Vol. 35 No. 7 Transactions of Information Processing Society of Japan July 1994

Regular Paper

A Method of Generating Textures by Using Skeleton Lines

KAZUNORI MIYATA†

A method is given for synthesizing texture by using a set of skeleton lines and attribute functions. The attributes of each point in a texture, such as color, bump, and optical attributes, are calculated by applying attribute functions to the defined skeleton lines. This method has the following three merits: First, combining a set of skeleton lines and attribute functions allows a variety of textures to be obtained. Second, this method has flexible data definitions; the skeleton data can be defined not only by means of mathematical functions with control parameters, but also graphically by means of an input device, such as a tablet or mouse. Third, the user can preview a texture together with its skeleton lines. This is useful, because the texture generation procedure takes a significant amount of time. We also show the language specifications for generating a texture.

1. Introduction

One of the main objectives in computer graphics is highly realistic image synthesis. Textures, which are attributes of objects' surfaces, are critical to generating realistic images. Simply by mapping a texture to a surface, we can obtain a far richer image than a flat color surface. The majority of current graphic workstations can do texture mapping in real time,[1] and this technology plays a very important role in applications for amusement and training, such as car racing games and flight simulators.

There have been several research reports on topics related to texture synthesis, such as noise-based textures,[2)-4)] sparse convolution textures,[5] solid textures,[6] enhanced solid textures,[7] and reaction-diffusion textures.[8),9)] These methods can generate a variety of textures that are adequate for generating a realistic image. But there is still a big problem; it is hard for a user to imagine what kind of texture will be generated simply by looking at its parameters. This is a problem of correspondence between a synthesized texture and its parameters. Furthermore, it is difficult to design a new texture entirely by conventional procedural approaches. A user can hardly design a texture freely without a knowledge of texture generation. Novice users are still in trouble.

† IBM Research, Tokyo Research Laboratory

Our method offers a solution to these problems, and has the following three merits:

1. Combining a set of skeleton lines and attribute functions allows a variety of textures to be obtained.
2. Data definitions are flexible; the skeleton data can be defined not only by means of mathematical functions with control parameters, but also graphically by means of an input device, such as a tablet or mouse.
3. The user can preview a texture together with its skeleton lines. This is useful, because the texture generation procedure takes a significant amount of time.

2. Basic Concept

A texture is a surface attribute of an object, and has many components, such as color, gloss, and roughness. In our method, a texture is divided into three classes: bump, color, and optics. A bump class has values of displacement, H, from a base level. A color class has surface color values made up of three components. These components are related to the color model: in this paper, we use the RGB model, whose components are red, green, and blue (R, G, B). An optics class has optical features of a surface, and these depend on the shading model: in this paper, we use Phong's model, whose components are ambient, diffuse, specular, and shiny (A, D, S, N). The attributes are stored in

bump, color, and optics buffers, respectively.

Texture generation involves the arrangement of these features. That is, a texture synthesis method can be defined as a combination of the definition of features and the arrangement of those features. In the proposed method, skeleton lines represent the arrangement of a texture's features, and attribute functions define the features themselves.

3. A Method for Texture Synthesis

In this section, we describe how to generate a variety of textures by combining skeleton lines and attribute functions. We first give an overview of the generation, and then explain each function in detail.

3.1 Overview

Figure 1 shows the data flow in this system. First, skeleton lines, the feature data of an intended texture, are defined by means of procedures or graphical input methods. An attribute function is defined for each skeleton line, and the function's parameters are then specified for each key point on the skeleton lines. Here, an attribute function must be unique for each skeleton line.

Next, each line segment of the skeleton lines is rasterized by means of the Digital Difference Analyzer (DDA)[10] module. At the same time, each parameter of an attribute function is interpolated linearly with each key point. Noise can be added to skeleton lines or sets of parameters by using a noise module if so desired.

For each rasterized point, an attribute function with these calculated parameters is then applied. The data generated by attribute functions are stored in data buffers by means of storing operators, and then converted into vari-

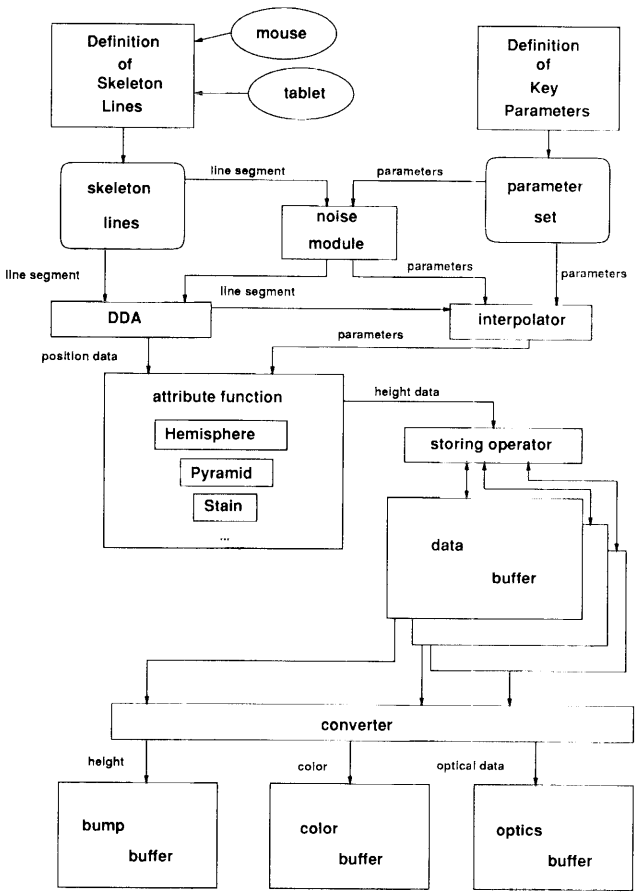


Fig. 1 Data flow.

ous attribute values by means of converters. The final texture is rendered by using these attribute buffers.

3.2 Skeleton lines

The term 'skeleton lines' means feature lines of a texture. For example, a wood grain texture, shown in **Fig. 2**, has a set of skeleton lines represented by distorted ellipses, and a marble texture, shown in **Fig. 3**, has a set of skeleton lines represented by perturbed stripes.

Skeleton lines only have position data, so the other parameters for an attribute function, such as the height and width, are specified for each key point on the skeleton lines. For a crack in a wall, for example, the width of the crack is specified at the top and bottom, and then interpolated linearly at each point.

A noise module can be used to perturb the skeleton lines or assigned parameters in order to make them more natural. Here, we use white noise and fractal noise ; white noise is generated by using uniform random numbers, and fractal noise is generated by the midpoint displacement method.[11]

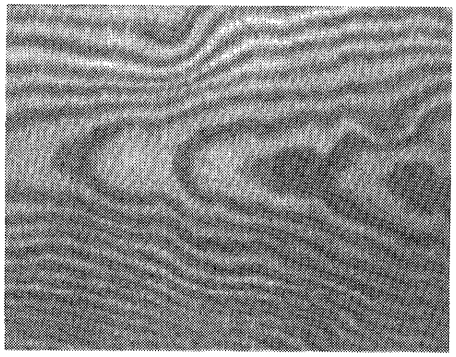


Fig. 2 Wood grain texture.

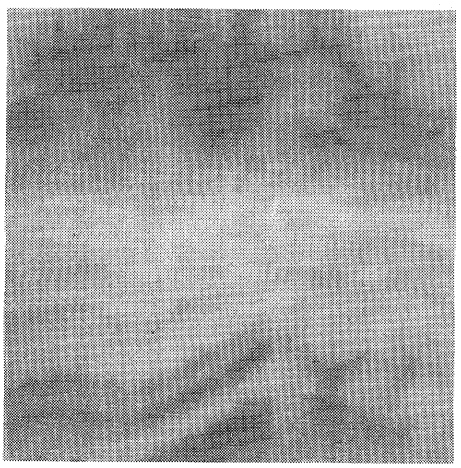


Fig. 3 Marble texture.

These defined skeleton lines are rasterized through the DDA module. At the same time, each parameter of an attribute function is interpolated linearly with each key point, and then stored.

The skeleton lines are defined not only by means of procedures, but also graphically by means of an input device, such as a mouse.

3.3 Attribute functions

An attribute function is used to generate attribute data for each point in a texture by tracing the defined skeleton lines, as shown in **Fig. 4**. The attribute function can generate only height data, which are stored in data buffers by means of storing operators and converted into attribute values by means of converters.

Some sample attribute functions are as follows ; here, the positioning parameters in each attribute function, C_x and C_y, correspond to the rasterized points of a skeleton line.

- **Hemisphere** (C_x, C_y, Z_{min}, Z_{max}, *Radius*)
  A hemisphere function generates height data for each point on the surface of a hemisphere whose center is (C_x, C_y, 0) and whose radius is given by Radius. This surface is clipped by its Z value, between

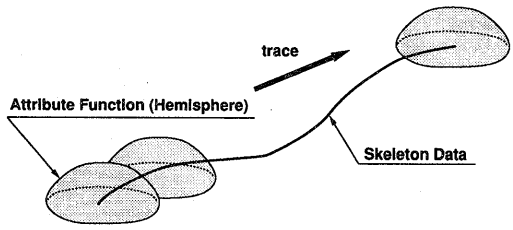


Fig. 4 Tracing on a skeleton lines.

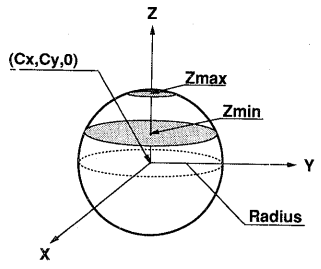


Fig. 5 Hemisphere.

Z_{min} and Z_{max}, as shown in **Fig. 5**.

- **Pyramid** (C_x, C_y, L_x, L_y, *Height*)
  A pyramid function, shown in **Fig. 6**, generates height data for each point on the surface of a pyramid the center of whose base is (C_x, C_y, 0), whose height is given by Height, and whose base is a rectangle with sides L_x and L_y.
- **Stain** (C_x, C_y, *Attack*, *Decay*)
  A stain function, shown in **Fig. 7**, generates height data for each point on the surface of the figure obtained by revolving an exponential function, given by Eq. (1), around the Z-axis. The axis of revolution passes through (C_x, C_y, 0).

$$Z(r) = Attack \cdot \exp(-Decay \cdot r^2) \quad (1)$$

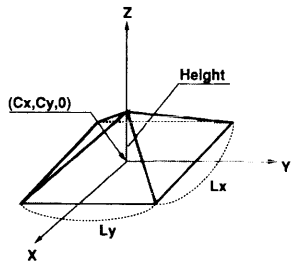


Fig. 6 Pyramid.

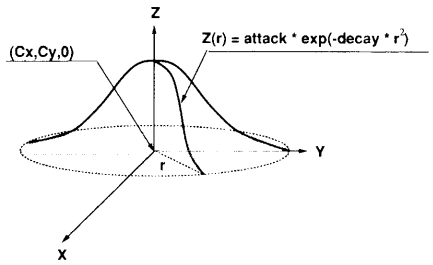


Fig. 7 Stain.

3.4 Storing operators

A storing operator is used when height data are stored in a data buffer. We now provide the following four types of storing operator:

MIN: stores the minimum value in a texture buffer

MAX: stores the maximum value in a texture buffer

ADD: stores the accumulated value in a texture buffer

RPL: replaces the value of a texture buffer with the current value

Figure 8 shows how each operator works.

3.5 Converters

A converter is used when height data are converted into attribute values. This module is made up of two functions: The selector, which selects an attribute buffer, and the mapper, which modifies the height data.

We now provide the following three types of mapper:

NOP: No operation.

CLIP (min, max): The data are clipped between min and max.

LINEAR (I_{min}, I_{max}, O_n, O_{min_0}, O_{max_0}, $\cdots$): The input data Z are assumed to lie in a range between I_{min} and I_{max}. O_n data OZ_i, $O_{min_i} < OZ_i < O_{max_i}$, are given by Eq. (2).

$$OZ_i = (Z - I_{min}) * (O_{max_i} - O_{min_i}) / (I_{min} - I_{max}) + O_{min_i} \quad (2)$$

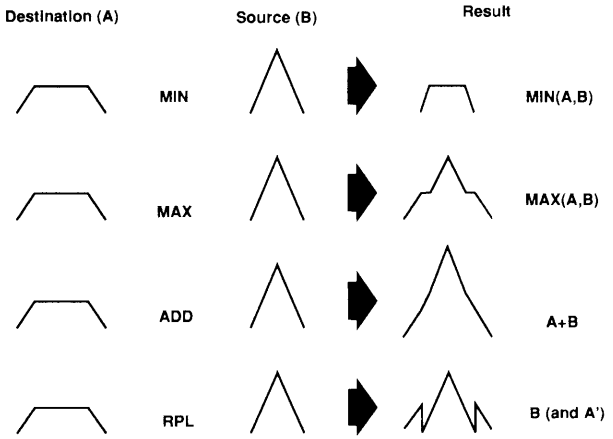


Fig. 8 Storing operators.

For example, coloring with the Z value Red = 100-200, Green = 0-255 and Blue = 150-180 is defined by LINEAR (0, 100, 3, 100, 200, 0, 255, 150, 180).

4. Descriptive Language

In this section, we gives the descriptive language of our texture generation method.

4.1 Functions

The descriptive language is made up of the following functions.

Skeleton lines are defined by bracketing each set of polygons between a call to **BgnSkeleton**() and a call to **EndSkeleton**(). A polygon is defined by bracketing each set of vertices between a call to **BgnPoly**() and a call to **EndPoly**(). The texture is defined by bracketing each set of **DefineClass**(), **DefineStrOp**(), **DefineMapper**(), and **CallSkeleton**() between a call to **BgnTexture**() and a call to **EndTexture**().

The descriptions of the various functions are as follows:

BgnSkeleton (id) : Marks the beginning of a

```
BgnSkeleton(1);
   BgnPoly(1);
      Vertex(1,   0,  0);
      Vertex(2, -32, 32);
   EndPoly();
EndSkeleton();

BgnSkeleton(2);
   BgnPoly(1);
      Vertex(1,  0,  0);
      Vertex(2, 32, 32);
   EndPoly();
EndSkeleton();

AssignParam(ALL, ALL, ALL, 0.0, 6.0, 6.0, NULL_P);
            /* ALL = -1, NULL_P = End of Parameter List */

BgnTexture();
  DefineClass(BUMP);
  DefineStrOp(MAX);
  DefineMapper(NOP);

  m[0][0] = 1.0; m[1][0] = 0.0; m[2][0] = 0.0;
  m[0][1] = 0.0; m[1][1] = 1.0; m[2][1] = 0.0;
  m[0][2] = 0.0; m[1][2] = 0.0; m[2][2] = 1.0;

  for(y = 0; y <= HEIGHT; y += 64){
    for(x = 0; x <= WIDTH; x += 64){
      m[2][0] = x;
      m[2][1] = y;

      CallSkeleton(1, HemiShpere, m);
    }
  }

  for(y = 32; y <= HEIGHT; y += 64){
    for(x = 0; x <= WIDTH; x += 64){
      m[2][0] = x;
      m[2][1] = y;

      CallSkeleton(2, HemiShpere, m);
    }
  }

EndTexture();
```

Fig. 9 Sample description.

definition of skeleton lines whose ID is given by the parameter id.

EndSkeleton () : Marks the end of a definition of skeleton lines.

CallSkeleton (id, afunc, matrix) : Associates the attribute function (given by afunc) with the skeleton lines whose ID is given by id, by applying a transformation matrix (given by matrix). Here, if the matrix is NULL, then it is an identity matrix.

Vertex (id, x, y) : Defines the vertex whose ID is given by id at the position (x, y). This function must be bracketed by a call to BgnPoly() and a call to EndPoly().

BgnPoly (id) : Marks the beginning of a polyline list whose ID is given by id.

EndPoly () : Marks the end of a polyline list.

BgnTexture () : Marks the beginning of a definition of a texture.

EndTexture () : Marks the end of a definition of a texture.

DefineClass (class) : Defines the selector for attribute buffers with a parameter class (={BUMP | COLOR | AMBIENT | DIFFUSE | SPECULAR|SHINY})

DefineStorOp (op) : Defines the storing operator with a parameter op (={MIN|MAX|ADD |RPL}).

DefineMapper (func, param) : Defines the mapper function (given by func (={NOP|CLIP| LINEAR}) and its parameter (given by parameter list=param).

AssignParam (SID, PID, VID, param) : Specifies parameters at key points. Here,

SID : skeleton line ID (for all skeleton lines, if SID is negative)

PID : polyline ID (for all polylines, if PID is negative)

VID : vertex ID (for all vertices, if VID is negative)

param : parameter list

4.2 Sample description

Figure 9 shows the sample description for the texture in **Fig. 10** 1.

First, two skeleton lines are defined, and the parameter list is defined for all skeleton lines. Then, the selector for the texture, its storing operator, and the mapper are called. After that, the two defined skeleton lines are called separately by applying a translation matrix to trace them with an assigned attribute function, hemisphere().

Table 1 Attribute functions, storing operators, and class.

No.	Attribute Function	Storing Op	Selector
1	hemisphere	MAX	bump
2	pyramid	MAX	bump
3	pyramid	MAX	bump
4	hemisphere (Zmin, Zmax<0)	MIN	bump
5	stain	MAX	color
6	pyramid (Height <0)	MIN	bump
7	hemisphere	MAX	bump
8	hemisphere	ADD	bump

5. Results

The sample textures shown in Fig. 10 were generated by using an IBM RISC System/6000 and C language for programming. Each result is a 512×512 (8 bits per R-, G-, B-plane) image. The calculation time depends on the number of skeleton lines and attribute functions, and averages about 1 minute, including the rendering process.

In the rendering process, the bump data are used to calculate the normal vector of each point with its adjoining points.

In Fig. 10, the left side shows the generated texture and the right side shows the texture's skeleton lines. The attribute functions and storing operators of these textures are listed in **Table 1**.

For these examples, we intended to generate the following textures, respectively :

(1) a non-slip plate, (2) a coarse green rug, (3) raked ground, (4) sprayed plaster, (5) a wine-colored board, (6) a crack in a wall, (7) a metallic hatch, and (8) blood vessels.

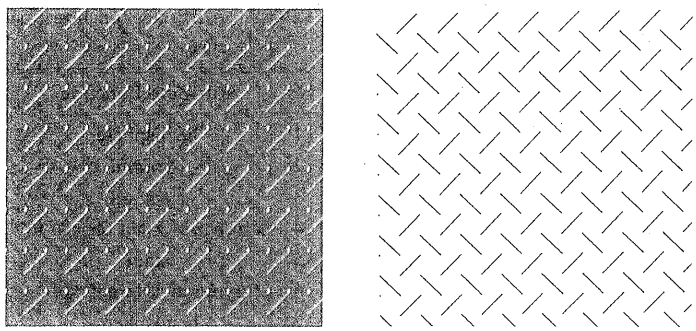
Figure 11 shows an image of cakes. Each texture (cakes and cups) was generated by this method.

Figure 12 shows an image of high-tech cuneiform. The cuneiform, which has an optical attribute of gold, is carved on a rough wall.

6. Conclusions

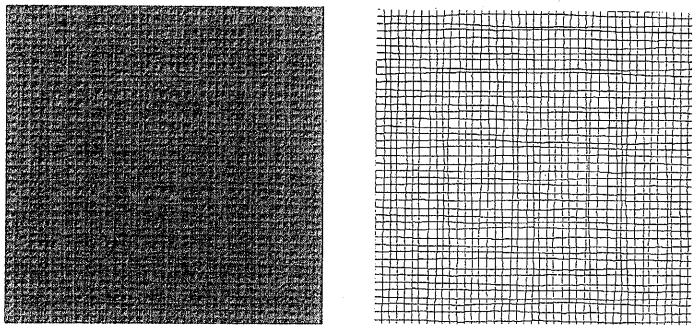
We have outlined a method for generating a veriety of textures by using skeleton lines and attribute functions.

This method has the limitation that it cannot



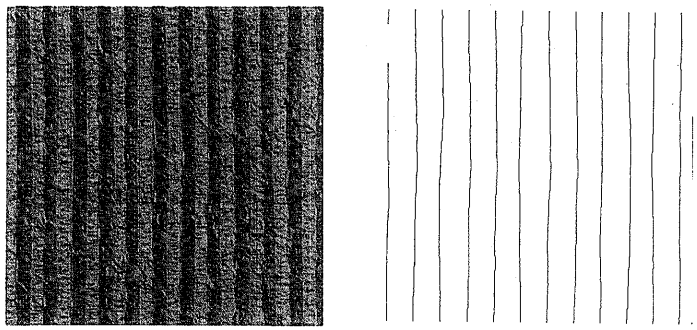
(a) Texture (b) Skeleton lines

Example # 1



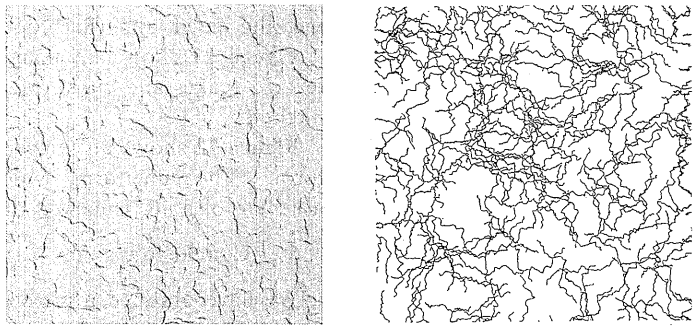
(a) Texture (b) Skeleton lines

Example # 2



(a) Texture (b) Skeleton lines

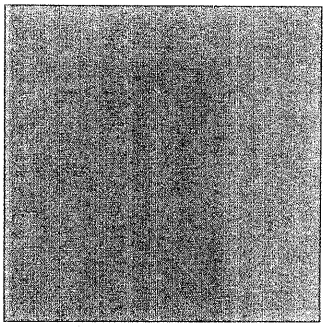
Example # 3



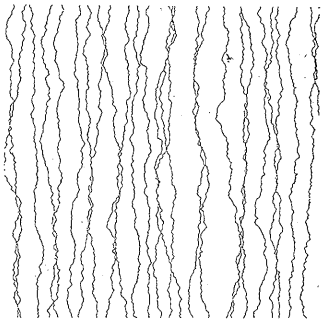
(a) Texture (b) Skeleton lines

Example # 4

Fig. 10 Examples.

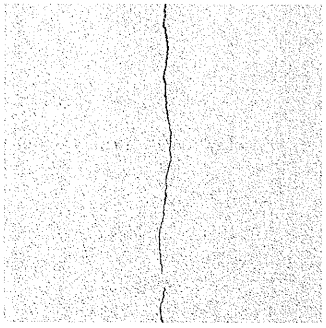


(a) Texture



(b) Skeleton lines

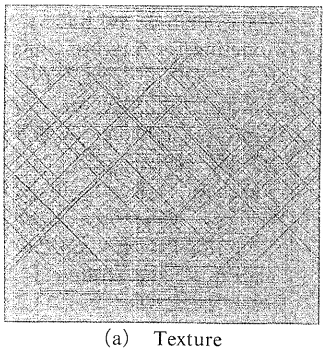
Example # 5



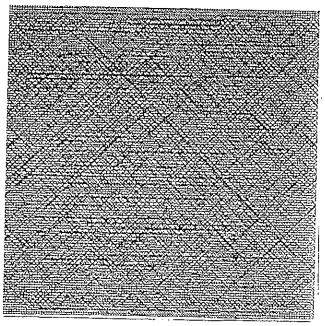
(a) Texture

(b) Skeleton lines

Example # 6

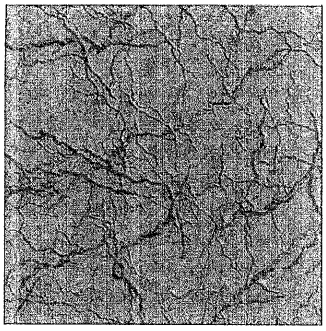


(a) Texture

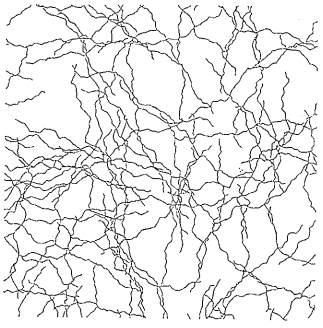


(b) Skeleton lines

Example # 7



(a) Texture



(b) Skeleton lines

Example # 8

Fig. 10 (Continued)

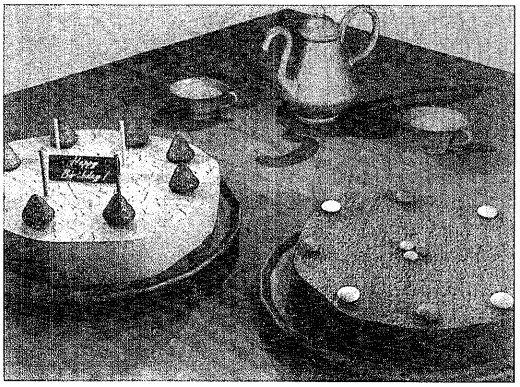


Fig. 11 Birthday cakes.

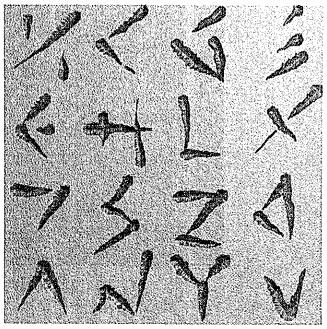


Fig. 12 High-tech cuneiform.

handle sectional textures, such as stone wall patterns and the scales of reptiles, because it is hard to define their skeleton lines.

In the future, we plan to work on the following areas:

1. Expansion of the texture library

Our method can treat only symmetrical attribute functions, such as hemispheres and pyramids. Therefore, it is difficult to generate non-symmetrical and directional textures, such as wood grains. We plan to enhance the method to enable it to treat such textures.

We also plan to extract skeleton lines and attribute functions from natural textures, and store them in a texture library.

2. Anti-aliasing

Laying down spheres or pyramids at integer pixel points can lead to aliasing problems. This problem is not noticeable in a complex texture, but it must be solved in order to generate a smooth texture.

3. Interactive texture generator

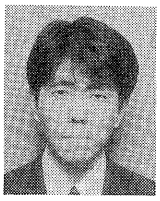
This method can be used to make an interactive texture generator. For example, skeleton lines can be defined for a texture by using a mouse, and an attribute function to be applied to the defined skeleton lines can be chosen in the same way as a brush or pen in a conventional drawing tool. That is, a user can design his or her own texture by means of this method without special knowledge of texture synthesis.

Acknowledgements I would like to thank researchers at the Tokyo Research Laboratory, IBM Japan, for their helpful suggestions.

References

1) Akeley, K.: RealityEngine Graphics, *SIGGRAPH '93 Conference Proceedings*, pp. 109–116 (1993).
2) Perlin, K.: An Image Synthesizer, *Computer Graphics*, Vol. 19, No. 3 (SIGGRAPH '85), pp. 287–296 (1985).
3) Lewis, J. P.: Algorithms for Solid Noise Synthesis, *Computer Graphics*, Vol. 23, No. 3 (SIGGRAPH '85), pp. 263–270 (1985).
4) Haruyama, S. and Barsky, A.: Using Stochastic Modeling for Texture Generation, *IEEE CG & A*, Vol. 4, No. 3, pp. 7–19 (1984).
5) Lewis, J. P.: Texture Synthesis for Digital Painting, *Computer Graphics*, Vol. 18, No. 3, pp. 245–251 (1984).
6) Peachey, D. R.: Solid Texturing of Complex Surfaces, *Computer Graphics*, Vol. 19, No. 3 (SIGGRAPH '85), pp. 279–286 (1985).
7) Perlin, K. and Hoffert, E.: Hypertexture, *Computer Graphics*, Vol. 23, No. 3 (SIGGRAPH '89), pp. 253–262 (1989).
8) Turk, G.: Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion, *Computer Graphics*, Vol. 25, No. 4 (SIGGRAPH '91), pp. 289–298 (1991).
9) Witkin, A. and Kass, M.: Reaction-Diffusion Texture, *Computer Graphics*, Vol. 25, No. 4 (SIGGRAPH '91), pp. 299–308 (1991).
10) Bresenham, J. E.: Algorithm for Computer Control of a Digital Plotter, *IBM Systems Journal*, Vol. 4, No. 1, pp. 25–30 (1965).
11) Fournier, A., Fussell, D. and Carpenter, L.: Computer Rendering of Stochastic Models, *CACM*, Vol. 25, No. 6, pp. 371–384 (1982).

(Received December 20, 1993)
(Accepted February 17, 1994)



Kazunori Miyata received a B.E. from Tohoku University in 1984 and an M.E. from Tokyo Institute of Technology in 1986. Since 1986, he has been a researcher at Tokyo Research Laboratory, IBM Japan, Ltd. His research interests include computer graphics, fractal theory, and natural phenomena. He is a member of IPSJ; the Institute of Electronics, Information, and Communication Engineers (IEICE), Japan; and the ACM.