture synthesis algorithm based on random patch pasting [22]. Praun has successfully adapted this patch-pasting algorithm for texture mapping on 3D surfaces [15]. Unfortunately, these patch-pasting algorithm suffers from the mismatching features across patch boundaries. The patch-based sampling algorithm, on the other hand, avoids mismatching features across patch boundaries by sampling texture patches according to the local conditional MRF density. Patch-based sampling includes patch pasting as a special case, in which the local PDF implies a null statistical constraint.

Patch-based sampling is amenable to acceleration and the fast speed of our algorithm is partially attributable to our carefully designed acceleration scheme. Our scheme is based on a quality-first principle: we do not use acceleration techniques that have a noticeably negative impact on the synthesized texture. The core computation in patch-based sampling can be formulated as a search for approximate nearest neighbors (ANN). We accelerate this search by combining an optimized technique for general ANN search, a novel data structure called the quad-tree pyramid for ANN search of images, and principal components analysis of the input sample texture.

The patch-based sampling algorithm is easy to use and flexible. It can generate tileable textures if so desired. It can be used for constrained synthesis as well. The algorithm has an intuitive randomness parameter. The user can use this parameter to interactively control the randomness of the synthesized texture.

The rest of the paper is organized as follows. In Section 2, we introduce patch-based sampling and describe a patch-based sampling algorithm for texture synthesis. In Section 3, we present texture synthesis results with an emphasis on the texture quality. Our acceleration techniques and texture synthesis speed are discussed in Section 4, followed by conclusions and suggestions for future work Section 5.

2 Patch-Based Sampling

The patch-based sampling algorithm uses texture patches of the input sample texture I_{in} as the building blocks for constructing the synthesized texture I_{out}. In each step, we paste a patch B_k of the input sample texture I_{in} into the synthesized texture I_{out}. To avoid mismatching features across patch boundaries, we carefully select B_k based on the patches already pasted in I_{out}, $\{B_0, ..., B_{k-1}\}$. The texture patches are pasted in the order shown in Figure 3. For simplicity, we only use square patches of a prescribed size $w_B \times w_B$.

2.1 Sampling Strategy

Let I_{R_1} and I_{R_2} be two texture patches of the same shape and size. We say that I_{R_1} and I_{R_2} *match* if $d(R_1, R_2) < \delta$, where the $d()$ represents the distance between two texture patches and δ is a prescribed constant.

Assuming the Markov property, the patch-based sampling algorithm estimates the local conditional MRF (FRAME or Gibbs) density $p(I_R|I_{\partial R})$ in a non-parametric form by an empirical histogram. Define the boundary zone ∂R of a texture patch I_R as a band of width w_E along the boundary of R as shown in Figure 2. When the texture on the boundary zone $I_{\partial R}$ is known, we would like to estimate the conditional probability distribution of the unknown texture patch I_R. Instead of constructing a model, we directly search the input sample texture I_{in} for all patches having the known $I_{\partial R}$ as their boundary zones. The results of the search form an empirical histogram Ψ for the texture patch I_R. To synthesize I_R, we just pick an element from Ψ at random. Mathematically, the estimated conditional MRF density is

$$p(I_R|I_{\partial R}) = \sum_i \alpha_i \delta(I_R - I_{R^i}), \quad \sum_i \alpha_i = 1,$$

where I_{R^i} is a patch of the input sample texture I_{in} whose boundary zone $I_{\partial R^i}$ matches the boundary zone $I_{\partial R}$. The weight α_i is a normalized similarity scale factor.

With patch-based sampling, the statistical constraint is implicit in the boundary zone ∂R. A large boundary zone implies a strong statistical constraint. Generally

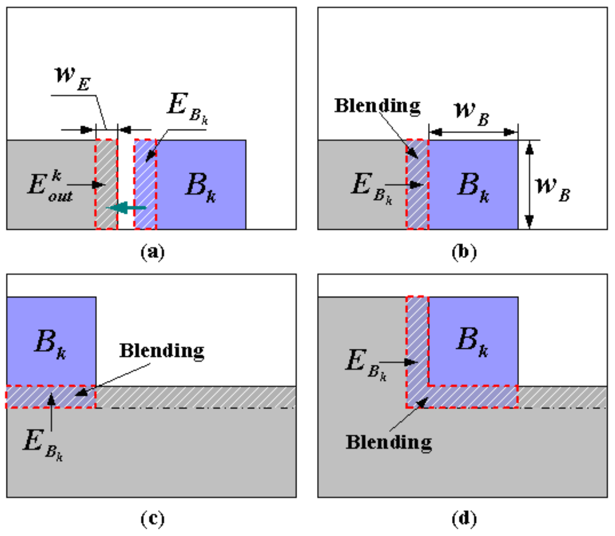

Figure 3: Texture synthesis by patch-based sampling. The grey area is already synthesized. The hatched areas are the boundary zones. (a) The boundary zones E^k_{out} and E_{B_k} should match. (b), (c), and (d) are three configurations for boundary zone matching. The overlapping boundary zones are blended together.

speaking, a non-parametric local conditional PDF such as in [4, 18] is faster to estimate than the analytical FRAME model in [24]. On the down side, the non-parametric density estimation is subject to greater statistical fluctuations, because in a small sample texture I_{in} there may be only a few sites that satisfy the local statistical constraints.

In practice, a more serious problem with existing non-parametric sampling techniques [4, 18] is that they tend to wonder into the wrong part of the search space and grow garbage in the synthesized texture. The patches in our sampling scheme implicitly provide constraints for avoiding garbage.

2.2 Synthesizing Texture

Now we use the patch-based sampling strategy to choose the texture patch B_k, the k-th texture patch to be pasted into the output texture I_{out}.

As Figure 3 (a) shows, only part of the boundary zone of B_k overlaps with the boundary zone of the already pasted patches $\{B_0, ..., B_{k-1}\}$ in I_{out}. We say that two boundary zones match if they match in their overlapping region. In Figure 3 (a), B_k has a boundary zone E_{B_k} of width w_E. The already pasted patches in I_{out} also have a boundary zone E^k_{out} of width w_E. According to the patch-based sampling strategy, E_{B_k} should match E^k_{out}.

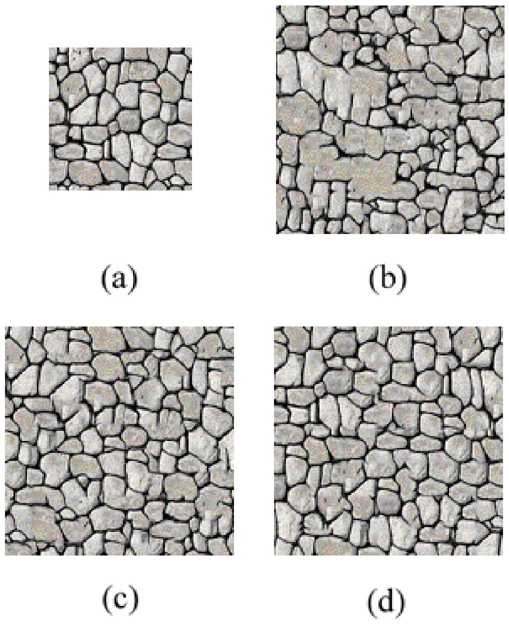

Figure 4: The effect of the patch size on synthesized texture. (a) Input sample texture of size 64×64. (b) The synthesized texture when the patch size $w_B = 16$. (c) The synthesized texture when $w_B = 24$. (d) The synthesized texture when $w_B = 32$. The randomness of the synthesized texture decreases as w_B increases.

For the randomness of the synthesized texture I_{out}, we form a set Ψ_B consisting of all texture patches of I_{in} whose boundary zones match E^k_{out}. Let $B_{(x,y)}$ be the texture patch whose lower left corner is at (x, y) in I_{in}. We form

$$\Psi_B = \{B_{(x,y)} | d(E_{B_{(x,y)}}, E^k_{out}) < d_{\max}, B_{(x,y)} \text{ in } I_{in}\}, \tag{1}$$

where $d_{\max}$ is the distance tolerance of the boundary zones. From Ψ_B we randomly select a texture patch to be the k-th patch to be pasted. For a given $d_{\max}$, the set Ψ_B could be empty. In that case, we choose B_k to be a texture patch in I_{in} with the smallest distance $d(E_{B_k}, E^k_{out})$.

The patch-based sampling algorithm proceeds as follows.

a) Randomly choose a $w_B \times w_B$ texture patch B_0 from the input sample texture I_{in}. Paste B_0 in the lower left corner of I_{out}. Set $k = 1$.
b) Form the set Ψ_B of all texture patches from I_{in} such that for each texture patch of Ψ_B, its boundary zone matches E^k_{out}.
c) If Ψ_B is empty, set $\Psi_B = \{B_{\min}\}$ where $B_{\min}$ is chosen such that its boundary zone is the closest to E^k_{out}.

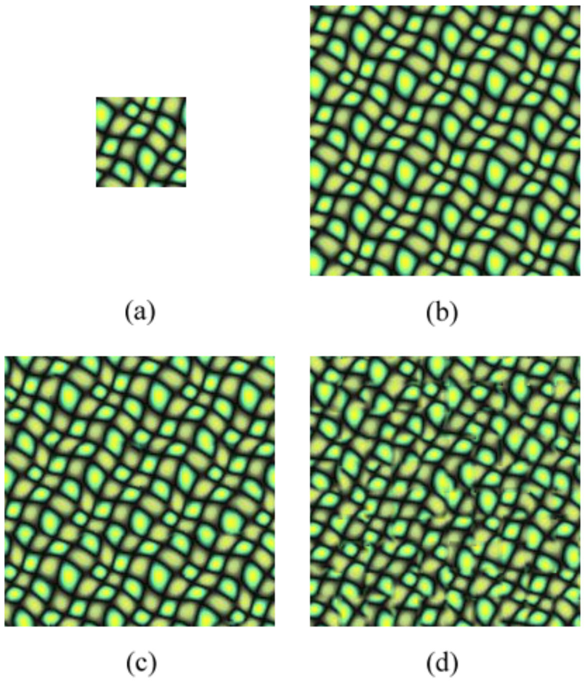
Figure 5: The effect of different relative error ϵ. (a) Input sample texture. (b) $\epsilon = 0$. (c) $\epsilon = 0.2$. (d) $\epsilon = 1.0$.

d) Randomly select an element from Ψ_B as the k-th texture patch B_k.
   Paste B_k onto the output texture I_{out}. Set $k = k + 1$.
e) Repeat steps (b), (c), and (d) until I_{out} is fully covered.
f) Perform blending in the boundary zones.

The blending step uses feathering [16] to provide a smooth transition between adjacent texture patches after I_{out} is fully covered with texture patches.

2.3 Implementation Details

Patch Size (w_B**)**: The size of the texture patch affects how well the synthesized texture captures the local characteristics of the input sample texture I_{in}. A smaller w_B implies weaker statistical constraints and less similarity between the synthesized texture I_{out} and the input sample texture I_{in}. Up to certain limit, a bigger w_B means better capturing of texture characteristics in the texture patches and thus more similarity between I_{out} and I_{in}.

Figure 4 shows the effect of w_B on the synthesized textures I_{out}. When $w_B = 16$, the texture patches contain less structural information of the input sample tex-

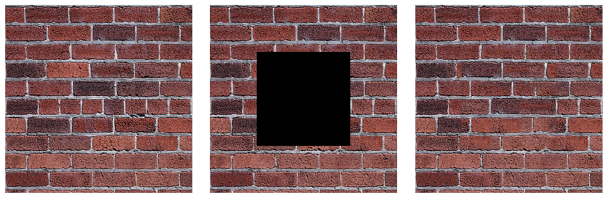

Figure 6: Constrained texture synthesis. Left: A 256×256 texture. Middle: A 128×128 hole is created. Right: Result of constrained synthesis.

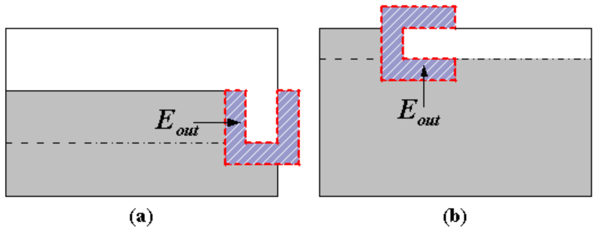

Figure 7: Boundary zone matching for tileable texture synthesis. The grey area is the texture already synthesized. The hatched purple areas are the areas to be matched. (a) Boundary zone matching for the last patch in a row. (b) Boundary zone matching for the last row.

ture I_{in}. As a result, the synthesized textures I_{out} appears more random. For patch size $w_B = 32$, the synthesized texture I_{out} become less random and resembles the input sample texture of I_{in} more.

For an input sample texture of size $w_{in} \times h_{in}$, the patch size w_B should be $w_B = \lambda \min(w_{in}, h_{in})$, where $0 < \lambda < 1$ is the *randomness parameter* of our system. The intuitive meaning of w_B is the scale of the texture elements in the input sample texture I_{in}. For texture synthesis, it is usually assumed that the approximate scale of the texture elements is known [4]. The patch size serves a similar function as the window size in [4], which is also a randomness parameter. The main difference is that a large window size in [4] drastically reduces the texture synthesis speed; the patch size w_B has little impact on the synthesis speed.

Unless stated otherwise, all examples in this paper are generated with λ values between 0.25 and 0.5.

Distance Metric: We choose the following measure of the distance between two boundary zones

$$d(E_{B_k}, E^k_{out}) = [\frac{1}{A}\sum_{i=1}^{A}(p^i_{B_k} - p^i_{out})^2]^{1/2} \tag{2}$$

where A is the number of pixels in the boundary zone. $p_{B_k}^i$ and p_{out}^i represent the values of the i-th pixel in the boundary zones E_{B_k} and E_{out}^k respectively.
Boundary Zone Width (w_E): The boundary zone width w_E should be sufficiently large to avoid mismatching features across patch boundaries. A wide boundary zone implies strong statistical constraints, which force a natural transition of features across patch boundaries. However, when the boundary zone is too wide, the statistical constraint will become so strong that there will be very few texture patches satisfying the constraints in a small sample texture I_{in}. In that case, patch-based sampling suffers serious statistical fluctuations. As we shall see, when w_E is too large it is also more costly to construct the k-d tree for accelerating the search for the texture patches of Ψ_B. As a balance, we typically set w_E to be four pixels wide.
Distance Tolerance ($d_{\max}$): When the distance between two boundary zones is defined by equation (2), we define $d_{\max}$ as

$$d_{\max} = [\frac{1}{A}\sum_{i=1}^{A}(\epsilon p_{out}^i)^2]^{1/2}$$

where A is the number of pixels in the boundary zone. p_{out}^i represent the values of the i-th pixel in the boundary zone E_{out}^k. $\epsilon \geq 0$ is the relative matching error between boundary zones.

The parameter ϵ controls the similarity of the synthesized texture I_{out} with the input sample texture I_{in} and the quality of I_{out}. The smaller the ϵ, the more similar are the local structures of the synthesized texture I_{out} and the sample texture I_{in}. If $\epsilon = 0$, the synthesized texture I_{out} looks like the tiling of the sample texture I_{in}, as Figure 5 (b) shows. When ϵ is too big, the boundary zones of adjacent texture patches may be very different and thus there may not be a natural transition across the patch boundaries, as Figure 5 (d) shows. To ensure the quality of the synthesized texture, we set $\epsilon = 0.2$.
Edge Handling: Let $B_{(x,y)}$ be the texture patch whose lower left corner is at (x, y) in I_{in}. To construct the set Ψ_B we have to test $B_{(x,y)}$ for inclusion in Ψ_B. For (x, y) near the border of the input sample texture I_{in}, part of $B_{(x,y)}$ may be out side of I_{in}. If the sample texture I_{in} is tileable, then we set the value of $B_{(x,y)}$ toroidally. $B_{(x,y)}(u, v) = I_{in}(u \bmod w_{in}, v \bmod h_{in})$ where (u, v) is the location of a pixel of $B_{(x,y)}$ inside the sample texture I_{in} and w_{in} and h_{in} are the width and the height of I_{in}. If the sample texture I_{in} is not tileable, we only search the texture patches that are completely inside I_{in}.
Constrained Synthesis: It is straightforward to extend the patch-based sampling algorithm to handle constrained texture synthesis. To better match the features

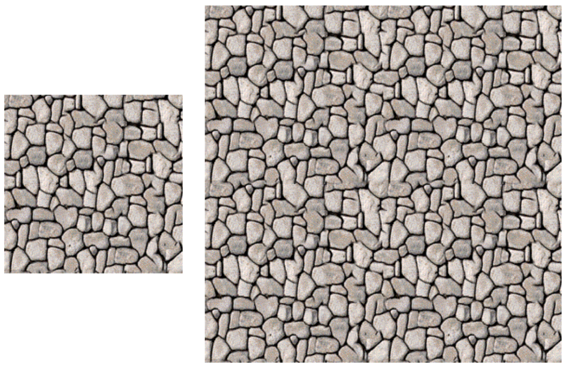

Figure 8: Result of tileable texture synthesis. Left: A tileable texture synthesized from the input sample in Figure 1. Right: A 2×2 tiling of the synthesized texture.

across patch boundaries between the known texture around the hole and newly pasted texture patches, we fill the hole in spiral order. Figure 6 shows an example.
Tileable Texture Synthesis: For tileable texture synthesis, Figure 7 shows the boundary zones to be matched for the last patch in a row and for the patches of the last row. In the synthesized texture I_{out}, the pixel values in the boundary zone should be

$$E_{out}(x, y) = I_{out}(x \bmod w_{out}, y \bmod h_{out})$$

where (x, y) is the location of the pixel in I_{out}. w_{out} and h_{out} are the width and height of the synthesized texture. Figure 8 shows the result of tileable texture synthesis.

2.4 Discussion

Patch- and Pixel-Based Non-Parametric Sampling: When the patch size $w_B = 1$, patch-based sampling becomes the non-parametric sampling of [4, 18]. When $w_B = 1$, the estimated conditional MRF density becomes

$$p(I(v)|I_{\partial v}) = \sum_i \alpha_i \delta(I(v) - I(v_i)), \quad \sum_i \alpha_i = 1,$$

where $I(v_i)$ is a pixel of the input sample texture I_{in} whose neighborhood $I_{\partial v_i}$ matches $I_{\partial v}$. The weight α_i is a normalized similarity scale factor. This is the non-parametric sampling described in [4, 18]. When $w_B = 1$, the window size of [4] is $w = 2w_E + 1$ where w_E is the boundary zone width.

Like [4, 18], the patch-based sampling algorithm is also greedy. However, the patches in the sampling scheme provide constraints for avoiding garbage. A texture

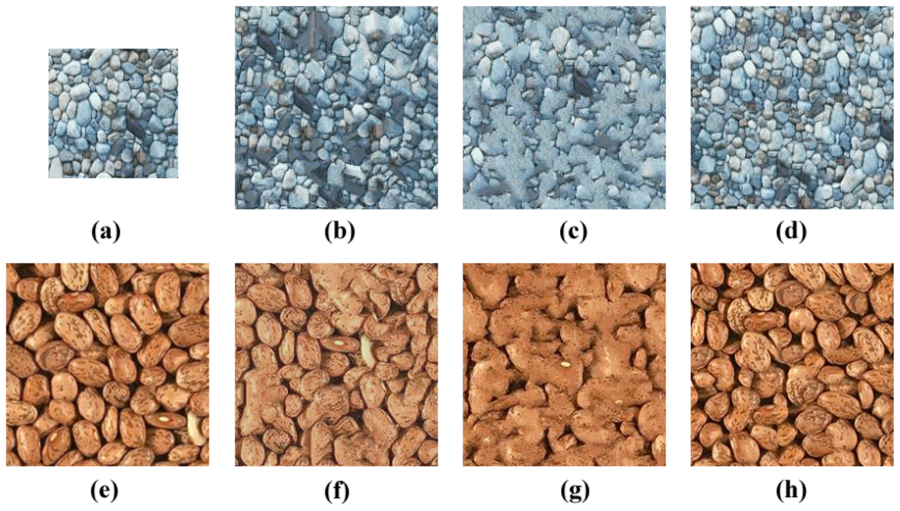
Figure 9: The patch-based sampling algorithm continues to synthesize high-quality texture even when other methods cease to be effective. (a) and (e) are input sample textures. (b) and (f) are results by Efros and Leung's algorithm [4]. (c) and (g) are the results by Wei and Levoy's algorithm [18]. (d) and (h) are the results of the patch-based sampling.

synthesized by patch-based sampling can be divided into two types of areas. The first type, which includes the majority of the synthesized texture, is the middle part of a pasted texture patch and this type of area has no garbage. The second type of area is a blend of two boundary zones. This type of area cannot have garbage either because the boundary zones themselves have no garbage and the blending is done on boundary zones with matched features.

Similar to [4, 18], the patch-based sampling algorithm can sometimes produces verbatim copies of the sample texture. With the parameter setting given earlier, this rarely occurs and when it does, we can solve the problem by slightly adjusting the patch size w_B. Adjusting patch size can be done interactively in our system.

Patch-Based Sampling vs Patch Pasting: When the relative matching error between the boundary zones becomes sufficiently large, say $\epsilon = 1.0$, the patch-based sampling algorithm is essentially the patch-pasting algorithm of [22].

3 Synthesis Results

We have tested the patch-based sampling algorithm on a wide variety of textures ranging from regular to stochastic. Figure 10 shows typical results. The companion CDROM contains more than 100 examples. Figure 11 shows some examples of failure. The left example in Figure 11 is a minor failure in which the algorithm tries to blend two different radishes together. This is not natural to people who are familiar with radishes. The right example is a failure because the sky progressively becomes brighter from left to right. The patch-based algorithm cannot detect the progressive transition very well and that leads to some boundary effects noticeable under close examination.

As mentioned, for some textures, [4] has a tendency to "slip" into a wrong part of the search space and start to grow garbage. The TSVQ acceleration [18] further aggravates this problem. Figure 9 show two examples. In contrast, patch-based sampling continues to synthesize high-quality textures. To synthesize these 200×200 textures from a 192×192 sample with Efros and Leung's algorithm, we spent about 20 hours on a mid-level PC. Patch-based sampling only takes 0.02 second on the same machine. The results of Wei and Levoy's algorithm were downloaded from their webpage.

For natural textures like the ones shown in Figure 9, the results of the patch-based sampling look subjectively better because the patches capture the fine nuances of natural textures that are often difficult to characterize by a statistical texture model such as MRF.

4 Performance Optimization

When constructing the set Ψ_B as defined in equation (1) in Section 2.2, we need to search the set of all $w_B \times w_B$ patches of the input sample texture I_{in} for patches whose boundary zones match E^k_{out}. Similar to [12, 18], we formulate our search as a k nearest neighbors search problem in the high dimensional space consisting of texture patches of the same shape and size as E^k_{out}. The k nearest neighbor problem is a well-studied problem. If we insist on getting the exact nearest neighbors in high dimensions, it is hard to find search algorithms that are significantly better than brute-force search. However, if we are willing to accept approximate nearest neighbors, there are efficient algorithms available [12, 1].

We adopt a quality-first principle for choosing acceleration techniques. If an acceleration technique has noticeable negative impact on the synthesized texture, we do not use the technique. With this principle in mind, we accelerate the approximate nearest neighbors search at three levels.

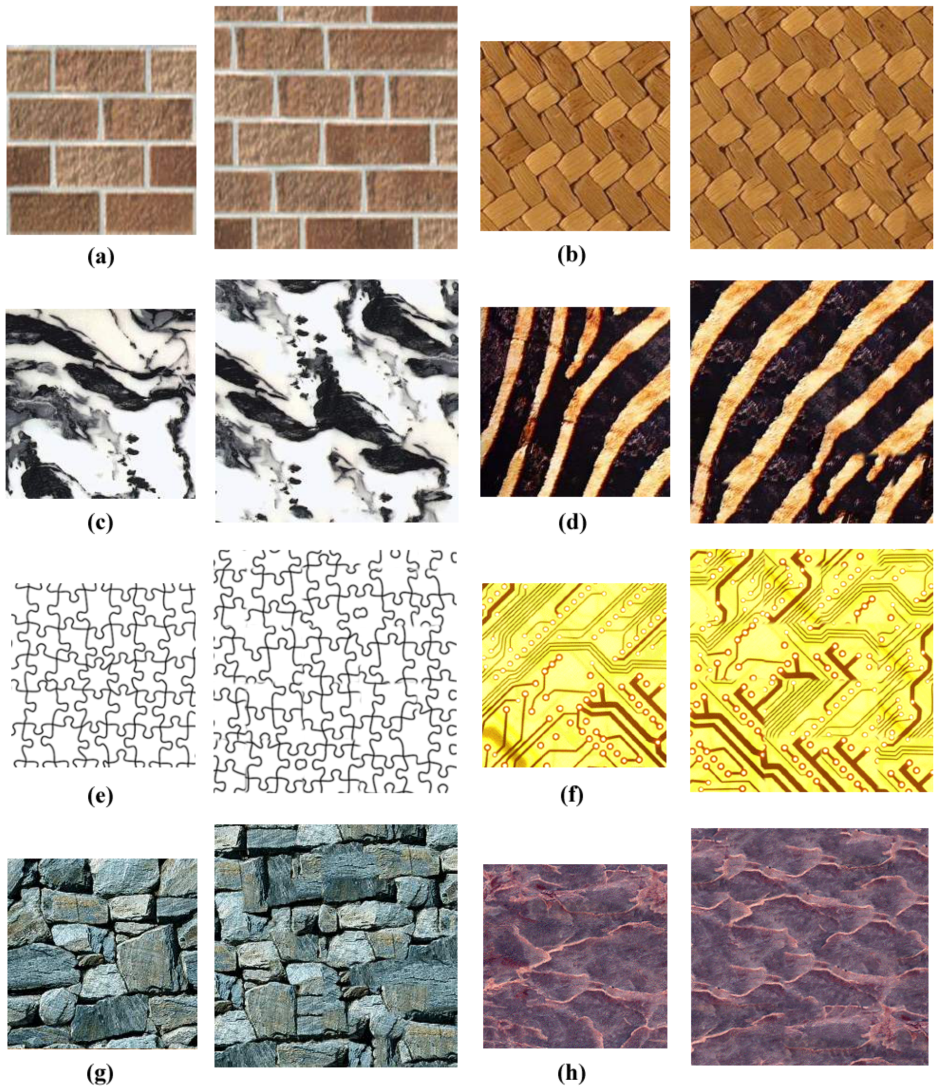

Figure 10: Texture synthesis results. Each example includes the 200×200 input sample and 256×256 result.

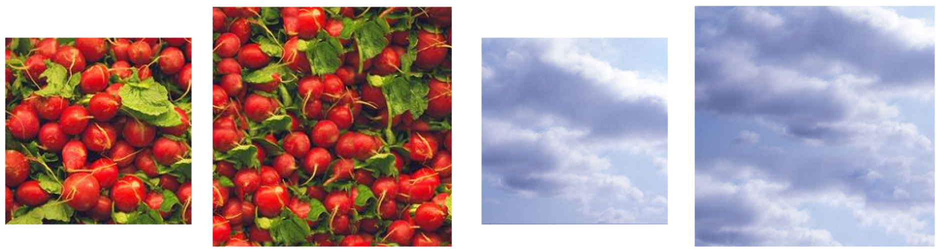

Figure 11: Examples of failure. The smaller texture is the input sample.

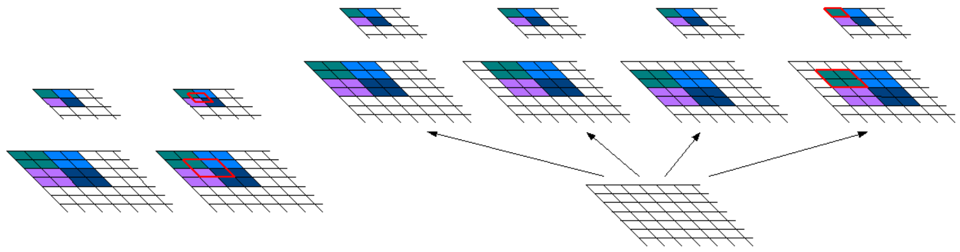

Figure 12: The quad-tree pyramid. The left figure shows two levels in a standard Gaussian pyramid. The pixels in the red rectangle in the lower level do not have a corresponding pixel in the higher level. The right figure shows two levels in a quad-tree pyramid. Every set of four pixels has a corresponding pixel in the higher level.

general acceleration We accelerate the search at general level using an optimized kd-tree [11].

domain specific acceleration We introduce a novel data structure called the quad-tree pyramid for accelerating the search based on the fact that the data points in our search space are images.

data specific acceleration We use principal components analysis (PCA) [9] to accelerate the search for the given input sample texture.

The acceleration at all three levels can be combined to get a compound speed-up of the ANN search.

4.1 Optimized KD-Tree

We use an optimized kd-tree [11] as our general technique for accelerating the ANN search. Initially we experimented with the bd-tree, which is optimal for ANN search [1]. However, our experiments indicate that bd-trees introduce minor but noticeable artifacts in the synthesized textures. In terms of speed, bd-trees and kd-trees are about the same for our searching needs. As pointed out in [1], the optimized kd-tree, with all its optimizations [1], performs as well as the bd-tree on most data sets[1].

A kd-tree partitions the data space into hypercubes using axis-orthogonal hyperplanes [6, 11]. Each node of a kd-tree corresponds to a hypercube enclosing a set of data points. When constructing a kd-tree, an important decision is to choose a splitting rule for breaking the tree nodes. We use the sliding mid-point rule [11]. An alternative choice is the standard kd-tree splitting rule, which splits the dimension with the maximum spread of data points. The standard kd-tree splitting rule has a good guarantee on the height and size of the kd-tree. However, this rule produces hypercubes of arbitrarily high aspect ratio. Since we only allow small errors in boundary zone matching, we want to avoid high aspect ratios. The sliding mid-point rule can also lead to hypercubes of high aspect ratios, but these hypercubes have a special property that prevents them from causing problems in nearest neighbor searching [11].

For searching a kd-tree, we use an adapted version of the search algorithm from [6] with the incremental distance computation of [1]. When the allowed matching errors are small, as is the case for us, the algorithm of [6] is slightly faster than the priority search algorithm, which is superior for finding exact nearest neighbors or for large matching errors [11].

For implementation, we use the Approximate Nearest Neighbor (ANN) library [11] to build a kd-tree for each one of the three boundary zone configurations shown in Figure 3.

4.2 Quad-Tree Pyramid

The kd-tree acceleration does not directly take advantage of the fact that our data points correspond to images. We introduce the quad-tree pyramid (QTP) to address this problem. QTP is a data structure for hierarchical search of image data. To find approximate nearest neighbors for a query vector $\mathbf{v}$, we find the m initial candidates using the low resolution data points and query vector $\mathbf{v}$. In general we should choose $m \ll n$, where n is the number of data points. In our system, we

[1]One case in which the bd-trees do perform significantly better is when the data points are clustered in low-dimensional subspaces, but this is not the case with our texture data.

set $m = 40$. From the initial candidates, we can find the k nearest neighbors using high-resolution query vector $\mathbf{v}$ and data points.

In order to use a kd-tree to accelerate the search of the m initial candidates, we need to filter all data points and the query vector $\mathbf{v}$ into low resolution. A naive approach to filter them is to do it one by one, which will be very expensive in terms of both time and storage because of the large number of data points. With QTP, we only need to filter the input sample texture I_{in}. The low resolution data can be extracted from the filtered I_{in}. As Figure 12 shows, a problem with the standard Gaussian pyramid is that a patch in the high-resolution image may not have a corresponding patch in the low-resolution image. QPT solves this problem by building a tree pyramid. The tree node in QTP is a pointer to an image and a tree level corresponds to a level in the Gauss pyramid. The root of the tree is the input sample texture I_{in}. When we move from one level of pyramid to the next lower resolution level, we compute four children (lower resolution images) with different shifts along the x- and y-directions as shown in Figure 12. With QTP constructed this way, each patch in the higher resolution image I does correspond to a patch in a child of I. In our system, we use a three-level QTP. There are 1, 4, and 16 images of the size of the sample texture at level 1, 2, and 3 respectively.

4.3 Principal Components Analysis

The approximate nearest neighbors search can be further accelerated by considering the special property of the input sample texture. Specifically, we reduce the dimension of the search space using PCA [9]. Suppose that we need to build a kd-tree containing n data points $\{\mathbf{x}_1, ..., \mathbf{x}_n\}$ where each $\mathbf{x}_i$ is a d-dimensional vector. PCA finds the eigenvalues and eigenvectors of the covariance matrix of these data points. The eigenvectors of the largest eigenvalues span a subspace containing the main variations of the data distribution. We choose the subspace dimension so that 97% of the variation the original data is retained. For example, if the texture patch size is $w_B = 64$ and the boundary zone width is $w_E = 4$, then the dimension of original data points is $d = 93$ and PCA typically reduces the dimension to about 20.

4.4 Acceleration Results

Table 1 summarizes the performance of the patch-based algorithm with and without acceleration techniques applied. The table also compares the speed of patch-based sampling with Heeger's algorithm [7] and Wei and Levoy's algorithm [18]. The timings for the patch-based sampling algorithm and Heeger's algorithm are measured by averaging the times of 500 trial runs with different textures. Wei and

Method	Analysis Time	Synthesis Time
QTP+KDTree+PCA	0.678	0.020
QTP+KDTree	0.338	0.044
QTP	0.017	0.256
Exhaustive	0.000	1.415
Heeger	0.0	32
Wei and Levoy**	22.0**	7.5**

Table 1: Timing comparison between the patch-based sampling algorithm and other algorithms. Timings are measured in seconds on a 667 MHz PC for synthesizing 200 $\times$ 200 textures from 128 $\times$ 128 samples. ”Exhaustive” means no acceleration is used. **Wei and Levoy’s timings are taken on a 195 MHz R10000 processor.

Levoy’s timings are taken from [18].

5 Conclusion

We have presented a patch-based sampling algorithm for texture synthesis. Our algorithm synthesizes high-quality textures for a wide variety of textures ranging from regular to stochastic. The algorithm is fast enough to enable real-time texture synthesis on a mid-level PC. For generating textures of the same size and comparable (or better) quality, our algorithm is orders of magnitude faster than existing texture synthesis algorithms. Our algorithm combines the strengths of non-parametric sampling [4, 18] and patch-pasting [22]. In fact, both patch-pasting and the pixel-based non-parametric sampling [4, 18] are special cases of the patch-based sampling algorithm. The patches in our sampling scheme implicitly provide constraints for avoiding garbage. For this reason, our algorithm continues to synthesize high-quality textures even when [4, 18] cease to be effective. For natural textures, the results of patch-based sampling look subjectively better.

For future work, we are interested in extending the ideas presented here for texture synthesis on surfaces of arbitrary topology. With patch-based sampling, we can eliminate a number of problems with existing techniques, e.g., the need for manual texture patch creation and feature mismatches across patch boundaries [15]. Other interesting topics include texture mixtures and texture movie synthesis [7, 2].

References

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching. *Journal of the ACM*, 45:891–923, 1998.

[2] Z. Bar-Joseph, R. El-Yaniv, D. Lischinski, and M. Werman. Texture Mixing and Texture Movie Synthesis Using Statistical Learning. *IEEE Trans. on Visualization and Computer Graphics*, 2001.

[3] J. S. De Bonet. Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Image. In *Computer Graphics Proceedings, Annual Conference Series*, pages 361–368, August 1997.

[4] A. A. Efros and T. K. Leung. Texture Synthesis by Non-Parametric Sampling. In *Proceedings of International Conference on Computer Vision*, 1999.

[5] A. Fournier, D. Fussell, and L. Carpenter. Computer Rendering of Stochastic Models. *Communications of the ACM*, 25(6):371–384, June 1982.

[6] J. Friedman, J. Bentley, and R. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. on Mathematical Software*, 3(3):209–226, 1977.

[7] D. J. Heeger and J. R. Bergen. Pyramid-Based Texture Analysis/Synthesis. In *Computer Graphics Proceedings, Annual Conference Series*, pages 229–238, July 1995.

[8] 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.

[9] I. T. Jollife. *Principal Component Analysis*. Springer-Verlag, New York, 1986.

[10] J.-P. Lewis. Texture Synthesis for Digital Painting. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 245–252, 1984.

[11] D. M. Mount. *ANN Programming Manual*. Deptartment of Computer Science, University of Maryland, College Park, Maryland, 1998.

[12] S.A. Nene and S.K. Nayar. A Simple Algorithm for Nearest-Neighbor Search in High Dimensions. *IEEE Trans. on PAMI*, 19(9):989–1003, September 1997.

[13] K. Popat and R.W. Picard. Novel Cluster-Based Probability Model for Texture Synthesis, Classification, and Compression. In *Proceedings of SPIE Visual Communication and Image Processing*, pages 756–768, 1993.

[14] J. Portilla and E. Simoncelli. Texture Modeling and Synthesis Using Joint Statistics of Complex Wavelet Coefficients. In *Proceedings of IEEE Workshop on Statistical and Computational Theories of Vision*, 1999.

[15] E. Praun, A. Finkelstein, and H. Hoppe. Lapped Texture. In *Computer Graphics Proceedings, Annual Conference Series*, pages 465–470, July 2000.

[16] R. Szeliski and H.-Y. Shum. Creating full view panoramic mosaics and environment maps. *Proceedings of SIGGRAPH 97*, pages 251–258, August 1997.

[17] G. Turk. Genereating Textures on Arbitrary Surfaces Using Reaction-Diffusion. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 289–298, July 1991.

[18] L. Y. Wei and M. Levoy. Fast Texture Synthesis Using Tree-Structured Vector Quantization. In *Computer Graphics Proceedings, Annual Conference Series*, pages 479–488, July 2000.

[19] A. Witkin and M. Kass. Reaction-Diffusion Textures. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 299–308, July 1991.

[20] S. P. Worley. A Cellular Texturing Basis Function. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 291–294, August 1996.

[21] Y. N. Wu, S. C. Zhu, and X. W. Liu. Equivalence of Julesz Ensemble and FRAME Models. *Int'l Journal of Computer Vision*, 38(30):245–261, 2000.

[22] Y. Q. Xu, B. Guo, and H. Y. Shum. Chaos Mosaic: Fast and Memory Efficient Texture Synthesis. In *Microsoft Research Technical Report MSR-TR-2000-32*, April 2000.

[23] S. C. Zhu, X. Liu, and Y. Wu. Exploring Texture Ensembles by Efficient Markov Chain Monte Carlo. *IEEE Trans. on PAMI*, 22(6), 2000.

[24] S. C. Zhu, Y. Wu, and D. B. Mumford. Minimax Entropy Principle and Its Application to Texture Modeling. *Neural Computation*, (9):1627–1660, 1997 (first appeared in CVPR96).

[25] S.C. Zhu, Y. Wu, and D. Mumford. Filters, Random-Fields And Maximum-Entropy (Frame). *International Journal of Computer Vision*, 27(2):107–126, March 1998.