Estimating Parameters for Procedural Texturing by Genetic Algorithms

Xuejie Qin and Yee-Hong Yang

Computer Graphics Research Group, Department of Computing Science, University of Alberta, Edmonton, AB T6G 2E8 Canada

Received February 10, 2001; revised April 2, 2002; accepted April 7, 2002

Procedural texturing has been an active research area in computer graphics with some open problems still unsolved (D. S. Ebert, F. K. Musgrave, K. P. Peachey, K. Perlin, and S. Worley, 1998, “Texturing and Modeling: A Procedural Approach,” Academic Press, San Diego). One major problem is how to estimate or recover the parameter values for a given procedural texture using the input texture image if the original parameter values are not available. In this paper, we propose a solution to this problem and present a genetic-based multiresolution parameter estimation approach. The key idea of our approach is to use an efficient search method (a genetic-based search algorithm is used in this paper) to find appropriate values of the parameters for the given procedure. During the search process, for each set of parameter values, we generate a temporary texture image using the given texturing procedure; then we compare the temporary texture image with the given target texture image to check if they match. The comparison between two texture images is done by using a multiresolution MRF texture model. The search process stops when a match is found. The estimated values of the parameters for a given procedure are the values of the parameters to the procedure to generate a texture image that matches the target texture image. Experimental results are presented to demonstrate the success of our approach. Application of our parameter estimation approach to texture synthesis is also discussed in the paper.

Key Words: crossover; fitness measure; genetic algorithm; parametric texture model; mutation; procedural texturing; texture analysis and synthesis; texturing.

1. INTRODUCTION

Textures exist in practically every naturally occurring object, and texturing can be viewed as a process of adding textures to synthesized images. In computer graphics, existing texturing techniques can be classified into two categories: nonprocedural and procedural.

In the nonprocedural approach, texturing can be done using two methods: texture mapping and texture analysis and synthesis. Texture mapping [2, 11, 12, 23] offers a shading technique...
in which 2D textures are mapped onto surfaces of 3D synthetic objects. However, texture mapping suffers several problems, one of which is the unacceptable artifact problem. To cover the surface of a large object, the algorithm must replicate the texture. This can cause either visible seams or visible repetitions, or both, none of which is acceptable. Another problem is the distortion problem. This is caused by the fact that there is no natural and well-defined mapping from 2D texture images to arbitrary 3D surfaces due to the complexities of textures and 3D models. To solve these problems, one method is to use the image perspective transformation approach [15, 33] or the procedural texturing approach [7].

Texture analysis and synthesis techniques generate synthetic textures based on analyzing input textures. It is in fact a two-phase process, namely, analysis and synthesis. In the analysis phase, input textures are first analyzed and then characterized either parametrically or nonparametrically. In the synthesis phase, synthesized textures with similar appearance are generated based on the texture characteristic information captured in the analysis phase. Some of the successful texture analysis and synthesis approaches include the Markov random field texture approach [5, 8, 17, 36], the frequency domain-based approach [14], the wavelet approach based on joint statistics [22, 26], the multiresolution sampling procedure [3, 13, 21, 31], the interactive texture synthesis [1], and the image quilting method [9].

All of the above analysis and synthesis techniques only apply to 2D; i.e., both analysis and synthesis work in the 2D image plane. Recent work in this area has achieved some success in synthesizing 2D textures onto 3D surfaces. For example, inspired by the idea of a local parameterization method in [23], both Wei and Levoy’s approach [32] and Turk’s approach [29] succeed in synthesizing textures onto 3D surfaces using a multiresolution analysis and synthesis approach. The key to their approaches is that they synthesize a point on a surface locally. In other words, the synthesized texture is generated by treating the small region surrounding a point on the surface as a plane. Thus, like texture mapping, these approaches will cause discontinuity, distortion, or both in their synthesized result if the local region is not chosen properly. In addition, these approaches could be slow because the synthesis is done locally, although speedup could be done using, for example, the tree-structured vector quantization, but with a reduction in quality in the synthesized result.

Another research direction in texturing is the procedural texturing approach, which was first introduced by Cook in 1984 [4]. With the introduction of solid texturing [18, 19] and a texture basis function such as Perlin’s noise [19], the use of procedural texturing has been widely accepted in the computer graphics community. In this approach, procedures are developed to generate synthetic textures without requiring input textures. By calling a compact (storage efficient) procedure, textures are generated directly on surfaces of 3D objects without seams and without discontinuity. Using different procedures, various kinds of realistic images containing marble, wood, stone, water, cloud, flame, and crumpled wrinkle can be generated in an efficient way. Some of the useful procedural-texturing techniques include the shade tree [4], pixel stream editor and solid texturing [18, 19], hyper-texture [20], the reaction-diffusion system [27, 28, 34], and cellular texture basis functions [25, 35].

Although, recent work [23, 29, 32] on statistical texture synthesis has achieved some success in applying textures onto 3D surfaces, the procedural approach still has some advantages over the nonprocedural approach, some of which are discussed below:

- Some natural phenomena with motions, such as gas, fire, fluid, and cloud, are difficult to synthesize using nonprocedural approaches, while it is more appropriate and relatively straightforward to model using procedural texturing [7].
It is easy and efficient to generate hypertexture [20] using procedural texturing, while it is difficult to do so using nonprocedural approaches.

Other advantages include compact representation, storage efficiency, no seams, no repetition, and no discontinuity.

There are, however, some major limitations with the procedural approach, which remain as open problems for further research. These limitations are summarized below (see [7] for a detailed discussion):

- Texturing procedures often require the setting of a large number of parameter values, a process that is time-consuming and nontrivial. This makes it difficult, if not impossible, to manually estimate the parameters of a given procedural texture.
- The design of texturing procedures is often done based on the experience of the designer and is largely a manual process, which makes procedures difficult to understand and design. Thus, designing texturing procedures is a difficult task.
- Although procedural texturing can generate realistic synthetic textures without requiring input textures, it also makes the texturing process difficult to predict and control. For example, if the value of a parameter is changed, a very different image may result.

This paper addresses the first problem mentioned above and gives a solution to estimate the parameters of a given procedural texture. We present and describe a genetic-based multiresolution approach to solving this problem. It is assumed that for a given procedural texture, the texturing procedure is known, and that this assumption is natural and reasonable (see the next section). The objective of this approach is to estimate or recover the values of parameters so that a similar (if not exactly the same) texture can be regenerated using the same procedure. The basic idea of our approach is to use an efficient search method, for example a genetic-search algorithm as described in this paper, to search the parameter space to find the values of the parameters whose corresponding texture closely matches the given procedural texture. In each search step, for each set of parameter values in the parameter space, we generate a temporary texture image on the fly since we already know the texturing procedure; then we compare the temporary texture image with the given texture image to check if they match closely. The comparison (we call this a match) between two texture images is done by using a new extended multiresolution version of the parametric MRF texture model described in [5]. The search process stops when there is a match found; then the estimated values for the parameters of the given procedural texture image are the values of the parameters of the matched texture image. Experimental results show that our approach works quite well.

The paper is organized as follows. Section 2 describes the basic idea of our approach and discusses some related issues on optimal search and texture-similarity measuring. Section 3 describes the details of our approach and presents the algorithm. Section 4 presents the experimental results for our genetic-based multiresolution parameter estimation approach. Section 5 describes the application of the parameter estimation approach to texture synthesis. Finally, conclusions and future work are described in Section 6.

2. BASIC IDEA

Procedural textures are generated by some texture generating procedures with a large number, typically 10–50, of parameters. If the values of these parameters were lost or not known, it would be very difficult, if not impossible, to recover the exact values of these
parameters used by the procedure to generate a particular given texture by trial-and-error even if the procedure were known. Therefore, it is desirable to have a systematic method to estimate the values of these parameters. In this section, we give the basic idea of our solution to this problem and present an overview of our approach.

For a given procedural texture, the parameter space is taken as the set of all parameters that are used to generate the texture by the procedure. For example, if we have a simple procedure called \( \text{Cloud}(p_1, p_2, p_3) \) for generating cloud-like textures, the parameter space is \( (p_1, p_2, p_3) \). For a given procedural texture (target texture), our objective is to estimate the values of procedural parameters in the parameter space systematically so that a similar, if not exactly the same, texture can be regenerated.

Note that we already know the procedure and its parameter space. Now the question arises of which point (a set of parameter values) in the parameter space the procedure uses to generate the input texture. Going a step further, suppose that for each point in the parameter space, we generate a temporary texture using the procedure. Then, we can just compare the temporary texture with the target texture to see if they look similar (we call this a match between two textures). Thus, intuitively, our solution to the problem is to search through the parameter space to find a point whose corresponding texture matches the target texture. The matched point that we find in the parameter space corresponds to the estimated parameter values for the target texture. Figure 1 illustrates our idea using a simple procedure \( \text{Cloud}(p_1, p_2, p_3) \).

Now we must determine which search technique to use and how to match two texture images. First, it is noted that for most of the procedural textures, the texture parameter space (TPS) is usually very large. For example, \( \text{TPS} = \{(a_0, a_1, \ldots, a_9) | a_i \in [0,1], 0 \leq i \leq 9\} \)

![Diagram](image_url)

**FIG. 1.** A diagram illustrating the basic idea of our approach using an example procedure. For reasons of simplicity, we assume that procedure \( \text{Cloud()} \) has three parameters: \( p_1, p_2, p_3 \). For each point \( P \) (a vector in \( R^3 \)) in the parameter space, a temporary texture is generated by \( \text{Cloud}(P) \), which is then compared with the target texture to see if it matches the target. If there is a match, then the search returns \( P \) as the estimated parameter values of the target texture and stops. Otherwise, the search goes to the next point in the parameter space.
with 16 different values for each $a_i$ has a cardinality of $16^{10} = 1099511627776 \approx 10^{12}$. In addition, our search algorithm should be capable of finding the best match. In other words, the search algorithm should be able to locate global optima from any starting point in the parameter space. A direct search method such as the iterative hill-climbing algorithm is local in scope; i.e., the optimum it seeks is the best in the neighborhood of the current starting point. Thus the solution found by a direct search method is not guaranteed to be a global solution. Another disadvantage of using a direct search method is that it is slow because it starts searching from a single point and proceeds in a single direction.

To overcome these two problems (local optima and low performance), random search techniques are more appropriate for our purpose. In this paper, a genetic-based search algorithm [16] is used, although other random search techniques (such as simulated annealing) may be used as well. It is noted that a genetic-based search method starts searching randomly from multiple points, i.e., a population of points, and thus proceeds searching in multiple directions, which is much more efficient than a direct search method, which performs searching in one single direction. In addition, a genetic-based search method can guarantee to find a nearly global optimal solution [10, 16].

The key to matching two texture images is how to measure the similarity between two texture images. There are some successful existing parametric texture estimators (such as [5, 36]). Although the FRAME model [36] is a powerful and general texture model, its applicability to solving the similarity problem is beyond the scope of this paper. In addition, the size of a filter used in the FRAME model [36] is large (e.g., $16 \times 16$ pixels compared with a $3 \times 3$ clique in [5]), which may cause low performance. Cross and Jain’s model [5] uses small size cliques (e.g., $3 \times 3$), but may not sufficiently model general textures. To overcome these problems, we extend the MRF texture estimator used in [5] to multiresolution. Such an extension is inspired by recent multiresolution techniques in texture analysis and synthesis [3, 13, 31]. In the next section, we describe the details of our approach.

3. DETAILS OF OUR APPROACH

First, we present a general description of our algorithm. We then describe the algorithm in detail including how to calculate the key (see Section 3.2 for its definition) of a texture image; that is, how to match the search key of an input texture image using a genetic-based search algorithm. The key of a texture image is first defined in single resolution, and then it is extended to multiresolution for more accurate search results. The genetic-based search process is the main part of the algorithm, and its basic operations such as selection, crossover, and mutation are described in detail.

3.1. The Algorithm

The diagram shown in Fig. 2 gives the outline of the algorithm of our approach. For a given input texture $I$, let $PPS = \{(a_0, a_1, \ldots, a_n) | a_i \in [0,1], 0 \leq i \leq n\}$ be its procedural parameter space, and let $s(I)$ be the search key (the definition is given later), which is calculated from our extended multiresolution MRF (MRFMRF) texture estimator. The genetic-based search algorithm first initializes randomly a population of search points $P_1, P_2, \ldots, P_m$ in the $PPS$ with $m$ known as a prior. For each $P_i$, the algorithm generates a temporary texture image $I_i$ and then calculates its key $s(I_i)$. After this step, each $s(I_i)$ is compared with the search key $s(I)$. If one of the $s(I_i)$ matches $s(I)$ within a user-specified error limit, the
3.2. Calculation of the Key of a Texture Image

One important step in our algorithm is to measure the similarity between two textures. This is done by first calculating the keys of the two textures and then checking the square errors between the keys. For reason of simplicity, we first give the definition of the key of a texture image in the sense of a single resolution. Later on, we describe how to calculate the key in multiresolution.

**DEFINITION 3.1.** The *key* of a given gray scale texture image \( I \) is defined as the values of the model parameters estimated by the MRF texture model described in [5]. For a color
texture image $I$, its key is defined as the combination of the three keys for the R, G, and B channels of $I$. The key of an input texture image is called the search key. The MRF texture model used in this used paper is described below.

Let $I = \{(i, j) | 0 \leq i, j \leq N - 1\}$ denote a texture image with $N \times N$ pixels. Let $X(q)$ denote the value of gray level at pixel location $(i, j)$ with index $q = N^*i + j$ in image $I = \{(i, j) | 0 \leq i, j \leq N - 1\}$. The pixel $q$ is said to be a neighbor of the pixel $r = N^*s + t$ at location $(s, t)$ if

$$p(X(r) | X(0), X(1), \ldots, X(r - 1), X(r + 1), \ldots, X(N^2 - 1))$$

depends on $X(q)$.

The MRF texture model is based on the fact that the gray level at a pixel location is highly dependent on the gray levels of its neighboring pixels. As in [5], we assume that the pixel $r = N^*s + t$ at location $(s, t)$ is a neighbor of the pixel $q = N^*i + j$ at location $(i, j)$ if $(s, t)$ is close to $(i, j)$. In this paper, we assume a binominal distribution for $p(X(q) = k | \text{neighbors of } q)$, which is given as

$$p(X(q) = k | \text{neighbors of } q) = \binom{G - 1}{k} \cdot \left[ \frac{e^{T(q)}}{1 + e^{T(q)}} \right]^k \cdot \left[ 1 - \frac{e^{T(q)}}{1 + e^{T(q)}} \right]^{G - 1 - k}$$

(3.1)

where $G$ is the total number of gray levels in image $I$, e.g., $G = 256$, and $k$ is the gray level value at pixel $q$.

Equation (3.1) is an example of the MRF texture model. The function $T(q)$ in this model is dependent on the gray level values of the neighboring pixels around $q$, which are given as

$$T(q) = a + \sum_{i=1}^{S} \sum_{j=1}^{4} b_{ij} s_{ij}(q)$$

(3.2)

where $S$ is called the order of the MRF texture model, whose value can be 1, 2, 3, or 4. Note that in [5], Cross and Jain use a different formula for (3.2). By experiments, we found that the original MRF model has some limitations; for example, regular textures are not modeled well and a large size is required for an image in order to obtain good parameter estimates (see [5] for a detailed discussion). Using the new formula (3.2) combined with the multiresolution technique to be discussed at the end of this section, more general textures can be modeled.

The function $s_{ij}(q)$ is the gray level value of a neighboring pixel around $q$ or the sum of gray level values of the neighboring pixels around $q$. Figure 3 shows the neighbors of $q$ and $s_{ij}(q)$ with order $S = 1$ and 2. For the cases of $S = 3$ and 4, the reader is referred to [24].

DEFINITION 3.2. The MRF model parameters of an image $I = \{(i, j) | 0 \leq i, j \leq N - 1\}$ are coefficients $a$ and $b_{ij}$ in Eq. (3.2) with $i = 1, \ldots, S, j = 1, 2, 3, 4$.

To estimate the MRF texture model parameters $a$ and $b_{ij}$ for an image $I$, we need to divide $I$ into disjoint sets of points called codings. We follow a coding scheme that is the same as that described in [5]. Each coding $C$ consists of a set of points from $I$ such that if $p$ and $q$ are two points in $C$, then $p$ is not a neighbor of $q$ in the sense of the MRF texture model defined in Eq. (3.1). Note that the first-order MRF texture model (i.e., $S = 1$
in (3.2)) requires at least two codings for image \( I \). The second-order MRF texture model requires at least four codings, and the third- and fourth-order MRF texture model requires at least nine codings. Theorems 3.3–3.5 give a solution on how to calculate the key of a texture image, and the proofs are given in the Appendix. To the best of our knowledge, these theorems are original contributions of this paper. Based on these three theorems, a nonlinear equation system solver for estimating MRF texture model parameters of a texture image is implemented.

**Theorem 3.3.** Let \( C_k \) denote the \( k \)th coding of image \( I \), let \( q \) be a point in \( C_k \), and let \( g_q \) be the gray level value at \( q \). Then the log likelihood \( L(C_k) \) defined as

\[
L(C_k) \overset{\text{def}}{=} \sum_{q \in C_k} \ln[p(X(q) = g_q | \text{neighbors of } q)]
\]

(3.3)
can be calculated by

\[
L(C_k) = \sum_{q \in C_k} \left[ T(q) \cdot g_q - \ln(g_q!) - \ln((n - g_q)! - n \cdot \ln(1 + e^{T(q)}) \right] + c_k \cdot \ln(n!)
\]

(3.4)

where \( p(X(q) = g_q | \text{neighbors of } q) \) is given in Eq. (3.1), \( T(q) \) is given in Eq. (4.2), \( n = G - 1 \), and \( c_k = |C_k| \), i.e., the number of pixels in \( C_k \).

**Theorem 3.4.** Assume the same notations as in the previous theorem; then the MRF texture model parameters \( a \) and \( b_{ij} \) over coding \( C_k \) of image \( I \) can be calculated by solving the nonlinear equation system (3.5) using a globally convergent method [6].

\[
\frac{\partial L(C_k)}{\partial a} = \sum_{q \in C_k} g_q - n \cdot \sum_{q \in C_k} \left( \frac{e^{T(q)}}{1 + e^{T(q)}} \right) = 0
\]

\[
\frac{\partial L(C_k)}{\partial b_{ij}} = \sum_{q \in C_k} (g_q \cdot s_{ij}(q)) - n \cdot \sum_{q \in C_k} \left( \frac{e^{T(q)}}{1 + e^{T(q)}} \cdot s_{ij}(g_q) \right) = 0
\]

(3.5)

where \( i = 1, \ldots, S \), \( j = 1, 2, 3, 4 \), and \( S \) and \( s_{ij} \) are defined in Eq. (3.2).
To solve the nonlinear equation system (3.5), we need the Jacobian matrix of the system. The following theorem gives a solution on how to compute the Jacobian matrix of the nonlinear equation system (3.5).

**Theorem 3.5.** Assume the same notations as in the previous theorem. Then the Jacobian matrix of the nonlinear equation system (3.5) can be computed using the following second-order partial derivatives:

\[
\frac{\partial^2}{\partial a^2} (L(C_k)) = -n \cdot \sum_{q \in C_k} \left[ \frac{e^{T(q)}}{(1 + e^{T(q)})^2} \right],
\]

\[
\frac{\partial^2}{\partial a \partial b_{ij}} (L(C_k)) = -n \cdot \sum_{q \in C_k} \left[ s_{ij}(q) \cdot \frac{e^{T(q)}}{(1 + e^{T(q)})^2} \right],
\]

\[
\frac{\partial^2}{\partial b_{lm} \partial b_{ij}} (L(C_k)) = -n \cdot \sum_{q \in C_k} \left[ s_{lm}(q) \cdot s_{ij}(q) \cdot \frac{e^{T(q)}}{(1 + e^{T(q)})^2} \right].
\]

After we have estimated the MRF texture model parameters \(a\) and \(b_{ij}\) on each coding \(C_k\) of image \(I\), the final estimated values of \(a\) and \(b_{ij}\) for \(I\) are obtained by taking the average value over all the codings, which are taken as the key for image \(I\). Note that the input texture image is assumed to be a gray scale image in Theorems 3.3–3.5. For a color image \(I\), we can calculate the key for each color channel. Thus the key of a color texture image is a vector of three keys, one for each of the R, G, and B channels. In particular, let \(I_1\), \(I_2\), and \(I_3\) be the gray scale images for the R, G, and B channels of \(I\), respectively; then we have \(s(I) = (s(I_1), s(I_2), s(I_3))\).

The key of a given texture described so far is for single resolution. As discussed in Section 2, it may be difficult to model general textures since the size of clique (i.e., neighborhood) used is too small to capture large-scale structures. Indeed, the experimental results in the next section demonstrate this shortcoming. To address this problem, we extend the key of a given texture to multiresolution. It is noted that the multiresolution techniques [3, 13, 31] can capture large-scale structures while using a small number of pixels and that the techniques have been successfully used in recent work on statistical texture analysis and synthesis [3, 13, 31] to a wide variety of textures.

It is straightforward to extend the key of a texture image to multiresolution. For each pyramid level, we view it as an image and calculate its key in the same way as that described before. Then, the key for the entire image pyramid is a vector of the keys from all the pyramid levels, which is then used as the key for the texture image. For example, for image \(I\), suppose that we build a pyramid of four levels: \(l_0, l_1, l_2, l_3\). Let \(s_0, s_1, s_2, s_3\) be the keys for \(l_0, l_1, l_2, l_3\), respectively. Then the key for \(I\) is \(s(I) = (s_0, s_1, s_2, s_3)\). In the rest of the paper, we assume that the key of a given texture image is calculated from its multiresolution pyramid. In the next section, we describe the operations of the genetic-based searching process.

### 3.3. Genetic-Based Searching Process

After the search key of the input image is calculated, the algorithm proceeds to the genetic-based search process. The purpose of the process is to match the search key by searching through the PPS using genetic algorithms. Before we describe the details of the genetic-based search process, we first give a brief description of genetic algorithms.
3.3.1. Genetic Algorithms

Genetic algorithms [10, 16] have been widely used in problems of searching, optimization, machine learning, and evolutionary computing. They are based on the mechanics of natural selection and genetics, and operate on populations of strings (or genes), with the string coded to represent some underlying parameter set. Basically, a genetic algorithm uses three basic operations for searching: selection, crossover, and mutation.

Selection is a process of reproduction in which individual strings are copied from the parent population to the child population according to their fitness. The better fitness an individual has, the better chance (probability) it will be selected. The selection operation may be implemented in an algorithmic form in a number of ways. A simple and efficient way is to create a biased roulette wheel [10]. Each current individual in the population has a slot in the roulette wheel with an area proportional to its fitness. To select an individual for the next generation, we simply spin the roulette wheel. When the roulette wheel stops, the individual in the slot when the wheel is stopped is selected. Note that the selection operation is a random process.

After selection, the crossover operation is performed. The newly selected individual strings are first mated at random. Then, each pair of the strings (parents) to be mated may undergo crossover as follows: an integer position \( k \) along the string is selected randomly between 1 and \( l - 1 \), where \( l \) is the string length. Two new strings (offsprings) are created by swapping the parts between \( k + 1 \) and \( l \) inclusively.

The mutation operation is performed within each individual string. It is an occasional random alteration of the value of a string position with a small probability. For example, in the binary coding scheme for strings, this operation simply means changing a 1 to a 0 and vice versa at a bit position. The purpose of mutation is to protect against premature loss of important information. However, as suggested in [10], the probability of mutation must be kept as low as 0.005–0.01, while the probability of crossover must be kept as high as 0.8–0.95.

3.3.2. Encoding and Initialization

Our genetic-based search process operates on a population of search points in the \( PPS \). To keep the demonstration simple, we assume a \( PPS \) of 10 procedural parameters, each with 16 different values; i.e., \( PPS = \{a_0, \ldots, a_9\} | a_i \in [0, 1], 0 \leq i \leq 9 \} \). Note that a point \( P \) in the \( PPS \) is a vector of real numbers. Thus, a simple and efficient encoding scheme for \( P \) is to represent it by a string (i.e., an array) of real numbers. For this reason, a point in the \( PPS \) is also called a string. Figure 4 gives a visual representation of the encoding scheme for a string \( P = (a_0, \ldots, a_9) \in PPS \).

The genetic-based search process starts from an initial population of search points. A random process is used to generate these search points, which are uniformly distributed in the search space. For example, to generate an initial population of 50 points in the \( PPS \) encoded as in Fig. 4, we can call a random number generator \( 10 \times 50 = 500 \) times, and

\[
\begin{array}{cccccccccc}
  a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 \\
\end{array}
\]

**FIG. 4.** Encoding scheme of a point in the \( PPS \). Each parameter \( a_i \) is a real number in \([0,1] \).
at each time, the generator generates a uniformly distributed random number in the range [0,1].

The size of the population, i.e., the number of points in the population, is crucial for the genetic-based search algorithm. A population too small will limit the search range in the PPS, which may not lead to an appropriate solution, while a population too large will slow down the algorithm. According to [10, 16], a good range of population size is 20–100. For satisfactory results, we found by experiments that a fixed value of 60 for population size generates good results. One can also use a more accurate and efficient scheme described in [24] to determine the population size based on the number of parameters in the PPS and the resolution of the parameters.

3.3.3. Matching the Search Key

DEFINITION 3.6. Let \( \varepsilon \) be a predefined small positive number; e.g., \( \varepsilon = 10^{-4} \). Suppose \( I_1, I_2 \) are two gray scale texture images and \( s_1(I_1), s_2(I_2) \) are the corresponding keys of \( I_1, I_2 \), respectively. Then \( s_1(I_1) \) is said to match \( s_2(I_2) \) within the predefined error limit \( \varepsilon \) if the square error between \( s_1(I_1) \) and \( s_2(I_2) \) is less than \( \varepsilon \); i.e.,

\[
(a^1 - a^2)^2 + \sum_{i=1}^{S} \sum_{j=1}^{4} (b^{1}_{ij} - b^{2}_{ij})^2 < \varepsilon
\] (3.7)

where \( s_1(I_1) = (a^1, b^{1}_{11}, b^{1}_{12}, \ldots, b^{1}_{S3}, b^{1}_{S4}) \) and \( s_2(I_2) = (a^2, b^{2}_{11}, b^{2}_{12}, \ldots, b^{2}_{S3}, b^{2}_{S4}) \). If \( I_1, I_2 \) are color images, then \( s_1(I_1) \) is said to match \( s_2(I_2) \) within the predefined error limit \( \varepsilon \) if their keys for each channel satisfy (3.7).

Suppose \( s(I) \) is the search key of a given input texture image \( I \). After an initial population of search points \( P_1, \ldots, P_m \) is generated, the genetic-based search process generates a temporary image \( I_i \) for each point \( P_i \) and calculates a temporary key \( s_i(I_i) \); then the algorithm checks whether there is a temporary key \( s_i(I_i) \) that matches with \( s(I) \) within a predefined error limit \( \varepsilon \). If there is such a key \( s_i(I_i) \), then the algorithm outputs the matched \( P_i \) as the final estimated parameter values and stops. Otherwise, the algorithm generates the next population of search points by selection, crossover, and mutation operations, and proceeds to the next iteration. These three genetic operations are described in the next section.

3.3.4. Generating New Population

In this section, search points are considered as strings for easy understanding of the three operations: selection, crossover, and mutation. We assume that the new population of strings to be generated has the same size as the previous population. The next population is generated completely based on the previous population by using selection, crossover, and mutation. In these three operations, the fitness of a string plays a crucial role. Thus we give its formal definition below.

DEFINITION 3.7. Let \( I \) and \( s(I) \) denote the input image and its search key, respectively. Suppose \( P \) is a string in the population and \( s(P) \) is the key of the temporary image generated from \( P \). Then the fitness of \( P \) relative to \( I \), denoted by \( \text{fitness}(P : I) \), is defined as the square
error between \( s(P) \) and \( s(I) \); i.e.,

\[
\text{fitness}(P : I) = (a - a_0)^2 + \sum_{i=1}^{S} \sum_{j=1}^{4} (b_{ij} - b_{ij}^0)^2
\]  
(3.8)

where \( s(I) = (a_0^I, b_{11}^I, b_{12}^I, \ldots, b_{53}^I, b_{54}^I) \) and \( s(P) = (a, b_{11}, b_{12}, \ldots, b_{53}, b_{54}) \). Note that a smaller value of \( \text{fitness}(P : I) \) means a better fitness of string \( P \). If the key is calculated from an image pyramid (i.e., multiresolution), the right side of (3.8) is replaced by the sum of several terms, where each term corresponds to one pyramid level and can be calculated in the same manner as that given in (3.8).

**Selection.** The purpose of selection is to select parent strings from the previous population so that the algorithm can use them to generate child strings (new strings) by performing crossover in the next step. A string is selected by its fitness value given in (3.8). The better fitness it has, the better chance it will be selected. The selection process can be implemented as a biased roulette wheel described in [10].

**Crossover.** After completing the selection process, a temporary population, called a parent population, is ready for the crossover operation. Child strings are generated by crossing over between two parent strings with a predefined crossover probability \( pc \in [0.8, 0.95] \) (see [10] for a discussion on this). The crossover position between two parent strings is determined randomly. Note that, if no crossover operation is required between a pair of parent strings, we just copy the parent strings into the child strings. Figure 5 shows the crossover operation between two parent strings \( P_1 \) and \( P_2 \).

**Mutation.** The mutation operation is performed at each position for every child string generated from the process of crossover, but with a predefined low probability \( pm \in [0.005, 0.01] \) (see [10] for a discussion on this). Mutation at the \( i \)th position of a string \( C = c_0c_1 \ldots c_n \) means that we randomly change the value of \( c_i \) to one of its possible values in \( \{c_0, c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n\} \). A better scheme for this is that we can change the value of \( c_i \) to one of its possible values so that string \( C \) has a better fitness. The detailed algorithm for mutation can be found in [24].

---

**FIG. 5.** The crossover operation between two parent strings \( P_1 \) and \( P_2 \). The strings \( C_1 \) and \( C_2 \) are the child strings generated by the crossover operation between \( P_1 \) and \( P_2 \) at the 4th position.
After the selection, crossover, and mutation operations, a new population of strings with the same size as the previous population is generated. In general, the best fitness and the average fitness of strings in the new population will both be improved. Therefore, in most cases, the genetic-based search process will find a solution within a predefined error limit although it may not always guarantee a solution (see [10, 16] for a detailed discussion). In our experiments, if there is no required solution found after a predefined number (e.g., 200) of search iterations, the best matched string is returned. In the next section, we present the experiment results.

4. EXPERIMENTAL RESULTS

In this section, we present some experimental results for parameter estimation; more results can be found in [24]. Four types of procedural textures are used as input textures for experimentation: marble, cloud, crumpled wrinkle, and wood. Texturing procedures used to generate these four types of input textures are represented as surface or displacement shaders written in the RenderMan shading language [30]. Therefore, the PPS for each type of input texture can be determined from the header of the shaders.

Figure 6 gives some examples of procedural input textures and their corresponding reconstructed textures generated by the estimated parameters using multiresolutions (three

**FIG. 6.** Examples of procedural input textures and their synthesized textures. Note that three pyramid levels are used to calculate the key of a texture image.

**FIG. 7.** Examples of the estimation results using different number of pyramid levels. The first texture is the original, and the rest are the reconstructed textures using 3, 2, and 1 pyramid levels with the predefined defined error limit \( \varepsilon = 10^{-4} \), respectively (from left to right).
### TABLE 1
The Original Texture Parameter Values for MR and Recovered Values from MR-S

<table>
<thead>
<tr>
<th></th>
<th>MR</th>
<th>MR-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>c2</td>
<td>0.00</td>
<td>0.0085</td>
</tr>
<tr>
<td>c3</td>
<td>0.00</td>
<td>-0.01</td>
</tr>
<tr>
<td>c4</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>c11</td>
<td>-1.00</td>
<td>-0.9</td>
</tr>
<tr>
<td>c12</td>
<td>0.00</td>
<td>0.135</td>
</tr>
<tr>
<td>c13</td>
<td>0.00</td>
<td>-0.14</td>
</tr>
<tr>
<td>c14</td>
<td>0.00</td>
<td>0.017</td>
</tr>
<tr>
<td>c22</td>
<td>0.00</td>
<td>-0.01</td>
</tr>
<tr>
<td>c23</td>
<td>0.00</td>
<td>-0.01</td>
</tr>
<tr>
<td>c24</td>
<td>0.00</td>
<td>0.05</td>
</tr>
<tr>
<td>c33</td>
<td>0.00</td>
<td>0.013</td>
</tr>
<tr>
<td>c34</td>
<td>0.00</td>
<td>-0.04</td>
</tr>
<tr>
<td>c44</td>
<td>0.00</td>
<td>-0.03</td>
</tr>
<tr>
<td>Scale</td>
<td>2.00</td>
<td>2.05</td>
</tr>
<tr>
<td>NNOISE</td>
<td>1.00</td>
<td>1.05</td>
</tr>
<tr>
<td>TextureScale</td>
<td>2.50</td>
<td>1.75</td>
</tr>
<tr>
<td>Lambda</td>
<td>3.00</td>
<td>3.27</td>
</tr>
</tbody>
</table>

### TABLE 2
The Original Texture Parameter Values for CL and Recovered Values from CL-S

<table>
<thead>
<tr>
<th></th>
<th>CL</th>
<th>CL-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>-1.00</td>
<td>0.600</td>
</tr>
<tr>
<td>c2</td>
<td>0.000</td>
<td>0.920</td>
</tr>
<tr>
<td>c3</td>
<td>0.000</td>
<td>0.640</td>
</tr>
<tr>
<td>c4</td>
<td>0.000</td>
<td>-0.160</td>
</tr>
<tr>
<td>c11</td>
<td>0.000</td>
<td>-0.160</td>
</tr>
<tr>
<td>c12</td>
<td>0.000</td>
<td>-0.280</td>
</tr>
<tr>
<td>c13</td>
<td>0.000</td>
<td>-0.160</td>
</tr>
<tr>
<td>c14</td>
<td>0.000</td>
<td>-0.120</td>
</tr>
<tr>
<td>c22</td>
<td>0.000</td>
<td>-0.480</td>
</tr>
<tr>
<td>c23</td>
<td>0.000</td>
<td>0.720</td>
</tr>
<tr>
<td>c24</td>
<td>0.000</td>
<td>-0.320</td>
</tr>
<tr>
<td>c33</td>
<td>0.000</td>
<td>-0.140</td>
</tr>
<tr>
<td>c34</td>
<td>1.000</td>
<td>-0.760</td>
</tr>
<tr>
<td>c44</td>
<td>1.000</td>
<td>-0.04</td>
</tr>
<tr>
<td>TextureScale</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Octaves</td>
<td>8.000</td>
<td>8.000</td>
</tr>
<tr>
<td>Omega</td>
<td>0.500</td>
<td>0.500</td>
</tr>
<tr>
<td>Lambda</td>
<td>2.000</td>
<td>2.000</td>
</tr>
<tr>
<td>Threshold</td>
<td>0.000</td>
<td>0.000</td>
</tr>
</tbody>
</table>
pyramid levels). The input textures are labeled MR, CL, WD, and WR, which represent marble, cloud, wood, and wrinkle, respectively. Their corresponding reconstructed textures are labeled by MR-S, CL-S, WD-S, and WR-S, respectively. The original parameter values used to generate MR, CL, WD, and WR, and the estimated parameter values from MR-S, CL-S, WD-S, and WR-S, are given in Tables 1–4. The fitness values of MR-S, CL-S, WD-S, and WR-S are given in Table 5.

Two important observations are made from the above experimental results. First, as was discussed earlier, it is insufficient to measure the similarity of two textures in a single resolution; therefore we do this in multiresolution, i.e., calculate the key of a texture based on its image pyramid. Figure 7 gives an example of the reconstructed textures using different numbers of pyramid levels. As the number of pyramid levels increases, the match between the reconstructed texture and the original texture improves. From the experimental results,
we found that in most cases the multiresolution parameter estimation performed much better visually than the single-resolution one.

Second, we found that the accuracy of the estimation depends on the value of the predefined error limit (see Definition 3.6 in Section 3). A smaller value of the predefined error limit results in a more accurate result. In other words, the smaller the fitness value a reconstructed texture has, the better it matches the input texture. Figure 8 gives an example of three reconstructed textures of an input texture (the left image) with increasing fitness values, which are obtained by three different values of the predefined error limit. One can see that from left to right, the visual pattern difference between the reconstructed texture and the original texture increases as the fitness value increases.

### 5. APPLICATION TO TEXTURE SYNTHESIS

Given an input texture, how do we use the parameter estimator approach described in this paper and procedural texturing to synthesize the required texture on surfaces of 3D synthetic objects? This issue is discussed in this section, and some experimental results are also presented. It is noted that all surface synthesis results presented in this section are generated using 3D solid texturing techniques [18, 19] and rendered by Blue Moon Rendering Tools.

For a procedural input texture, it is straightforward to synthesize it onto surfaces of synthetic objects because we already know the texturing procedure. Basically, we first use the parameter estimation approach described earlier in this paper to estimate the parameter values of the input texture. Once the parameter values are known, we can then ask the procedure to generate synthetic textures (which should look similar to the input texture) onto arbitrary surfaces. The first two rows of Fig. 9 give two examples of procedural input textures and their synthesis results onto surfaces of different objects.

For an arbitrary input texture, such as a real texture image, it is relatively difficult to synthesize it onto arbitrary surfaces. Recent research on statistical texture analysis and synthesis [29, 32] has achieved some success in this area. An alternative method is to use a texturing procedure to model each type of input textures, for example a brick procedure to model brick-like textures. In this way, each input texture can be treated as a procedural texture, and then it is possible to synthesize it onto surfaces by first estimating the parameter values for the input texture using our parameter estimation approach. The last two rows of Fig. 9 give two examples of using real input textures and their synthesized textures onto surfaces of different objects.

There are two subtle issues of synthesizing an arbitrary input texture onto surfaces using this approach. First, it is difficult to have a general texturing procedure to model all types of real input textures. In this paper, we assume that for each type of real input textures, there is a corresponding texturing procedure to model it. Second, for a given input real texture, it is difficult to decide in a systematic way which texturing procedure we should use to

<table>
<thead>
<tr>
<th>TABLE 5</th>
<th>Fitness Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR-S</td>
<td>0.0000016</td>
</tr>
<tr>
<td>CL-S</td>
<td>0.0000035</td>
</tr>
<tr>
<td>WD-S</td>
<td>0.0000109</td>
</tr>
<tr>
<td>WR-S</td>
<td>0.000045</td>
</tr>
</tbody>
</table>

we found that in most cases the multiresolution parameter estimation performed much better visually than the single-resolution one.

Second, we found that the accuracy of the estimation depends on the value of the predefined error limit (see Definition 3.6 in Section 3). A smaller value of the predefined error limit results in a more accurate result. In other words, the smaller the fitness value a reconstructed texture has, the better it matches the input texture. Figure 8 gives an example of three reconstructed textures of an input texture (the left image) with increasing fitness values, which are obtained by three different values of the predefined error limit. One can see that from left to right, the visual pattern difference between the reconstructed texture and the original texture increases as the fitness value increases.

### 5. APPLICATION TO TEXTURE SYNTHESIS

Given an input texture, how do we use the parameter estimator approach described in this paper and procedural texturing to synthesize the required texture on surfaces of 3D synthetic objects? This issue is discussed in this section, and some experimental results are also presented. It is noted that all surface synthesis results presented in this section are generated using 3D solid texturing techniques [18, 19] and rendered by Blue Moon Rendering Tools.

For a procedural input texture, it is straightforward to synthesize it onto surfaces of synthetic objects because we already know the texturing procedure. Basically, we first use the parameter estimation approach described earlier in this paper to estimate the parameter values of the input texture. Once the parameter values are known, we can then ask the procedure to generate synthetic textures (which should look similar to the input texture) onto arbitrary surfaces. The first two rows of Fig. 9 give two examples of procedural input textures and their synthesis results onto surfaces of different objects.

For an arbitrary input texture, such as a real texture image, it is relatively difficult to synthesize it onto arbitrary surfaces. Recent research on statistical texture analysis and synthesis [29, 32] has achieved some success in this area. An alternative method is to use a texturing procedure to model each type of input textures, for example a brick procedure to model brick-like textures. In this way, each input texture can be treated as a procedural texture, and then it is possible to synthesize it onto surfaces by first estimating the parameter values for the input texture using our parameter estimation approach. The last two rows of Fig. 9 give two examples of using real input textures and their synthesized textures onto surfaces of different objects.

There are two subtle issues of synthesizing an arbitrary input texture onto surfaces using this approach. First, it is difficult to have a general texturing procedure to model all types of real input textures. In this paper, we assume that for each type of real input textures, there is a corresponding texturing procedure to model it. Second, for a given input real texture, it is difficult to decide in a systematic way which texturing procedure we should use to
FIG. 8. Examples of reconstructed textures with varying fitness values. The first texture is the original, and the rest are the reconstructed textures with corresponding fitness values (i.e., errors) given underneath. From left to right, the fitness values increase.

FIG. 9. Examples of procedurally synthesizing input textures onto surfaces of 3D synthetic objects using the genetic-based multiresolution parameter estimation technique. Images in the first column are input textures, and the corresponding synthesized results are to the right of the input textures. The input textures in the first two rows are procedural textures, and the last two input textures are real images taken by a digital camera.
model that type of texture. To limit the scope of our work, for each input texture image, we require the user to tell the system the type of input texture so that the system can choose an appropriate procedure for it. For example, for a given wood input texture, the user must specify a wood type to the system. In the future, a texture recognition system could be developed and used with this approach.

6. CONCLUSIONS AND FUTURE WORK

Estimating the parameter values for procedural texturing has been an open problem in the literature. In this paper, a genetic-based multiresolution approach for estimating procedural texture parameters is presented. An important feature of our approach is to use a genetic-based algorithm to search the parameter space for a match with the target texture. The match between two texture images is done using an extended multiresolution version of Cross and Jain’s MRF texture parameter estimator. The experimental results have demonstrated the feasibility of our approach in estimating procedural texture parameter values.

We also show how to use our parameter estimation approach in the application of procedurally synthesizing an input texture onto surfaces of 3D synthetic objects. However, this is assumed that for each type of input textures, the user provides an appropriate texturing procedure to model them. For a procedural input texture, there is no such requirement. For a nonprocedural input texture, this may cause problems as discussed in Section 5. Therefore, future work should be directed toward finding a systematic way to design texturing procedures based on input textures. We are currently investigating issues along this very important direction.

APPENDIX: PROOFS OF THEOREMS 3.3–3.5

Proof of theorem 3.3. From Eq. (3.4), we have:

\[ L(C_k) \overset{\text{def}}{=} \sum_{q \in C_k} \ln[p(X(q) = g_q \mid \text{neighbors of } q)] \]

\[ = \sum_{q \in C_k} \ln \left[ \frac{n}{g_q} \left( \frac{e^{T(q)} g_q}{1 + e^{T(q)}} \right)^{g_q} \left( 1 - \frac{e^{T(q)}}{1 + e^{T(q)}} \right)^{n-g_q} \right] \]

\[ = \sum_{q \in C_k} \ln \left[ \frac{n!}{g_q! \cdot (n - g_q)!} \cdot \frac{e^{T(q) \cdot g_q}}{(1 + e^{T(q)})^n} \right] \]

\[ = \sum_{q \in C_k} \left[ \ln(n!) - \ln(g_q!) - \ln((n - g_q)!) + T(q) \cdot g_q \cdot \ln e - n \cdot \ln(1 + e^{T(q)}) \right] \]

\[ = \ln(n!) \cdot \sum_{q \in C_k} 1 + \sum_{q \in C_k} \left[ T(q) \cdot g_q - \ln(g_q!) - \ln((n - g_q)!) - n \cdot \ln(1 + e^{T(q)}) \right] \]

\[ = c_k \cdot \ln(n!) + \sum_{q \in C_k} \left[ T(q) \cdot g_q - \ln(g_q!) - \ln((n - g_q)!) - n \cdot \ln(1 + e^{T(q)}) \right]. \]
Proof of theorem 3.4. Using Eq. (3.4), we have:

\[
\frac{\partial L(C_k)}{\partial a} = \frac{\partial}{\partial a} \left[ c_k \cdot \ln(n!) + \sum_{q \in C_k} (T(q) \cdot g_q - \ln(g_q!) - \ln(n - g_q!) - n \cdot \ln(1 + e^{T(q)})) \right]
\]

\[
= \sum_{q \in C_k} g_q \cdot \frac{\partial}{\partial a} (T(q)) - n \cdot \sum_{q \in C_k} \frac{\partial}{\partial a} (\ln(1 + e^{T(q)}))
\]

\[
= \sum_{q \in C_k} g_q \cdot \frac{\partial}{\partial a} (T(q)) - n \cdot \sum_{q \in C_k} \left( \frac{e^{T(q)}}{1 + e^{T(q)}} \cdot \frac{\partial}{\partial a} (T(q)) \right).
\]

Since \( T(q) = a + \sum_{s=1}^{S} \sum_{j=1}^{4} b_{ij} s_{ij}(q) \), it is easy to see that \( \frac{\partial}{\partial a} (T(q)) = 1 \); thus we have:

\[
\frac{\partial L(C_k)}{\partial a} = \sum_{q \in C_k} g_q - n \cdot \sum_{q \in C_k} \left( \frac{e^{T(q)}}{1 + e^{T(q)}} \right).
\]

Similarly, note that \( \frac{\partial}{\partial b_{ij}} (T(q)) = s_{ij}(g_q) \); we have:

\[
\frac{\partial L(C_k)}{\partial b_{ij}} = \frac{\partial}{\partial b_{ij}} \left[ c_k \cdot \ln(n!) + \sum_{q \in C_k} (T(q) \cdot g_q - \ln(g_q!) - \ln(n - g_q!) - n \cdot \ln(1 + e^{T(q)})) \right]
\]

\[
= \sum_{q \in C_k} g_q \cdot \frac{\partial}{\partial b_{ij}} (T(q)) - n \cdot \sum_{q \in C_k} \frac{\partial}{\partial b_{ij}} (\ln(1 + e^{T(q)}))
\]

\[
= \sum_{q \in C_k} g_q \cdot \frac{\partial}{\partial b_{ij}} (T(q)) - n \cdot \sum_{q \in C_k} \left( \frac{e^{T(q)}}{1 + e^{T(q)}} \cdot \frac{\partial}{\partial b_{ij}} (T(q)) \right)
\]

\[
= \sum_{q \in C_k} g_q \cdot s_{ij}(g_q) - n \cdot \sum_{q \in C_k} \left( \frac{e^{T(q)}}{1 + e^{T(q)}} \cdot s_{ij}(g_q) \right).
\]

Using the method of maximum likelihood described in Section 2.1, we let \( \frac{\partial L(C_k)}{\partial a} \) and \( \frac{\partial L(C_k)}{\partial b_{ij}} \) be zero and solve the nonlinear equations for \( a, b_{ij} \). ■

Proof of theorem 3.5. From (3.5), it is straightforward to prove (3.6) using the chain rule by noting that

\[
\frac{\partial}{\partial a} \left( \frac{e^{T(q)}}{1 + e^{T(q)}} \right) = \frac{e^{T(q)}}{(1 + e^{T(q)})^2},
\]

and

\[
\frac{\partial}{\partial b_{lm}} \left( \frac{e^{T(q)}}{1 + e^{T(q)}} \right) = \frac{e^{T(q)} \cdot s_{lm}}{(1 + e^{T(q)})^2}.
\]

■

ACKNOWLEDGMENTS

The authors thank NSERC for the financial support. The first author thanks the University of Saskatchewan and the University of Alberta for the Graduate Scholarships. We also thank the anonymous referees for their valuable comments so that the quality of presentation of this paper could be significantly improved. Finally, we thank Ronald Pang for proofreading the paper.
REFERENCES

27. J. Stam and E. Fiume, Depicting fire and other gaseous phenomena using diffusion process, in 
28. G. Turk, Generating textures for arbitrary surfaces using reaction-diffusion, in ACM SIGGRAPH, Las Vegas, 
Wesley, Reading, MA, 1989.
31. L. Wei and M. Levoy, Fast texture synthesis using tree-structured vector quantization, ACM SIGGRAPH, 
Louisiana, 2000, pp. 479–488.
32. L. Y. Wei and M. Levoy, Texture synthesis over arbitrary manifold surfaces, in ACM SIGGRAPH, Los Angeles, 
33. A. E. Whiteside, Preparing data bases for perspective scene generation, in Proceedings of SPIE, Bellingham, 
36. S. C. Zhu, Y. Wu, and D. Mumford, Filters, random fields and maximum entropy: Towards a unified theory 