University of Saskatchewan

Proceedings of the Twelfth Annual Graduate Symposium on Computer Science

April 11, 2000

Department of Computer Science
Saskatoon, Saskatchewan, Canada

A Generalized Cellular Texture Basis Function

Xuejie Qin
Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, SK. S7N 5A9
Canada
E-mail: xuq414 @cs.usask.ca

Supervisor: Dr. Herbert Yang

Abstract

Texture basis function plays a very important role in procedural texturing. In the literature, there are only two texture basis functions proposed, namely Perlin's noise and Worley's noise. In this paper, a new generalized cellular texture basis function is proposed. This new texture basis function is a generalization of Worley's noise. The proposed basis function can be used in procedural texturing algorithms to generate some interesting texture patterns, such as crumpled wrinkle, wood, marble, cloud, flame and cell-like patterns.

1. Introduction

Texturing, a process of generating textures on the surface of 3D objects, is an important and active research topic in the field of computer graphics and image processing. In the literature, various texturing techniques have been proposed. All of these techniques can be classified into two categories: non-procedural and procedural.

In non-procedural techniques, synthesized textures are generated after analyzing or mapping input textures. Some of the successful models include texture mapping [Blinn, 1976; Bunker, 1984; Dungan, 1978; Green, 1986], and statistical and random models [Cross, 1983; Bonet, 1997; Heeger, 1995]. In procedural techniques, various procedures (algorithms) are developed to generate textures without requiring input textures. Textures are generated directly on the surface of 3D objects from calling a procedure. By defining different procedures, different kinds of textures can be generated. Some of the successful procedural techniques include shade tree [Cook, 1984], pixel stream editor [Perlin, 1985], reaction-diffusion system [Turk, 1991; Witkin, 1991], and a genetic texture-programming system [Sims, 1991].

Compared with non-procedural techniques, procedural techniques are extremely fast, highly realistic and storage-efficient. Nonetheless, there are still some challenging problems unsolved or ill solved in those procedural techniques. One of the challenging problems is to define some well-behaved texture basis functions so that they can be called by texturing procedures in different ways to generate various kinds of textures. In the literature, there are only two basis functions proposed, namely Perlin's noise [Perlin, 1985] and Worley's noise [Worley, 1996].

In this paper, a new generalized cellular texture basis function is proposed. This new basis function is a generalization of Worley's noise [Worley, 1996]. Worley's noise is a linear combination of n i^{th} closest feature point basis functions, whose values at a given point in the space are determined by that point and the feature points around that point. The proposed basis function extends the Worley's noise by including a non-linear combination of those i^{th} closest feature point basis functions. We found that using this generalized basis function will generate more interesting texture patterns than Worley's noise. In addition, this basis function can be used as a complement of Perlin's noise. Wood, marble, clouds and flame can be generated by using this new generalized basis function. In some cases, the textured images generated by the proposed basis function look

more interesting and realistic than those generated by Perlin's noise. Experimental results are presented to show these facts.

The remainder of this paper is organized as follows. The next section presents the literature review of previous works on texturing, focuses on procedural texturing. Section 3 presents the proposed generalized cellular texture basis function and the algorithm to compute this basis function. In section 4, we describe some applications of the proposed basis function in procedural texturing, and present some experimental results. Finally, conclusions are drawn in section 5.

2. Previous Works on Texturing

The work in this paper is closely related to texturing in the field of computer graphics. Texturing is a process of creating textures on the surface of 3D objects. The purpose of texturing is to add visual realism to computer synthetic images. Techniques for texturing can be classified into two categories: non-procedural and procedural.

2.1 Non-Procedural Texturing

In non-procedural approach, two different kinds of techniques may be used to generate synthesized textures: texture mapping techniques and texture analysis/synthesis techniques. With texture mapping techniques, input texture patterns are mathematically transferred first onto the surfaces of 3D models that are used to represent real world scenes. Then, the textured 3D surfaces are perspectively projected onto the output image viewing plane. Texture tiling [Dungan, 1978], cell texturing [Bunker, 1984], reflection mapping [Blinn, 1976] and environment mapping [Green, 1986] are some of the specific names to describe variations of the texture mapping techniques.

With texture analysis and synthesis techniques, an input texture patch is first analyzed, then texture information is characterized by a set of parameters, and finally the synthesized texture is generated based on those parameters. Some of the successful texture analysis and synthesis models include Markov Random Field texture models [Cross, 1983], multiresolution sampling procedure [Bonet, 1997] and pyramid-based texture analysis and synthesis [Heeger, 1995].

2.2 Procedural texturing

Procedural texturing has been proven to be a powerful approach to add visual detail to the surface of rendered objects in image synthesis. In this approach, textures are generated directly on the surface of 3D objects from calling a procedure, which may call other procedures. By defining different procedures, different kinds of textures can be generated.

Since the middle of 1980s, procedural techniques for generating realistic textures, such as marble, wood, stone, water, smoke, clouds, flame and other natural materials have gained widespread use in the field of computer graphics. Some useful procedural texturing techniques are summarized below. For a complete discussion of the procedural texturing techniques, we refer the readers to [Ebert, 1998].

Cook in [Cook, 1984] described a system called shade trees, which was one of the first systems in which it was convenient to generate procedural textures during rendering. Shade trees enable the use of different tree-structured shading models for different surfaces such as copper, wood, grass, and etc. The input parameters to the shading models, called *appearance parameters*, can be manipulated procedurally. In this way, shade trees make it possible to use textures to control any part of the shading calculation. Color and transparency textures, reflection mapping, bump mapping, displacement mapping and solid texturing can all be implemented using shade trees.

Perlin in [Perlin, 1985] described a complete procedural texture generation language (Pixel Stream Editor) and laid the foundation for the most popular class of procedural textures in use today, in particular those based on *noise*, a stochastic texture generation basis function. By using his noise basis function, Perlin has generated very convincing representations of clouds, fire, water, stars, marble, wood, rock, soap films, and crystal.

Turk in [Turk, 1991] and Witkin in [Witkin, 1991] described synthesis texture models inspired by reaction-diffusion. Reaction-diffusion is a process in which two or more chemicals diffuse at unequal rates over a surface and react with one another to form stable patterns such as spots and strips in the skins of animals.

Sims in [Sims, 1991] describes an interactive system to apply evolutionary techniques of variation and selection to create complex simulated structures, textures, and motions for use in computer graphics and animation. In his system, procedural textures are represented as LISP expressions. By interactively selecting among the resulting textures, the user of the system can direct the simulated evolution of a texture in some desired direction.

Worley in [Worley, 1996] proposed a cellular texture basis function, called Worley's noise, which can be used as solid texturing primitive to generate textured surfaces resembling flagstone-like tiled areas, organic crusty skin, crumpled paper, ice, rock, mountain ranges, and craters. Worley's noise is a linear combination of n i^{th} closest feature point basis functions, whose values at a given point in the space are the distances from the point to the feature points around that point.

2.3 Non-Procedural Texturing versus Procedural Texturing

Since the non-procedural texturing depends on input texture, the synthesized texture can be predicted and controlled as expected, while on the other hand, the synthesized texture is at most as good as the input texture. In addition, each model may only handle some specific kinds of textures, and may not work for other kinds of textures.

Compared with non-procedural techniques, procedural techniques have the advantages of extremely fast, high quality of realism and storage efficiency. Nonetheless, procedural approach has its own disadvantages. In general, it is difficult to develop a texturing procedure; there is no general procedure for all or at least some kinds of textures; and the resulting texture is difficult to predict and measure with a set of parameters.

3. The Generalized Cellular Texture Basis Function (GCTBF)

Texture basis function plays a very important role in procedural texturing. It can be used as a solid texturing primitive for building different kinds of textures in various texturing procedures. Unfortunately, there are only two texture basis functions proposed in the literate, namely Perlin's noise [Perlin, 1985] and Worley's noise Worley [Worley, 1996]. In this section, a new generalized cellular texture basis function is first formularized, and then the algorithm to compute this basis function is given.

3.1 Mathematical Formulation

The generalized cellular texture basis function, denoted by $GCTBF$, is a scalar function whose domain is the whole 3D space and its range is the set of non-negative real numbers, i.e. $GCTBF : \Re^3 \rightarrow \Re^+ \cup \{0\}$. For a given point x in $\Re^3$, $GCTBF(x)$ is defined as:

$$GCTBF(x) = GCTBF(F_1, F_2, ..., F_n)(x) = \sum_{i=1}^{n} c_i F_i(x) + \sum_{i,j=1}^{n} c_{ij} F_i(x) F_j(x) \quad (1)$$

where n, c_i, c_{ij} are constants known as priors for $i, j = 1, 2, 3, \ldots, n$.

In the above formula, the function F_i is called the i^{th} closest feature point basis function, whose value at x measures the distance from x to the i^{th} closest feature point around x. For example, $F_1(x)$ measures the distance from x to the first closest feature point, $F_2(x)$ measures the distance from x to the second closest feature point, and so on. It is easy to see that F_i is a continuous function of x, and that $F_i \leq F_{i+1}$ for $i = 1, 2, \ldots, n$. The feature points around x are generated by a stochastic distribution function, such as Poisson

distribution, Gamma-type distribution. We will discuss in more detail on how to generate the feature points around x in next subsection.

If we let $c_{ij} = 0$ in (1), then $GCTBF(x) = \sum_{i=1}^{n} c_i F_i(x)$, this is the Worley's basis function (Worley's noise) [Worley, 1996], which is a linear combination of $F_1, F_2, \ldots, F_n$. This implies that the proposed new texture basis function is a generalization of Woley's basis function, since $GCTBF(x)$ includes a non-linear combination of $F_i F_j$ for $i, j = 1, 2, \ldots, n$. Mathematically, the non-linear term $F_i F_j$ has more complex behaviour than the linear term F_i. Thus, using this new generalized basis function may generate more interesting texture patterns. In other words, the set of texture patterns generated by the new generalized basis function is a superset of the set of texture patterns generated by using Worley's basis function. Experimental results are presented in section 4 to show this fact.

In the formula (1), the coefficients c_i, c_{ij} are known as priors. Theoretically, they can be any value of a real number. In the actual implementation of this basis function in this paper, we leave the task of setting values for c_i, c_{ij} to the texturing procedures that call this basis function. For computation efficiency reason, we restrict the value of n in (1) up to 4.

3.2 Algorithm to Compute GCTBF

From (1) in the previous section, for a given point x in the space, the value of $GCTBF$ at x is easy to calculate if we know the value of F_i at x for $i = 1, 2, \ldots n$. By definition, $F_i(x)$ is the distance from x to the i^{th} closest feature point around x. If we could locate all the feature points around x, then it is easy to calculate $F_i(x)$. In this subsection, we first present the general algorithm, then the detailed algorithm to compute $F_i(x)$.

3.2.1 General Algorithm to Compute $F_i(x)$

As in [Worley, 1996], we generate the feature points around x by using Poisson distribution function. By this distribution function, the probability of m feature points occurring in a unit cube is given by the following formula:

$$p(m) = \lambda^m / (m! e^{\lambda}) \qquad (3)$$

where λ measures the mean density of feature points per unit cube, which is known as a prior, say $\lambda = 4$.

Using formula (3), we pre-compute the values of $p(m)$ for $m = 0, 1, 2, \ldots, M$, where M is the maximum number of feature points known as a prior, for example $M = 50$. Then, we store the values of $p(m)$ in a distribution look-up table.

For each point x in $\Re^3$, we use a unit cube to surround x. The base of the unit cube, which is defined as the coordinates of the bottom-left-front vertex of the cube, is equal to the integer part of the coordinates of x. Here we assume a left-handed coordinate system. We use the base of the unit cube to define a seed to initialize a fast random number generator. We generate a random number r using the already initialized random number generator, and look up r in the distribution look-up table. The value of m such that the corresponding $p(m)$ is the closest one to r will be the number of feature points required in the unit cube surrounding x.

After the number of feature points is calculated, the location of feature points must be calculated by using the same initialized random number generator; this is crucial to ensure that the algorithm works correctly to compute $F_i(x)$. In fact, the coordinates of each feature point in the cube can be computed by calling the same initialized random number generator three times. With the locations of all feature points in the cube computed,

the calculation of the distances from those feature points to x is easy. To get $F_1(x), F_2(x), \ldots, F_n(x)$, we just sort the distances into increasing order.

The algorithm should not stop at this point. Since the distance from a feature point to x may reach $\sqrt{3}$ in the extreme case, a feature point in the neighboring cube may even be closer to x than those already computed. Thus the algorithm should keep going to process the other 26 neighboring cubes. For each of the neighboring cubes, the algorithm just repeats the same steps described before to compute the number and the locations of feature points but using a different seed for the random number generator based on that cube. The distances from the feature points to x are then computed, and the list of $F_1(x), F_2(x), \ldots, F_n(x)$ is finally updated using a simple sorting algorithm. Figure 1 shows in three neighboring cubes the relationship between the point x, the feature points, and $F_1(x), F_2(x), \ldots, F_n(x)$ around x with $n = 4$.

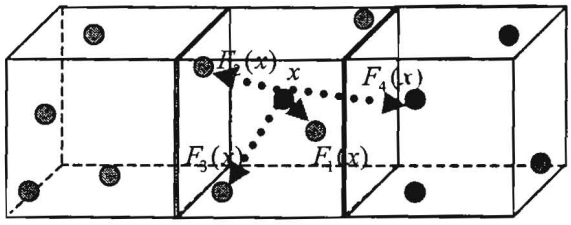

Figure 1: The relationship between the point x, the feature points, and $F_i(x)$ in the three neighboring cubes around x for $i = 1, 2, 3, 4$. The point x is represented by a red dot; the feature points are represented by green dots.

3.2.2 Detailed Algorithm to Compute $F_i(x)$

For a given point x in the space, we use six steps to compute $F_i(x)$. The first step is the initializations of the distribution look-up table and the permutation table that will be used in later steps. These two tables are pre-computed only once and stored as arrays. The steps from two to six are the main parts of the algorithm used to compute $F_i(x)$ for $i = 1,2,\ldots,n$. In this algorithm, we assume that a left-handed coordinate system is used. The detailed algorithm is given as follows.

1. Initialization and pre-computation.
   - Pre-compute the values of $p(m)$, for $m = 0,1,2,\ldots,M$, where $p(m) = \lambda^m / (m! e^{\lambda})$, $\lambda = 4$, and $M = 50$. Store them as a Possion distribution look-up table. Name it as *DistributionTable*[M].
   - Pre-compute a permutation array of size 2^{16}, each element of the array is a random number between 0 and $2^{16} - 1$ inclusively. Name it as *PermutationTable*[N], where $N = 2^{16}$.
2. For each point $x = (x_1, x_2, x_3)$ in the space, define a unit cube C_x containing x. The base of C_x is computed by $base(C_x) = (floor(x_1), floor(x_2), floor(x_3))$. For example, if $x = (1.1, 2.3, 3.4)$, then $base(C_x) = (1, 2, 3)$.
3. Compute the number of feature points in cube C_x using the pre-computed Possion distribution look-up table.
   - Compute a seed s for a fast random number generator using the pre-computed permutation table.

(1) $TABMASK = 2^{16} - 1$.

(2) $PERM(x) = PermutationTable[x \,\&\, TABMASK]$, *where & is the bitwise and operator.*

(3) $INDEX(x_1, x_2, x_3) = PERM(x_1 + PERM(x_2 + PERM(x_3)))$.

(4) $x_1 = xcomp(base(C_x))$, $x_2 = ycomp(base(C_x))$, $x_3 = zcomp(base(C_x))$

(5) $s = INDEX(x_1, x_2, x_3)$

- Initialize a random number generator using seed s.
- Generate the first random number r using the above initialized random number generator.
- Look up r in the table $DistributionTable[M]$. The value of m, such that $p(m)$ is the closest one to r in $DistributionTable[M]$ is the number of feature points required in C_x. Return the value of m.

4. Compute the location of the m feature points in the cube C_x. For each of the feature points, compute its xyz coordinates using the already initialized random number generator. These coordinates are relative to the base of C_x.
5. For each of the feature points, compute its distance to x, keep a sorted list of the n smallest distances $F_1(x), F_2(x), \ldots, F_n(x)$, where n is known as a prior, for example $n = 4$.
6. For each of the 26 neighboring cubes of C_x, repeat step 2 to step 5, and update the list of $F_1(x), F_2(x), \ldots, F_n(x)$ at the end of step 5.

4. Applications in Procedural Texturing Using GCTBF

The new proposed cellular texture basis function can be used as a solid texturing primitive for building various kinds of textures in procedural texturing. As with Perlin's noise, mapping the value of this basis function at a surface point into a color and normal-displacement can provide visually interesting and impressive effects. Combining color or bump mapping with fractal technique, a variety of texturing procedures can be implemented based on this new basis function to generate wrinkle, wood, marble, cloud, terrain, and flame-like textures. In this paper, the texturing procedures are implemented using RenderMan Shading Language (SL). We also implement the new basis function in C++ and link them as dynamic shared objects (DSO) so that we can call it as a SL built-in function in surface or displacement shaders written in Shading Language. In the rest of this section, some applications using this basis function in procedural texturing are described, and experimental results are presented.

4.1 Using GCTBF in Color Mapping

Color mapping is used to map a number into a color. The commonly used color mappings are the *spline* function and the *mix* function. The value of $mix(c_1, c_2, t)$ at t is equal to $(1-t)*c_1 + t*c_2$ with $t \in [0,1]$, where c_1, c_2 are colors known as priors. To map a surface point to a color, we can first use *GCTBF* to map that surface point into a number, then use a color mapping to map that number into a color. For example, for a given point x on the surface of a 3D synthetic object, the color at x, denoted $pattern(x)$, can be determined by the formula:

$$pattern(x) = color\ spline(clamp(GCTBF(x), 0, 1), c_1, c_2, \ldots, c_n) \qquad (4)$$

In the above formula, $c_1, c_2, \ldots, c_n$ are given colors. The function $clamp(s, a, b)$ returns a if s is less than a, b if s is greater than b; otherwise it returns s. To obtain different texture patterns, we can use different values for n and define different color values for $c_1, c_2, \ldots, c_n$. Note that some c_is may have the same value. For example, if we let $n = 13$, and $c_1 = c_2 = c_6 = c_7 = c_{12} = (0.25, 0.25, 0.35)$, $c_3 = c_4 = c_5 = (0.10, 0.10, 0.30)$, $c_8 = c_9 = (0.05, 0.05, 0.26)$, $c_{10} = c_{11} = c_{13} = (0.03, 0.03, 0.20)$, then we can generate blue-marble-like patterns on

the surface of objects. As a color mapping, the *mix* function can be used in a similar way as *spline* function. An example of using *mix* can be found in example 2 in section 4.3.

Figure 2 shows some patterns generated using (4) with $n=13$, and the values of $c_1, c_2, ..., c_n$ are given as before. To compare, we present the corresponding pattern generated by Perlin's noise. Figure 2 also shows that patterns generated using non-linear combinations of F_i are more interesting than patterns generated by linear combination of F_i for $i = 1, 2, ..., n$.

4.2 Using GCTBF in Bump Mapping

Bump mapping technique was first introduced by Blinn in [Blinn, 1978]. This technique involves modifying the surface normal vectors to give the appearance that the surface has bumps or indentations. As with Perlin's noise, the new basis function can also be used in bump mapping to create bumps on the surface of synthetic objects. For a given point x on the surface, the normal N at x can be modified using an algorithm like this:

$$\begin{aligned} &bump(x) = abs(0.5 - GCTBF(kx)) \\ &x = x - bump(x) * normalize(N) \\ &N = calculationnormal(x) \end{aligned} \qquad (5)$$

Figure 3 shows an example of creating bumps using (5) with $k = 3$ (the last four images). To compare, we also present the original image without bumps (the first image), and the one with bumps generated by Perlin's noise (the second image).

4.3 Using GCTBF in Fractal

Fractal technique offers a very easy way to generate geometrically complex objects like crumpled wrinkle, cloud, flame, water and terrain. The complexity arises simply through the repetition of form over some range of scale. Mathematically, we get a simple fractal when we take a basis function, scale it down, and add it again to the same function. For example, if we let $\sin(p)$ be a basis function of p, then $f(p) = \sum_{i=0}^{n} (\sin(l^i p)/l^i)$ is a fractal of p based on $\sin(p)$. In the expression, n is called *octaves*, which tells exactly how many times we should scale and add the basis function into itself; and l is called *lacunarity*, which tells how much we should scale the basis function for each octave, a typical value for l is 2. The composite of two fractal functions or a fractal with a simple function is also a fractal. For a complete discussion of fractal, the reader is referred to [Ebert, 1998].

Combining fractal with color mapping or bump mapping when using *GCTBF* in procedural texturing, we can generate even more interesting textures such as crumpled wrinkle, cloud, flame, water, and terrain. In the rest of this subsection, we present two examples to demonstrate how to combine fractal with color or bump mapping when using *GCTBF* as a solid texturing primitive.

Example 1: Combining fractal with bump mapping to generate crumpled wrinkles. To create crumpled wrinkles on a surface, we can use a fractal version of bump mapping in the algorithm described in (5), which is given in (6) below. Figure 4 shows some images generated using (6) with $GCTBF = F_2 - F_1 - F_1F_1$ and $octaves = 0, 1, 2, 3, 4$ respectively.

$$
\begin{aligned}
&fractal(x) = \sum_{i=0}^{octaves} abs(0.5 - GCTBF(2^i x))/2^i \\
&bump(x) = fractal(x) * fractal(x) * fractal(x) \\
&x = x - bump(x) * normalize(N) \\
&N = calculationnormal(x)
\end{aligned} \tag{6}
$$

Example 2: Combining fractal with color mapping to generate cloud. For a given point x on a surface, the cloud-like pattern, denoted $cloudPattern(x)$, can be given by the following algorithm:

$$
\begin{aligned}
&skycolor = (0.15, 0.15, 0.60) \\
&coludcolor = (1.00, 1.00, 1.00) \\
&fractal(x) = \sum_{i=0}^{octaves} \frac{1}{2^i}((1 - 2 * GCTBF(2^i x)) * smoothscale(2^i x)) \\
&coludPattern(x) = mix(skycolor, cloudcolor, smmothstep(0, 1, fractal(x)))
\end{aligned} \tag{7}
$$

In the above cloud-like-pattern-generating algorithm, the function $smoothscale(p)$ has a complex form and its value at the point p depends on the differential surface area at p. For more information on this function, we refer the reader to chapters 10-13 in the book [Ebert, 1998]. The *mix* function is the actual color mapping used in this algorithm, whose definition is given in section 4.1. The $smoothstep(\min, \max, t)$ function returns 0 if t is less than *min*, 1 if t is greater than or equal to *max*, and performs a smooth *Hermite* interpolation between 0 and 1 in the interval *min* to *max*. Figure 5 shows some cloud patterns generated using the algorithm described in (7). To compare, we also present the corresponding cloud pattern generated by Perlin's noise in Figure 5.

Before we end this subsection, we give three synthetic images textured by using *GCTBF*, as shown in Figure 6. Figure 6(a) shows a scene with a vase placed on a wooden table under a cloudy sky. The marble-like patterns on the vase, the wood patterns on the table and the clouds in the sky are generated by three different texturing procedures (surface shaders written in RenderMan Shading Language) that call *GCTBF*. Precisely, the marble-like pattern, the wood pattern and the cloud pattern used in the scene are generated by $\sum_{i=1}^{4}(-1)^i F_i + \sum_{i,j=1}^{4}(-1)^{i+j} F_i F_j$, $F_1 - F_1 F_1$, and $F_1 F_2 + F_2 F_3$ respectively. Figure 6(b) shows a scene with a tree trunk placed on a rocky ground under a cloudy sky. The wood pattern on the tree trunk and the cloud pattern in the sky are generated by $F_3 - F_3 F_3$ and F_1 respectively. The rocky ground is generated by the technique described in (6) of example 1 with $GCTBF = F_2 - F_1 - F_1 F_1$ and $octaves = 4$. Figure 6(c) shows a simple scene of flame. This scene is rendered using a single patch textured by a flame-texturing procedure that calls *GCTBF* with $GCTBF(x) = F_1(x)$. This flame-like texture is actually generated by combining fractal technique with *spline* color mapping.

5. Conclusion

Texture basis function plays an important role in procedural texturing. In the literature there are only a couple of texture basis functions proposed, namely Perlin's noise and Worley's noise. In this paper, a new generalized cellular texture basis function, called *GCTBF*, is proposed. This basis function is a generalization of Worley's noise. Worley's noise is a linear combination of n i^{th} closest feature point basis functions, whose values at a given point in the space are the distances from that point to the feature points. The new basis function extends Worley's noise by including a non-linear combination of those i^{th} closest feature point basis functions. Mathematically, the non-linear terms have more complex behaviors than the linear terms. Experimental results have also demonstrated this fact.

The new basis function can be used as a solid texturing primitive for building various kinds of textures in procedural texturing. There are three ways in using *GCTBF* in the application of procedural texturing: color

mapping, bump mapping and fractal. By combining color or bump mapping with fractal technique, a variety of texturing procedures can be implemented based on this new basis function to generate crumpled wrinkle, wood, marble, cloud, water and flame-like textures.

Reference

[Blinn, 1976] J.F. Blinn and M.E. Newell. Texture and Reflection in Computer Generated Images. CACM 19, 10(May), pp.542-547.

[Blinn, 1978] J.F. Blinn. Simulation of wrinkled surface. Computer Graphics, 12(3), pp.286-292, July 1978.

[Bonet, 1997] J.S. Bonet. Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Images. Computer Graphics, pp.361-368, 1997.

[Bunker, 1984] M. Bunker, R. Economy and J. Harvey. Cell texture - Its Impact on Computer Image Generation. In Proceedings of the Sixth Interservice/Industrial Association, Washington, DC, pp149-155, October 1984.

[Cook, 1984] R.L. Cook. Shade Tress. Computer Graphics, 18(3), pp.223-231, July 1984.

[Cross, 1983] G.C. Cross and A.K. Jain. Markov Random Field Texture Models. IEEE Transactions on Pattern Analysis and Machine Intelligence 5, pp25-39, 1983.

[Dungan, 1978] W. Dungan, A. Stenger and G. Sutty. Texture Tile Considerations for Raster Graphics. Computer Graphics, 12(3), pp.130-134, August 1978.

[Ebert, 1998] D.S. Ebert, F.K. Musgrave, K.P. Peachey, K. Perlin and S. Worley. Texturing and Modeling: A Procedural Approach. Academic Press, 1998.

[Francos, 1993] J.M. Francos, A.Z. Meriri and B. Porat. A Unified Texture Model Based on a 2D Wold-Like Decomposition. IEEE Transactions on Signal Processing 41, pp.2665-2678, 1993.

[Green, 1986] N. Green. Environment Mapping and Other Applications of World Projection. IEEE Computer Graphics Application, Nov. pp.21-29, 1986.

[Heeger, 1995] A.J. Heeger and J.R. Bergen. Pyramid-Based Texture Analysis/Synthesis. Computer Graphics, pp.229-238, 1995.

[Lewis, 1984] J.P. Lewis. Texture Synthesis for Digital Painting. Computer Graphics, 18(3), pp.245-252, 1984.

[Perlin, 1985] K.Perlin. An Image Synthesizer. Computer Graphics, 19(3), pp.187-296, July 1985.

[Pixar, 1989] Pixar. The RenderMan Interface: Version 3.1. Pixar, San Rafael, California, 1989.

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

[Schachter, 1979] B.J. Schachter and N. Ahuia. Random Pattern Generation Processes. Computer Graphics and Image Processing, 10, pp.95-114, 1979.

[Sims, 1991] K. Sims. Artificial Evolution for Computer Graphics. Computer Graphics, 25(4), pp. 319-328, July 1991.

[Turk, 1991] G. Turk. Generating Textures for Arbitrary Surfaces Using Reaction-Diffusion. Computer Graphics, 25(3), pp.289-298, July 1991.

[Witkin, 1991] A. Witkin and M.Kass. Reaction-Diffusion Textures. Computer Graphics, 25(3), pp.299-308, July 1991.

[Worley, 1996] S. Worley. A Cellular Texture Basis Function. Computer Graphics, pp.291-294, July 1996.

Acknowledgment

I would like to thank Dr. Herbert Yang for his patient supervision of this paper.

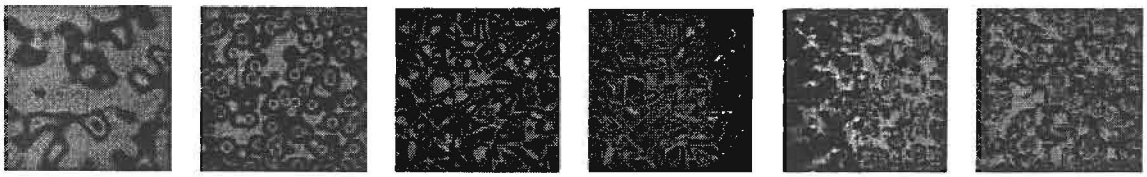

Figure 2: An example of using *GCTBF* in color mapping. The first pattern is generated by Perlin's noise; the rest of them are generated by *GCTBF* with five different combinations shown above.

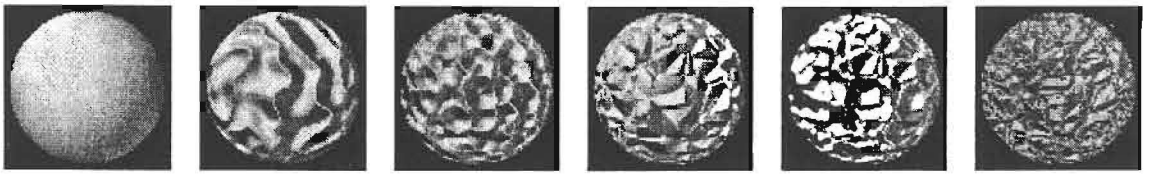

Figure 3: An example of creating bumps using *GCTBF* (the last four). The first is the original image without bumps; the second is the one with bumps generated by Perlin's noise.

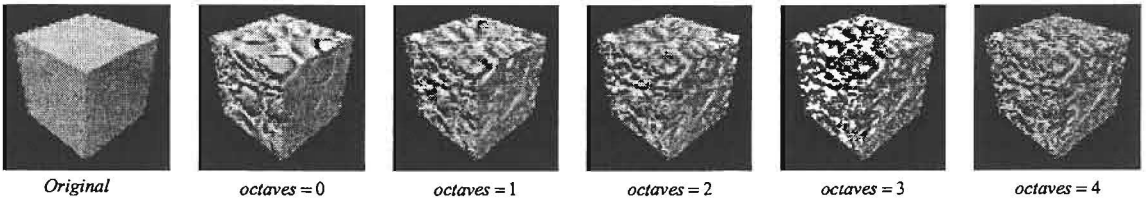

Figure 4: Sample images generated by using algorithm described in (6) with $GCTBF = F_2 - F_1 - F_1F_1$. From left to right, the values of octaves used are 0, 1, 2, 3, and 4. The first is the original image.

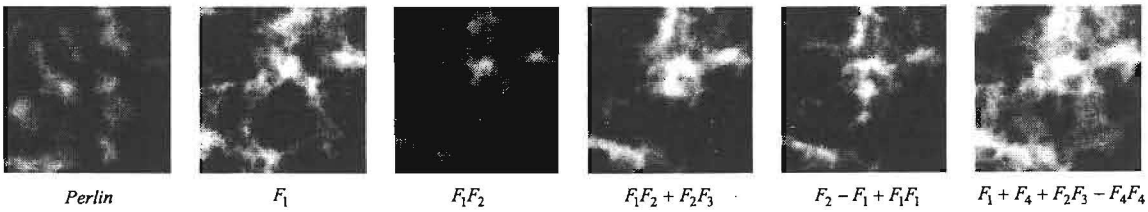

Figure 5: Sample cloud patterns (the last five) generated using the algorithm described in (7) using *GCTBF* with five different combinations shown above. The first pattern is generated by Perlin's noise.

Figure 6: An example of textured images using *GCTBF*

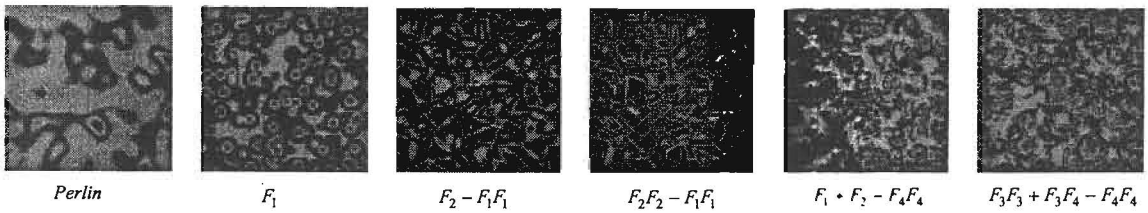

Figure 2: An example of using *GCTBF* in color mapping. The first pattern is generated by Perlin's noise; the rest of them are generated by *GCTBF* with five different combinations shown above.

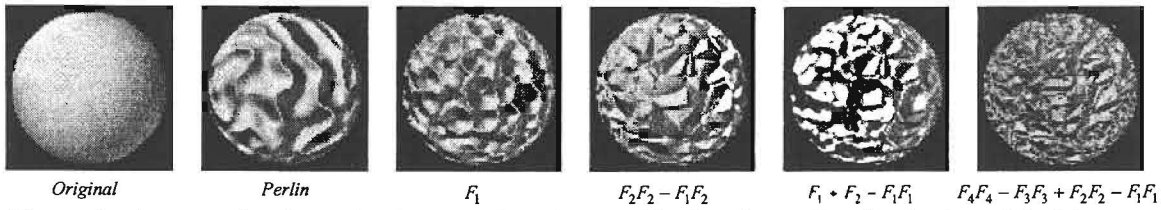

Figure 3: An example of creating bumps using *GCTBF* (the last four). The first is the original image without bumps; the second is the one with bumps generated by Perlin's noise.

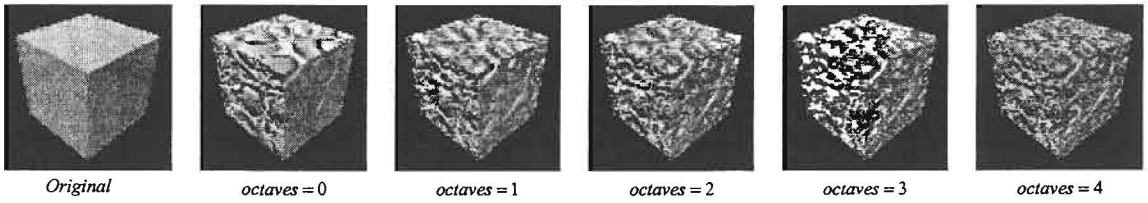

Figure 4: Sample images generated by using algorithm described in (6) with $GCTBF = F_2 - F_1 - F_1F_1$. From left to right, the values of octaves used are 0, 1, 2, 3, and 4. The first is the original image.

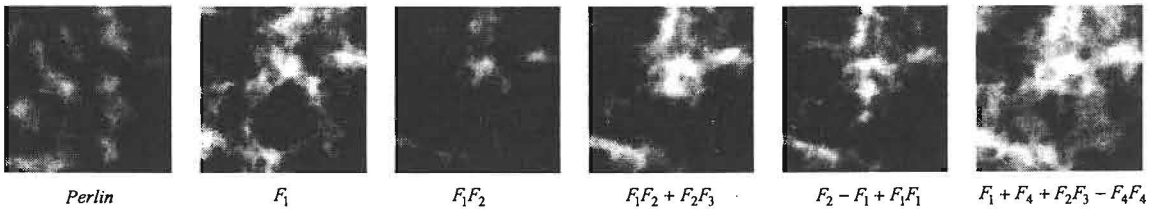

Figure 5: Sample cloud patterns (the last five) generated using the algorithm described in (7) using *GCTBF* with five different combinations shown above. The first pattern is generated by Perlin's noise.

Figure 6: An example of textured images using *GCTBF*